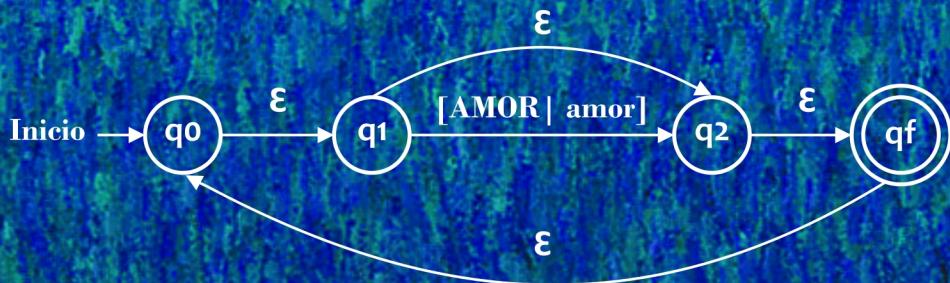


TEORÍA DE LA COMPUTACIÓN



Sara Méndez García

TEORÍA DE LA COMPUTACIÓN

Sara Méndez García

Teoría de la Computación
Sara Méndez García

Primera Edición: Julio 2016
© Sara Méndez García, 2016
Todos los derechos reservados.

Sara Méndez García
smendez@ipn.mx

Diseño de Portada: GM Producciones

ISBN: 978-607-97197-1-5

Derechos Reservados

Esta obra es propiedad intelectual de su autor y los derechos de publicación en lengua española han sido legalmente transferidos al editor. Prohibida su reproducción parcial o total por cualquier medio sin permiso por escrito del propietario de los derechos del copyright.

Impreso en México. Printed in México



"Gracias a Dios porque he comprobado que soy una de sus hijas predilectas"

Agradezco y ofrezco este trabajo a:
*Autoridades del Instituto Politécnico Nacional
y de la Unidad Profesional Interdisciplinaria
de Ingeniería y Ciencias Sociales y
Administrativas, "mi alma mater"*

A mis hijas Erika, Gladys y Marisol porque las amo profundamente, quiero ser la madre en la que puedan confiar, me gusta escucharlas con amor, paciencia y anhelo vivir en sus corazones, aunque ya no esté en este plano físico.

A mi hijo Esteban y su linda esposa Violeta, porque me gusta ser su apoyo y compañía siempre que ustedes también quieran.

A mis nietos, Pisho y Saúl, quiero ser su abuela favorita.

A ti Víctor, porque siempre deseo tener una buena relación de pareja y comprensión mutua.

A mi madre, que al ser yo tu única hija, voy a ser tú apoyo para el resto de nuestras vidas.

A mis estudiantes, aplico el saber más para servirles mejor.

A los profesores de la academia de computación por el tiempo dedicado a este mi primer libro.

A mis amigos de ayer, nuevos amigos y hasta a mis enemigos todos son bendecidos en mis oraciones diarias.

Gracias mil por todo y a todos los que contribuyeron en este libro.

CONTENIDO

UNIDAD I MARCO CONCEPTUAL	1
1.1 Sistemas Formales	2
1.2 Teoría de conjuntos	11
1.3 Conceptos de lenguajes	17
1.4 Conceptos De Expresiones Regulares	42
1.5 Conceptos de autómatas finitos	45
1.6 Temas complementarios	52
UNIDAD II AUTÓMATAS FINITOS	61
2.1 Transformación de expresión regular a autómata finito no determinista con épsilon transiciones	63
2.2 Transformación de Autómata Finito no Determinista con Épsilon	71
Transiciones a Autómata Finito no Determinista	
2.3 Transformación de Autómata Finito no Determinista a Autómata Finito Determinista	77
2.4 Otros Métodos De Transformación	79
UNIDAD III CONCEPTOS DE GRAMÁTICAS	85
3.1 Conceptos básicos relacionados con gramáticas	86
3.2 Jerarquía de Chomsky	88
3.3 Gramáticas Libres de Contexto	93
3.3.1 Árboles de derivación	97
3.3.2 Derivación por la derecha e izquierda (LL, LR)	99
3.3.3 Recursividad por la derecha e izquierda	103
3.4 Gramáticas Regulares	106
3.5 Otros conceptos relacionados	122

UNIDAD IV NORMALIZACIÓN DE GRAMÁTICAS**131**

4.1	Obtener la forma positiva de una gramática	133
4.2	Hacer admisible y arborescente una gramática	137
4.3	Obtener la Forma Normal de Chomsky de una gramática libre de contexto	142
4.4	Forma Normal de Greibach de una gramática libre de contexto y generar su Autómata de pila	145

UNIDAD V EQUIVALENCIAS Y MÁQUINA DE TURING**159**

5.1	Equivalencias de una Gramática Regular Lineal por la derecha	163
5.1.1	Método para pasar de una Gramática Regular Lineal por la Derecha a un AFN con Épsilon Transiciones	173
5.1.2	Método para pasar de un AFN con Épsilon Transiciones a una Gramática Regular Lineal por la Derecha	182
5.2	Equivalencias de una Gramática Regular Lineal por la izquierda	185
5.2.1	Método para pasar de una Gramática Regular Lineal por la Izquierda a un AFN con Épsilon Transiciones	185
5.2.2	Método para pasar de un AFN con Épsilon Transiciones a una Gramática Regular Lineal por la Izquierda	190
5.3	Generación de los 'Items' de un Gramática	193
5.4	Simplificación de un AFD	197
5.5	Máquina de Turing	202

GLOSARIO**225**

UNIDAD I

“Hay tres cosas que cada persona debería hacer durante su vida: 1) plantar un árbol, 2) escribir un libro y 3) tener un hijo”, recomendaba el poeta cubano José Martí. Sin embargo, por experiencia propia, no sé dónde plantar el árbol pues no tengo casa, solo que sea en el camellón de mi calle, tengo cuatro hijos y he tenido la osadía de escribir un libro, pero el chiste es que alguien lo lea y el milagro que lo compren. GRACIAS

MARCO CONCEPTUAL

“Explica los conceptos relacionados con Autómatas Finitos (AF), a través de dinámicas expositivas, aplicables a los métodos de transformación y en la industria de automatismos secuenciales”.

TEMAS:

- 1.1 Sistemas Formales
- 1.2 Teoría de conjuntos
- 1.3 Conceptos de lenguajes
- 1.4 Conceptos de expresiones regulares
- 1.5 Conceptos de autómatas finitos
- 1.6 Temas complementarios

1.1 Sistemas Formales

Recordemos que, de forma muy simplificada, un sistema es un conjunto de elementos organizados, heterogéneos -diferentes- que interactúan entre sí para lograr un fin común, incluye la aceptación de entradas, estímulos del medio ambiente para que mediante procesos idóneos que realiza el sistema, permite generar salidas o resultados que cumplen con la razón de ser del sistema. Ver figura 1.0, incluye la posibilidad de que las salidas vuelvan a entrar, retroalimentar el sistema. En un sistema formal, los elementos que lo integran son de tipo abstracto, representativo del comportamiento de un sistema “real”, también se puede decir que en un sistema formal se tiene la abstracción, esto es, se utilizan símbolos para cada uno de los elementos del sistema que se pretende representar y así analizar su comportamiento en base a la interacción que se establece en cada uno de sus elementos.



Figura 1.0 Diagrama esquemático de un Sistema

Analizar **figura 1.1** donde se muestra la relación que existe entre: El MUNDO REAL que tiene problemas por resolver, si se logran simplificar o esquematizar, podemos compartir nuestra lógica o comportamiento del sistema, es factible cambiar de un mundo real a un mundo posible, simplificado o bien, un modelo funcional conjuntamente con recursos (económicos, materiales, financieros y humanos) nos permite crear nuevas tecnologías, esto es, la INNOVACIÓN y produce ÉXITO.

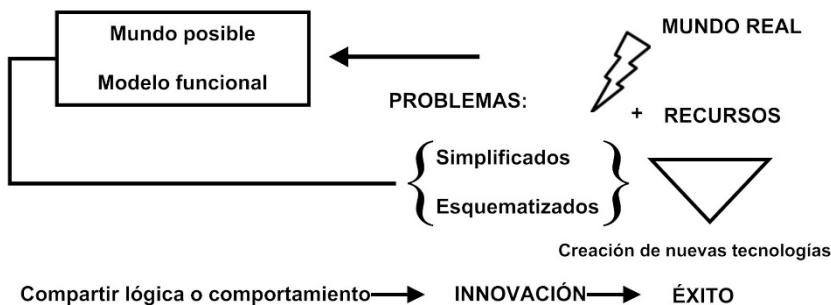


Figura 1.1 Muestra la relación entre: **El MUNDO REAL** y los elementos simbólicos que genera.

Un **ejemplo** muy común es un sistema de encendido y apagado de cualquier interruptor, circuito o mecanismo, lo podemos modelar o esquematizar de la **figura 1.2**, la cual muestra tres estados, mismo que desde el inicio parte a otros dos estados más, el estado de encendido o activación al cual se transita mediante un símbolo 1 que permite la transición (arco o línea con flecha) al estado E y al estado de apagado al cual se transita con un símbolo 0 mismo que permite la transición al estado A. Ambos estados E y A son estados finales (con doble círculo).

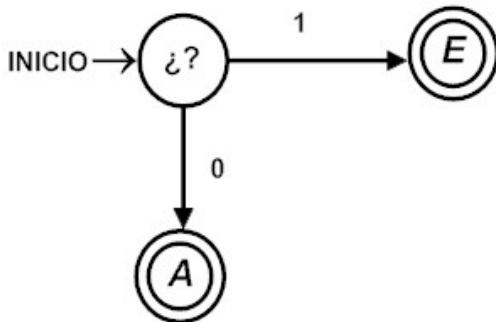


Figura 1.2. Autómata Finito para cualquier interruptor (¿?) que apaga (A) y enciende (E) cualquier dispositivo.

En la **figura 1.2** mostramos un diagrama de transiciones con tres estados. El primero, etiquetado como inicio, genera dos transiciones (arcos con flecha o direcciones) que tienen los símbolos 1 y 0 respectivamente. El PRIMER estado con los símbolos ¿? Lo cual significa el hecho de que pueda ser cualquier elemento manual, mecánico o eléctrico que acepta dos estados.

UNIDAD I • MARCO CONCEPTUAL

Dentro de estos dos estados está el SEGUNDO círculo, el estado de encendido que tiene una variable o símbolo identificado con la letra E y se llega con el número 1. El TERCER círculo, que es el estado de apagado, etiquetado con la letra A, donde el encendido le permite llegar al estado E y que tiene una variable o símbolo identificado con el número 0 que le permite llegar a el estado E. De aquí podemos derivar el modelo a formar de un autómata el cual consiste en un conjunto de estados simbolizados con la letra Q, que contiene los tres estados, el alfabeto que son los símbolos que acepta, y para éste ejemplo son el 0 y el 1 y de ahí tenemos los elementos del conjunto Q, y queda:

$$Q = \{?, A, E\}$$
 (Representados en la figura 1.2 con círculos)

$$\Sigma = \{0, 1\}$$
 (Representados como transiciones en aristas dirigidas o flechas)

$$\delta = Q \times \Sigma$$
 (Eje cartesiano con columnas etiquetadas con Q y los símbolos 0, 1)

$$S = \text{estado inicial} = \{?\}$$
 y (primer círculo)

$$F = \text{estados finales} = \{A, E\}$$
 (dos círculos dobles)

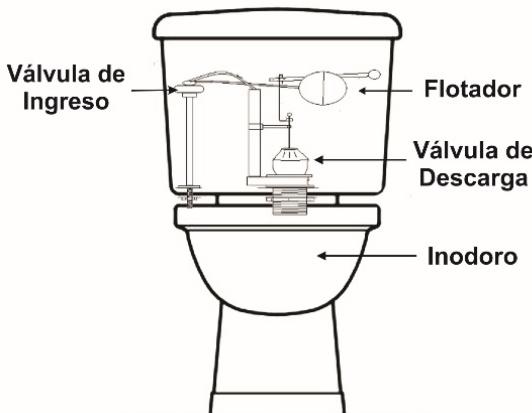


Figura. 1.3 Este es un sistema manual-mecánico, muy común y muy funcional el cual todos conocemos y usamos a diario.

Todo esto puede ser confuso, mejor es describir un sistema real, mismo que tiene un conjunto de elementos como, por **ejemplo**, **figura 1.3** el sistema “manual-mecánico” de un depósito de agua que se llena de forma autónoma, se vacía mediante un mecanismo de mano como es jalar una palanca o bien, oprimir un botón, mismo que puede abrir una llave especial que se activa con base al nivel del agua que se retiene en el contenedor o depósito de agua. En este sistema, el recurso principal es el agua, el contendor de la misma, la llave (que abre y cierra mediante un flotador o bombilla), la palanca o botón que manipula el usuario y que abre la llave, empaques, etc.)

EJERCICIO 1

Desarrolle un diagrama de flujo en el cual se representen tres variables: **E**, **N** y **S**, cada una con la posibilidad de contener sólo dos valores (binarias): **0** y **1**, mismos que permiten abstraer al sistema manual mecánico del depósito de agua de nuestro inodoro, vamos a aprovechar tres de sus elementos para hacer una abstracción muy simple, utilizando también tres letras mayúsculas para los elementos que serán representados (simbolizados) con las letras mayúsculas:

- E** => representa la válvula de ingreso (entrada)
- N** => representa la cazoleta de nivel (flotador o proceso)
- S** => representa la válvula de descarga (salida)

Cada una de estas letras mayúsculas representan una variable binaria dentro de nuestra abstracción, esto es, si la variable E tiene el valor de 0 (cero), significa que el agua no entra por la válvula de ingreso, si toma el valor de 1 (uno), esto significa que el agua entra por la válvula de ingreso. Los valores binarios son para cada una de las tres variables que tiene representaciones tales como:

- E** Si es **0** => el agua no entra. Si es **1** => la válvula se abre y entra el agua.
- N** Si es **0** => no hay agua, está vacío el contenedor. Si es **1** => está lleno de agua el contenedor.
- S** Si es **0** => No hay salida de agua. Si es **1** => Sale el agua y abre válvula de ingreso E.

Este ejercicio consiste en relacionar las tres variables con sus dos valores cada una y mostrarlo mediante un diagrama de flujo que contiene algunos de los símbolos, Según **figura 1.4**, donde se muestran los diagramas de: ENTRADA, PROCESO, SALIDA, DECISION y FIN.

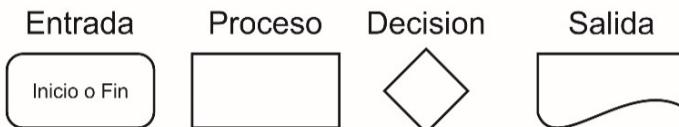


Figura 1.4 Símbolos utilizados para elaborar un diagrama de flujo

UNIDAD I • MARCO CONCEPTUAL

Algunos ejemplos de sistemas formales son aquellos en los que su **modelado** lo integran elementos matemáticos o gráficos, tales como: **expresiones regulares**, los **autómatas finitos** (determinísticos AFD, no determinísticos AFN, diagramas que se presentan en el siguiente sub-tema). Estos gráficos dirigidos equivalen a la representación del comportamiento de diferentes elementos que son parte de un sistema real, tales como: estímulos de entrada, procesos, transiciones entre los procesos, decisiones, recursos, eventos, etapas, salidas.

Finalmente, también la razón de ser de los sistemas son los resultados o salidas. En base a un problema que se plantea de forma tangible, esto significa que el problema es real, pero **los sistemas representados formalmente son representaciones simbólicas de algunos sistemas reales** y al abstraerse **se convierten en sistemas o modelos matemáticos**, mismos que son representaciones de problemas que se resuelven con sistemas en un tipo de matemática conocida como la **teoría de conjuntos, matemáticas discretas y el álgebra de Boole** (conceptos todos descritos en el glosario de éste sub-tema). Los anteriores tres conceptos son herramientas que permiten el modelado de sistemas que han generado como obra maestra las calculadoras, computadoras, sus lenguajes y muchos más elementos conocidos como tecnologías de información y comunicación (TIC's).

Todo lo anterior, se puede expresar también de forma jerárquica, mediante el uso de diferentes gráficos, que pueden ser tan sencillos como **árboles de análisis** (descritos en el siguiente capítulo) en éstos árboles se pueden representar de una forma evidente y basándose en reglas de **derivación** sencillas así como la definición de los elementos que integran el sistema y, es aquí donde se aplica la teoría de conjuntos para definir, integrar, **modelar** o representar todos y cada uno de los elementos reales de un sistema. Los cuales son una abstracción factible, manipulable y económica de los elementos verdaderos que integran el sistema en el mundo real. Mismo que al expresarse con los elementos de la matemática se convierte en un sistema formal con soluciones probadas en el modelo creado que se aplicarán con éxito a la realidad con un sistema computacional.

Entrando al mundo de los sistemas computacionales, allí existen elementos que pueden ser tangibles conocidos como **hardware** o elementos intangibles, lógicos o inteligentes conocidos como **software** ver **figura 1.5**.

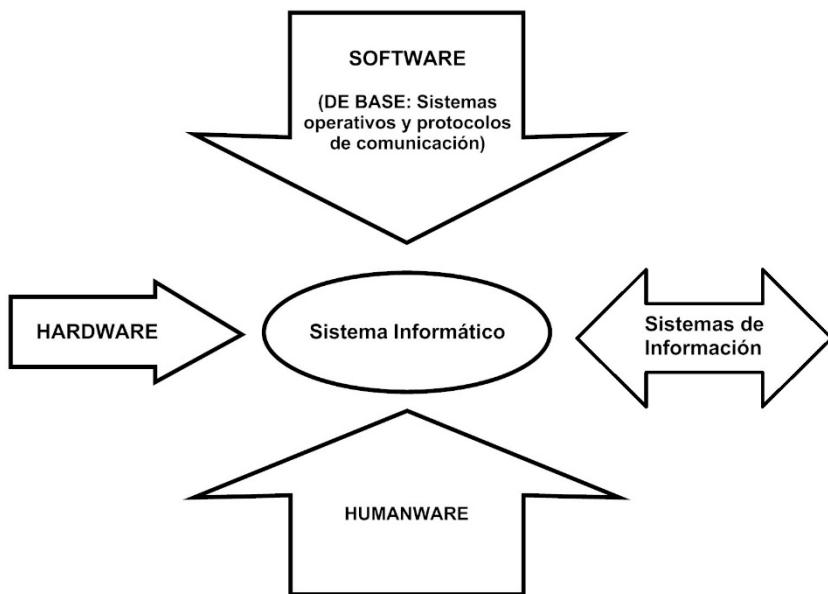


Figura 1.5 Diagrama esquemático de un sistema informático

En este tipo de sistemas, podemos entrar a lo que se conoce como software de base, un ejemplo de este es el sistema operativo, los protocolos de comunicación. En un sistema informático tenemos aún más elementos, dentro de los cuales se integran elementos como el humano (también definido como el **humanware**); recursos materiales (el equipamiento o hardware, hardware de red o comunicaciones, etc.); elementos financieros, económicos, dinero; la infraestructura, el centro de cómputo con instalaciones, sistemas de seguridad, oficinas; etc. Todo lo anterior es un reflejo de lo que puede ser un sistema informático que produce sistemas de información, mismo que al influir el factor humano, también se le considera un sistema inteligente.

Los sistemas de información inteligentes, son aquellos que además dentro del software, se integran artefactos que emulan el comportamiento y la toma de decisiones que un humano ha programado, esto es, que ha grabado las instrucciones dentro de los programas que los hacen funcionar. Entramos al mundo de la **programación** y dentro de éste se requiere el elemento conocido como software básico y son, además

UNIDAD I • MARCO CONCEPTUAL

de los ya descritos (sistemas operativos, protocolos de comunicación), **programas traductores** que son de tres tipos; **1) intérpretes, 2) ensambladores y 3) compiladores**. Es aquí donde se han desarrollado miles de estos componentes lógicos o **lenguajes de programación**.

CONCLUSIÓN SOBRE SISTEMAS FORMALES

Los sistemas formales son los que:

- Aplican la matemática para poder obtener la solución modelada a dicho sistema.
- Formalizan aplicaciones (Hacen factible la abstracción para lograr la solución).
- Tienen relación con los autómatas matemáticos, que permiten modelar su comportamiento.
- Obedecen a un lenguaje de símbolos (*lingüística*).
- Ejemplos de este tipo de sistemas son: Representaciones formales como la forma de Backus y Naur, abreviada como BNF's, diagramas de autómatas, las calculadoras, los **lenguajes artificiales**, usados por el hombre para dar instrucciones a una máquina, artefacto o computadora.
- Se pueden encontrar en **lenguajes de programación formales** o regulares.
- Utilizan la teoría de conjuntos (parte de la matemática).

Un sistema formal también es considerado como un sistema lógico-deductivo que está constituido por un lenguaje formal. Lenguaje cuyos símbolos y reglas están formalmente (matemáticamente) especificados; además de ser la base de la formación/construcción de un sistema formal. A partir de esto, una de las mayores aplicaciones de un sistema formal es la: **Demostración**.

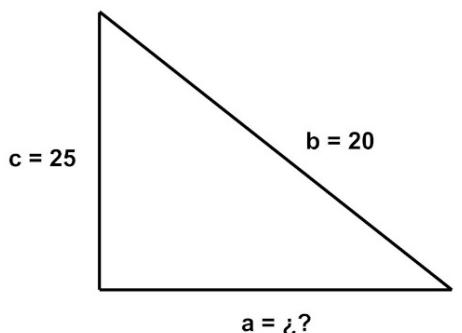
Para que a un Sistema se le pueda considerar como formal tiene que presentar una característica principal, la cual es: **Capturar y Abstraer la esencia de determinadas características del mundo real en un modelo (paradigma) funcional y operacional aplicando alguna rama de la matemática**.

Un sistema formal necesita de la matemática para poder desarrollarse, es por ello, conjuntamente con la lingüística y la teoría de autómatas son las áreas involucradas para proporcionar el sustento teórico para el estudio, modelado y creación de sistemas

de forma concreta, directa, económica y factible. Además de tienen un objetivo, el cual es: **Señalar como válidas determinadas cadenas (teoremas) de información o de símbolos que se estén utilizando dentro del lenguaje formal que se encuentra en el mismo sistema**". Todo es para hacerlos reales con la asignación de los recursos que ya se describieron y son: humanos, materiales, financieros, etc., dependiendo de cada sistema en particular.

Los Sistemas Formales poseen propiedades, las cuales son:

- **Coherencia:** Si cada teorema o cadena al ser interpretado(a) corresponde a una decisión verdadera.
- **Completitud:** Si cada proposición verdadera puede ser representada mediante un teorema. Es incompleto si alguna verdad no puede expresarse.
- **Decidibilidad:** Un sistema formal es decidible, si existe un algoritmo que diga en tiempo finito si una cadena cualquiera es un teorema o no lo es. Aquí colocamos como ejemplo el conocido teorema de Pitágoras donde se reúnen los tres elementos ya descritos: Coherencia, completitud y decidibilidad según se muestra en la **figura 1.6**.



Teorema: El cuadrado de la hipotenusa (b) es igual a la suma de los cuadrados de los dos catetos (a y c)

$$a^2 + b^2 = c^2$$

Figura 1.6 Teorema de Pitágoras como ejemplo de que reúne los elementos de un sistema formal.

UNIDAD I • MARCO CONCEPTUAL

Algunas de las características principales y fundamentales que ayudan a identificar y a definir los sistemas formales son:

- En un sistema formal hay siempre una realidad que se modela.
- Una manera rápida de explicar un sistema formal es mediante la siguiente "ecuación" o igualdad.
- Sistema formal = Lenguaje formal + Aparato deductivo.
- Con esto se quiere significar que un sistema formal consta de dos partes bien definidas:
- Un lenguaje formal, con el que se denotan los elementos de la realidad modelada, en cierto sentido, estática.
- Un aparato deductivo, que sirve para establecer elementos de la realidad que tienen alguna cualidad interesante (valor numérico, ser o no verdaderos, etc.).
- El sistema tiene un lenguaje para denotar objetos de la realidad.
- Modelan realidades en las que hay afirmaciones, y éstas pueden ser ciertas o falsas.

Es por todas estas razones que los Sistemas Formales dentro de la Teoría de la Computación son de suma importancia, ya que modelan gran parte de la realidad y se considera que permiten el mejor entendimiento con ayuda de la matemática y en particular está la teoría de conjuntos, lo referente a la teoría de autómatas, teoría de la computación y lo que de todas éstas teorías se derivan. Según se verá en los siguientes temas.

EJERCICIO 2.

Desarrolle su propio concepto de lo que es:

- Sistema,
- Sistema de información,
- Sistema formal,
- Software,
- Hardware,
- Humanware,
- Sistema de control

Puede desarrollar estos conceptos en formato texto, utilizando un diagrama o un mapa conceptual.

1.2 Teoría de conjuntos

La palabra “**conjunto**” generalmente la asociamos con la idea de agrupar cosas, objetos, personas, etc. Por ejemplo, un conjunto de animales, de letras, de computadoras, es decir, denotamos una colección de elementos claramente identificados entre sí, que guardan alguna característica en común que puede ser general o específica, así, de esta forma, al agrupar el conjunto de computadoras, nos referimos en general a cualquier computadora y podemos ser más específicos al decir conjunto de computadoras personales, donde nos referimos a un tipo muy particular dentro de todas las computadoras.

También se pueden agrupar conjuntos ya sean números, personas, figuras, ideas y conceptos. La característica esencial de un conjunto es la de estar bien definido, esto es que, dado un objeto particular, determinar si este pertenece o no al conjunto. Por ejemplo, el conjunto de los días de la semana:

Días = {lunes, martes, miércoles, jueves, viernes, sábado, domingo}

Simbolizando este conjunto tenemos: D = Días, l = lunes, m = martes, w = miércoles, j = jueves, v = viernes, s = sábado, d = domingo

Para denotar el nombre de los conjuntos, se usan letras mayúsculas y para denotar a todos los elementos que integra se utilizan letras minúsculas.

$$D = \{l, m, w, j, v, s, d\}$$

Algunos de los aspectos más importantes de un Conjunto son:

- No hay elementos que se repiten.
- No tienen un orden específico.
- Los elementos son símbolos o abstracciones de lo que representan.

Ejemplo: Conjunto A de hombre y mujer, donde el símbolo “h” puede representar un hombre y ‘m’ puede representar una mujer.

Simbólicamente: A = {h, m} o A = {m, h}

UNIDAD I • MARCO CONCEPTUAL

Al tener en claro lo que es un conjunto, se puede empezar a definir con base a lo que es una teoría de conjuntos relacionados con la matemática y a su vez con la teoría de la computación, la cual, como ya se ha dicho, los sistemas formales se basan en la matemática para modelar sistemas. La teoría de conjuntos además es una rama de la lógica que estudia las propiedades de los conjuntos: colecciones abstractas de objetos, consideradas como objetos en sí mismas.

Los conjuntos y sus operaciones más elementales son una herramienta básica en la formulación de cualquier teoría. En Matemáticas existen muchos problemas que involucran conjuntos. Generalmente, estos problemas están relacionados con un grupo de elementos que deben cumplir una cierta propiedad.

En Computación existen también problemas de conjuntos. Generalmente, estos problemas están relacionados, en si poseen un **algoritmo** eficiente que lo resuelva. Dentro de esta área la teoría de conjuntos juega un papel muy importante y preponderante ya que no solamente se enfoca en la computación como tal, sino que también se enfoca en ciencias de la computación, en la matemática discreta, **teoría de computabilidad**, algoritmos computacionales, programación y especificación formal de software.

La Teoría de Conjuntos se encuentra involucrada en diversas áreas de estudio, por lo que es importante tomar en cuenta su objeto de estudio, y éste es: “Los Conjuntos”. Estas entidades matemáticas son objetos que ayudan a representar conceptos. Un aspecto importante de la teoría de conjuntos dentro de la teoría de la computación es que: *La computación es la rama de las ciencias computacionales que estudia la ejecución de un algoritmo el cual se describe como el conjunto de instrucciones, pasos o reglas bien definidas, ordenadas y finitas que permite realizar una actividad mediante pasos sucesivos que no generen dudas a quien deba realizar dicha actividad, por medio de varios sitios (lugares o espacios), dispersos geográficamente, que comparten recursos.*

Dentro de la teoría de la computación se hace hincapié en que tiene una enorme importancia que, sin embargo, sólo afecta a matemáticos, lógicos y a todos aquellos que se dedican a tareas de programación **informática**. Respecto a la teoría de conjuntos se tiene que mediante ésta se pueden llevar a cabo diferentes tipos de operaciones y dichas operaciones nos pueden servir para realizar diversas aplicaciones (dentro de la computación) y estas operaciones son:

1. UNIÓN.

La unión de dos conjuntos A y B la denotaremos por $A \cup B$ y es el conjunto formado por los elementos que pertenecen al menos a uno de ellos o a los dos. Lo que se denota por: $A \cup B = \{ x \mid x \in A \text{ ó } x \in B \}$

Ejemplo: Sean los conjuntos $A = \{ 1, 3, 5, 7, 9 \}$ y $B = \{ 10, 11, 12 \}$
 $A \cup B = \{ 1, 3, 5, 7, 9, 10, 11, 12 \}$

2. INTERSECCIÓN.

Sean $A = \{ 1, 2, 3, 4, 5, 6, 8, 9 \}$ y $B = \{ 2, 4, 8, 12 \}$

Los elementos comunes a los dos conjuntos son: $\{2, 4, 8\}$. A este conjunto se le llama intersección de A y B; y se denota por $A \cap B$, algebraicamente se escribe así:

$A \cap B = \{ x \mid x \in A \text{ y } x \in B \}$

Y se lee el conjunto de elementos x que están en A y están en B.

Ejemplo: Sean $Q = \{ a, n, p, y, q, s, r, o, b, k \}$ y $P = \{ l, u, a, o, s, r, b, v, y, z \}$

$Q \cap P = \{ a, b, o, r, s, y \}$

3. CONJUNTO VACÍO.

Un conjunto que no tiene elementos es llamado conjunto vacío, también conocido como conjunto nulo lo que denotamos por el símbolo \emptyset .

Ejemplo: Sean $A = \{ 2, 4, 6 \}$ y $B = \{ 1, 3, 5, 7 \}$ encontrar $A \cap B$

$A \cap B = \{\emptyset\}$ (Significa: Conjunto, cuyo único elemento es el conjunto vacío)

El resultado de $A \cap B = \{\emptyset\}$ muestra que no hay elementos entre las llaves, si este es el caso se le llamará conjunto vacío o bien nulo y se puede representar como:

$A \cap B = \emptyset$

4. CONJUNTOS AJENOS.

Sí la intersección de dos conjuntos es igual al conjunto vacío, entonces a estos conjuntos les llamaremos conjuntos ajenos, es decir:

Si $A \cap B = \emptyset$ entonces A y B son ajenos.

5. COMPLEMENTO.

El complemento de un conjunto respecto al universo U es el conjunto de elementos de U que no pertenecen a A y se denota como A' y que se representa por comprensión como: $A' = \{ x \in U \mid x \notin A \}$

UNIDAD I • MARCO CONCEPTUAL

Ejemplo: Sea $U = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$ $A = \{1, 3, 5, 7, 9\}$ donde $A \subset U$
El complemento de A estará dado por: $A' = \{2, 4, 6, 8\}$

6. DIFERENCIA.

Sean A y B dos conjuntos. La diferencia de A y B se denota por $A - B$ y es el conjunto de los elementos de A que no están en B y se representa por comprensión como:
 $A - B = \{x / x \in A ; X \in B\}$

Ejemplo: Sea $A = \{a, b, c, d\}$ y $B = \{a, b, c, g, h, i\}$

Por lo tanto $A - B = \{d\}$

En el ejemplo anterior se observa que solo interesan los elementos del conjunto A que no estén en B. Si la operación fuera $B - A$.

El resultado es: $B - A = \{g, h, i\}$ e indica los elementos que están en B y no en A.

7. POTENCIA.

Es la posibilidad de que en un conjunto sus elementos puedan ser infinitos por ejemplo se tiene la representación del conjunto de letras a desde una a, hasta un número infinito de a's y esto se representa de la siguiente forma:

Cerradura positiva:

$A = \{a\}^+$ esto significa el conjunto integrado por una a, dos a, tres a, etc.: a, aa, aaa, aaaa, aaaaa, ..., ∞

También se tiene el conjunto: $B = \{b\}^*$ y esto significa: ninguna b, o solo una b, dos b, tres b, etc.

Cerradura de Kleene

$B = \{b\}^* = \emptyset, b, bb, bbb, bbbb, bbbbb, \dots, \infty$

8. IGUALDAD.

Si un conjunto está definido como $L_3 = \{0, 1\}$ y $L_4 = \{0, 1\}$
Entonces, podemos decir que: $L_3 = L_4$

9. PRODUCTO CARTESIANO.

Si un conjunto está definido como $L_1=\{0,1\}$ y $L_2=\{00,11\}$ Su producto cartesiano es L_1L_2 . Lo que significa fusionar los elementos de L_1 , que son el digito 0 con el digito 1 con los elementos de L_2 que son dos dígitos 00 como primer elemento y el conjunto de dos dígitos 11 como segundo elemento. Para esto tomamos el primer elemento de L_1 que es 0 y lo unimos al primer y segundo elementos de L_2 y de la misma forma para el segundo elemento de L_1 que es 1 y lo unimos a el primer y segundo elementos de L_2 y queda: $L_1L_2 = \{000, 011, 100, 111\}$

EJERCICIO 3.

Desarrolle una representación simbólica, basada en la teoría de conjuntos de todos los números binarios.

EJERCICIO 4.

Desarrolle una representación en forma de conjuntos de todo tipo de computadoras, por ejemplo, si ubicamos a las computadoras por su tamaño, podemos decir que hay super-computadoras, mini-computadoras, macro-computadoras, servidores, computadoras personales, etc. Trate de hacer una abstracción lo más completa posible para que simbólicamente represente al menos cinco tipos de computadoras y realice una abstracción simbólica en forma de conjuntos.

Como se muestra en el conjunto de hombres (h) y mujeres (m) se escriben los elementos de un conjunto entre llaves ({ }), y cada elementos separados por comas (,). El detallar a todos los elementos de un conjunto entre las llaves, se denomina forma tabular, extensión o enumeración de los elementos.

EJERCICIO 5.

Desarrolle una representación simbólica, basada en la teoría de conjuntos de todos los símbolos del alfabeto castellano.

EJERCICIO 6.

Desarrolle el producto cartesiano de $L_5 = \{01,10\}$ y $L_6 = \{0,1\}$,

EJERCICIO 7.

Desarrolle un resumen o bien organizador gráfico que describa un sistema computacional y un sistema informático y permita definir sus analogías y diferencias.

1.3 Conceptos de lenguajes

Dentro de las diversas definiciones que se encuentran en la bibliografía de este material, primeramente, tenemos una definición general desde el punto de vista de los lenguajes que utilizamos para comunicarnos como humanos:

ELEMENTOS PARA EL DISEÑO DE LENGUAJES PRATT (1987).

Algunas de las características básicas a considerar en el diseño de los lenguajes de programación son:

- **Abstracción.** El lenguaje debe evitar forzar a los programadores a tener que enunciar algo más de una vez. El lenguaje debe permitir al programador la identificación de patrones repetitivos y automatizar tareas mecánicas, tediosas o susceptibles de cometer errores. Ejemplos de técnicas de abstracción son los procedimientos y funciones, la generalidad, los lenguajes de patrones de diseño, etc.
- **Concisión notacional.** El lenguaje proporciona un marco conceptual para pensar algoritmos y expresar dichos algoritmos con el nivel de detalle adecuado. El lenguaje debe ser una ayuda al programador (incluso antes de comenzar a codificar) proporcionando un conjunto de conceptos de forma clara, simple y unificada. La sintaxis debe ser legible por el programador (o por otras personas que vayan a utilizar esos programas). Deben buscarse soluciones de compromiso entre lenguajes demasiado crípticos. Ejemplo, lenguaje *C* y lenguajes con código demasiado largo o dilatado con exceso como *COBOL*, *XSLT* tienen concisión notacional.
- **Eficiencia.** El programador debe poder expresar algoritmos suficientemente eficientes o el lenguaje debe incorporar técnicas de optimización de los programas escritos en él. Ejemplo, lenguaje *C*.
- **Entorno.** Aunque el entorno no forma parte del lenguaje, muchos lenguajes débiles técnicamente son ampliamente utilizados debido a que disponen de un entorno de desarrollo potente o agradable. De la misma forma, la disposición de documentación, ejemplos de programas e incluso programadores pueden ser factores clave de la popularidad de un lenguaje de programación como es el caso de *COBOL* que es ejemplo en el que el entorno están tan ligado al código que se tiene que modificar y re-compilar tantas veces cambien los equipos donde se implementa.

UNIDAD I • MARCO CONCEPTUAL

- **Expresividad.** El programador debe poder expresar sus intenciones. De hecho, algunos sistemas limitan la expresividad para mejorar la fiabilidad de los programas. Por ejemplo, la aritmética de punteros no es permitida en algunos lenguajes.
- **Extensibilidad.** El lenguaje debe facilitar mecanismos para que el programador pueda aumentar la capacidad expresiva del lenguaje añadiendo nuevas construcciones. En *Haskell*, por ejemplo, el programador puede definir sus propias estructuras de control.
- **Librerías e interacción con el exterior.** La inclusión de un conjunto de librerías que facilita el rápido desarrollo de aplicaciones es un componente esencial de la popularidad de los lenguajes. Si no se dispone de tales librerías, es necesario contemplar mecanismos de enlace con otros lenguajes que facilitan la incorporación de librerías externas. Un ejemplo son las librerías estándar del lenguaje *C* y *COBOL*.
- **Ortogonalidad.** Ofrece la posibilidad de combinar características de todas las formas posibles. La falta de *versatilidad* (que se muda o cambia fácilmente de un estado a otro), puede suponer la enumeración de situaciones excepcionales o la aparición de incoherencias. Esto es, si se define alfanumérico puede devolver datos lo mismo alfabéticos que numéricos. Los ejemplos de lenguajes que no tienen *ortogonalidad* son los lenguajes fuertemente “*tipeados*”, esto es, que deben tener una definición explícita, clara y precisa del tipo de datos a utilizar, así como los lenguajes *PASCAL*, *COBOL* y otros.
- **Portabilidad.** El lenguaje debe facilitar la creación de programas que funcionen en el mayor número de entornos computacionales. Este requisito es una garantía de supervivencia de los programas escritos en el lenguaje y, por tanto, del propio lenguaje. Para conseguir la portabilidad, es necesario limitar las características dependientes de una arquitectura concreta. El mejor ejemplo de esta característica lo es *Java*.
- **Seguridad.** La fiabilidad de los productos software es cada vez más importante. Lo ideal es que los programas incorrectos no pertenezcan al lenguaje y sean rechazados por el compilador. Por ejemplo, los sistemas con chequeo de tipos establecen restricciones a los posibles programas que pueden escribirse en un lenguaje para evitar que en tiempo de ejecución se produzcan errores. Existen lenguajes como *Charity* que garantizan la terminación de sus programas.

PREVENCIÓN Y DETECCIÓN DE ERRORES PRATT (1987).

Tener una serie de defensas tal que, si un error no es detectado por el programador que desarrolla el código, este error probablemente sea detectado por otro programador. Los errores deben ser detectados por el compilador, si un mecanismo no es capaz de detectar un error es necesario implementar otro que lo detecte, pero nunca ignorarlo.

Prevención de errores

- El programador comete errores. Hay que prevenir los errores
- El programador es su fuente. El programador no sabe lo que hace y el compilador ha de limitar sus acciones (*EUCLID, PASCAL*). Hacer imposible cierto tipo de errores.

Ejecutar datos → control de flujo limitado.

Errores en el uso de datos → código fuente fuertemente “tipado” (con previa definición de tipos de datos).

Apuntadores erróneos → Gestión de memoria implícita (*LISP, PROLOG, ML*, etc.).

Hay que facilitar su detección, identificación y corrección. Tener que declarar antes de utilizar. Evitar coerciones inductoras de errores.

- ***float* a *int*** por su pérdida de precisión.
- **Comprobaciones en tiempo de ejecución.** Índice de arreglos “array” fuera de límites. Control sobre los apuntadores a *NULL*.

Prevención y tolerancia de fallos.

Hay dos formas de aumentar la fiabilidad de un sistema:

1. **Prevención de fallos:** Se trata de evitar que se introduzcan fallos en el sistema antes de que entre en funcionamiento. Se realiza en dos etapas:
 - **Evitación de fallos:** Se trata de impedir que se introduzcan fallos durante la construcción del sistema.
 - **Eliminación de fallos:** Consiste en encontrar y eliminar los fallos que se producen en el sistema una vez construido.

UNIDAD I • MARCO CONCEPTUAL

2. **Tolerancia de fallos:** Se trata de conseguir que el sistema continúe funcionando, aunque produzcan fallos.

En ambos casos el objetivo es desarrollar sistemas con modos de fallo bien definidos. Detección de errores ya sea por el entorno de ejecución o hardware, por ejemplo: Instrucción ilegal o por el núcleo del sistema operativo (dentro de éste, puntero nulo). También por el software de aplicación o bien duplicación (redundancia con dos versiones) por ejemplo: Comprobaciones de tiempo, inversión de funciones o códigos detectores de error o bien validación de estado o validación estructural.

EFICACIA.

Los lenguajes de programación contienen un repertorio de tipos básicos o primitivos junto con una serie de operaciones sobre dichos tipos. Los tipos básicos más habituales son:

- Números enteros: Se representan mediante un número de bytes fijo, limitando su rango. Algunos lenguajes, como *Haskell*, incluyen una representación ilimitada.
- Caracteres: Como en el caso anterior, suelen representarse mediante un número fijo de bytes. En algunos lenguajes, los caracteres se identifican con los enteros.
- Números reales representados en punto flotante. La representación también suele realizarse mediante un número fijo de bytes, limitando la precisión de los programas.
- Booleanos: Con los valores verdadero o falso.
- Referencias. Algunos lenguajes incluyen un tipo básico que se utiliza para hacer referencia a otros elementos. Estas referencias pueden implementarse mediante direcciones de memoria.

Cada tipo básico contiene un conjunto de operaciones primitivas. Por ejemplo, los enteros y flotantes incluyen operaciones aritméticas, los caracteres, operaciones de conversión y los booleanos. Los lenguajes con chequeo estático de tipos permiten comprobar en tiempo de compilación que en tiempo de ejecución no se van a producir errores de tipos. El chequeo estático de tipos aumenta la seguridad de los programas, al detectar errores antes de la ejecución. Otra ventaja es la eficiencia, ya que en la fase de ejecución no es necesario realizar comprobaciones de tipo.

Otros lenguajes, como *LISP*, *BASIC*, *Perl*, *Prolog*, etc., no incluyen chequeo estático de tipos. Las ventajas de no incluirlo son una mayor flexibilidad (es posible construir más programas) y sencillez para el programador. El programador no se preocupa de incluir declaraciones de tipos y los programas dan menos errores de tipo al compilar (aunque pueden darlos al ejecutarse).

Algunos lenguajes, como *Haskell* o *ML*, incluyen además un sistema de inferencia de tipos. El programador no tiene obligación de declarar el tipo de las expresiones, ya que el sistema es capaz de inferirlo. En caso de que el programador lo hubiese declarado, el sistema puede comprobar que el tipo declarado coincide con el tipo inferido.

COMPATIBILIDAD

La elección de un lenguaje de programación depende de sus conocimientos del lenguaje y del ámbito de la aplicación que está generando. Las aplicaciones de pequeño tamaño se suelen crear utilizando un único lenguaje, y es frecuente implementar aplicaciones grandes utilizando varios lenguajes. Por ejemplo, si está ampliando una aplicación con servicios Web de XML existentes, (ver *GLOSARIO*), podría utilizar un lenguaje de secuencias que no requiera apenas tareas de programación. Para aplicaciones cliente-servidor, utilizaría probablemente el lenguaje del que tiene más conocimientos para toda la aplicación. Para nuevas aplicaciones empresariales, donde un equipo grande de programadores crea componentes y servicios para implementarlos en varios sitios remotos, la mejor opción sería utilizar varios lenguajes dependiendo de los conocimientos de los programadores y de las expectativas de mantenimiento a largo plazo.

Los lenguajes de programación de la plataforma .NET, incluidos *Visual Basic .NET*, *Visual C#* y las Extensiones administradas de *C++*, y otros lenguajes de distintos fabricantes, utilizan los servicios y características de .NET Framework a través de un conjunto común de clases unificadas. Las clases unificadas de .NET proporcionan un método coherente de acceso a la funcionalidad de la plataforma. Si aprende a utilizar la biblioteca de clases, observará que todas las tareas siguen la misma arquitectura uniforme. Ya no necesitará aprender ni administrar distintas arquitecturas *API* para escribir las aplicaciones.

UNIDAD I • MARCO CONCEPTUAL

En la mayoría de las situaciones, puede utilizar de manera eficiente todos los lenguajes de programación de Microsoft. Sin embargo, cada lenguaje de programación tiene sus puntos fuertes, y es recomendable comprender las características únicas para cada uno de ellos. Las secciones siguientes ayudarán a seleccionar el lenguaje de programación que mejor se ajuste a su aplicación. Una noción importante de estos lenguajes es la herencia que facilita la reutilización de código.

- **Simula.** Desarrollado por O. J. Dahl y K. Nygaard entre 1962 y 1967 como un dialecto de ALGOL para simulación de eventos discretos. Fue el primer lenguaje que desarrolló el concepto de clases y subclases.
- **Smalltalk.** Desarrollado por A. C. Kay en 1971 como un sistema integral orientado a objetos que facilitaba el desarrollo de aplicaciones interactivas.
- **C++.** Creado en 1985 por B. Stroustrup añadiendo capacidades de orientación a objetos al lenguaje C. El objetivo era disponer de un lenguaje con las capacidades de abstracción y reutilización de código de la orientación a objetos y la eficiencia de C++. El lenguaje fue ganando popularidad a la par que se incluían numerosas características como genericidad, excepciones, etc.
- **Java.** Desarrollado por J. Gosling en 1993 como un lenguaje orientado a objetos para dispositivos electrónicos empotrados. Alcanzó gran popularidad en 1996 cuando Sun Microsystems hizo pública su implementación. La sintaxis del lenguaje se inspira en la de C++ pero contiene un conjunto más reducido de características. Incluye un sistema de gestión de memoria y un mecanismo de tratamiento de excepciones y concurrencia. Las implementaciones se basan en una máquina virtual estándar (denominada JVM - Java Virtual Machine). El lenguaje alcanza gran popularidad como lenguaje para desarrollo de aplicaciones en Internet puesto que la JVM es incorporada en muchos servidores y clientes.
- **Python.** Creado por Guido van Rossum en 1990 como un lenguaje orientado a objetos interpretado. Utiliza una sintaxis inspirada en C++ y contiene un sistema dinámico de tipos. El lenguaje ha ganado popularidad debido a su capacidad de desarrollo rápido de aplicaciones de Internet, gráficos, bases de datos, etc.
- **C#.** Creado por Microsoft para la plataforma .NET en 1999. Utiliza una sintaxis similar a la de Java y C++. También contiene un sistema de gestión dinámica de memoria y se apoya en CLR (Ver significado en GLOSARIO). El lenguaje aprovecha muchas de las características de esta plataforma, como el mecanismo de excepciones, sistema de componentes, control de versiones.

Los lenguajes de programación tienen como objetivo la construcción de programas, normalmente escritos por personas humanas. Estos programas se ejecutarán por un computador que realizará las tareas descritas. El programa debe ser comprendido tanto por personas como por computadores.

La utilización de un lenguaje de programación requiere, por tanto, una comprensión mutua por parte de personas y máquinas. Este objetivo es difícil de alcanzar debido a la naturaleza diferente de ambos. En un lenguaje natural, el significado de los símbolos se establece por la costumbre y se aprende mediante la experiencia. Sin embargo, los lenguajes de programación se definen habitualmente por una autoridad, que puede ser el diseñador individual del lenguaje o un determinado comité.

Los lenguajes de programación son, por tanto, una solución de compromiso entre las necesidades del emisor (programador - persona) y del receptor (computador - máquina). De esa forma, las declaraciones, tipos, nombres simbólicos, etc. son concesiones de los diseñadores de lenguajes para que los humanos podamos entender mejor lo que se ha escrito en un programa. Por otro lado, la utilización de un vocabulario limitado y de unas reglas estrictas son concesiones para facilitar el proceso de traducción.

Morris (1938) realiza una división del estudio de los signos (semiótica) en tres partes:

- Sintaxis: relación de los signos entre sí.
- Semántica: Relación de los signos con los objetos a los que se aplican.
- Pragmática: Relación de los signos con sus intérpretes.

Adaptando dichas definiciones al caso particular de lenguajes de programación, la sintaxis se refiere al formato de los programas del lenguaje, la semántica estudia el comportamiento de los programas y la pragmática estudia aspectos relacionados con las técnicas empleadas para la construcción de programas.

Es el sistema de códigos directamente interpretable por una máquina, como el procesador de un ordenador o computadora. Este lenguaje está compuesto por un conjunto de instrucciones que determinan acciones a ser tomadas por la máquina. Un programa de computadora consiste de una cadena de estas instrucciones de lenguaje de máquina (más datos). Estas instrucciones son normalmente ejecutadas en secuencia, con eventuales cambios de flujo causados por el propio programa o eventos externos.

UNIDAD I • MARCO CONCEPTUAL

El lenguaje de máquina es específico de cada máquina o arquitectura de la máquina, aunque el conjunto de instrucciones disponibles pueda ser similar entre ellas.

El lenguaje-máquina sólo utiliza dos signos. Y éstos se corresponden exactamente con las únicas dos constantes del álgebra booleana, y también con los dos estados exclusivos de un conmutador. Estos signos, constantes o estados de conmutación son el 1 y el 0, llamados dígitos binarios. Claude Elwood Shannon, en su *Análisis de los relés y circuitos de conmutación*, “*Analysis of Relay and Switching Circuits*”, y con sus experiencias en redes de conmutación sentó las bases para la aplicación del álgebra de Boole a redes de conmutación. Una Red de Comutación es un circuito de interruptores eléctricos que, al cumplir ciertas combinaciones booleanas con las variables de entrada, define el estado de la salida. Este concepto es el núcleo de las Compuertas Lógicas, las cuales son por su parte, los ladrillos con que se construyen sistemas lógicos cada vez más complejos.

Shannon utilizaba al relé “*relay*” como dispositivo físico de conmutación en sus redes. El “*relay*”, igual que el interruptor de una lámpara eléctrica, es 1 o es 0; está prendido o está apagado.

El desarrollo tecnológico ha permitido evolucionar desde las redes de “*relays*” electromagnéticos de Shannon, pasar a circuitos con tubos de vacío, luego a redes transistorizadas, hasta llegar a los modernos circuitos integrados. El Microprocesador de la CPU de la computadora opera en lenguaje-máquina, ya que su repertorio de instrucciones consiste en la ejecución de conjuntos binarios. Por cierto, que Shannon fue quien aportó el término de bit para los guarismos 1 y 0, acrónimo de las palabras inglesas *binary digits* o el equivalente en español: “dígitos binarios”.

LENGUAJE ENSAMBLADOR.

El lenguaje ensamblador consiste en una selección de abreviaturas (mnemónicos o mnemoclaves) tomadas de palabras del lenguaje inglés, las cuales dan una idea de su significado. Este lenguaje está muy cercano a la arquitectura de la computadora y se dan instrucciones indicando el direccionamiento para cada instrucción. Este es una versión mnemotécnica del código de máquina, donde se usan nombres (identificadores) en lugar de códigos binarios para operaciones, y también se usan nombres (también identificadores) para las direcciones de memoria.

Existe un lenguaje ensamblador para cada tipo de computadora en especial, por lo que se dice que son dependientes del hardware y de bajo nivel, por lo que suelen ser complejos dependiendo de la capacidad de la computadora, a mayor capacidad mayor complejidad en la elaboración del código en este tipo de lenguaje. Un ejemplo puede ser...

<i>Dirección en hexadecimal</i>	<i>Instrucción en lenguaje ensamblador</i>	<i>Código de máquina</i>
0000 0001	MOV a, R1	0001 01 00 00000000
0000 0010	ADD #2, R1	0010 01 10 00000010
0000 0011	MOV R1, a	0001 01 00 00000100

Se concibe una pequeña palabra de instrucción, en la que los cuatro primeros bits son el código de la instrucción, donde 0001 y 0010 representan hipotéticamente las instrucciones MOV (mover) y ADD (sumar) respectivamente, el 00 representa (también hipotéticamente) la dirección de la variable a, el 01 representa la dirección de la variable R1, de donde se concluye que las tres instrucciones en ensamblador corresponden a:

MOV a, R1

Se mueve el contenido de la constante cuya dirección es a (00) a la variable R1 (cuya dirección en 10) y esto equivale a igualar el valor de la variable a= R1

ADD #2, R1

Se incremente el valor de la variable R1 en dos unidades (que convertidas a binario es 10) y equivale a R1 = R1 + 2

MOV R1, a

El valor que tenga R1 se guarda en la variable a (Igualándolos) R1 = a

De esta forma podemos imaginar que todo lo que se escribe en mnemónicos conlleva una traducción a binario. Según se mostró con un tipo muy particular (ya simplificado) de lenguaje ensamblador, solo con el objetivo de entender una traducción simulada de ensamblador a binario.

UNIDAD I • MARCO CONCEPTUAL

Ventajas y desventajas de los lenguajes de bajo nivel

La programación en el lenguaje de la máquina o en lenguaje simbólico tiene ciertas ventajas:

1. Mayor adaptación al equipo, sencillez siempre que se domine la arquitectura de la computadora.
2. Posibilidad de obtener la máxima velocidad con mínimo uso de memoria.

También tienen importantes inconvenientes:

- Imposibilidad de escribir código independiente de la máquina.
- Mayor dificultad en la programación y en la comprensión de los programas.

Por esta razón, a finales de los años 1950 surgió un nuevo tipo de lenguaje que evitaba los inconvenientes, a costa de ceder un poco en las ventajas.

Estos lenguajes se llaman "de tercera generación" o "de alto nivel", en contraposición a los "de bajo nivel" o "de nivel próximo a la máquina".

LENGUAJE DE ALTO NIVEL

Principales lenguajes de alto nivel

- | | | |
|-----------------|-----------|------------|
| • Léxico | • C | • Perl |
| • Basic | • C# | • Ada |
| • Logo | • C++ | • Cobol |
| • ALGOL | • Clipper | • Java |
| • Pascal | • Python | • Fortran |
| • Object Pascal | • Rubi | • Modula-2 |

Lenguajes funcionales

- Lisp
- Haskell

Nota: Dentro de los lenguajes de nivel alto, el lenguaje C y sus derivados son considerados lenguajes de nivel intermedio.

TRADUCTORES

Los traductores son un conjunto de programas, necesarios para lograr la conversión de un código a otro, de un lenguaje a otro y de diferentes niveles. Antes de mostrar los diferentes tipos que hay, es importante remarcar que la traducción de un código a otro es indispensable para poder lograr la comunicación hombre máquina, simplificando esta labor, se han creado herramientas “traductores” cada vez más cercanos al hombre y más lejanos del lenguaje de máquina, Por esta razón trataremos de definir brevemente que es un nivel.

Los niveles en los programas se consideran tomando en cuenta las características del lenguaje que se utiliza. En términos muy generales, los lenguajes pueden ser:

De alto nivel. Cercanos al hombre y lejanos a las características particulares de la computadora o al lenguaje de máquina.

De bajo nivel. Cercanos a la arquitectura de las computadoras y con gran detalle para el código que realiza el hombre debido a su cercanía con el tipo particular de computadora.

De nivel intermedio. Estos son poderosos como los lenguajes de bajo nivel (ensambladores) y accesibles al hombre, como son los lenguajes de alto nivel. El ejemplo clásico de este tipo de lenguajes de nivel medio, son los que tienen alguna relación con los diferentes tipos de lenguaje C, ya que estos tienen una gran variedad por la cantidad de derivados que se han generado, incluyendo a Java. Se considera que los lenguajes de control de tareas o lenguajes que se utilizan para dar instrucciones a los sistemas operativos, son lenguajes conocidos como lenguajes de control de trabajos (del Inglés *“Job Control Language”* o JCL). Corresponden a este nivel. Con base a lo anterior, existen infinidad de niveles intermedios de traducción o interpretación, los cuales llevan diferentes nombres tales como:

Ensambladores. Estos existen en gran variedad y en diferentes tipos, de acuerdo a sus particularidades, ya que dentro de los mismos ensambladores, también incluyen varios tipos de niveles de traductores debido a que son de bajo nivel y dependen de las características específicas de cada computadora. Esto hace que se tenga un tipo especial de ensamblador para cada tipo de procesador en particular, dependiendo de su tamaño de palabra, el número de registros internos con los que cuenta, así como características específicas de sus tipos de

UNIDAD I • MARCO CONCEPTUAL

direcciónamiento. Además de las características físicas con las que se diseña un lenguaje ensamblador, estos lenguajes pueden ser identificados de forma genérica, en los siguientes tipos:

Los pseudo-ensambladores o emuladores de ensamblador, un ejemplo de este tipo diseñado para el procesador *INTEL* modelo 80286 es el programa que se ejecuta como comando externo, conocido como el *DEBUG*. Otro tipo de ensambladores son los macro ensambladores, también para procesadores *INTEL*, un ejemplo de este es *MASM* y los ensambladores en sí.

Es importante hacer la diferencia entre el concepto de ensambladores como procesos y como lenguajes, ya que en el proceso de traducción de un lenguaje de programación de alto nivel se requiere traducir a bajo nivel y para esto se tiene un programa o procesador que por lo general proporcionan los sistemas operativos llamado ensamblador (*ASSEMBLER*), y este adecua el código de un lenguaje de alto nivel a otro de bajo nivel, mismo que es el que la computadora interpreta y ejecuta. Los lenguajes ensambladores son a los que nos referimos en la clasificación de los niveles que aquí se describen, ya que un compilador puede incluir una fase de síntesis o ensamble.

INTÉRPRETES O TRADUCTORES INMEDIATOS.

Estos tienen como principal característica la posibilidad de emitir una respuesta inmediata (en tiempo real), esto es, para cada instrucción se genera una acción o respuesta que se obtiene de la misma máquina al mismo tiempo que se realiza la interpretación del código escrito por el usuario programador. Este es un nivel de traducción eficiente y económica, debido a que no requiere gran capacidad de almacenamiento, ya que no genera código intermedio y por eso es eficiente al producir una respuesta inmediata. Su inconveniente es que deja el código fuente a disposición del usuario.

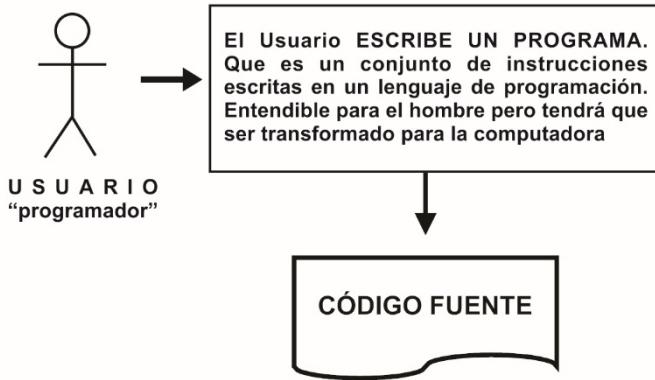
Existen lenguajes como *BASIC* y *C* de los cuales se han construido versiones compiladas e intérpretes con respuesta inmediata, la diferencia entre uno y otro es la cantidad de recursos que requiere cada tipo de herramienta y los niveles de traducción que a continuación se describen. Es importante ubicar a los traductores, intérpretes y compiladores como herramientas de desarrollo de programas para la utilización de computadoras cualquiera que sea su capacidad y complejidad. Por lo que se abrevian

como herramientas de desarrollo o productividad, existiendo una gran variedad por la combinación de sus características.

Niveles de traducción. Recordemos la necesidad de realizar una comunicación hombre máquina para poder controlar los trabajos que realiza una computadora. Para esto, ya se explicó que el nivel más alto de traducción es el lenguaje cercano al hombre, los cuales se conocen como lenguajes naturales tales como el inglés, español, francés, alemán, etc. Estos lenguajes son entendibles para el hombre, pero difíciles de definir en sus reglas gramaticales debido a la gran cantidad de excepciones, así como la variedad de palabras y distintos significados que se requieren para expresar ideas que por lo general están sujetas a un contexto. (Que son un tipo de gramática que se explicará posteriormente).

Recapitulando lo anterior, para poder lograr que la máquina identifique palabras, se necesita establecer reglas gramaticales y realizar procesos tales como: **Codificación, Análisis, Síntesis**, estos tres conocidos como traducción y la parte de interpretación o ejecución del código conocido como tiempo de ejecución o *RUN TIME*. La codificación es una operación consistente en representar una información o instrucción mediante un código (ver el glosario).

En la **figura 1.7** se presenta un diagrama esquemático de un proceso de codificación, en este se incluye un usuario programador, el cual escribe un código en



algún lenguaje de programación. El código se puede introducir directamente en la computadora (de forma intérprete) o bien grabar el código en un archivo llamado programa fuente.

Figura 1.7 Elementos que intervienen en el proceso de codificación

La codificación es la que realiza el hombre, con conocimientos de programación. Él escribe las instrucciones en la computadora, las cuales se guardan en un archivo conocido como

UNIDAD I • MARCO CONCEPTUAL

programa fuente o código fuente. Este programa fuente entra a un proceso de análisis y síntesis conocido como compilación, donde además se realiza la labor de análisis de las palabras estructuradas en el código fuente para verificar que sean léxica, semántica y sintácticamente correctas.

Después de realizado lo anterior, si son correctas, pasan a la fase de análisis y después se aceptan para pasar a la siguiente etapa, que es de síntesis y generación de un código que será el que la máquina interpretará y ejecutará. De aquí que en esa cantidad de pasos intermedios se tienen una variedad de compiladores que en términos generales y basándonos en sus niveles de traducción, los podemos describir como:

- **Pre-compiladores o Pre-procesadores**
- **Compiladores**
- **Pseudo-compiladores.**

El concepto de compilación se describe brevemente en la **figura 1.8** y en glosario al final de este material. Fases y etapas que intervienen, en el proceso de compilación y elementos que intervienen en el proceso de codificación. Este es el *COMPILE TIME*. Los compiladores también se pueden clasificar por el número de pasadas, entendiéndose por pasada la revisión que se realiza sobre el código fuente, pero todos estos tipos de compiladores requieren de otro proceso de invocación o ejecución conocido como *RUN-TIME*. Los dos tiempos que tiene un compilador: *COMPILE-TIME* y *RUN-TIME* es otra característica que diferencia a un compilador de un intérprete, ya que éste último solo tiene un tiempo que es una invocación y su inmediata ejecución.

Intérpretes. Programas que conjuntan el tiempo de revisión, análisis, síntesis y ejecución (*RUN-TIME*), en una sola orden o instrucción. Son eficientes y económicos porque no requiere de grandes recursos de almacenamiento y se realiza todo en un solo tiempo.

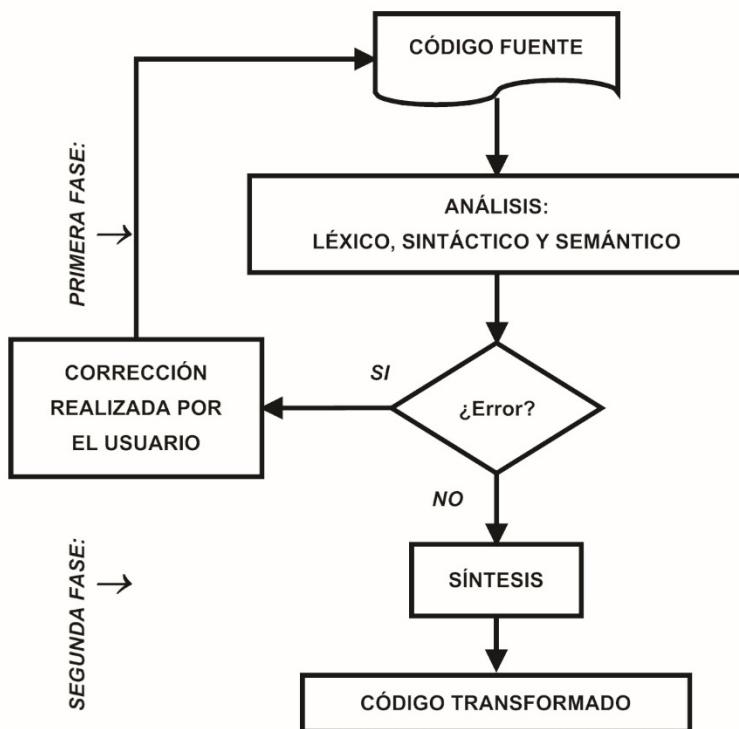


Figura 1.8:
Fases y etapas que intervienen, en el proceso de compilación y elementos que intervienen en el proceso de codificación.
Este es el COMPILE TIME

Compiladores. Como ya se describió, los diferentes nombres y tipos de compiladores están en función del nivel de traducción que se tiene del código fuente. Cuando el código fuente es de muy alto nivel, por ejemplo, herramientas que se conocen como 5GL (Quinta generación).

Los lenguajes 5GL pueden requerir de un pre-proceso o etapa previa de adaptación o traducción agregada para disminuir el nivel de 5° a 4° nivel y con esto agregarle instrucciones que completen la información que la computadora requiere.

El código transformado puede ser un conjunto de instrucciones agregadas al código fuente original, puede ser un código en alto nivel, y solo adecuarlo como es el caso de la compilación que se realiza parcialmente denominada pre-compilación o pseudo-compilación o puede generarse un código conocido como objeto que es el

UNIDAD I • MARCO CONCEPTUAL

lenguaje de máquina que corresponde al código fuente que el usuario programador escribió. Sobre los niveles en los lenguajes de programación, ya se había descrito una clasificación sólo de tres niveles:

El *alto nivel* que corresponde a los lenguajes de programación conocidos como formales y el *bajo nivel* que corresponde a los lenguajes más cercanos a la arquitectura de la computadora. Pero, dada la diversidad en los lenguajes de programación actuales de los cuales ya son miles. Se han establecido *niveles intermedios*, como ya se mencionó el ejemplo de un nivel intermedio es el lenguaje C con la variedad de herramientas que se desprenden de este, tales como *TURBO C*, *AWK*, *C++*, gracias a que como ya se mencionó.

El lenguaje C y sus derivados, se considera de alto nivel por su facilidad, flexibilidad y de bajo nivel por la posibilidad de administrar los espacios en memoria y su poderío en el uso de recursos computacionales, mismas características que hasta han servido como herramienta para desarrollar el sistema operativo más evolucionado con el que se cuenta actualmente, este es *UNIX* y sus cientos de clones (el más famoso y actual *LINUX*).

Otros niveles que a continuación se describen son a juicio particular y temporal, ya que este tema es polémico y no se ha llegado a criterios que se fijen como una norma o estándar para definir a los lenguajes por niveles. Se utiliza de forma indistinta el término nivel con el concepto de generaciones de lenguajes de programación, en el entendido de que los lenguajes de programación son creados por el hombre, diferentes a los lenguajes naturales que se les considera de origen Divino y muchos de los lenguajes naturales se forman de la combinación entre ellos, generando también los conocidos dialectos, lenguas y en el caso particular de los lenguajes de programación o artificiales, también existen infinidad de versiones del mismo lenguaje, por ejemplo del lenguaje C existen versiones intérpretes, compilados, derivaciones como C++, C#, etc. También existen dialectos, versiones y actualizaciones. Esto hace de los lenguajes de programación un mundo que es importante conocer en el sitio o “site” de Kinnersley (2015).

Con base a lo anterior, sucede algo similar en los lenguajes de programación o artificiales, de los cuales surgen niveles intermedios a lo largo de la historia, misma que se trata de resumir a continuación. Aquí el definir los niveles de lenguajes se presentan al menos dos puntos de vista diferentes.

1. Autores que definen el nivel 0 como lenguaje ensamblador sin considerar como lenguaje nativo el lenguaje de máquina.
2. Autores que cuentan a partir de los lenguajes que son de programación y
3. Autores que consideran al lenguaje de máquina y ensamblador ambos como de nivel cero por ser de bajo nivel, dependientes del hardware y ambos complejos.

Para efectos de este material se describen los niveles a juicio del autor de este material con lo que se puede considerar una descripción lógica y lo más completa posible.

Nivel cero o de más bajo nivel. Este nivel corresponde al lenguaje de máquina, el cual como ya se mencionó, consiste en unos y ceros integrados en códigos diversos, controlados de forma alambrada o bien con micro-instrucciones.

Nivel uno corresponde al lenguaje ensamblador basado en abreviaturas, mnemónicos, mnemotécnicos o mnemoclaves del idioma inglés. Estos son dependientes del hardware y existe un tipo de lenguaje ensamblador para cada tipo de procesador.

Nivel dos o segundo nivel, corresponde al de los lenguajes de programación formales. Estos tienen estructuras y reglas gramaticales muy restrictivas, sólo permiten el uso de datos que se tienen que declarar con anterioridad a su uso (esta es una característica que se conoce como fuertemente “*tipeado*”) y comprenden un gran número de palabras reservadas o instrucciones particulares para cada lenguaje con formatos específicos, algunos ejemplos de estos lenguajes de segundo nivel son, el compilador de *ALGOL*, *COBOL* y *FORTRAN*. Aquí cabe hacer la aclaración de que algunos autores consideran a este nivel como de primera generación de los lenguajes de programación, considerando solo a los lenguajes de nivel alto como lenguajes, y los del nivel bajo tales como lenguaje de máquina y ensamblador, de bajo nivel, o bien como de nivel cero o generación cero.

Nivel tres, son lenguajes de programación informales que tienen como base la restrictiva formalidad del anterior nivel, pero incluyen estructuras gramaticales más flexibles y dinámicas, mismas que le dan mayor libertad al usuario programador. Aquí nace el lenguaje *Basic*, después *Smalltalk*, *PASCAL* y culmina con el lenguaje *C*. Al que se le considera también de nivel medio (entre alto y bajo nivel ya mencionado con anterioridad).

UNIDAD I • MARCO CONCEPTUAL

Nivel cuatro, conocidos como de cuarta generación, se abrevian como 4GL, estos lenguajes tienen como principal característica facilitarle la labor de codificar al usuario programador. Con esto se crean herramientas como DBASE, SQL, INFORMIX, Java Creator, Blue Jeen, etc. Estas herramientas son generadoras de código automático. Cabe aclarar que, a partir de la tercera, cuarta y quinta generación existe mayor diversidad de opiniones en base a que no se tiene un consenso estándar sobre las características que guarda cada tipo de lenguaje, este tema se hace cada día más polémico y difícil de definir, se puede decir que al hablar de generaciones o niveles de los lenguajes de programación estamos entrando a un mundo extenso, en constante crecimiento y variabilidad.

Por lo anterior es importante aclarar que se pueden encontrar discrepancias en la forma de clasificar los lenguajes por niveles y generaciones mismas que se puede decir se manejan de forma indistinta en los libros sobre esta materia. Según apreciación personal, es más fina la clasificación de los lenguajes en base a paradigmas de programación y estos son descritos con mayor detalle después de los tipos de traductores sobre los lenguajes.

Es importante concluir que entre más alto es el nivel que alcanza el lenguaje, más pasos intermedios se deben incluir para lograr una traducción e interpretación de la computadora. Por lo anterior, quizá hasta sea necesario crear pre-procesos esto es, procesos previos a la compilación conocidos como pre-compiladores. Este aspecto está abierto para futuros desarrollos los cuales no presentan un límite, sólo el que la creatividad del programador decida.

EJERCICIO 8.

Desarrolle un resumen o bien organizador gráfico que describa los diferentes autores (al menos dos) que han escrito sobre niveles, descripciones y generaciones de lenguajes de programación en el cual se encuentren al menos dos de tres puntos de vista diferentes para clasificar o caracterizar a los lenguajes artificiales o creados con el hombre para dar instrucciones a la computadora.

Ejemplo de un pre-procesador o pre-compilador:

En el diagrama de la **figura 1.9** del lado izquierdo, sólo se esquematiza de forma genérica la función de un pre-procesamiento que es bajar el nivel del lenguaje a procesar.

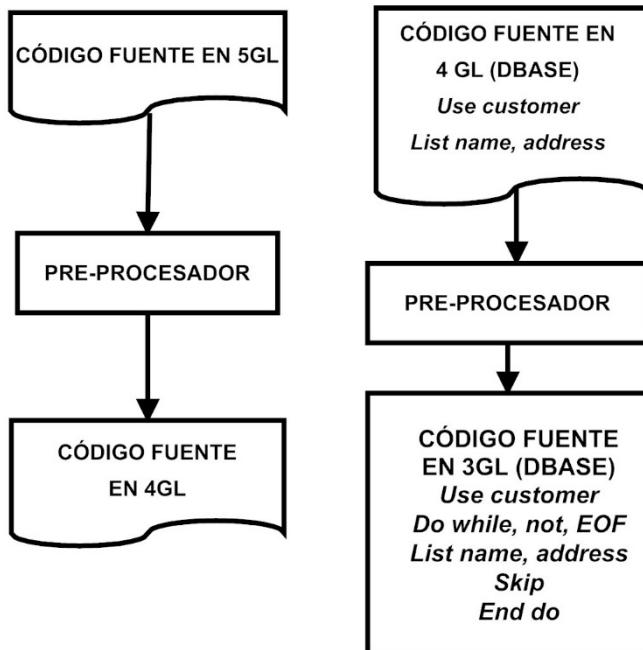


Figura 1.9 Diagrama esquemático de un pre-procesador o pre-compilador

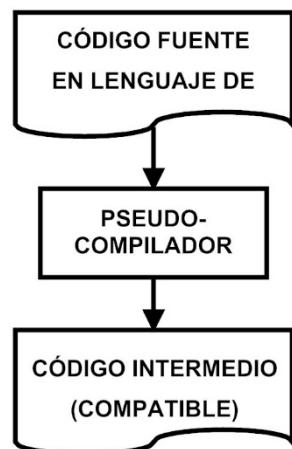
En el lado derecho del diagrama se puede observar en el código que se escribe para una herramienta 4GL, solo se tienen dos líneas de instrucciones y en una herramienta 3GL, se agregan más instrucciones para realizar la misma función descrita en la herramienta 4GL, que consiste en tomar la base de datos conocida como clientes “customer” y de ella obtener una lista con el nombre y la dirección de los clientes. Este ejemplo corresponde a: “SQL Informix”.

UNIDAD I • MARCO CONCEPTUAL

El último ejemplo relacionado con los niveles de traducción es el caso de un pseudo-compilador el cual se puede describir como un generador de código intermedio, esto es, se crea un código estándar para lograr algunas ventajas con respecto a la arquitectura de la computadora, de forma muy general tenemos en el siguiente diagrama de bloques **figura 1.10**

En este caso el pseudo-compilador es un generador de código intermedio, cuya principal ventaja es su compatibilidad. El ejemplo clásico, conocido y actual de este tipo de herramientas es la máquina virtual de Java, donde se tiene un generador de código intermedio conocido con *BYTECODE* el cual es independiente de la arquitectura de la computadora donde se corre para hacer posible que las aplicaciones corran en cualquier plataforma, lo mismo para *UNIX*, *MACINTOSH* Y *MICROSOFT*.

Figura 1.10 Diagrama esquemático de un pseudo-compilador



Una máquina virtual según se muestra en la **figura 1.11**, es una capa intermedia de software que se le agrega al hardware para lograr la compatibilidad y el enmascaramiento o bien el ocultamiento de las particularidades del hardware donde se ejecutan las aplicaciones diseñadas para la máquina virtual en cuestión.

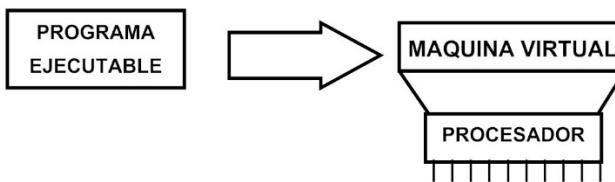


Figura 1.11 Diagrama esquemático de una máquina virtual

Gracias a los pseudo-compiladores se tiene una compatibilidad y transportabilidad (fácil implementación en diferentes plataformas) de los programas que se desarrollen con generación de código intermedio. Antes de concluir con el tema de los niveles de traducción, el cual está relacionado con los diferentes tipos de

interpretación, es importante remarcar que entre más bajo es el nivel del lenguaje que se utiliza en el programa fuente, más sencillo es el proceso que se realiza para obtener la traducción al lenguaje de máquina.

Otro ejemplo es el codificar un programa fuente al lenguaje ensamblador, donde se tiene solo un proceso para lograr obtener un código de máquina ejecutable. Es importante remarcar la importancia que tiene este tipo de herramientas de bajo nivel, pues siguen dando eficiencia y poderío en el uso de recursos computacionales.

Ya para concluir sobre los niveles de traducción es importante remarcar que con el éxito de la computadora personal, se tienen muchas facilidades en cuanto a herramientas de desarrollo y paqueterías, pero es difícil, por ejemplo, pensar que se tienen microprocesadores con velocidades medibles en gigahertzs (mil millones de hertzs) y aún no se tiene un ensamblador para procesador INTEL PENTIUM. Por lo que en este tema se pretende aprovechar la necesidad de crear niveles de traducción más eficientes y realizarlos con herramientas de bajo nivel como son los ensambladores. Ejemplo de un proceso de ensamble en la **figura 1.12**

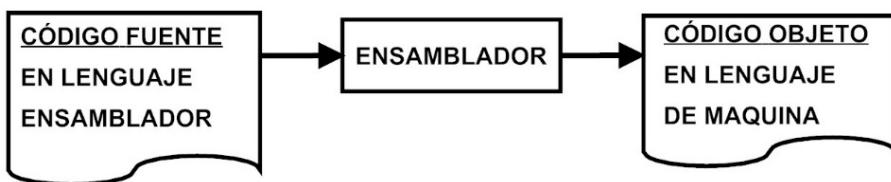


Figura 1.12 Diagrama esquemático de un ensamblador

Paradigmas de los lenguajes de programación. Para completar el tema sobre los niveles de traducción, ya se mencionó que una cualidad más detallada sobre los diferentes lenguajes que se han diseñado hasta la fecha son los paradigmas o modelos a los que responden los actuales lenguajes. A continuación, se da una lista con los paradigmas o modelos que tienen los lenguajes actuales de programación, no sin olvidar que cada día se puede desarrollar un nuevo lenguaje o un nuevo paradigma. Ya que los lenguajes que se conocen hasta la fecha corresponden a los modelos actuales o paradigmas conocidos con la tecnología que existe en este momento, pero estos generarán nuevos modelos, mismos que guardan características heredadas, tal como los hijos se parecen a sus padres.

UNIDAD I • MARCO CONCEPTUAL

Los paradigmas conocidos hasta la fecha son:

Imperativo. Corresponden a la arquitectura de John Von Newmann “programa almacenado en memoria”, con instrucciones estructuradas, funciones y manejo de memoria dinámica con apuntadores a direcciones de memoria que controla el usuario. Ejecución señalizada. Ejemplos: *ALGOL, COBOL y FORTRAN*.

Funcional (operacional). Los lenguajes que responden a este modelo optimizan la representación interna de los datos y le dan particular importancia al orden de las decisiones que se toman en la lógica del código. Este paradigma está basado en la optimización de los datos que se almacenan en memoria. El ejemplo más representativo de este tipo de paradigma es *LISP*.

Orientado a objetos. Este quizá es el paradigma más difundido y conocido para los nuevos programadores, ya que pretende facilitar la labor de codificar el detalle de cómo resolver un problema. Los lenguajes orientados a objetos proporcionan una gran cantidad de elementos con los cuales no se tiene que definir el cómo, sólo lo que se desea obtener del lenguaje y aprovechar el encapsulamiento (ocultamiento y protección del código), la inferencia, el paso de mensajes, las clases, subclases, la herencia, etc. El primer lenguaje representativo de este paradigma es el *Smalltalk* (en 1980), luego *C++* (en 1986), el *Object Pascal* (en 1989), *Object Cobol* (en 1993) hasta el actual y exitoso *Java* (en 1995).

Asíncrono. Este paradigma y el que se describe a continuación corresponde a las nuevas arquitecturas avanzadas de las computadoras, pueden optimizar las características de los microprocesadores con tipo de procesamiento paralelo o vectorial. Este paradigma corresponde a procesos asíncronos (en diferentes tiempos de ejecución) los cuales se hacen posibles mediante la comunicación entre procesos. El lenguaje de programación *LINDA* es un ejemplo representativo de uno de los lenguajes que agrega estas facilidades, tomando como base a los lenguajes *C* y *FORTRAN*.

Síncrono. En este modelo se tiene un procesamiento paralelo con alto grado de acoplamiento. Debe de haber una sincronía en todos los procesos que se ejecutan al mismo tiempo. Un ejemplo de un lenguaje que corresponde a este paradigma es el *ACTUS, Paralation LISP* que proveen construcciones independientes de la arquitectura de la computadora.

Funcional (definicional). Este paradigma tiene una gran variedad de funciones que el usuario programador puede utilizar para simplificar su código. Un ejemplo de este tipo de lenguajes es el lenguaje de programación llamado *Miranda* y el *Haskell*. Este último es un “poliformismo estático”, significa, variedad de formas pre-definidas, derivadas del lenguaje *Miranda* y ambos lenguajes (*Miranda* y *Haskell*) se caracterizan por tener funciones de alto orden, usar datos algebraicos definidos por el usuario y un sistema de clases. También proporciona operadores con re-direcciónamiento dinámico conocido con el término en inglés “overloading”, sistema funcional de entradas y salidas, arreglos funcionales y compilación por separado.

Transformacional. Este paradigma permite definir reglas sobre argumentos y términos que se re-escriben. El ejemplo representativo de este tipo de lenguaje es *Bertrand*, que corresponde a la característica de ser un lenguaje de especificación basado en reglas, sobre argumentos y términos que se pueden re-escribir. En los lenguajes restrictivos el usuario programador debe especificar explícitamente los árboles de búsqueda y la propagación restrictiva que es la característica básica de este paradigma transformacional.

Lógico. Este paradigma emplea la lógica y las reglas de inferencia, tratando de adecuarse al lenguaje natural. El ejemplo representativo de este paradigma es *PROLOG* y todos sus derivados.

Basado en formas (Form-based). Este es el paradigma que corresponde a todas las hojas electrónicas de cálculo que se han diseñado, desde *Visical*, *Lotus* hasta *Excel*.

De flujo de datos (dataflow). Los que corresponden a este paradigma se programan a través de elementos gráficos. Un ejemplo de este lenguaje es “Show and Tell”.

Restringido (constraint). Especifica la relación que debe permanecer. El usuario programador es el responsable de especificar la relación y el sistema es el encargado de mantenerla. Aunque no se tienen muchos lenguajes que respondan a este paradigma, un ejemplo que corresponde a este paradigma es el lenguaje visual *ThigLab*.

Demostracional. Mejor conocido como de pregunta con ejemplos, en inglés, “Query by example”. En este paradigma se presenta el “¿qué quiero?” con un ejemplo y automáticamente se genera código, mismo que le indica a la computadora el cómo obtener lo que quiero con una generación inmediata de resultados.

UNIDAD I • MARCO CONCEPTUAL

Otro enfoque de clasificar a los traductores de forma genérica en ciencias de la computación, es definir a los procesadores de lenguajes como aquellos programas destinados a trabajar sobre una entrada que por la forma como ha sido elaborada, pertenece a un lenguaje en particular reconocido o aceptado por el programa en cuestión. Los procesadores de lenguajes se clasifican como traductores o intérpretes.

Un traductor es un programa que recibe una entrada escrita en un lenguaje (el lenguaje fuente) a una salida perteneciente a otro lenguaje (el lenguaje objeto), conservando su significado. En términos computacionales esto significa que tanto la entrada como la salida sean capaces de producir los mismos resultados. Un intérprete, por otra parte, no lleva a cabo tal transformación; en su lugar obtiene los resultados conforme va analizando la entrada. Los traductores son clasificados en compiladores, ensambladores y preprocesadores.

Lenguaje fuente: Lenguaje origen que traduce el compilador.

- **Lenguaje objeto:** Lenguaje meta, al cual traduce el compilador.
- **Lenguaje del compilador:** Lenguaje en el que está escrito el compilador.



Figura 1.13 Diagrama esquemático de los lenguajes involucrados a un proceso de compilación.

En la **figura 1.13** se muestra la relación de tres lenguajes distintos involucrados en el proceso que realiza un compilador. El lenguaje fuente, mismo que puede ser lenguaje C, el lenguaje objeto que puede ser un lenguaje de máquina y el lenguaje compilado que puede ser un lenguaje ensamblador.

Supongamos que se quiere implementar un nuevo lenguaje A(N) en una máquina determinada. Disponemos solamente de un ensamblador para dicha máquina. En principio parece que la solución es escribir un compilador en lenguaje ensamblador (E) que traduzca desde el lenguaje A(N) al lenguaje máquina LM según se muestra en la **figura 1.14**.

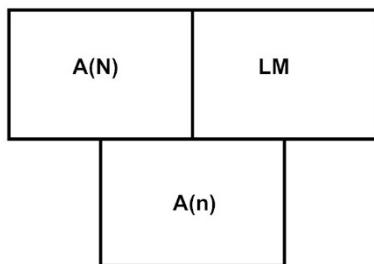


Figura 1.14 Diagrama esquemático de Lenguaje fuente: Lenguaje origen que traduce el compilador con bootstrapping.

Esto en la práctica resulta bastante complicado, dado que programar en ensamblador es muy laborioso. Lo que se hace en estos casos es desarrollar un lenguaje restringido A(1), parecido al A(N) pero más simple, y para este lenguaje escribir el compilador en ensamblador, o en cualquier otro lenguaje soportado por la máquina. Una vez construido este compilador, y dado que nuestra máquina es ya capaz de entender el lenguaje A(1), se puede desarrollar un compilador para otro lenguaje A(2) escribiéndolo en el lenguaje A(1), y así sucesivamente hasta llegar a obtener un auto-compilador del lenguaje A(N). Esta técnica se conoce como **bootstrapping** (en glosario).

El primer compilador de Pascal desarrollado en Zurich por Wirth fue posible gracias a esta técnica, “bootstrapping”. Según se muestra en el diagrama esquemático que se puede ver en la **figura 1.15**. Lenguaje fuente, Lenguaje origen que traduce el compilador con el conocido compilador C de GNU que emplea también este mecanismo en tres pasos.

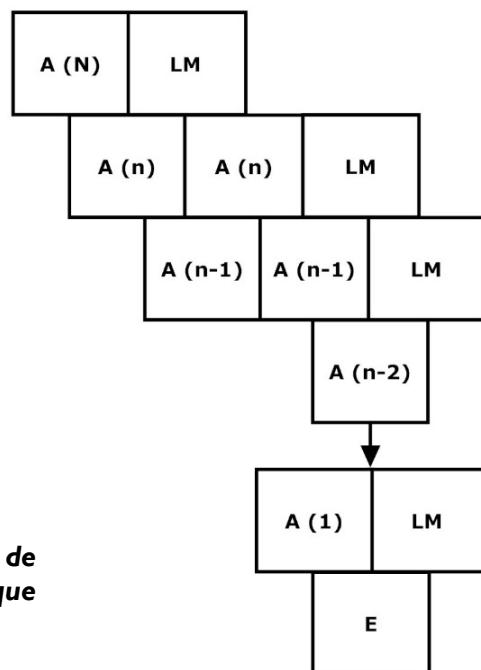


Figura 1.15 Diagrama esquemático de Lenguaje fuente: Lenguaje origen que traduce el compilador con bootstrapping

1.4 Conceptos De Expresiones Regulares

Las expresiones regulares son especificaciones sobre un patrón de símbolos en el orden y forma que se define siguiendo una nomenclatura basada en la teoría de conjuntos. Estas expresiones o representaciones son procesadas directamente por lenguajes como AWK, incluyendo Perl, PCRE, PHP,.NET, Java, JavaScript, XRegExp, VBScript, Python, Ruby, Delphi, R, Tcl, POSIX, y muchos otros lenguajes. Son ideales para definir filtros que simplifican el diseño de cualquier aplicación. En las expresiones regulares se aplican los siguientes conceptos: Cadenas, Alfabetos y Lenguajes.

Ejemplos de Alfabetos $\Sigma = \text{Alfabeto}$

$$\Sigma_1 = \{0, 1\}$$

$$\Sigma_2 = \{a, b, c\}$$

Ejemplos de Cadenas $w = \text{Cadena}$

$$w = 011100 \quad v = 11 \quad u = bcabb$$

Cuando hay cadenas de un mismo Alfabeto, se pueden concatenar o pegar:

$$wv = 01110011$$

La concatenación **no** es comutativa.

Cadena vacía ϵ

$$\epsilon X = X\epsilon = X$$

La longitud de una cadena es el número de símbolos y tienen un valor absoluto $|x|$

Ejemplos:

$$|w| = 6 \quad \text{Longitud } 6$$

$$|\epsilon| = 0 \quad \text{Longitud de } \epsilon \text{ es } 0$$

PREFIJOS Y SUBFIJOS.

Prefijo: Porción inicial de una palabra o cadena subyacente.

Subfijo: Cualquier porción final de una cadena o parte final de la palabra.

Para prefijos y sufijos a manejar, pueden tener un significado por sí solos. ϵ_1 es un prefijo.

b₁ es un prefijo, al igual que los siguientes:

- bc
- bca Debido a que es cualquier porción inicial de la cadena u.
- bcab
- bcabb
- ε_1 es un sufijo

b es un sufijo de la cadena, al igual que los siguientes:

- bb
- abb
- cabb
- bcabb

Cuando el prefijo no es igual a la palabra completa es un prefijo propio. Los sufijos propios son todos aquellos sufijos que no sean la palabra completa. Conjunto que se puede obtener a partir de cualquier alfabeto.

Σ^* Consta de todas las cadenas que se pueden formar con los símbolos de Σ y es infinito.

Ejemplo:

$\Sigma_1^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, \dots, \infty\}$ (Siguen los de tres dígitos, cuatro, etcétera, por eso es infinito)

Lenguajes: L ⊂ Z* Lenguaje sobre Σ

Por lo que un lenguaje puede ser finito e infinito.

Lenguajes.

\emptyset Formalmente nos es útil.

{ ε } Tiene el elemento de la palabra vacía.

Σ^* Lenguaje

Los elementos también pueden operarse dado que estos son conjuntos.

Operaciones entre lenguajes.

La concatenación.

$L_1 L_2 = \{WV \mid W \in L_1, V \in L_2\} = u u u u \dots u \quad (n \text{ veces}) = \epsilon$

Cerradura de Kleene.

Se denota como L^* (V^*)

En L^* están concatenadas todas las palabras contra todas.

Cerradura positiva

Siempre tendrá a ϵ como un elemento

Tendrá a ϵ si L la tenía desde el principio.

Ejemplos:

$L_1 = \{a, ab\}, L_2 = \{b, ba\}$

$L_1 L_2 = \{ab, aba, abb, abba\} = \{\epsilon, a, ab, aa, aab, aba, abab, \dots\}$

Se suprime ϵ porque no aparece $L_1 = \{a, ab, aa, aab, aba, abab, \dots\}$

ϵ está en la cerradura positiva. Si y sólo si ϵ está en L (el lenguaje) originalmente desde un principio.

1.5 Conceptos de autómatas finitos

Un autómata es un diagrama dirigido “dígrafo” el cual tiene reglas muy simples en su construcción para modelar el comportamiento de cualquier sistema que recibe entradas.

La entrada o inicio, se representa con una línea o arista o flecha en dirección a un círculo que es el estado inicial. Las entradas son desglosadas en símbolos (caracteres) que pueden ser dígitos los cuales representan la transición entre un estado y otro. Hasta llegar a la aceptación de un conjunto de símbolos o cadenas. A este conjunto de símbolos, se le conoce como lenguaje aceptado. Los símbolos o abstracciones del mundo real. Los procesos o conjuntos de actividades son estados o nodos o vértices los cuales se relacionan hasta el final del diagrama que se muestra en la **figura 1.16**.

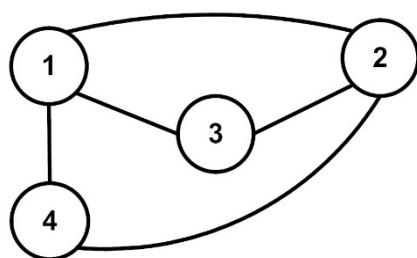


Figura 1.16
Diagrama de un conjunto no ordenado de vértices (círculos) y relaciones (líneas)

Es un conjunto no ordenado de puntos o vértices los cuales se describen de la siguiente forma:

- $V = \{1, 2, 3, 4\}$ Conjunto de vértices.
- δ = Conjunto de aristas.
- $\delta = \{ \{1,2\}, \{2,3\}, \{1,3\}, \{2,4\}, \{1,4\} \}$

Trayectorias: Sucesión de vértices que entre dos consecutivos hay una arista.

Longitud de trayectorias: Número de aristas que aparezcan en la trayectoria.

Número de vértices: 1.

Ciclo: Trayectoria cerrada de la misma.

Longitud: número de aristas que aparecen en el ciclo.

Gráficas dirigidas: Digráficas en la **figura 1.17**.

Nodo: punto de intersección o unión de varios elementos que confluyen en el mismo lugar.

Arcos: Parejas ordenadas de nodos.

$$\delta = \{ (1,2), (2,3), (3,1), (4,2), (1,4) \}$$

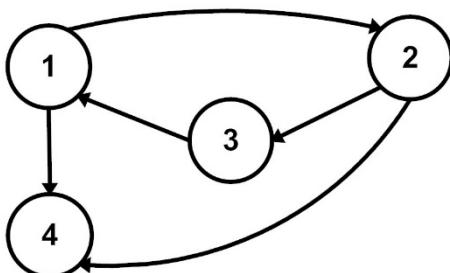
Trayectoria dirigida: $1 \rightarrow 2 \rightarrow 3$

Ciclo: $1 \rightarrow 2 \rightarrow 3 \rightarrow 1$

(1 es predecesor de 2)

(2 es sucesor de 1)

Figura 1.17. Diagrama dirigido con cuatro estados (círculos del 1 al 4) y cinco transiciones o arcos dirigidos.



AUTÓMATAS FINITOS

El autómata finito es un modelo matemático que consta de una quíntupla, cinco elementos (5-tupla) $M = \{q, \Sigma, \delta, q_0, F\}$ de un sistema, con entradas y salidas discretas. El sistema puede estar en cualquiera de un número finito de configuraciones o “estados”. El estado del sistema resume la información concerniente a entradas anteriores y que es necesaria para determinar el comportamiento del sistema para entradas posteriores.

El mecanismo de control de un elevador es un buen ejemplo de un sistema de estados finitos. El mecanismo **no** recuerda todas las demandas previas de servicio, sino solo el piso en el que se encuentra, la dirección de movimiento (hacia arriba o hacia abajo) y el conjunto de demandas de servicio aún no satisfechas.

Otro ejemplo que tenemos es que ciertos programas como los editores de texto y los analizadores léxicos que se encuentran en la mayoría de los compiladores, son diseñados como sistemas de estado finito.

AUTÓMATA FINITO DETERMINISTA

Entonces podemos definir que un autómata finito determinista AFD consiste en un conjunto finito de estados y un conjunto de transiciones de estado a estado, que se dan sobre el símbolo de entrada tomado de un alfabeto Σ . Un estado q_0 , que es el estado inicial, en el que el autómata comienza.

Para determinar los distintos miembros de un lenguaje podemos construir un diagrama, este diagrama tiene la forma de un grafo dirigido con información adicional, y se llama diagrama de transición. Los nodos del grafo se llaman estados y se usan para

señalar hasta que lugar se ha analizado la cadena. Las aristas del grafo que se etiquetan con caracteres del alfabeto determinan las transiciones. Naturalmente nosotros debemos comenzar por un estado inicial, y cuando se hayan tratado todos los caracteres de la cadena correspondiente, necesitamos saber si la cadena es legal. Para ello se marcan ciertos estados como estados de aceptación o estados finales.

Ejemplo: En la siguiente figura 1.18 se ilustra el diagrama de transiciones de un Autómata Finito. El estado inicial q_0 está indicado por la flecha con la etiqueta “inicio” (en este ejemplo existe solo un estado final, pero pueden existir más en otros autómatas), también q_0 en este caso indicado por el doble círculo. Este autómata acepta todas las cadenas de ceros y unos, en las que el número de 0's y el número de 1's son pares. Dado este ejemplo, formalmente, un autómata finito determinista **AFD** es una colección de cinco elementos:

1. Un alfabeto de entrada Σ .
2. Una colección finita de estados Q .
3. Un estado inicial S .
4. Una colección F de estados finales o de aceptación.
5. Una función $\delta: Q \times \Sigma \rightarrow Q$, conocida como función de transición que determina el único estado siguiente para el par (q_i, σ) correspondiente al estado actual q_i y la entrada σ .

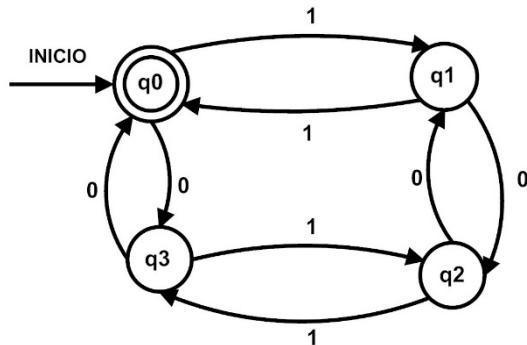


Figura 1.18 Diagrama de transiciones de un Autómata Finito Determinístico

Denominaremos a un **AFD** con M , por lo que usaremos $M = \{Q, \Sigma, S, F, \delta\}$ para indicar el conjunto de estados, el alfabeto, el estado inicial, el conjunto de estados finales y la función asociada con el **AFD**. Se dice que una cadena x es aceptada por un **AFD**.

Si $\delta(q_0, x) = p$ para algún p en F , es decir que la cadena termine en uno de los estados de aceptación o bien estados finales. El lenguaje aceptado por M , denominado $L(M)$, es el conjunto $\{x \mid \delta(q_0, x) \text{ está en } F\}$. Por lo que un lenguaje es regular, si es el conjunto de cadenas es aceptado por un autómata finito.

UNIDAD I • MARCO CONCEPTUAL

Ejemplo: Considerando el diagrama de nuestro ejemplo anterior, en notación formal, este **AFD** se denota como $M = \{Q, \Sigma, S, F, \delta\}$ de la siguiente manera:

$$Q = \{q_0, q_1, q_2, q_3\}$$

$$\Sigma = \{0, 1\}$$

$$S = q_0$$

$$F = \{q_0\}$$

δ = según la tabla 1.1

		Entradas		Finales
		0	1	
Estados	q_0	q_3	q_1	1
	q_1	q_2	q_0	0
	q_2	q_1	q_3	0
	q_3	q_0	q_2	0

Tabla 1.1 Diagrama de transiciones δ del autómata.

Dada la cadena **110101** vemos que pertenece a $L(M)$.

Ejemplo: El siguiente **AFD** que se representa mediante la siguiente manera

$$M = \{Q, \Sigma, S, F, \delta\}.$$

$$Q = \{q_0, q_1\}$$

$$\Sigma = \{a, b\}$$

$$S = q_0$$

$$F = \{q_0\}$$

δ = según tabla 1.2

		Entradas		Finales
		a	b	
Estados	q_0	q_3	q_1	1
	q_1	q_2	q_0	0

Tabla 1.2 Diagrama de transiciones δ de un autómata finito determinístico

La secuencia completa es $q_0^a q_0^b q_1^a q_1^b q_0^a q_0$ por lo tanto..

Dada la siguiente cadena **ababa**, vemos que pertenece a **L(M)**. (Está integrado por las letras o símbolos de las transiciones en la parte superior izquierda de la secuencia completa que se forma).

AUTÓMATA FINITO NO DETERMINISTA

Si se permite que desde un estado se realicen cero o más transiciones mediante el mismo símbolo de entrada, se dice que el autómata finito es no determinista **AFDN** o **AFN**, pero llegaremos a la conclusión de que cualquier lenguaje aceptado por un **AFN** también es aceptado por un **AFD**. A veces es más conveniente diseñar **AFN** en lugar de **AFD**. De manera formal denotamos a un **AFN** mediante una colección de cinco objetos **(Q, Σ, S, F, δ)**, donde:

1. **Q** es un estado finito de estados.
2. **Σ** es el alfabeto de entrada.
3. **S** es uno de los estados de **Q** designado como estado de partida.
4. **F** es una colección de estados de aceptación o finales.
5. **δ** es una relación sobre **(Q x Σ)** y se llama relación de transición o función de transición expresada en una tabla de transiciones que hacen objetiva la función de producto cartesiano.

Ejemplo: En la figura 1.19 se muestra un diagrama de transiciones de un **AFN**. Observe que existen dos aristas con la etiqueta **0** que salen del estado **q₀**, uno que regresa a **q₀** y otro que va al estado **q₃**. Por lo que su relación de transición **δ** está en la siguiente página.

Tabla 1.3 Transiciones del diagrama de la figura 1.19

Estados	Entradas		
	0	1	Finales
q ₀	{ q ₀ ,q ₃ }	{ q ₀ ,q ₁ }	0
q ₁	{ δ }	{ q ₂ }	0
q ₂	{ q ₂ }	{ q ₂ }	1
q ₃	{ q ₄ }	{ δ }	0
q ₄	{ q ₄ }	{ q ₄ }	1

AUTÓMATA ϵ - TRANSICIONES

Este autómata es también un **AFN**, que incluye transiciones sobre la cadena vacía ϵ , cuando hablamos de una cadena vacía conecta a dos estados, nos referimos a que no se consume ningún símbolo del alfabeto.

Como ejemplo de estos autómatas tenemos los siguientes, en este primer ejemplo tenemos a un autómata que puede cambiar su estado de q_0 a q_1 sin consumir nada en la entrada. Observemos que q_1 es el único estado de aceptación de este **AFN**. Si w es cualquier cadena de 0 o más a 's, este autómata cicla sobre q_0 hasta que consume las a 's. Una vez que la cadena se vacía, se desplaza a q_1 con ϵ y lo acepta.

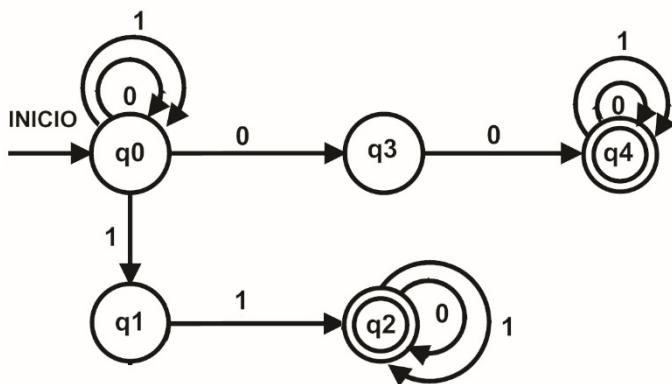


Figura 1.19 Diagrama de un autómata finto no determinístico.

En la **figura 1.20** el **AFN** puede moverse del estado q_0 al estado q_1 sin consumir nada en la entrada. En ambos **AFN**, la decisión de elegir una ϵ -transición se realiza de la misma forma que la de cualquier otra transición con elección múltiple que exista en un **AFN**. Según se muestra en la **figura 1.21**.

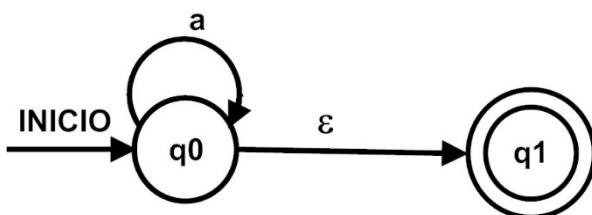


Figura 1.20 Se muestra un diagrama de un autómata finto no determinístico con ϵ -transiciones

En la **figura 1.20** el **AFN** puede moverse del estado q_0 al estado q_1 sin consumir nada en la entrada. En ambos **AFN**, la decisión de elegir una ϵ -transición se realiza de la misma forma que la de cualquier otra transición con elección múltiple que exista en un **AFN**. Según se muestra en la **figura 1.21**.

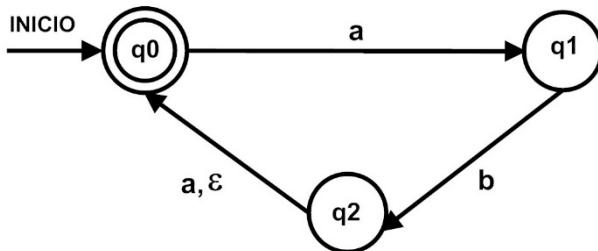


Figura 1.21 Diagrama de un autómata finito no determinístico con ϵ -transiciones.

Este **AFN** asocia $Q \times (\Sigma \cup \{ \epsilon \}) \times Q$ con subconjuntos Q . Por lo que la figura anterior genera la siguiente tabla de transición:

Estados	Entradas			Finales
	0	1		
q0	{ q1 }	{ δ }	{ δ }	0
q1	{ δ }	{ q2 }	{ δ }	1
q2	{ q0 }	{ δ }	{ q0 }	1

Tabla 1.4 de la figura 1.19

1.6 Temas complementarios

Estructura de un compilador. Fases y pasos del compilador

Un compilador es un programa, en el que pueden distinguirse dos subprogramas o fases principales: una fase de análisis, en la cual se lee el programa fuente y se estudia la estructura y el significado del mismo; y otra fase de síntesis, ésta por lo general genera el programa objeto, pero puede generar un código intermedio y un código optimizado.

En un compilador pueden distinguirse, además, algunas estructuras de datos comunes, la más importante de las cuales es la tabla de símbolos, junto con las funciones de gestión de ésta y de los demás elementos del compilador, y de una serie de rutinas auxiliares para detección de errores.

Esquema de un compilador.

Las funciones de estos módulos son las siguientes:

- 1. Analizador lexicográfico:** Las principales funciones que realiza son:
 - Identificar los símbolos.
 - Eliminar los blancos, caracteres de fin de línea, etc...
 - Eliminar los comentarios que acompañan a la fuente.
 - Crear unos símbolos intermedios llamados “**tokens**”.
 - Avisar de los errores que detecte.

Ejemplo: A partir de la sentencia en PASCAL:

nuevo := viejo + RAZON*2

Genera un código simplificado para el análisis sintáctico posterior, por ejemplo:

<id1> <:=> <id2> <+> <id3> <*> <ent>

Nota: Cada elemento encerrado entre <> representa un único token. Las abreviaturas *id* y *ent* significan identificador y entero, respectivamente.

- 2. Analizador sintáctico:** Comprueba que las sentencias que componen el texto fuente son correctas en el lenguaje, creando una representación interna

que corresponde a la sentencia analizada. De esta manera se garantiza que sólo serán procesadas las sentencias que pertenezcan al lenguaje fuente. Durante el análisis sintáctico, así como en las demás etapas, se van mostrando los errores que se encuentran.

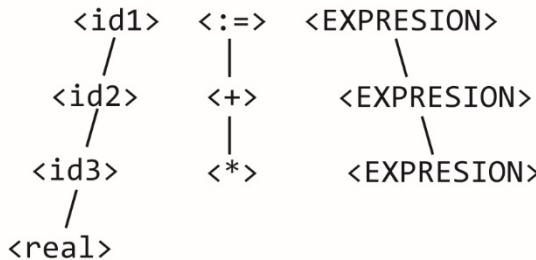
Ejemplo: El esquema de la sentencia anterior corresponde al de una sentencia de asignación del lenguaje Pascal. Estas sentencias son de la forma:

<id> <:=> <EXPRESION>

Y la parte que se denomina <EXPRESION> es de la forma:

- <id> <+> <EXPRESION> o bien
- <id> <*> <EXPRESION> o bien
- <real>

La estructura de la sentencia queda, por tanto, de manifiesto mediante el siguiente esquema:



Análisis semántico: Se ocupa de analizar si la sentencia tiene algún significado. Se pueden encontrar sentencias que son sintácticamente correctas pero que no se pueden ejecutar porque carecen de sentido. En general, el análisis semántico se hace a la par que el análisis sintáctico introduciendo en éste unas rutinas semánticas.

UNIDAD I • MARCO CONCEPTUAL

Ejemplo: En la sentencia que se ha analizado existe una variable entera. Sin embargo, las operaciones se realizan entre identificadores reales, por lo que hay dos alternativas: o emitir un mensaje de error "Discordancia de tipos", o realizar una conversión automática al tipo superior, mediante una función auxiliar inttoreal.

Generador de código intermedio: El código intermedio es un código abstracto independiente de la máquina para la que se generará el código objeto. El código intermedio ha de cumplir dos requisitos importantes: ser fácil de producir a partir del análisis sintáctico, y ser fácil de traducir al lenguaje objeto. Esta fase puede no existir si se genera directamente código máquina, pero suele ser conveniente emplearla.

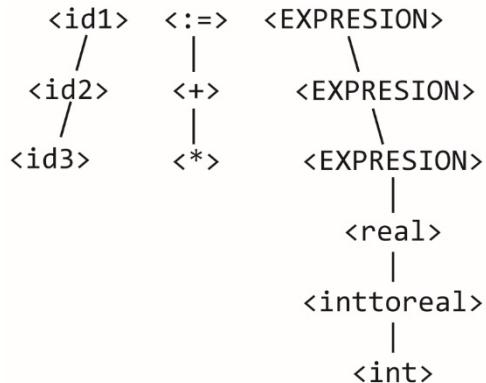
Ejemplo: Consideremos, por ejemplo, un código intermedio de tercetos (tres elementos), llamado así porque en cada una de sus instrucciones aparecen como máximo tres operandos. La sentencia traducida a este código intermedio quedaría:

```
temp1 := inttoreal temp2 := id3 * temp1 temp3 := id2 + temp2 id1 := temp3
```

Optimizador de código: A partir de todo lo anterior, crea un nuevo código más compacto y eficiente, eliminando, por ejemplo, sentencias que no se ejecutan nunca, simplificando expresiones aritméticas. La profundidad con que se realiza esta optimización varía mucho de unos compiladores a otros. En el peor de los casos esta fase se suprime.

Ejemplo: Siguiendo con el ejemplo anterior, es posible evitar la función **inttoreal** mediante el cambio de 2 por 2.0, obviando además una de las operaciones anteriores. El código optimizado queda como sigue: `temp1:= id3 * 2.0id1:= id2 + temp1`

Generador de código: A partir de los análisis anteriores y de las tablas que estos análisis van creando durante su ejecución produce un código o lenguaje objeto que es directamente ejecutable por la máquina. Es la fase final del compilador. Las instrucciones del código intermedio se traducen una a una en código máquina reubicable.



Nota: Cada instrucción de código intermedio puede dar lugar a más de una de código máquina.

Ejemplo: El código anterior traducido a ensamblador DLX quedaría:

LW R1,id3 MUL R1,R1,2 LW R2,id2 ADD R2,R2,R1 SW id1,R2

En donde id1, id2 y id3 representan las posiciones de memoria en las que se hallan almacenadas estas variables; R1 y R2 son los registros de la máquina; y las instrucciones *LW*, *SW*, *MUL* y *ADD* representan las operaciones de colocar un valor de memoria en un registro, colocar un valor de un registro en memoria, multiplicar en punto flotante y sumar, respectivamente.

La tabla de símbolos: Es el medio de almacenamiento de toda la información referente a las variables y objetos en general del programa que se está compilando.

Ejemplo: Hemos visto que en ciertos momentos del proceso de compilación debemos hacer uso de cierta información referente a los identificadores o los números que aparecen en nuestra sentencia, como son su tipo, su posición de almacenamiento en memoria, etc. Esta información es la que se almacena en la tabla de símbolos.

Rutinas de errores: Están incluidas en cada uno de los procesos de compilación (análisis lexicográfico, sintáctico y semántico), y se encargan de informar de los errores que encuentran en texto fuente.

Ejemplo: El analizador semántico podría emitir un error (o al menos un aviso) cuando detectase una diferencia en los tipos de una operación.

ENSAMBLADOR

Un ensamblador es el programa encargado de llevar a cabo un proceso denominado de ensamble o ensamblado. Este proceso consiste en que, a partir de un programa escrito en lenguaje ensamblador, se produzca el correspondiente programa en lenguaje máquina (sin ejecutarlo), realizando:

- La integración de los diversos módulos que conforman al programa.
- La resolución de las direcciones de memoria designadas en el área de datos para el almacenamiento de variables, constantes y estructuras complejas; así como la determinación del tamaño de éstas.

UNIDAD I • MARCO CONCEPTUAL

- La identificación de las direcciones de memoria en la sección de código correspondientes a los puntos de entrada en saltos condicionales e incondicionales junto con los puntos de arranque de las subrutinas.
- La resolución de los diversos llamados a los servicios o rutinas del sistema operativo, código dinámico y bibliotecas de tiempo de ejecución.
- La especificación de la cantidad de memoria destinada para las áreas de datos, código, pila y montículo necesaria y otorgada para su ejecución.
- La incorporación de datos y código necesarios para la carga del programa y su ejecución.

Turbo Assembler.- De Borland Intl., es muy superior al Turbo Editasm. Trabaja de la misma forma, pero proporciona una interfaz mucho más fácil de usar y un mayor conjunto de utilerías y servicios.

Un pre-compilador, también llamado preprocesador, es un programa que se ejecuta antes de invocar al compilador. Este programa es utilizado cuando el programa fuente, escrito en el lenguaje que el compilador es capaz de reconocer (de aquí en adelante denominado lenguaje anfitrión-- en inglés “*host language*”), incluye estructuras, instrucciones o declaraciones escritas en otro lenguaje (el lenguaje empotrado en inglés “*embeded language*”). El lenguaje empotrado es siempre un lenguaje de nivel superior o especializado (por ejemplo de consulta, de cuarta generación, simulación, cálculo numérico o estadístico, etcétera). Siendo que el único lenguaje que el compilador puede trabajar es aquel para el cual ha sido escrito, todas las instrucciones del lenguaje empotrado deben ser traducidas a instrucciones del lenguaje anfitrión para que puedan ser compiladas. Así pues un pre-compilador también es un traductor.

Los pre-compiladores, también conocido como pre-procesadores son una solución rápida y barata a la necesidad de llevar las instrucciones de nuevos paradigmas de programación (ejemplo, los lenguajes de cuarta generación), extensiones a lenguajes ya existentes (como el caso de C y C++) y soluciones de nivel conceptual superior (por ejemplo paquetes de simulación o cálculo numérico) a código máquina utilizando la tecnología existente, probada, optimizada y confiable (lo que evita el desarrollo de nuevos compiladores). Facilitan la incorporación de las nuevas herramientas de desarrollo en sistemas ya elaborados (por ejemplo, la consulta a bases de datos relacionales substituyendo las instrucciones de acceso a archivos por consultas en SQL).

Resulta común encontrar que el flujo de proceso en los lenguajes de cuarta generación o de propósito especial puede resultar demasiado inflexible para su implantación en los procesos de una empresa, flujos de negocio o interacción con otros elementos de software y hardware, de aquí que se recurra o prefiera la creación de sistemas híbridos soportados en programas elaborados en lenguajes de tercera generación con instrucciones empotradas de nivel superior o propósito especial.

Un pseudo-compilador es un programa que actúa como un compilador, salvo que su producto no es ejecutable en ninguna máquina real, sino en una máquina virtual. Un pseudo-compilador toma de entrada un programa escrito en un lenguaje determinado y lo transforma a una codificación especial llamada código de byte. Este código no tendría nada de especial o diferente al código máquina de cualquier microprocesador salvo por el hecho de ser el código máquina de un microprocesador ficticio. Tal procesador no existe, en su lugar existe un programa que emula a dicho procesador, de aquí el nombre de máquina virtual.

La ventaja de los pseudo-compiladores que permite tener tantos emuladores como microprocesadores reales existan, pero sólo se requiere un compilador para producir código que se ejecutará en todos estos emuladores. Este método es una de las respuestas más aceptadas para el problema del tan ansiado lenguaje universal o código portable independiente de plataforma.

Un intérprete es un programa que ejecuta cada una de las instrucciones y declaraciones que encuentra conforme va analizando el programa que le ha sido dado de entrada (sin producir un programa objeto o ejecutable). La ejecución consiste en llamar a rutinas ya escritas en código máquina cuyos resultados u operaciones están asociados de manera única al significado de las instrucciones o declaraciones identificadas.

Los intérpretes son útiles para el desarrollo de prototipos y pequeños programas para labores no previstas. Presentan la facilidad de probar el código casi de manera inmediata, sin tener que recurrir a la declaración previa de secciones de datos o código, y poder hallar errores de programación rápidamente. Resultan inadecuados para el desarrollo de complejos o grandes sistemas de información por ser más lentos en su ejecución. Los programas interpretados suelen ser más lentos que los compilados, pero los intérpretes son más flexibles como entornos de programación y depuración.

UNIDAD I • MARCO CONCEPTUAL

Comparando su actuación con la de un ser humano, un compilador equivale a un traductor profesional que, a partir de un texto, prepara otro independiente traducido a otra lengua, mientras que un intérprete corresponde al intérprete humano, que traduce de viva voz las palabras que oye, sin dejar constancia por escrito.

COMPILADOR

En Resumen, un compilador es un programa que recibe como entrada un programa escrito en un lenguaje de nivel medio o superior (el programa fuente) y lo transforma a su equivalente en lenguaje ensamblador (el programa objeto), e inclusive hasta lenguaje máquina (el programa ejecutable) pero sin ejecutarlo. Un compilador es un traductor. La forma de como llevará a cabo tal traducción es el objetivo central en el diseño de un compilador.

EJERCICIO 9.

Desarrolle un resumen o bien organizador gráfico que describa de forma breve y objetiva las características más significativas de los lenguajes de programación en el cual se encuentren la mayor cantidad de características que se describen en el desarrollo de este tema.

EJERCICIO 10.

1. ¿Qué entiende por Sistemas Formales? (Definición personal de forma coloquial). Describa un ejemplo
 2. ¿Cuál es la relación de la teoría de conjuntos con los sistemas formales? Describa con tres ejemplos.
-

EJERCICIO 11.

Desarrolle un cuadro resumen que muestre las analogías y diferencias entre los tres diferentes tipos de traductores: ensambladores, intérpretes y compiladores.

EJERCICIO 12.

Seleccione diez lenguajes, los más populares como ensambladores, *Basic*, *C*, *Java*, *Pascal*, *Cobol*, *C++*, etc. Describa sus dos niveles, propósito, si son o no formales, aspectos desde el punto de vista del usuario programador y de su diseño.

EJERCICIO 13.

$$r_1 = a$$

$$r_2 = b$$

$$r_1 \ r_2 = ab$$

$$r_3 = a + b$$

$$r_4 = a^*$$

$$r_5 = b^+$$

UNIDAD I • MARCO CONCEPTUAL

UNIDAD II

“Asegúrate de no hacer nada como autómata dormido. Observa cada uno de tus actos, de tus pensamientos y de tus sentimientos. Observa y actúa.” OSHO

AUTÓMATAS FINITOS

“Utiliza autómatas finitos, mediante diferentes métodos de transformación (Thompson, Rabin y Scott entre otros), para el diseño de sistemas secuenciales en empresas de electrónica digital y áreas de informática”.

TEMAS:

- 2.1 Transformación de Expresión Regular a Autómata Finito no Determinista con Épsilon Transiciones.
- 2.2 Transformación de Autómata Finito no Determinista con Épsilon Transiciones a Autómata Finito no Determinista.
- 2.3 Transformación de Autómata Finito no Determinista a Autómata Finito Determinista.
- 2.4 Otros métodos de transformación.

INTRODUCCIÓN:

“Podemos convertir un Autómata Finito Determinista en una expresión regular por medio de la construcción inductiva en la que se crean expresiones para las etiquetas de los caminos que pueden pasar cada vez por conjunto más grande de los estados. Alternativamente, podemos emplear un procedimiento de eliminación de estados para construir la expresión regular equivalente a un autómata finito determinista. En otra dirección podemos construir de forma recursiva un autómata finito no determinista con épsilon transiciones en un autómata finito determinista, si así se desea”. Todo esto ya quedo demostrado “formalmente”, esto es, aplicando lemas, teoremas y sustituciones de forma matemática.

La descripción de la relación descrita en el anterior párrafo, es la que se intenta esquematizar de forma más objetiva en la **figura 2.0**. Esta figura muestra cómo una expresión regular se puede convertir en un autómata finito determinista con épsilon transiciones o sin épsilon transiciones o bien en un autómata finito no determinista y viceversa. Esto es, de un autómata finito no determinista con ϵ transiciones a otro sin épsilon transiciones o bien un autómata finito determinista y viceversa.

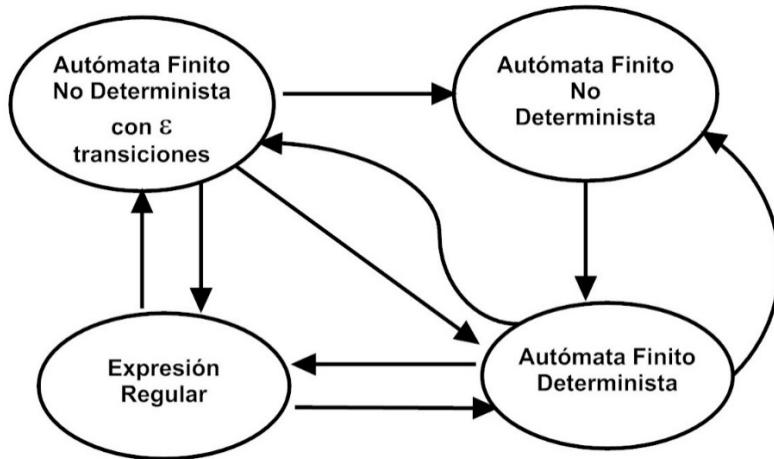


Figura 2.0 Equivalencia entre autómatas finitos y expresiones regulares

2.1 Transformación de expresión regular a autómata finito no determinista con épsilon transiciones.

Recordemos que un autómata es la representación abstracta, gráfica y formal de elementos de una gramática definida como un conjunto de **elementos terminales y no terminales**. Los elementos terminales también pueden ser un solo símbolo, conjunto de símbolos o cadenas de caracteres, simplemente definidos como cadenas. Éstos símbolos o cadenas pueden ser de tipo numérico, alfabético, alfanumérico o caracteres especiales como el asterisco, los símbolos de admiración “¡！”, interrogación “¿？”, etc.

Los **elementos no terminales** los vamos a definir como cadenas, palabras o como se conocen en el mundo de los lenguajes de programación, palabras reservadas que son las instrucciones aceptadas por el lenguaje de programación. También los elementos no terminales son todos aquellos identificadores que nos sirven como nombres de variables, por ejemplo, en la programación tenemos en el lenguaje “BASIC” la sentencia o expresión siguiente: INPUT X

En este caso, de una sola instrucción, según el lenguaje “BASIC”, estamos aceptando o solicitando el valor de la variable X, que debe ser introducido por el usuario. De aquí se desprenden dos elementos no terminales que son la palabra INPUT y la X que tendrán que desglosarse hasta elementos terminales como pueden ser, en el caso de la palabra INPUT sólo una secuencia de cinco caracteres la “l” o “i”, la “N” o “n”, la “P” o “p”, la “U” o “u” y la “T” o “t”, (esto es, mayúsculas o minúsculas), en el mismo orden en el que se sugiere en la instrucción predefinida. Esto significa que solo se acepta la palabra INPUT o input y se rechaza cualquier otra posibilidad como tinpu, nitup, TUPNI y cualquier otra combinación de estas cinco letras. A estas tres últimas palabras, se le marcará como un error de sintaxis o “syntax error”. Cuando se describan los árboles de análisis sintáctico y semántico se retomará este sencillo ejemplo de “BASIC”

Convirtiendo este elemento en una expresión regular etiquetada como “X” tenemos que le asignamos la variable r_x y formalmente el que una expresión regular solo acepte dos palabras que pueden ser INPUT con mayúsculas e input con minúsculas, nos queda el diagrama que se muestra en la **figura 2.1**

UNIDAD II • AUTÓMATAS FINITOS

$rx = /INPUT|input/$

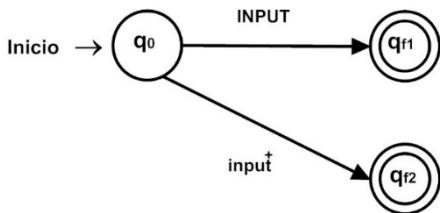


Figura 2.1 Diagrama de transiciones con bien de estados finitos para dos cadenas, INPUT con mayúsculas e input con minúsculas

Este es un diagrama de autómata que procesa los cinco símbolos de este pequeño ejemplo, nos puede quedar un autómata mínimo con solo tres estados, en el cual el nivel de abstracción es por cada una de las dos palabras, al no incluir el símbolo de épsilon ϵ , se denomina **autómata finito no determinista sin épsilon transiciones**. Pero también se puede abstraer a nivel símbolos la representación de estas dos cadenas, en donde cada uno de los cinco símbolos que integran cada cadena formara el pequeño lenguaje de únicamente dos palabras y cada una de ellas se puede desglosar dentro de las transiciones hasta llegar al último símbolo que llega al estado final, según se muestra en la **figura 2.2**. El diagrama que se muestra en la figura 2.2 representa la posibilidad de que el comportamiento de la máquina “M”, puede quedar definido de la siguiente forma: $M = \{Q, \Sigma, \delta, S, F\}$

Dónde: $Q = \{q_0, q_1, q_2, q_3, q_4, q_5\}$

El alfabeto, que es el conjunto de símbolos que se utilizan para el lenguaje (de solo dos palabras) es: $\Sigma = \{i, n, p, u, t, I, N, P, U, T\}$

Las transiciones son: $\delta = Q \times \Sigma$

$Q = \text{estados}$	$\Sigma = I i$	$\Sigma = N n$	$\Sigma = P p$	$\Sigma = U u$	$\Sigma = T t$	F
q_0	q_1	\emptyset	\emptyset	\emptyset	\emptyset	0
q_1	\emptyset	q_2	\emptyset	\emptyset	\emptyset	0
q_3	\emptyset	\emptyset	q_3	\emptyset	\emptyset	0
q_4	\emptyset	\emptyset	\emptyset	q_4	\emptyset	0
q_5	\emptyset	\emptyset	\emptyset	\emptyset	q_5	0

$$S = \{q_0\}$$

$$F = \{q_5\}$$

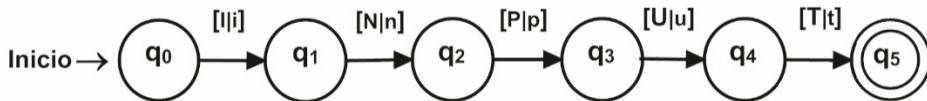


Figura 2.2. Diagrama de estados finitos para las palabras INPUT o input o InPuT, iNpUt, etc.

Cuando no existe un símbolo que nos lleva de una transición a otra, se coloca ese símbolo, de lo contrario, se le coloca en la tabla de transiciones el símbolo vacío, \emptyset

Dentro de la instrucción INPUT X (con mayúsculas) e input X (con minúsculas). El otro elemento no terminal que es la X, tiene una definición más completa, pues se le pueden asignar valores numéricos, alfabéticos, alfanuméricos y hasta caracteres especiales. Esta definición estricta crea la necesidad de hacer una descripción formal, misma que se verá más adelante con las expresiones regulares. Estas son las representaciones más amigables de un conjunto de caracteres agrupados en cadenas, mediante la identificación de su inicio y fin con el carácter especial “/”, esta es la forma de representación “formal” de una expresión regular y su orden o patrón en el que se expresan, es mediante la agrupación de estos símbolos, mismos que si son elementos terminales pueden ser palabras o cadenas.

Suponiendo que la definición de la variable X sea sólo para aceptar símbolos alfabéticos, tanto en letras mayúsculas como minúsculas, la expresión formal queda:

$$X = \{(A, B, C, \dots, Z | a, b, c, \dots, z)^*\}$$

Considerando que aquí los nombres de variables pueden ser escritos con mayúsculas o minúsculas y se acepta cualquier combinación de las letras del alfabeto castellano desde la letra A hasta la z con todas sus posibles combinaciones y queda según la **figura 2.3**.

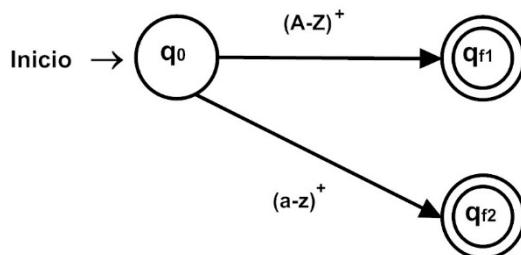


Figura 2.3 Cerradura positiva +

UNIDAD II • AUTÓMATAS FINITOS

Aquí el símbolo de “+” juega un papel muy importante, significa que puede haber un nombre de variable usando mayúsculas o minúsculas. Cuando además de aceptar símbolos también se puede aceptar el carácter o símbolo épsilon que representa a la cadena vacía ϵ . Esto es, que existan nombres de variables con todas las combinaciones de caracteres, tanto mayúsculas y minúsculas o que no existan, también se interpreta como que se aceptan un número infinito de caracteres y esto es la definición de un lenguaje infinito, formalmente se expresa.

$X = \{(A, B, C, \dots, Z | a, b, c, \dots, z)^*\}$ Siendo esta la expresión regular equivalente a:
 $r_x = [/INPUT|input/ \epsilon X]$

Y el desglose de los cinco elementos para una máquina de estado finito o autómata queda:

$M = \{Q, \Sigma, \delta, S, F\}$ donde:

$Q = \{q_0, q_1, q_2, q_3, q_4, q_5\}$

El alfabeto, que es el conjunto de símbolos que se utilizan para el lenguaje (de solo dos palabras) es:

$\Sigma = \{i, n, p, u, t, I, N, P, U, T\}$

Las transiciones son: $\delta = Q \times \Sigma$

$Q = \text{estados}$	$\Sigma = I i$	$\Sigma = N n$	$\Sigma = P p$	$\Sigma = U u$	$\Sigma = T t$	F
q_0	q_1	\emptyset	\emptyset	\emptyset	\emptyset	0
q_1	\emptyset	q_2	\emptyset	\emptyset	\emptyset	0
q_3	\emptyset	\emptyset	q_3	\emptyset	\emptyset	0
q_4	\emptyset	\emptyset	\emptyset	q_4	\emptyset	0
q_5	\emptyset	\emptyset	\emptyset	\emptyset	q_5	0

$$S = \{q_0\}$$

$$F = \{q_5\}$$

Convertiendo el anterior autómata a uno que sea no determinístico con ϵ transiciones puede ser de la forma que se muestra en la **figura 2.4**

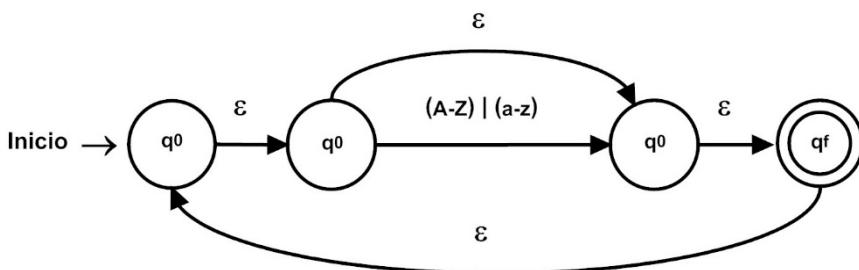


Figura 2.4 Autómata finito para procesar cadenas de caracteres con mayúsculas y minúsculas

Por ejemplo, expresiones regulares que quedan definidas con un solo símbolo es la letra “a” minúscula y el símbolo “/” (slash) muestran que es una cadena o expresión regular de un solo símbolo, la letra “a”. Si se toma como un solo símbolo, su representación queda de la siguiente forma, pero si se trata de un símbolo como la X de la instrucción en “BASIC” el cuál más allá de una letra, carácter o símbolo, se trata un elemento variable o **no terminal**. Esta se describirá formalmente y con más detalle en la siguiente unidad. $r_a = /a/$

Podemos expresar una cadena de cinco símbolos ordenados, unidos o concatenados uno después de otro e integrar la palabra o cadena “banco” y si lo representamos de la siguiente forma, esta también puede ser el ejemplo de una expresión regular: $r_b = /banco/$

Aquí la expresión regular tiene una longitud de cinco símbolos, los cuales son, el primero “b”, seguido por el segundo símbolo, la “a”, el tercer símbolo la “n”, el cuarto, “c” y el quinto y último símbolo, la letra “o”.

Las expresiones regulares definen el comportamiento, orden o secuencia lógica de un conjunto de elementos conocidos como caracteres, y en el ejemplo anterior, letras que integradas, unidas o concatenadas forman cadenas. Las cadenas al ser presentadas de una forma particular, pueden ser consideradas expresiones regulares simples que a su vez pueden indicarse formalmente con las operaciones de unión o concatenación (según lo descrito en el tema de teoría de conjuntos, capítulo anterior). Se pueden representar por disyunción, y para el caso de la palabra banco, si deseamos

UNIDAD II • AUTÓMATAS FINITOS

definir que el último símbolo la “o” puede ser “a” entonces podemos definir como expresión regular la siguiente posibilidad: $r_x = /banco|banca/$

El carácter “|” (pipe) o pequeña línea vertical, nos indica dos posibilidades, la palabra banco o banca. Lamentablemente no hay una nomenclatura estándar para representar la disyunción, alternancia o dos posibilidades de cadenas o símbolos, por lo que la expresión anterior, también se puede expresar de la siguiente forma: $r_x = /banco+banca/$

Esto significa las dos posibilidades de expresiones regulares, r_x lo mismo puede ser la palabra “**banco**” o bien la palabra “**banca**”. La misma expresión regular, puede ser definida en un diagrama conocido como **autómata**. Mismo que define el comportamiento de dos cadenas que pueden ser como se muestran en la **figura 2.5**

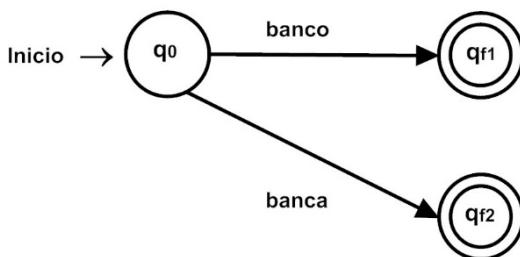


Figura 2.5.- Autómata simplificado para dos palabras

Este diagrama que mostramos con tres círculos, también pueden representarse en varios estados, con la transición por cada uno de los símbolos, el primero etiquetado con el estado inicial q_0 y al menos dos estados finales etiquetados como, q_{f1} y q_{f2} ambos con doble círculo, los cuales, en la expresión regular están representados con dos cadenas de cinco símbolos cada uno, esto es, una para la palabra banco y otra para la cadena con también cinco símbolos definida como banca.

Las dos cadenas muestran un ejemplo de lo que puede ser dos elementos terminales en una expresión regular y éstos son terminales porque cada uno de ellos nos lleva a un estado final que define el comportamiento de un sistema para dos cadenas predefinidas con cinco símbolos alfabéticos en un único orden. Esto visto desde un sistema formal que también podemos definir como ejemplo de un pequeño lenguaje que solo se integra por dos únicas palabras (banco y banca). Se puede describir como una gramática regular. En la cual tenemos, los siguientes elementos:

Un conjunto de estados (Q) los cuales para este sencillo ejemplo ésta definido por un estado inicial, etiquetado como q_0 y dos estados finales, etiquetados como q_{f1} y q_{f2}

Formalmente queda $\mathbf{M} = \{Q, \Sigma, \delta, S, F\}$

Dónde: $Q = \{q_0, q_{f1}, q_{f2}\}$

El alfabeto, que es el conjunto de símbolos que se utilizan para el lenguaje y estos son

$\Sigma = \{a, b, c, n, o\}$

Las transiciones son: $\delta = Q \times \Sigma$

$Q = \text{estados}$	$\Sigma = \text{banco}$	$\Sigma = \text{banca}$	F
q_0	q_{f1}	q_{f2}	0
q_{f1}	\emptyset	\emptyset	1
q_{f2}	\emptyset	\emptyset	1

$S = \{q_0\}$

$F = \{q_{f1}, q_{f2}\}$

Ésta descripción muestra que un lenguaje es un conjunto de al menos cinco elementos, que se pueden describir con una expresión regular o con un autómata finito que en este caso es: *Si utilizamos el sistema numérico binario. En un Sistema numérico binario solo intervienen dos dígitos, el cero “0” y el uno “1”. Con dos dígitos y la posibilidad de repetirlos de forma finita o infinita vamos a realizar los primeros ejemplos de cómo convertir una expresión “regular” en un autómata.*

Ejemplo: Supongamos una expresión regular que contiene un solo dígito, el cero.

A la expresión regular la identificaremos con el carácter r y para definir que es la primera, la identificamos con el sub-índice uno. Tenemos r_1 recordemos que, debe estar determinada por dos elementos, el lado izquierdo que representa la expresión y pude forma muy simplificada con la equivalencia o signo de igual $=$ para termina n con el digito cero. Tenemos: $r_1 = 0$ su representación en un autómata que es un conjunto de al menos cinco elementos.

UNIDAD II • AUTÓMATAS FINITOS

- 1) Un conjunto de Estados, representado con la letra Q (en mayúscula),
- 2) Un estado inicial o “Start”, simbolizado con la letra S, en este ejemplo empezaremos a enumerar los estados con la letra q y para identificar su identificación particular les daremos un sub-índice empezando con el cero, uno, dos y así sucesivamente.
- 3) Un sub-conjunto de estados finales, simbolizado con la letra F, mismo que para este ejemplo será un solo estado, por la simplicidad de la expresión regular
- 4) Un conjunto de transiciones que es la relación entre las entradas y los estados “Q” esto es las transiciones, que son un eje cartesiano entre el alfabeto simbolizado como Σ y el conjunto de estados Q. esto se representa con δ y son igual a $\Sigma \times Q$
- 5) Por último, pero no menos importante, daremos el alfabeto, que son los símbolos que se utilizan para cambiar de un estado a otro y se anotan en las transiciones o uniones mediante un arco o línea dirigida, tal como se ve en la **figura 2.6**

La representación de los elementos formales (agregados al gráfico arriba dibujado con dos líneas, una que marca el inicio y otra que marca el paso de un estado q_0 a q_1) $r_1 = 0$ donde descrito de la más simple expresión regular como: $r_1 = 0$ con los elementos de un autómata queda de la siguiente forma:

$$Q = \{q_0, q_1\}$$

$$S = q_0$$

$$F = q_1$$

δ se representa mejor como un eje cartesiano donde $\Sigma = \{0,1\}$, esto es el alfabeto que para un sistema binario son sólo dos símbolos el cero y el uno como únicas entradas

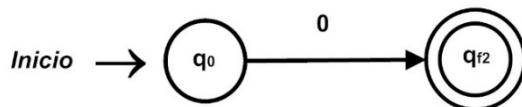


Figura 2.6 Autómata para la expresión regular $r_1 = 0$

Estados	Símbolos de Entrada=	0	1	Estados Finales
q_0	q_1	\emptyset	\emptyset	0
q_1	\emptyset	\emptyset	\emptyset	1

$$S = \{q_0\} \quad Y \quad F = \{q_1\}$$

2.2 Transformación de Autómata Finito no Determinista con Épsilon Transiciones a Autómata Finito no Determinista.

Las transformaciones tienen una base matemática comprobada en relación a que una expresión regular tiene una representación formal y ésta a su vez, tiene una representación gráfica conocida como autómata. Dentro de los autómatas pueden ser determinísticos y no determinísticos, como ya se explicó en el capítulo anterior. Las épsilon transiciones pueden ser omitidas o definidas según la conveniencia para expresar el lenguaje. Retomando el ejemplo de la expresión regular con un solo símbolo tenemos de nuevo el autómata sin épsilon transiciones que también se puede considerar un autómata mínimo y es la **figura 2.6**

De este, simplemente le podemos agregar cuatro símbolos épsilon y lo convertimos en una expresión regular de un solo símbolo misma que nos genera un número infinito de cadenas, las cuales, al ser representadas mediante una expresión regular formar un lenguaje infinito que es lo que significa el exponente ya sea con el signo de + “cerradura positiva” o con el signo de épsilon, que se incluye en la cerradura de “Kleene”, representado con el exponente de asterisco *

$r_1 = 0$

Lenguaje regular de un solo símbolo $L = \{0\}$ y $r_1 = 0^*$

Lenguaje infinito donde $L = \{ \epsilon, 0, 00, 000, 0000, \dots, \infty \}$

Su autómata puede ser de la forma que se muestra en la **figura 2.7**

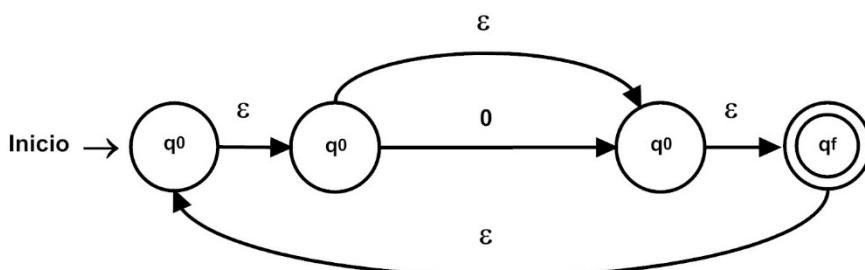


Figura 2.7 Autómata de la expresión regular $r_1 = 0^*$

UNIDAD II • AUTÓMATAS FINITOS

Aquí tenemos un autómata finito determinista con épsilon transiciones y para lograr la transformación a un autómata finito no determinista, hace falta al menos otro símbolo. Ya que recordemos que para que esto se cumpla, debe de haber por cada símbolo del alfabeto una transición.

Ejemplo: Si tenemos a y b como símbolos del alfabeto, entonces nuestro autómata no determinista puede quedar según la muestra la **figura 2.8**. En este autómata se observa que del estado q_0 a q_1 solo se tiene la transición con el símbolo “a” y falta una transición para el símbolo “b”. Del estado q_1 es q_1 a q_f también solo se tiene una sola transición que es con la letra “b” y falta la “a”, por estos dos motivos se trata de un autómata finito no determinista.

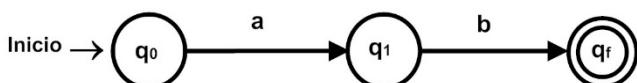


Figura 2.8 Autómata de la expresión regular $r=ab$

El autómata de la **figura 2.9** es sólo un ejemplo de como se ve un autómata finito determinista de uno no determinista, para el mismo lenguaje, esto es los mismos símbolos a y b se tienen en el diagrama. En este autómata, para cada estado se tiene dos transiciones, una con la letra “a” y otra con la letra “b”, esto sólo es un ejemplo de un autómata determinista, sin considerar otras características que se comentarán más adelante.

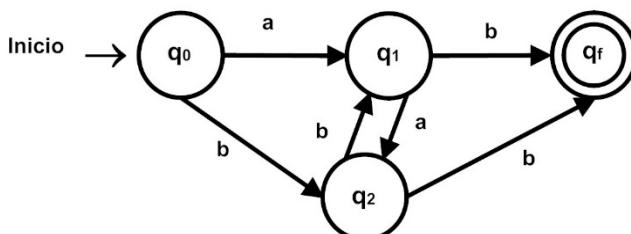


Figura 2.9 Autómata finito determinista para los símbolos a y b

Existe una situación importante entre dos autómatas y esto es el concepto de que cuando dos autómatas aceptan el mismo lenguaje y uno de ellos es determinista y el otro es no determinista, o bien, uno tiene a épsilon como símbolo aceptado y se conoce como “autómata finito con épsilon transiciones” y otro que no incluye a épsilon o cadena vacía, y a este autómata se le

conoce como “autómata finito (que puede ser determinista o no determinista) sin épsilon transiciones”

También se le denomina simplemente autómata finito determinista o no deterministas, agregándosele la característica de épsilon transiciones, cuando incluye la cadena vacía y omitiendo esta característica cuando no incluye a épsilon, sólo podemos describir autómata finito “determinista o no determinista”. Tal como se especifica en el título de este subtema.

Cuando dos autómatas son diferentes, pero aceptan el mismo lenguaje, a esto se le conoce como autómatas equivalentes. Entre dos autómatas equivalentes, también uno puede ser mínimo, esto es que tiene el menor número de estados y el otro puede tener mayor número de estados. Una vez que se trató de dejar claros estos conceptos, procedemos a mostrar un ejemplo clásico y sencillo de una conversión de un autómata finito no determinista con épsilon transiciones a un autómata finito también no determinista con épsilon transiciones de una forma “formal” la cual ésta comprobada su efectividad.

A continuación, se muestra en la **figura 2.10** el diagrama de un autómata finito no determinista con épsilon transiciones, mismo que se convertirá en un autómata finito también no determinista pero ahora sin las “épsilon” transiciones.

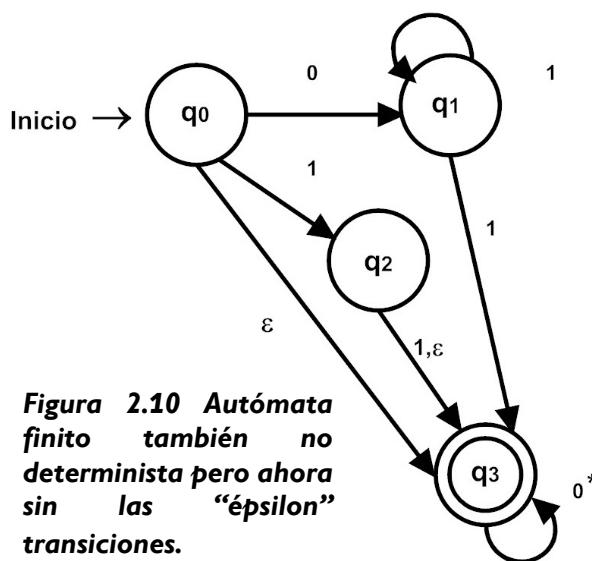


Figura 2.10 Autómata finito también no determinista pero ahora sin las “épsilon” transiciones.

Para lograr la conversión, es necesario explicar el concepto de la épsilon cerradura.

¿Qué es la épsilon cerradura? Son el conjunto de todos aquellos estados en los cuales se accede a ellos mediante la cadena vacía o épsilon cerradura. La épsilon cerradura de un estado, es al menos el estado en cuestión, por ejemplo:

UNIDAD II • AUTÓMATAS FINITOS

La épsilon cerradura de q_0 es q_0 . de una forma “formal” se especifica:

$$\epsilon\text{-c}(q_0, \epsilon) = \delta^*(q_0, \epsilon) = \{q_0\}$$

Aclaramos que, para este material, los símbolos $\epsilon\text{-c}$ se leen como la **épsilon cerradura**

El símbolo δ se lee como la transición y δ^* es la transición modificada de la δ transición.
La descripción formal del autómata es:

$$M = \{Q, \Sigma, \delta, S, F\}$$

donde:

$$Q = \{q_0, q_1, q_2, q_3\}$$

El alfabeto, que es el conjunto de símbolos que se utilizan para el lenguaje (de solo dos palabras de longitud uno) es: $\Sigma = \{\epsilon, 0, 1\}$

Las transiciones son: $\delta = Q \times \Sigma$

$Q = \text{estados}$	ϵ	0	1	F
q_0	q_0	q_1	q_2	0
q_1	q_1	\emptyset	$\{q_1, q_3\}$	0
q_2	q_2	\emptyset	q_3	1
q_3	q_3	q_3	q_3	0

$$S = \{q_0\}$$

$$F = \{q_3\}$$

El lenguaje aceptado para este autómata, son todas y cada una de las cadenas que llegan al estado final de q_3 y este lenguaje a su vez puede incluir una expresión regular o varias de éstas. Consideradas las expresiones regulares como metalenguajes, esto es, el lenguaje de los lenguajes y queda:

$$L = \{0^*, 11, 01^*1, 110^*\}$$

A continuación, vamos a obtener la épsilon cerradura de los estados que se definen en el autómata de la página anterior. Empezamos con la épsilon cerradura de todos y cada uno de los cuatro estados del autómata desde q_0, q_1, q_2, q_3 ... primero con épsilon y luego con cada uno de los símbolos del alfabeto. Que para este autómata el alfabeto es:

$\Sigma = \{ \epsilon, 0, 1 \}$ (según descripción “formal” anterior) Σ

$$\epsilon\text{-c}(q_0, \epsilon) = \delta^*(q_0, \epsilon) = \{q_0, q_3\}$$

$$\epsilon\text{-c}(q_1, \epsilon) = \delta^*(q_1, \epsilon) = \{q_1\}$$

$$\epsilon\text{-c}(q_2, \epsilon) = \delta^*(q_2, \epsilon) = \{q_2, q_3\}$$

$$\epsilon\text{-c}(q_3, \epsilon) = \delta^*(q_3, \epsilon) = \{q_3\}$$

$$\delta^*(q_0, 0) = \epsilon\text{-c}(\delta(\delta^*(q_0, \epsilon), 0)) = \epsilon\text{-c}(\delta(\{q_0, q_3\}, 0)) = \epsilon\text{-c}(\{q_1\} \cup \{q_3\}) = \epsilon\text{-c}(\{q_1, q_3\}) = \{q_1, q_3\}$$

$$\delta^*(q_1, 0) = \epsilon\text{-c}(\delta(\delta^*(q_1, \epsilon), 0)) = \epsilon\text{-c}(\delta(\{q_1\}, 0)) = \{\emptyset\}$$

$$\delta^*(q_2, 0) = \epsilon\text{-c}(\delta(\delta^*(q_2, \epsilon), 0)) = \epsilon\text{-c}(\delta(\{q_2, q_3\}, 0)) = \epsilon\text{-c}(\{q_3\}) = \{q_3\}$$

$$\delta^*(q_3, 0) = \epsilon\text{-c}(\delta(\delta^*(q_3, \epsilon), 0)) = \epsilon\text{-c}(\delta(\{q_2\}, 0)) = \epsilon\text{-c}(\{q_3\}) = \{q_3\}$$

$$\delta^*(q_0, 1) = \epsilon\text{-c}(\delta(\delta^*(q_0, \epsilon), 1)) = \epsilon\text{-c}(\delta(\{q_0, q_3\}, 1)) = \epsilon\text{-c}(\{q_2\}) = \{q_2, q_3\}$$

$$\delta^*(q_1, 1) = \epsilon\text{-c}(\delta(\delta^*(q_1, \epsilon), 1)) = \epsilon\text{-c}(\delta(\{q_1\}, 1)) = \epsilon\text{-c}(\{q_1, q_3\}) = \{q_1, q_3\}$$

$$\delta^*(q_2, 1) = \epsilon\text{-c}(\delta(\delta^*(q_2, \epsilon), 1)) = \epsilon\text{-c}(\delta(\{q_2, q_3\}, 1)) = \epsilon\text{-c}(\{q_3\} \cup \{\emptyset\}) = \epsilon\text{-c}(\{q_3\}) = \{q_3\}$$

$$\delta^*(q_3, 1) = \epsilon\text{-c}(\delta(\delta^*(q_3, \epsilon), 1)) = \epsilon\text{-c}(\delta(\{q_3\}, 1)) = \epsilon\text{-c}(\{\emptyset\}) = \{\emptyset\}$$

Conviene comprobar en el modelo gráfico, si este autómata que se forma de las transiciones anteriores es ya un autómata finito no determinístico, pero sin la cadena vacía o épsilon dentro de sus transiciones, por lo que es evidente que ya no está, y también gráficamente podemos verificar si el lenguaje aceptado es el mismo que en el autómata anterior para validar que

ambos son autómatas equivalentes.

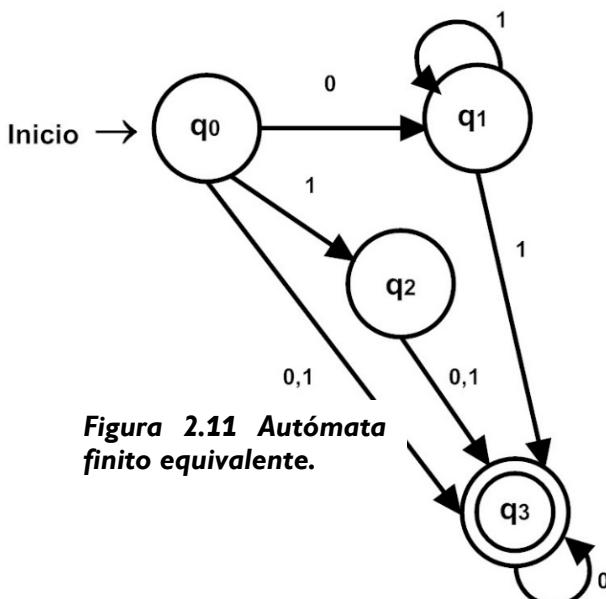


Figura 2.11 Autómata finito equivalente.

$$L = \{0^*, 11, 01^*1, 110^*\}$$

Con todo lo anterior, obtenemos un nuevo autómata y su descripción es la que se muestra en la **figura 2.11**

UNIDAD II • AUTÓMATAS FINITOS

La descripción formal del autómata es:

$$M = \{Q, \Sigma, \delta, S, F\}$$

Donde:

$$Q = \{q_0, q_1, q_2, q_3\}$$

El alfabeto, que es el conjunto de símbolos que se utilizan para el lenguaje (de solo dos palabras de longitud uno) es: $\Sigma = \{0, 1\}$ (este ya no incluye a la cadena vacía o épsilon)

Las transiciones son: $\delta = Q \times \Sigma$

$Q = \text{estados}$	0	1	F
q_0	$\{q_1, q_3\}$	$\{q_2, q_3\}$	1
q_1	$\{\emptyset\}$	$\{q_1, q_3\}$	0
q_2	$\{q_3\}$	$\{q_3\}$	0
q_3	$\{q_3\}$	$\{\emptyset\}$	1

$$S = \{q_0\}$$

$$F = \{q_3\}$$

2.3 Transformación de Autómata Finito no Determinista a Autómata Finito Determinista.

Como ya se explicó anteriormente, Estas transformaciones nos permiten llegar a un autómata mínimo cuyo procesamiento resulta más objetivo y práctico, y recordemos que un autómata finito no determinista, no incluye una transición para cada uno de los símbolos del alfabeto, sin embargo, un ejemplo sencillo de presentar es cuando se tiene como únicos símbolos del alfabeto al 0 y al 1.

Es por esto la razón de que las computadoras se hayan diseñado con sistemas binarios en su lenguaje de máquina.

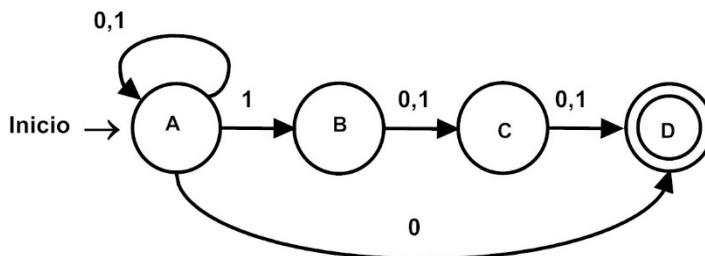


Figura 2.12 Autómata finito no determinista AFN

Partimos del autómata finito no determinista de la **figura 2.12**. Añadimos un estado de error o “X” puede ser etiquetado con cualquier nombre según se muestra en la **figura 2.13**

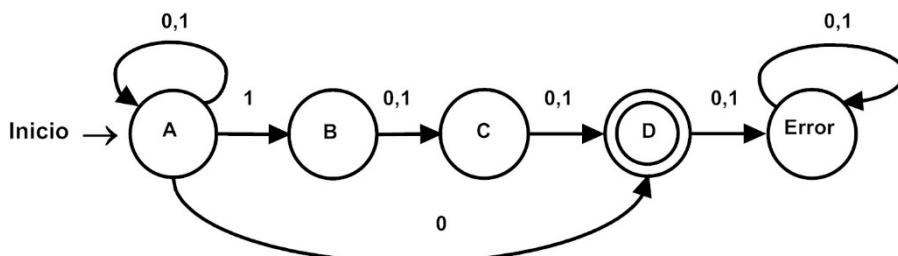


Figura 2.13 Autómata finito no determinista AFN

UNIDAD II • AUTÓMATAS FINITOS

Generamos una tabla considerando la transición de error y agrupando nuevas transiciones, agregando las nuevas transiciones y nos queda la siguiente tabla

Q = estados	0	1	F
{A}	{A}	{A,B}	0
{A,B}	{A,C,D}	{A,B,C}	0
{A,C,D}	{A ,D, Error}	{A, B, D, Error}	1
{A,B,C}	{A,C,D}	{A,B,C,D}	0
{A, D, Error}	{A, Error}	{A, B, Error}	1
{A,B,C,D}	{A,C,D, Error}	{A, B, C, D, Error}	1
{A, Error}	{A, Error}	{A, Error}	0
{A, B, Error}	{A,C,D, Error}	{A, B, C, Error}	0
{A,C,D, Error}	{A, D, Error}	{A, B, D, Error}	1
{A, B, C, Error}	{A,C,D, Error}	{A, B, C, D, Error}	0
{A, B, C, D, Error}	{A,C,D, Error}	{A, B, C, D, Error}	1

Con esta tabla se construye el autómata finito determinista ya sin épsilon transiciones y dado que acepta el mismo lenguaje es un autómata equivalente. Se conserva el estado D como estado final, considerándose como final el grupo de estados que contiene a D y también el nuevo estado de error es final.

2.4 Otros Métodos De Transformación

EQUIVALENCIA ENTRE GRAMÁTICAS

TEOREMA DE MYHILL- NERODE Y MINIMIZACIÓN DE AUTÓMATAS FINITOS

Este teorema nos permite asociar con un lenguaje cualquiera, el cual lo podemos identificar con la letra “L” a una relación de equivalencia natural “RL”; es decir, “ $x R$ ” y si y sólo si para cada z , xz y yz están en “L” o no lo están. En el peor de los casos cada cadena está, por sí misma, en una clase de equivalencia, pero puede haber menos clases. En particular, el índice (número de clases de equivalencia) siempre es finito si “L” es un conjunto regular (o expresión regular).

Existe también una relación de equivalencia natural sobre las cadenas asociadas con un autómata finito.

Sea $M = (Q, \Sigma, \delta, q_0, F)$ un DFA (Autómata Finito Determinista, del inglés “Deterministic Finite Automata”).

Para “ x ” y “ y ” en Σ^* sea $x R_M y$ si y sólo si $\delta(q_0, x) = \delta(q_0, y)$. La relación R_M es reflexiva, simétrica y transitiva, puesto que el símbolo “=” tiene tales propiedades, y por tanto R_M es una relación de equivalencia. R_M divide el conjunto Σ^* en clases de equivalencia, una por cada estado que se puede alcanzar desde q_0 .

Además, si $x R_M y$, entonces $xz R_M yz$ para toda z en Σ^* , ya que:

$$\delta(q_0, xz) = \delta(\delta(q_0, x), z) = \delta(\delta(q_0, y), z) = \delta(q_0, yz).$$

Una relación de equivalencia R tal que xRy implique $xzRyz$ se dice que es invariante por la derecha (con respecto a la concatenación). Vemos que cada autómata finito induce una relación de equivalencia invariante por la derecha, definida como se definió R_M sobre las cadenas de entrada. Este resultado se formaliza en el teorema de Myhill-Nerode. Las tres proposiciones siguientes son equivalentes:

El conjunto $L \subseteq \Sigma^*$ es aceptado por algún autómata finito.

L es la unión de algunas clases de equivalencia de una relación de equivalencia invariante por la derecha de índice finito.

UNIDAD II • AUTÓMATAS FINITOS

Sea R_L la relación de equivalencia definida por: $x R_L y$ si y sólo si para toda z en Σ^* , xz está en L exactamente, cuando yz está en L . Entonces R_L es de índice finito.

- x y y tienen un 1 cada una, 0
- x y y tienen cada una más de un 1.

Por ejemplo, si $x = 010$ y $y = 1000$, entonces xz está en L si y sólo si z está en 0^* .

Pero yz está en L bajo exactamente las mismas condiciones.

Como otro ejemplo, si $x = 01$ y $y = 00$, entonces podemos escoger $z = \emptyset$ para mostrar que xR_Ly es falso.

Esto es, $xz = 010$ está en L , pero $yz = 000$ no lo está.

Podemos representar las tres clases de equivalencia de R_L con

$C_0 = 0^*$, $C_1 = 0 * 10$ y $C_2 = 0 * 10 * 1 (0 + 1)^*$, L es el lenguaje que consiste en sólo una de estas clases, C_2 .

La relación es de C_a, \dots, C_f con C_1, C_2 y C_3 .

Por ejemplo, $C_a \cup C_b = (00)^* + (00)^* 0 = 0^* = C_0$.

A partir de R_L podemos construir un DFA de la manera siguiente. Tómense representantes de C_1, C_2 y C_3 , digamos $\epsilon, 1$ y 11 . Entonces sea M' el DFA.

Por ejemplo, $\delta'([1], 0) = [1]$, ya que si w es cualquier cadena de $[1]$

(nótese que $[1]$ es C_1), digamos $0^i 10^j$, entonces w 0 es $0^i 10^j \in L$, que también está en $C = 0 * 10^*$.

Minimización de autómatas finitos

El teorema de Myhill-Nerode tiene, entre otras consecuencias, la implicación de que existe un DFA de estado mínimo esencialmente único para cada conjunto regular.

Teorema. El autómata de estado mínimo que acepta a un conjunto L es único hasta un isomorfismo (es decir, un renombramiento de estados) y está dado como M' en la demostración del Teorema.

Demostración. En la demostración del Teorema vimos que cualquier DFA $M = (Q, \Sigma, \delta, q_0, F)$ que acepta a L define una relación de equivalencia que es un refinamiento de R_L . Por consiguiente, el número de estados de M es mayor o igual que el número de estados de M del Teorema. Si la igualdad es válida, entonces cada uno de los estados de M puede ser identificado con uno de los estados de M !. Esto es, sea q un estado de M . Debe existir alguna x en Σ^* , tal que $\delta(q_0, x) = q$, de otra manera q puede ser eliminada de Q y hallar, así, un autómata más pequeño. Identifíquese a q con el estado $\delta(q_0, X)$ de M !. Esta identificación será consistente. Si $\delta(q_0, z) = \delta(q_0, y) = q$, entonces, según la demostración del teorema, x y y están en la misma clase de equivalencia de R_L . En consecuencia, $\delta'(q'_0, x) = \delta'(q'_0, y)$.

UN ALGORITMO DE MINIMIZACIÓN

Existe un método simple para encontrar el DFA de estado mínimo M de los Teoremas anteriores, equivalente a un DFA dado $M = (Q, \Sigma, \delta, q_0, F)$. Sea el símbolo $=$ la relación de equivalencia sobre los estados de M , tal que $p = q$ si y sólo si para cada cadena de entrada x , $\delta(p, x)$ es un estado de aceptación si y sólo si $\delta(q, x)$ es un estado de aceptación. Obsérvese que existe un isomorfismo entre aquellas clases de equivalencia de $=$ que contienen un estado que se puede alcanzar desde q_0 mediante alguna cadena de entrada y los estados del FA de estado mínimo M !. Así los estados de M pueden identificarse con estas clases.

Más que dar un algoritmo formal para encontrar las clases de equivalencia $=$ de M primero veremos un ejemplo. En primer lugar, se necesita cierta terminología. Si $p = q$, decimos que p es equivalente a q . Decimos que p es distingüible de q si existe una x tal que $\delta(p, x)$ esté en F y $\delta(q, x)$ no, o viceversa.

Ejemplo. Sea M el autómata finito hemos construido una tabla con una entrada por cada par de estados. Ponemos una X en la tabla cada vez que descubrimos un par de estados que no pueden ser equivalentes. Inicialmente se sitúa una X en cada entrada que corresponda a un estado final y uno no final. En nuestro ejemplo, ponemos una X en las entradas $(a, c), (b, c), (c, d), (c, e), (c, f), (c, g)$ y (c, h) .

En seguida, para cada par de estados p y q que aún no se sabe si son distingüibles, consideramos los pares de estados $r = \delta(p, a)$ y $s = \delta(q, a)$ para cada símbolo de entrada a . Si se ha mostrado que los estados r y s son distingüibles mediante

UNIDAD II • AUTÓMATAS FINITOS

alguna cadena x , entonces p y q son distinguibles mediante la cadena a x . Por consiguiente, si la entrada (r, s) de la tabla tiene una x , también se debe poner una en la entrada (p, q) . Si la entrada (r, s) aún no tiene una x , entonces el par (p, q) se coloca en una lista que está asociada con la entrada (r, s) , si en lo futuro, la entrada (r, s) recibe una x , entonces cada par de la lista asociada con la entrada (r, s) también deberá tener una x .

Continuando con el ejemplo, colocamos una x en la entrada (a, b) , puesto que la entrada $(\delta(b, 1), \delta(a, 1)) = (c, f)$ ya tiene una x . De forma similar, la entrada (a, d) recibe una x ya que la entrada $\delta(a, 0), \delta(d, 0) = (b, c)$ tiene una. Tomando en consideración la entrada (a, e) sobre el símbolo de entrada 0 trae como resultado que el par (a, e) sea colocado en la lista de los pares asociados con (b, h) . Obsérvese que sobre la entrada 1 , tanto a como e van al mismo estado y en consecuencia ninguna cadena que comience con un 1 puede distinguir a de e . Debido a la entrada 0 , el par (a, g) se coloca en la lista asociada con (b, g) . Cuando se considera la entrada (b, g) , ésta recibe una x debido a una entrada 1 , y por tanto el par (a, g) también debe tener una x , pues está en la lista asociada con (b, g) . La cadena 01 distingue a de g .

Al completar la tabla, concluimos que los estados equivalentes son $a = e, b = h$ y $d = f$.

EJERCICIO 14.

Desarrolle el diagrama del autómata original y del autómata mínimo generado con la tabla de la descripción de este método de minimización en la descripción de los párrafos anteriores.

EJERCICIO 15.

¿Cómo podemos probar que dos autómatas son equivalentes?

EJERCICIO 16.

Desarrolle un diagrama de autómata que represente la siguiente expresión regular en awk $^{+}[-]?(0-9)+[.]?0-9^{*}|[.]0-9^{+})\$$

Considere como condición de error si esta expresión no se cumple y si se cumple, realice las operaciones aritméticas básicas como son la suma, resta, multiplicación, división y exponenciación.

EJERCICIO 17.

Proporcione la definición formal del diagrama de autómata de la pregunta anterior.

EJERCICIO 18.

Desarrolle un autómata finito no determinístico que genere el siguiente lenguaje
 $L = \{ab, abb, abb, abbb, \dots, \infty\}$

EJERCICIO 19.

Convierte el autómata anterior a un autómata finito determinístico y compruébelo.

UNIDAD II • AUTÓMATAS FINITOS

UNIDAD III

Mama, ¿Qué haces en frente de la computadora con los ojos cerrados???

- Nada hijo es que Windows me dijo que cierre las pestanas.

CONCEPTOS DE GRAMÁTICAS

“Explica los conceptos relacionados con gramáticas, mediante la simbología idónea, aplicables a los métodos de normalización y en áreas de desarrollo de software de base”

TEMAS:

- 3.1 Conceptos básicos relacionados con gramáticas
- 3.2 Jerarquía de Chomsky
- 3.3 Gramáticas Libres de Contexto
 - 3.3.1 Árboles de derivación
 - 3.3.2 Derivación por la derecha e izquierda
 - 3.3.3 Recursividad por la derecha e izquierda
- 3.4 Gramáticas Regulares.
- 3.5 Otros conceptos relacionados

3.1 Conceptos básicos relacionados con gramáticas

La palabra “gramática” la define el diccionario de la lengua española como “Arte que enseña a hablar y escribir correctamente”, desde el punto de vista de la lingüística es; “Estudio sistemático de los elementos constitutivos de una lengua”, por último, de forma general se define como; “el libro que contiene reglas para el buen uso del idioma”. Definiciones tomadas de la página 204. Autores varios. (2005). A cada tipo de lenguaje le corresponde un tipo particular de gramática L (G). Según la descripción de la unidad uno de este material, recordemos que los lenguajes pueden ser “naturales” es decir, aquellos que utilizan los seres humanos para comunicarse, éstos son conocidos como idiomas, de los cuales se derivan dialectos, lenguas y tienen sus propias gramáticas cada uno de ellos.

Noam Chomsky, lingüista de la década de los cincuenta, inició el estudio de las “gramáticas formales”. Aunque no son máquinas estrictamente, estas gramáticas están estrechamente relacionadas con los autómatas abstractos y sirven actualmente como base de algunos importantes componentes de software, entre los que se incluyen componentes de los compiladores. Los lenguajes independientes del contexto, mismos que se definen más adelante, también se conocen con el nombre de gramáticas libres de contexto, son un método recursivo sencillo de especificación de reglas gramaticales con las que se pueden generar cadenas (palabras) de un lenguaje.

Es factible producir de esta manera todos los lenguajes regulares, además de que existen ejemplos sencillos de gramáticas libres de contexto que generan lenguajes no regulares. Las reglas gramaticales de este tipo permiten que la sintaxis tenga variedad y refinamientos mayores que los realizados con los lenguajes regulares, en gran medida sirven para especificar la sintaxis de lenguajes de alto nivel y otros lenguajes formales. Todos estos desarrollos teóricos afectan directamente a lo que los expertos en computadoras hacen.

Algunos de los conceptos, como el de “autómata finito” y determinados tipos de gramáticas formales, se emplean en el diseño y la construcción de importantes clases de software. Otros conceptos, como la “máquina de Turing”, nos ayudan a comprender lo que podemos esperar de nuestro software. Una “gramática formal” es una estructura matemática con un conjunto de reglas de formación que define las cadenas de caracteres admisibles en un determinado lenguaje “formal”.

UNIDAD III • CONCEPTOS DE GRAMÁTICAS

Tales gramáticas formales aparecen en varios contextos diferentes como, la lógica matemática, las ciencias de la computación y la lingüística teórica.

EJERCICIO 20.

Desarrolle un resumen o bien organizador gráfico que muestre de forma objetiva las definiciones coloquiales de gramáticas relacionándolas con tres criterios de clasificarlas.

1) El criterio basado en sus definiciones formales según Chomsky (cuatro tipos); 2) Sus contenidos como por ejemplo una gramática aumentada, con o sin atributos, etc. Y 3) Según su recorrido LR, LL, etc. Solo tome en consideración sus nombres.

Algunos términos son importantes y parte fundamental para comprender lo que se planteará en adelante acerca de las gramáticas, tipos de gramáticas, y algunos ejemplos.

CONCEPTO	BREVE DESCRIPCIÓN
Producción	Llamamos producción (o regla de producción) definida sobre ese alfabeto, a un par ordenado de palabras (x,y) . A las producciones también se las llama reglas de derivación. Se representa $x : : = y$. Se lee de izquierda a derecha, y dice: x produce y
Producción Compresora	Una producción es compresora si la longitud de su parte derecha es menor que la de la parte izquierda.
Derivación Directa	Sea Σ un alfabeto, P un conjunto de producciones definidas sobre ese alfabeto: $v, w \in \Sigma^*$. La notación utilizada para representar una derivación directa es $v \rightarrow w$
Derivación	Sea Σ un alfabeto, P un conjunto de producciones definidas sobre ese alfabeto y $v, w \in \Sigma^*$. La notación utilizada en este caso es $v \rightarrow *w$.

UNIDAD III • CONCEPTOS DE GRAMÁTICAS

3.2 Jerarquía de Chomsky

Noam Chomsky es profesor de lingüística y activista político estadounidense, nacido el 7 de diciembre de 1928 en Filadelfia, Pennsylvania, en 1955 se doctoró en la universidad de Pennsylvania y fue profesor e investigador del MIT (Massachusetts Institute of Technology)

Hay distintos tipos de gramáticas formales que generan a su vez lenguajes formales, dentro de estos están los definidos por la jerarquía descrita por Noam Chomsky en 1956. Chomsky definió cuatro tipos distintos de gramáticas en función de la forma de las reglas de la derivación P. La clasificación comienza con un tipo de gramáticas que pretenden ser universal, aplicando restricciones a sus reglas de derivación se van obteniendo los otros tipos de gramáticas. Esta clasificación es jerárquica, es decir cada tipo de gramáticas engloba a todos los tipos siguientes.

Gramáticas Tipo 0: Recursivamente enumerables (LRE)

Para este tipo de gramáticas no restringidas o gramáticas con estructura de frase. Las reglas de derivación son de la forma:

$$\alpha \rightarrow \beta$$

Siendo $\alpha \rightarrow \beta(V \cup V)^+$ y $\beta \in (V \cup V)^*$, es decir la única restricción es que no puede haber reglas de la forma $\gamma \rightarrow \beta$ donde γ la cadena vacía.

Ejemplo: Sea la gramática $G = (\{A,S\}, \{a,b\}, S, P)$ donde las reglas de producción son:

- 1) $S \rightarrow aS$
- 2) $S \rightarrow aA$
- 3) $A \rightarrow bA$
- 4) $A \rightarrow b$

Estas cuatro reglas de producción también se pueden expresar en sólo dos, ya que la siguiente representación de P (producciones) puede ser también:

- 1) $S \rightarrow aS|aA$ Esta línea | conocida como el carácter “pipe” expresa una u otra opción (aS o aA)
- 2) $A \rightarrow bA|b$

Problema. Aquí el planteamiento es determinar el lenguaje que genera esta gramática (con 4 o 2 reglas de producción, equivale al mismo lenguaje).

Solución. Esto que se presenta a continuación son solo algunos ejemplos de las sentencias, cadenas o palabras generadas por las reglas de producción arriba descritas.

$$S \rightarrow aS \rightarrow aaA \rightarrow aab$$

$$S \rightarrow aA \rightarrow ab$$

$$S \rightarrow aS \rightarrow aaS \rightarrow aaaS \rightarrow \dots \rightarrow a^nS \rightarrow a^n a A \rightarrow a^{n+1} b$$

$$S \rightarrow aA \rightarrow abA \rightarrow abbA \rightarrow abbbA \rightarrow ab^n A \rightarrow ab^{n+1}$$

El lenguaje generado se puede describir en la siguiente expresión regular:

$$L(G) \rightarrow aa^*bb^*$$

Tipo	Lenguaje	Autómata
0	Recursivamente enumerable (LRE)	Máquina de Turing (MT)
1	Dependiente del contexto (LSC)	Autómata linealmente acotado
2	Independiente del contexto (LLC)	Autómata de pila
3	Regular (RL)	Autómata finito

Tabla 3.1. “Jerarquía de Gramáticas”

Gramáticas Tipo 1: Dependientes del contexto (LSC)

También llamadas gramáticas sensibles al contexto, en estas las reglas de producción son de la siguiente forma:

$$\alpha A \beta \rightarrow \alpha \gamma \beta$$

Siendo $A A \in V$; $\alpha, \beta (V \cup V)^*$ y $\gamma (V \cup V)^+$

Estas gramáticas se llaman sensibles al contexto, pues se puede remplazar A por γ siempre que estén en el contexto $\alpha \dots \beta$

Ejemplo: La gramática $G = (VN, VT, S, P)$ donde

$$VN = \{\langle \text{numero} \rangle, \langle \text{dígito} \rangle\}$$

$$VT = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

$$S = \langle \text{número} \rangle$$

Y las reglas de producción que se muestran a continuación son de tipo 1.

UNIDAD III • CONCEPTOS DE GRAMÁTICAS

- 1) $\langle \text{número} \rangle \rightarrow \langle \text{dígito} \rangle \langle \text{número} \rangle$
- 2) $\langle \text{número} \rangle \rightarrow \langle \text{dígito} \rangle$
- 3) $\langle \text{dígito} \rangle \rightarrow 0|1|2|3|4|5|6|7|8|9$

Gramáticas Tipo 2: Independiente del contexto (LLC)

Estás gramáticas se denominan gramáticas libres de contexto, las reglas de producción de estás solo admiten tener un símbolo no terminal en su parte izquierda, es decir son de la forma:

$$A \rightarrow \alpha$$

Siendo $A \in V_N$ y $\alpha \in (V_N \cup V_T)^+$

Si cada regla se representa como un par ordenado (A, α) , el conjunto P es un subconjunto del conjunto producto cartesiano $\{ VN \times (\{ VN \} \cup \{ VT \}) \}^+$ es decir:

$$P \subseteq \{ VN \times (\{ VN \} \cup \{ VT \ }) \}^+$$

La denominación contexto libre se debe a que se puede cambiar A por α , independiente del contexto en que aparezca A .

La gramática $G = (VN, VT, S, P)$

Donde

$$VN = \{ \langle \text{número} \rangle, \langle \text{dígito} \rangle \}$$

$$VT = \{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 \}$$

$$S = \langle \text{número} \rangle$$

Y las reglas de producción que se muestran a continuación son de tipo 2.

- 1) $\langle \text{número} \rangle \rightarrow \langle \text{dígito} \rangle \langle \text{número} \rangle$
- 2) $\langle \text{número} \rangle \rightarrow \langle \text{dígito} \rangle$
- 3) $\langle \text{dígito} \rangle \rightarrow 0|1|2|3|4|5|6|7|8|9$

Gramáticas Tipo 3: Regular (RL)

Las gramáticas de tipo 3 también se denominan regulares o gramáticas lineales al derecha comienzan sus reglas de producción por un símbolo terminal, puede ser seguido o no por un símbolo no terminal, es decir son de la forma:

$$\begin{array}{lcl} A & \rightarrow & ab \\ A & \rightarrow & a \end{array}$$

Donde $A, B \in VN$ y $\alpha \in VT$

UNIDAD III • CONCEPTOS DE GRAMÁTICAS

La gramática $G = (\{a,b\}, \{A,S\}, S, P)$ donde P son las reglas de producción que se muestran a continuación es de tipo 3.

- 1) $S \rightarrow aS$
- 2) $S \rightarrow aA$
- 3) $A \rightarrow bA$
- 4) $A \rightarrow b$

La diferencia entre estos tipos es que cada uno de ellos tiene reglas más particulares y restringidas y por tanto generan lenguajes formales menos generales. Dos tipos importantes son las gramáticas libres de contexto (Tipo 2) y las gramáticas regulares (Tipo 3).

Los lenguajes que pueden ser descritos mediante esos tipos de gramáticas se les conoce como lenguajes libres de contexto y lenguajes regulares respectivamente. Estos dos tipos son mucho menos generales que las gramáticas no restringidas de Tipo 0 (es decir, que pueden ser procesadas o reconocidas mediante máquinas de Turing). Estos dos tipos de gramáticas se usan más frecuentemente puesto que los analizadores sintácticos para estos lenguajes pueden implementarse de manera eficiente. Por ejemplo, todos los lenguajes regulares pueden ser reconocidos por un autómata finito.

Para subconjuntos de gramáticas libres de contexto, existen algoritmos para generar analizadores sintácticos LL y analizadores sintácticos LR eficientes, mismos que se explicaran en el tema de los árboles de derivación y, que permiten reconocer los correspondientes lenguajes generados por esas gramáticas.

Limitación de las gramáticas formales

Las gramáticas como la usada en los primeros modelos de gramática generativa requieren ciertas restricciones para ser computacionalmente tratables. Para entender esa restricción debe considerarse la interacción entre un hablante y un oyente; el primero genera una oración o secuencia de acuerdo con las reglas de la gramática, el segundo, para entender dicha secuencia, debe analizar la secuencia para entenderla, encontrando los elementos formantes, interpretándolos y reconstruyendo la relación que hay entre ellos (estructura interna). Para que esto sea posible, se requiere que la estructura interna tenga una forma suficientemente simple como poder analizar sintácticamente las secuencias con un bajo grado de ambigüedad (concepto que se

UNIDAD III • CONCEPTOS DE GRAMÁTICAS

explicara más adelante). Pues bien, computacionalmente se ha encontrado que la clase de complejidad frente al análisis inverso de ciertas gramáticas es excesiva. Para estas gramáticas basadas en reglas de reescritura se tiene (en la siguiente página):

RESTRICCIONES EN LAS REGLAS	TIPO DE GRAMÁTICA	TIPO DE LENGUAJE	GRADO DE COMPLEJIDAD
Tipo 3	Gramática ES regular	lenguajes regulares	Lineal
Tipo 2	Gramática ES libre de contexto	lenguajes libres de contexto	Polinómica
Tipo 1	Gramática ES dependiente del contexto	lenguajes dependientes del contexto	Exponencial
Tipo 0	Gramática ES no restringida	lenguajes recursivamente enumerables	Indecidible

Tabla 3.2. “Gramáticas y sus características”

Teorema de la Jerarquía: Sobre un alfabeto dado, el conjunto de los *lenguajes recursivamente enumerables* (tipo 0) contienen propiamente al conjunto de los *lenguajes sensibles al contexto* (tipo 1), que contiene propiamente al conjunto de los *lenguajes independientes del contexto* (tipo 2), que a su vez contienen propiamente a los lenguajes regulares (tipo 3).

Especialmente resaltaremos que en el año de 1956 Chomsky introdujo las gramáticas independientes del contexto como parte de un estudio sobre los lenguajes naturales. Su utilización para especificar la sintaxis de los lenguajes de programación surgió independientemente, mientras trabajaba en el borrador del lenguaje ALGOL 60, en el cual John Backus adaptó de inmediato las producciones de Emil Post a ese uso. La notación resultante fue una variante de las gramáticas independientes del contexto.

3.3 Gramáticas Libres de Contexto (GLC)

También conocidas como Gramáticas independientes del contexto (GIC). Una gramática independiente del contexto es una notación formal que sirve para expresar las definiciones recursivas de los lenguajes. Una gramática consta de una o más variables que representan las clases de cadenas, es decir, los lenguajes de programación que son manipulables mediante la abstracción matemática. Lo contrario a los lenguajes de programación para computadoras son los **lenguajes naturales**.

Concepto de un lenguaje natural.

Podemos definir como lenguajes naturales a todos aquellos que utilizan los humanos para comunicarse, tales como el Chino, Japonés, Español, Inglés, Francés, Italiano, Portugués, etc. Estos tienen reglas gramaticales regidas por la sintaxis y la semántica de cada uno, ésta siempre ligada a un contexto. Por su gran cantidad de lenguajes naturales, expresiones idiomáticas, y excepciones a las reglas antes mencionadas, se tiene gran complejidad, misma que es motivo de estudio de la inteligencia artificial.

Ambigüedades en lenguajes naturales

En el manejo de cualquier lenguaje natural se tiene este problema, la ambigüedad significa que de una palabra se pueden tener distintas interpretaciones y su significado queda incierto, dudoso y poco claro, como un ejemplo podemos decir de la palabra “papa”, misma que puede tener al menos dos significados Papa, autoridad suprema de la iglesia católica, apostólica romana y papa como tubérculo.

Gramáticas para describir lenguajes naturales

Para poder describir a los lenguajes naturales, podemos escribir reglas como las que enumeramos en el siguiente conjunto P_n

$P_n = \{<\text{oración}> \rightarrow <\text{su}jeto> <\text{predicado}>,$
 $<\text{su}jeto> \rightarrow <\text{sustantivo}> | <\text{artículo}> <\text{sustantivo}> | <\text{artículo}> <\text{sustantivo}> <\text{adjetivo}>,$
 $<\text{predicado}> \rightarrow <\text{verbo}> | <\text{verbo}> <\text{modificador}>,$
 $<\text{artículo}> \rightarrow \text{el} | \text{la} | \text{los} | \text{las} | \text{un},$
 $<\text{sustantivo}> \rightarrow \text{n}i\text{o} | \text{m}adre | \text{ár}bol | \text{v}ehículos,$
 $<\text{adjetivo}> \rightarrow \text{pe}queñ)o | \text{al}to | \text{j}oven,$

UNIDAD III • CONCEPTOS DE GRAMÁTICAS

<verbo> → aprende|corre|juega|crece,
<modificador>→ velozmente|risueño|temprano}

Vale la pena destacar que esta es “una versión muy simplificada” de la gramática castellana, y si bien da lugar a oraciones que son correctas en nuestro idioma, también puede generar oraciones que resulten incorrectas desde el punto de vista tanto semántico (de su significado), como sintáctico (de su estructura).

Este es el caso de:

El árbol juega (incorrecta semánticamente)
Los madre come risueño (incorrecta sintácticamente)

Estos errores se solucionan en la lengua castellana agregando reglas que regulan y restringen las combinaciones de terminales y que corresponden al análisis morfológico y semántico del lenguaje. (En los lenguajes formales no existe el análisis morfológico que es el de género, número y persona, es decir el que controla la concordancia entre sujeto y verbo o la concordancia entre artículo, sustantivo y adjetivo)

COMUNICACIÓN HOMBRE MÁQUINA

Por todo lo que hemos visto, podemos decir que la comunicación hombre máquina se logra creando los lenguajes de programación, los cuales son un tipo muy especial de software que permite al hombre dar instrucciones a la máquina de una forma factible, estructurada, ordenada, siguiendo las reglas gramaticales que el mismo hombre diseña para su fácil utilización y para lograr resolver cualquier tipo de programas.

Gramáticas formales

Estas gramáticas permitirán en forma intencional describir un determinado lenguaje; esto se hará definiendo el alfabeto sobre el que se construirán sus palabras, denominadas símbolos terminales; un símbolo inicial del que se partirá para la obtención de cualquier de las palabras del lenguaje llamado axioma inicial, un conjunto de símbolos especiales denominados no terminales, los que permitirán expresar representaciones o estados intermedios en el proceso de generación de las palabras del lenguaje; y un conjunto de reglas de producción o de reescritura, que serán las que

permitan realizar las transformaciones necesarias, partiendo desde el axioma inicial, produciendo los reemplazos de símbolos no terminales, mediante la utilización de las reglas de producción hasta obtener las palabras del lenguaje.

Definición

Chomsky: $G_3 \subset G_2 \subset G_1 \subset G_0$

Una gramática formal **G**, es una 4-tupla, que queda definida de la siguiente manera:
 $G = (\Sigma_T, \Sigma_N, S, P)$

En donde:

Σ_T Conjunto de símbolos que representan el alfabeto de símbolos terminales, en donde toda palabra del lenguaje generado por esta gramática, estará formada por símbolos o caracteres definidos en este conjunto.

Σ_N Conjunto de símbolos que representan el alfabeto de símbolos NO terminales. Este conjunto de símbolos será utilizado como símbolos auxiliares en la derivación de cadenas, pero no formarán parte de las cadenas del lenguaje.

De las definiciones anteriores se observa lo siguiente: $\Sigma = \Sigma_T \cup \Sigma_N$ y $\Sigma_T \cap \Sigma_N = \emptyset$

S: Símbolo NO terminal especial, que pertenece al conjunto de símbolos NO terminales ($S \in \Sigma_N$), denominado símbolo inicial o axioma de la gramática.

P: Conjunto finito de reglas o producciones que tienen como única restricción que en la parte izquierda debe haber al menos un símbolo no terminal.

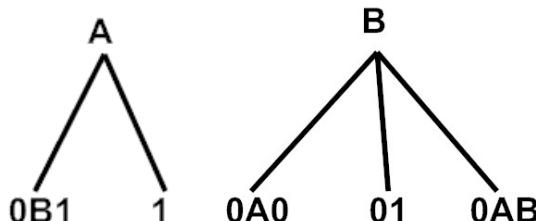
Ejemplo: dada la gramática: $G_1 = (\{0,1\}, \{A, B\}, A, P)$

Dónde: $P = \{(A := 0B1), (A := 1), (B := 0A0), (B := 01), (B := 0AB)\}$

También podríamos escribir las:

$A := 0B1|1$

$B := 0A0|01|0AB$



UNIDAD III • CONCEPTOS DE GRAMÁTICAS

Producciones

Una producción, o regla de producción es un par ordenado de palabras (x, y), con $x \in \Sigma^*$, donde la presencia de x se encuentra como parte integrante de cualquier otra palabra, puede ser sustituida por y ; lo que permite transformar palabras en otras.

Como notación suele utilizarse $x := y$, denominada notación BNF (por *Backus - Naur Form*, por sus creadores).

La palabra x es denominada la parte izquierda o primer miembro de la producción; y la palabra y , la parte derecha o segundo miembro.

Simbología :=

Serán producciones el conjunto P de la gramática de ejemplo

$G_1: \{(A := 0B1), (A := 1), (B := 0A0), (B := 01), (B := 0AB)\}$

EJERCICIO 21.

Desarrolle una tabla que contenga al menos tres columnas; la primera que contenga siglas o acrónimos, del acrónimo de la primera columna en la segunda columna describir su significado en inglés y la tercera columna con su significado en español de los siguientes acrónimos: LRE, LSC, LLC, RL, ES, LL, LR.

3.3.1 Árboles de derivación.

Un árbol de derivación permite mostrar gráficamente cómo se pueden derivar cualquier cadena de un lenguaje a partir del símbolo distinguido de una gramática que genera ese lenguaje. Un árbol es un conjunto de puntos, llamados nodos, unidos por líneas llamadas arcos. Un arco conecta dos puntos distintos. Para ser un árbol un conjunto de nodos y arcos debe satisfacer ciertas propiedades:

Hay un único nodo distinguido, llamado raíz (se dibuja en la parte superior) que no tiene arcos incidentes. Todo nodo “c” excepto el nodo raíz está conectado con un arco a otro nodo “k”, llamado el padre de “c” (“c” es el hijo de “k”). El padre de un nodo, se dibuja por encima del nodo.

Todos los nodos están conectados al nodo raíz mediante un único camino. Los nodos que no tienen hijos se denominan hojas, el resto de los nodos se denominan nodos interiores.

Propiedades de un árbol de derivación.

Sea $G = (N, T, S, P)$ una gramática libre de contexto, sea $A \in N$ una variable. Diremos que un árbol $TA = (N, E)$ etiquetado es un árbol de derivación asociado a G si verifica las propiedades siguientes:

La raíz del árbol es un símbolo no terminal

Cada hoja corresponde a un símbolo terminal o ϵ .

Cada nodo interior corresponde a un símbolo no terminal.

Para cada cadena del lenguaje generado por una gramática es posible construir (al menos) un árbol de derivación, en el cual cada hoja tiene como rótulo alguno de los símbolos de la cadena, según se muestra en la **figura 3.0**.

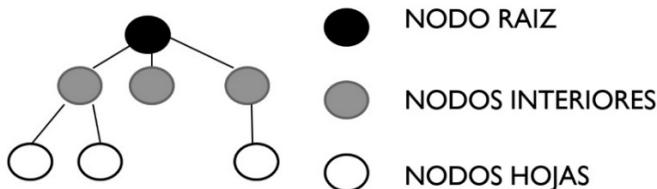


Figura 3.0 Elementos de la estructura de datos de un árbol

UNIDAD III • CONCEPTOS DE GRAMÁTICAS

Para cada cadena del lenguaje generado por una gramática es posible construir (al menos) un árbol de derivación, en el cual cada hoja tiene como rótulo uno de los símbolos de la cadena.

Si un nodo está etiquetado con una variable X y sus descendientes (leídos de izquierda a derecha) en el árbol son X_1, \dots, X_k , entonces hay una producción $X \rightarrow X_1 \dots X_k$ en G .

Sea $G=(N,T,S,P)$ una GLC. Un árbol es un árbol de derivación para G si:

1. Todo vértice tiene una etiqueta tomada de $T \cup N \cup \{\epsilon\}$
2. La etiqueta de la raíz es el símbolo inicial S
3. Los vértices interiores tienen etiquetas de N
4. Si un nodo n tiene etiqueta A y $n_1 n_2 \dots n_k$ respectivamente son hijos del vértice n , ordenados de izquierda a derecha, con etiquetas x_1, x_2, \dots, x_k respectivamente, entonces: $A \rightarrow x_1 x_2 \dots x_k$ debe ser una producción en P
5. Si el vértice n tiene etiqueta ϵ , entonces n es una hoja y es el único hijo de su padre.

3.3.2 Derivación por la derecha e izquierda (LL, LR)

Derivación directa

Es la aplicación directa de una producción ($x := y$), a una determinada palabra v para convertirla en w .

Simbología =

Ejemplo: dado $v = z \cdot x \cdot u$; y aplicando ($x = y$) obtenemos $w = z \cdot y \cdot u$

Con $v, w, z, u \in \Sigma^*$

Para el caso de la gramática G1, podemos, partiendo de una palabra 0B1, obtener, aplicando las reglas de producción ($B := 0A0$) lo siguiente: $0B1 = 00A01$

Derivación:

Es la aplicación de una secuencia de producciones a una palabra.

Simbología: \rightarrow

Ejemplo: En el ejercicio anterior continuaremos aplicando reglas de producción y pasaremos de una palabra 0B1 a una palabra 0000101

$0B1 \rightarrow 00A01 \rightarrow 000B101 \rightarrow 00001101$

Descripción: En 0B1 se reemplaza la B por 0A0 y queda $0(0A0)1 \rightarrow 00A01$

En 00A01 se reemplaza la A por 0B1 y queda $00(0B1)01 \rightarrow 000B101$

Y finalmente En 000B101 se reemplaza la B por 01 y queda $000B101 \rightarrow 000(01)101$

Según reglas de producción: $\{(A \rightarrow 0B1), (A \rightarrow 1), (B \rightarrow 0A0), (B \rightarrow 01), (B \rightarrow 0AB)\}$

Derivaciones utilizando una gramática

Aplicamos las producciones de un GIC para inferir que determinadas cadenas pertenecen al lenguaje de una cierta variable. Para llevar a cabo esta inferencia hay disponibles dos métodos. El más convencional de ellos consiste en emplear las reglas para pasar del cuerpo a la cabeza.

Procedimiento de inferencia cursiva

- Tomamos cadenas que sabemos que pertenecen al lenguaje de cada una de las variables del cuerpo.
- Concatenar en el orden apropiado con cualquier símbolo terminal.
- Inferir que la cadena resultante pertenece al lenguaje de la variable de la cabeza.

UNIDAD III • CONCEPTOS DE GRAMÁTICAS

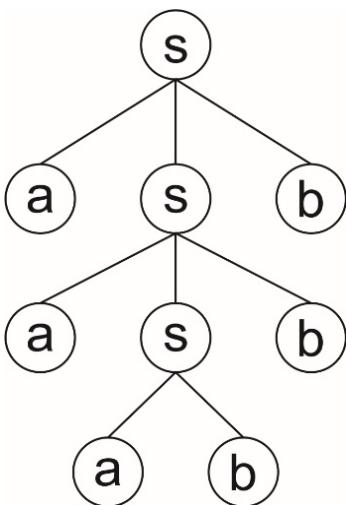
1. $E \rightarrow I$
2. $E \rightarrow E + E$
3. $E \rightarrow E * E$
4. $E \rightarrow (E)$
5. $I \rightarrow a$
6. $I \rightarrow b$
7. $I \rightarrow Ia$
8. $I \rightarrow Ib$
9. $I \rightarrow I0$
10. $I \rightarrow I1$

Imaginemos una gramática con estas dos reglas:

$$A \rightarrow bA \quad 2. A \rightarrow c$$

El elemento en mayúsculas es el símbolo inicial. Los elementos en minúsculas son los símbolos terminales. Para generar cadenas de caracteres, la idea es sustituir el símbolo inicial de la izquierda por los símbolos de la derecha, y luego repetir el proceso hasta que sólo haya símbolos terminales.

Por ejemplo: $A \rightarrow bA \rightarrow bbA \rightarrow bbbA \rightarrow bbcb$



Esta gramática da lugar a un lenguaje formal que consiste en el conjunto de todas las cadenas de caracteres que pueden ser generadas por medio ellas. Por ejemplo: bbcb, bbbbbbbbc, c, bc, etc.

Ejemplo de un árbol de derivación.

Sea $G = (N, T, S, P)$ una GLC con $P: S \rightarrow ab|aSb$

La derivación de la cadena: aaabbb será:

$S \rightarrow aSb \rightarrow aaSbb \rightarrow aaabbb$ y el árbol de derivación se muestra en la **figura 3.1**

Figura 3.1 árbol de derivación de izquierda a derecha para la cadena “aaabbb”

La palabra está formada por los nodos hojas tomados de izquierda a derecha y de arriba para abajo.

Relación entre derivaciones y árboles. Si leemos las etiquetas de las hojas de izquierda a derecha tenemos una sentencia. Llamamos a esta cadena la producción del árbol de derivación.

Teorema. Sea $G=(N, T, S, P)$ una GLC. Entonces $S \rightarrow^* \alpha$ (de S se deriva α) si y sólo si hay un árbol de derivación en la gramática G con la producción α .

Si w es una cadena de $L(G)$ para la gramática libre de contexto G , entonces w tiene al menos un árbol de derivación. Referido a un árbol de derivación particular, w tendrá una única derivación a la izquierda y otra única a la derecha. El ejemplo se muestra en la **figura 3.2** representa la derivación a la izquierda.

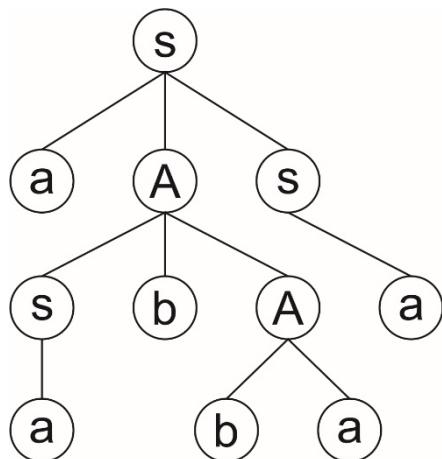


Figura 3.2 árbol de derivación de derecha a izquierda para la cadena “aabbaa”

Aquí la derivación se toma en la primera cadena aAS , el nodo hoja de la izquierda con los dos nodos intermedios no terminales (AS , con mayúscula). La segunda cadena: $aSbAS$, es el primer nodo hoja de la izquierda con el nodo intermedio unido al nodo intermedio de A que es la S y su nodo hoja también de A , para terminar con el tercer nodo de A que también es una A y el último nodo intermedio S . La tercera cadena $aabAS$ toma todos los nodos hojas desde S (raíz) y deja los dos últimos nodos intermedios de la derecha que es la AS . La cuarta cadena, $aabbAS$, toma todos los nodos hojas desde S (raíz) y solo deja el último nodo intermedio de la derecha que es la misma S . Por último, la quinta cadena, toma todos los nodos hojas desde S (raíz) y queda $aabbaa$.

UNIDAD III • CONCEPTOS DE GRAMÁTICAS

Conclusión derivación de izquierda a derecha, derivando nodos intermedios generadores de nodos hojas, desde el nodo raíz a las hojas.

Derivación a la derecha: $S \rightarrow aAS \rightarrow aAa \rightarrow aSbAa \rightarrow aSbbaa \rightarrow aabbaa$

Aquí la derivación se toma en la primera cadena aAS, al igual que en la derivación por la izquierda.

La segunda cadena: aSbAS, es el primer nodo hoja de la izquierda con el nodo intermedio unido al nodo intermedio de A que es la S y su nodo hoja también de A, para terminar con el tercer nodo de A que también es una A y el último nodo intermedio S.

La tercera cadena: aabAS toma todos los nodos hojas desde S (raíz) y deja los dos últimos nodos intermedios de la derecha que es la AS.

La cuarta cadena: aabbaS, toma todos los nodos hojas desde S (raíz) y solo deja el último nodo intermedio de la derecha que es la misma S.

Por último, la quinta cadena, toma todos los nodos hojas desde S (raíz) y queda aabbaa. Conclusión derivación de izquierda a derecha, derivando nodos intermedios generadores de nodos hojas, desde el nodo raíz a las hojas.

3.3.3 Recursividad por la derecha e izquierda

La recursividad es la característica de una gramática formal, la cual puede ser recursiva o repetitiva cuando existe una producción que se incluye a sí misma dentro de las reglas de producción de la gramática. Ejemplo, $S \rightarrow aS$ (Aquí la regla inicial de S también produce la cadena “ a ” de longitud 1 y la S que al repetirse a sí misma se convierte en infinita).

La recursividad tiene como característica principal la sensación de infinito, de algo que es continuo y que por tanto no puede ser delimitado en el espacio o el tiempo porque se sigue replicando y multiplicando de manera lógica y matemática. Así, es común encontrar casos de recursividad por ejemplo en imágenes de espejos que hacen que la imagen sea replicada al infinito, una dentro de otra hasta que deja de verse, pero no por eso deja de existir.

Otro caso típico de recursividad en las imágenes es cuando encontramos una publicidad en la que el objeto tiene la propaganda de sí mismo en su etiqueta y así al infinito, o cuando una persona está sosteniendo una caja de un producto en cuya etiqueta aparece esa misma persona sosteniendo el mismo producto y así hasta el infinito. En estos casos, la recursividad pasa por el hecho de que se busca definir algo con lo misma información que ya se tiene.

En la definición de una gramática recursiva (repetitiva hasta el infinito), se presentan algunos problemas en la derivación de palabras que sea de fácil derivación o generación. Es por esto que se tiene la necesidad de eliminar la recursividad por la izquierda o por la derecha para tener palabras definidas en el alfabeto.

Una gramática G se llama recursiva en A , $A \in \Sigma_N$

$$A \rightarrow + xAy$$

Si x es la palabra vacía, se dice que la gramática es recursiva a la izquierda

$$A \rightarrow + Ay$$

Si y es la palabra vacía, se dice que la gramática es recursiva a la derecha

$$A \rightarrow + xA$$

UNIDAD III • CONCEPTOS DE GRAMÁTICAS

Se dice que una producción es recursiva si

$A ::= xAy$

La producción es recursiva a izquierdas si $x = \epsilon$ (cadena o palabra vacía)

$A ::= Ay$

Será recursiva a derechas si $y = \epsilon$ (cadena o palabra vacía)

$A ::= xA$

Si un lenguaje es infinito, la gramática que lo representa ha de ser recursiva. Eliminación de la recursividad por la izquierda en producciones de un mismo símbolo no terminal.

$\forall A \in \Sigma_N$

Si $P1 = (A ::= A\alpha_1 | A\alpha_2 | \dots | A\alpha_n | \beta_1 | \beta_2 | \dots | \beta_m)$ entonces //crear un símbolo nuevo $A' \Sigma'_N = \Sigma_N \cup \{A'\}$;

$P = (P - P1) \cup \{A ::= \beta_1 A' | \beta_2 A' | \dots | \beta_m A'; A' ::= \alpha\}$

Donde $\beta_1 A' | \alpha_1$

$A' | < | \alpha$

$A' | \epsilon$

fsi

f \forall

Eliminación de la recursividad por la izquierda en más de un paso

1. Disponer los Σ_N en algún orden A_1, A_2, \dots, A_n

2. Para $i := 1$ hasta n

 Para $j := 1$ hasta n

 Si $i \neq j$ entonces

 reemplazar cada producción $A_i ::= A_{ij}$ por:

$A_i ::= \delta_{1j} | \delta_{2j} | \dots | \delta_{kj}$

 donde $A_j ::= \delta_1 | \delta_2 | \dots | \delta_k$ son todas las reglas de A

 fsi

 Eliminar la recursividad por la izquierda de las A

 fpara

 fpara

G =({{i,+,*,(,)},{E,T}},E, P={E ::= T+E | T*T | i ; T ::= E|(E)})

1. A₁=E; A₂=T

2. Bucles:

i=1 (A₁=E); j=1 (A₁=E). Se reemplazarían E ::= E_α. No hay.

i=1 (A₁=E); j=2 (A₂=T). Se reemplazan E ::= T_α. El nuevo P' es

E ::= E+E | E*E | (E)+E | (E)*E | i

T ::= E | (E)

Se elimina la recursión en E quedando P'':

E ::= (E)+EE' | (E)*EE' | i E'

E' ::= +EE' | *EE' | ε

T ::= E | (E)

i=2 (A₂=T); j=1 (A₁=E). Se reemplazan T ::= T_α.

El nuevo P''' es

E ::= (E)+EE' | (E)*EE' | i E'

E' ::= +EE' | *EE' | λ

T ::= (E)+EE' | (E)*EE' | i E' | (E)

Se eliminaría la recursión en T si la hubiera.

i=2 (A₂=T); j=2 (A₂=T). Se reemplazarían T ::= T_α.

No hay el conjunto final de producciones es P''''

EJERCICIO 22.

Desarrolle de la Gramática G los árboles de derivación e indique si las siguientes cadenas pueden ser generadas o no por esta gramática (demostrarlo mediante la derivación). Para esto, considere las siguientes cadenas:

x = “abbaaab”, y = “baabbaaaabb”, z = “baabab”, p = “abbaab”

G = ({a, b}, {A, B, S}, S, P), con P: S ::= aB|bA

B ::= bS|b

A ::= aS|a

3.4. Gramáticas Regulares.

Gramáticas Tipo 3.- Gramáticas regulares o de estado finito.

Sus producciones son de la forma:

Lineal por la derecha: $A \rightarrow aB$ o $A \rightarrow a$, donde $A, B \in \Sigma_N$, $a \in \Sigma_T$

Lineal por la izquierda: $A \rightarrow Ba$ o $A \rightarrow a$, donde $A, B \in \Sigma_N$, $a \in \Sigma_T$

Se permiten producciones de la forma $S \rightarrow \epsilon$

Los lenguajes representados por este tipo de gramáticas se denominan lenguajes regulares.

Ejemplos: $G1 = (\{0, 1\}, \{A, B\}, A, \{A ::= B1 \mid 1, B ::= A0\})$

Gramática lineal por la izquierda que describe el lenguaje

$$L1 = \{1, 101, 10101, \dots\} = \{1(01)^n \mid n = 0, 1, 2, \dots\}$$

$$G2 = (\{0, 1\}, \{A, B\}, A, \{A ::= 1B \mid 1, B ::= 0A\})$$

Gramática lineal derecha que genera el mismo lenguaje que la gramática anterior. Para cada Gramática lineal por la derecha existe una Gramática lineal izquierda que genera el mismo lenguaje y viceversa.

Algoritmo:

1. Se transforma la gramática de forma que no haya ninguna regla en cuya parte derecha esté el axioma. Para lo cual:
 - 1.1. Se crea un nuevo $S' \in \Sigma_N$
 - 1.2. \forall regla $S ::= x$ (con S axioma y $x \in \Sigma^*$) se crea una nueva regla $S' ::= x$.
 - 1.3. Cada regla $A ::= xSy$, se transforma en $A ::= xS'y$.
2. Se crea un grafo G dirigido:
 - 2.1. $\forall A \in \Sigma_N \cup \{\epsilon\}$ se crea un nodo.
 - 2.2. \forall producción $(A ::= aB) \in P$ se crea un arco etiquetado con a que va del nodo A al B .
 - 2.3. \forall producción $(A ::= a) \in P$ se crea un arco etiquetado con a que va del nodo A al $\lambda(F)$.
 - 2.4. Si $\exists S ::= \lambda$ se crea un arco sin etiqueta que va del nodo del axioma al nodo λ .

3. Se crea otro grafo G' a partir de G :

- 3.1. Se intercambian las etiquetas del axioma y λ .
- 3.2. Se invierte la dirección de todos los arcos.

4. Se transforma en un conjunto de reglas:

- 4.1. \forall nodo, se crea un símbolo no terminal excepto para el nodo ϵ .
- 4.2. \forall arco etiquetado con a que va del nodo A al nodo $B \in \Sigma_N \cup \{\epsilon\}$, se crea una producción $A ::= Ba$.
- 4.3. Si \exists un arco del nodo del axioma al nodo λ se crea una regla $S ::= \epsilon$

El algoritmo para el paso inverso es similar al descrito.

Ejemplo: Gramática lineal derecha:

$$G1 = (\{0, 1\}, \{A, B\}, A, \{A ::= 1B \mid \epsilon, B ::= 0A, B ::= 0\})$$

1. Se transforma la gramática para que no haya regla como: $B ::= 0A$

- 1) Se crea un nuevo A'
- 2) Se crean las reglas $A' ::= 1B \mid \epsilon$
- 3) Se crea la regla $B ::= 0A'$
- 4) Se borra la regla $B ::= 0A$

La Gramática resultante es:

$$G11 = (\{0, 1\}, \{A, B, A'\}, A, \{A ::= 1B \mid \epsilon, A' ::= 1B \mid \epsilon, B ::= 0A', B ::= 0\})$$

Se elimina $A' ::= \epsilon$ quedando:

$$G12 = (\{0, 1\}, \{A, B, A'\}, A, \{A ::= 1B \mid \epsilon, A' ::= 1B, B ::= 0A', B ::= 0\})$$

$$G12 = (\{0, 1\}, \{A, B, A'\}, A, \{A ::= 1B \mid \epsilon, A' ::= 1B, B ::= 0A', B ::= 0\})$$

Se crea un grafo G que representa las transiciones de la gramática según se muestra en la **figura 3.3**.

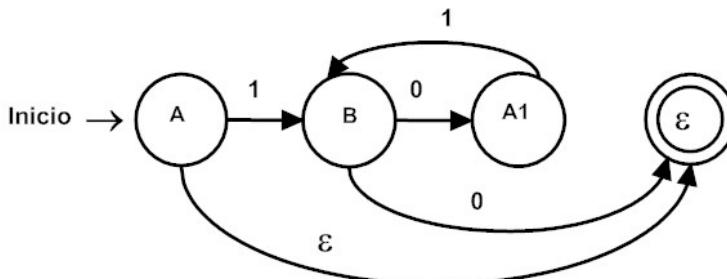


Figura 3.3
Autómata Finito
No determinístico
(AFN) para la
expresión regular
 $r = (10) + | 1(01) +$

2. Se crea el grafo G' invirtiendo G según figura 3.4

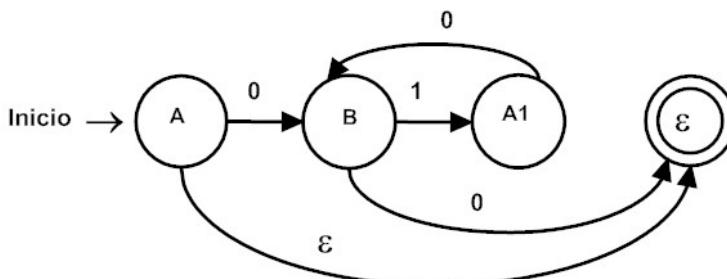


Figura 3.4
Autómata Finito
No determinístico
(AFN) para la
expresión regular
 $r = (00) + | 0(10) +$

3. Se transforma en un conjunto de reglas, en la Gramática lineal por la izquierda equivalente:

$$G2 = (\{0, 1\}, \{A, B, A'\}, A, \{A ::= B0 \mid \epsilon, A' ::= B0, B ::= A'1, B ::= 1\})$$

Para cada Gramática lineal por la izquierda existe una Gramática lineal derecha que genera el mismo lenguaje y viceversa.

Algoritmo:

1. Se transforma la gramática de forma que no haya ninguna regla en cuya parte derecha esté el axioma. Para lo cual:
 - 1.1. Se crea un nuevo $S' \in \Sigma_N$
 - 1.2. \forall regla $S ::= x$ (con S axioma y $x \in \Sigma^*$) se crea una nueva regla $S' ::= x$.
 - 1.3. Cada regla $A ::= xSy$, se transforma en $A ::= xS'y$.

2. Se crea un grafo G dirigido:
 - 2.1. $\forall A \in \Sigma_N \cup \{F\}$ se crea un nodo.
 - 2.2. \forall producción $(A ::= Ba) \in P$ se crea un arco etiquetado con a que va del nodo A al B .
 - 2.3. \forall producción $(A ::= a) \in P$ se crea un arco etiquetado con a que va del nodo A al F .
 - 2.4. Si $\exists S ::= \varepsilon$ se crea un arco con etiqueta ε que va del nodo del axioma al nodo F .
3. Se crea otro grafo G' a partir de G :
 - 3.1. Se intercambian las etiquetas del axioma y F .
 - 3.2. Se invierte la dirección de todos los arcos.
4. Se transforma en un conjunto de reglas:
 - 4.1. \forall nodo, se crea un símbolo no terminal excepto para el nodo F .
 - 4.2. \forall arco etiquetado con a que va del nodo A al nodo $B \in \Sigma_N \cup \{F\}$, se crea una producción $A ::= aB$.
 - 4.3. Si \exists un arco etiquetado con λ del nodo del axioma al nodo F se crea una regla $S ::= \varepsilon$

Ejemplo: Gramática lineal izquierda:

$$G1 = (\{0, 1\}, \{A, B\}, A, \{A ::= B1 \mid 1, B ::= A0\})$$

1. Se transforma la gramática para que no haya regla como: $B ::= A0$
 - 1.1. Se crea un nuevo A'
 - 1.2. Se crean las reglas $A' ::= B1 \mid 1$
 - 1.3. Se crea la regla $B ::= 0A'0$
 - 1.4. Se borra la regla $B ::= A0$

La Gramática resultante es:

$$G11 = (\{0, 1\}, \{A, B, A'\}, A, \{A ::= B1 \mid 1, A' ::= B1 \mid 1, B ::= A'0\})$$

2. Grafo G para las transiciones de la gramática, según se muestra en la **figura 3.5**

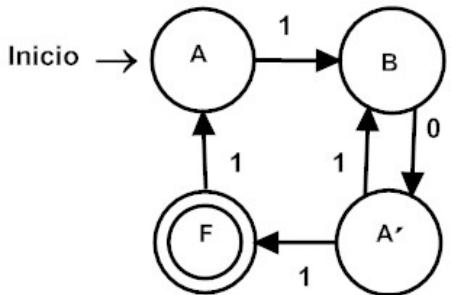


Figura 3.5. Autómata Finito No determinístico (AFN) para la expresión regular $r=1|1(01)^+$

3. Se crea el grafo G' invirtiendo y se muestra en la **figura 3.6**

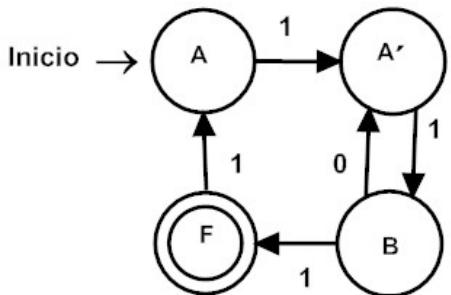


Figura 3.6. Autómata Finito No determinístico (AFN) para la expresión regular $r=1(01)^+$

4. Se transforma en las reglas, de la Gramática lineal por la derecha equivalente:
 $G2 = (\{0, 1\}, \{A, B, A'\}, A, \{A ::= 1A' | 1, A' ::= 0B, B ::= 1A' | 1\})$

Gramática regular asociada a un AFD

Si L es aceptado por un AFD entonces L puede generarse mediante una gramática regular.

Sea $M = (Q, \Sigma, f, q_0, F)$, la gramática regular equivalente es aquella definida como $G=(Q, \Sigma, P, q_0)$, donde P viene dado por:

Si $f(q,a)=p$ entonces añadir a P la producción $q \rightarrow ap$.

Si $f(q,a)=p$ y $p \in F$ entonces añadir a P la producción $q \rightarrow a$.

Si $q_0 \in F$ entonces añadir a P la producción $q_0 \rightarrow \epsilon$

Ejemplo: De AFD a Gramática regular:

Dado un lenguaje regular L reconocido por un AFD $M=(Q,\Sigma,f,q_0,F)$, se puede obtener una gramática regular $G=(\Sigma_N,\Sigma_T,S,P)$, que genere el mismo lenguaje de la siguiente forma:

$$\Sigma_N = Q$$

$$\Sigma_T = \Sigma$$

$$S = q_0$$

$$P=\{(q,ap)|f(q,a)=p\} \cup \{(q_0, \epsilon)|p, q_0 \in F\}$$

Ejemplo: Considera el siguiente DFA que acepta el lenguaje $a^* b$

El autómata se muestra en la **figura 3.7**

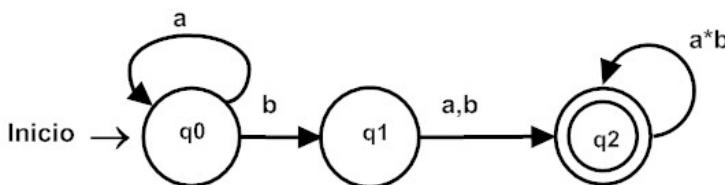


Figura 3.7
Autómata Finito
Determinístico
(AFD) para la
expresión regular
 $r=a^*b$

La gramática será:

$$q_0 \rightarrow aq_0|bq_1|b$$

$$q_1 \rightarrow aq_2|bq_2$$

$$q_2 \rightarrow aq_2|bq_2$$

Autómatas finitos y gramáticas regulares

AFD asociado a una gramática regular

Si L es un lenguaje generado por una gramática regular, entonces existe un AFD que lo reconoce:

Sea la gramática regular $G=(\Sigma_N, \Sigma_T, P, S)$, se construye el AFND- ϵ equivalente como:

$M=(\Sigma_N \cup \{F\}, \Sigma_T, f, S, \{F\})$, donde f viene dado por:

Si $A \rightarrow aB$ entonces $f(A,a)=B$

Si $A \rightarrow a$ entonces $f(A,a)=F$

Si $S \rightarrow \epsilon$ entonces $f(S, \epsilon)=F$

F es un “nuevo” símbolo no terminal.

UNIDAD III • CONCEPTOS DE GRAMÁTICAS

Convertir el **AFND-ε** obtenido a un **AFD**.

Ejemplo: De Gramática regular a **AFN**:

Dado un lenguaje regular L generado por una gramática regular $G=(\Sigma_N, \Sigma_T, S, P)$, se puede obtener un AFD definido como $M=(Q, \Sigma, f, q_0, F)$ que reconozca el mismo lenguaje de la siguiente forma:

$$Q = \Sigma_N \cup \{F\}$$

$$\Sigma = \Sigma_T$$

$$q_0 = S$$

Ejemplo: Considera la siguiente gramática:

$S \rightarrow aS|b$ que acepta el lenguaje a^*b :

El AFD equivalente se muestra en la **figura 3.8**

Las expresiones regulares son una notación especial que se utiliza habitualmente para describir los lenguajes de tipo regular. La notación más utilizada para especificar

patrones son las **expresiones regulares**, sirven como nombres para conjuntos de cadenas.

Los usos más habituales de las expresiones regulares son en la creación de analizadores léxicos para compiladores, en la búsqueda de patrones dentro de los editores de texto, en la descripción de redes neuronales y circuitos electrónicos, etcétera.

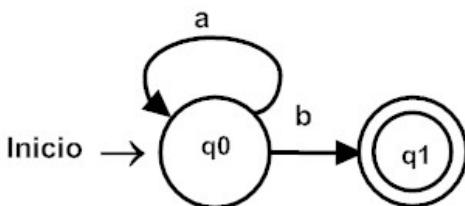


Figura 3.8 Autómata Finito No determinístico (AFN) para la expresión regular $r=a^*b$

En las expresiones regulares pueden aparecer símbolos de dos tipos:

Símbolos base: son los del alfabeto del lenguaje que queremos describir; λ para describir la palabra vacía y \emptyset para describir un lenguaje vacío.

Símbolos de operadores: “+” unión , “.” o concatenación y “*” para la clausura. En ocasiones se utiliza para la unión el símbolo “|” y el punto se omite en la concatenación de expresiones regulares.

El lenguaje de las expresiones regulares se construye de forma inductiva a partir de símbolos para expresiones regulares elementales y operadores.

Expresiones regulares

Dado un alfabeto Σ y los símbolos: \emptyset (lenguaje vacío), λ (palabra vacía), (\cup) concatenación, $+$ (unión), $*$ (clausura), se cumple:

Los símbolos \emptyset y ϵ son expresiones regulares.

Cualquier símbolo $a \in \Sigma$ es una expresión regular.

Si u y v son expresiones regulares, entonces, $u+v$, uv , u^* y v^* son expresiones regulares.

Sólo son expresiones regulares las que se pueden obtener aplicando un número finito de veces las reglas anteriores.

Se establece la siguiente prioridad en las operaciones:

1. Paréntesis ()
2. Clausura *
3. Concatenación ·
4. Unión +

A cada expresión regular le corresponde un **lenguaje regular**:

Si $\alpha = \emptyset$, $L(\alpha) = \emptyset$

Si $\alpha = \epsilon$, $L(\epsilon) = \{ \epsilon \}$.

Si $\alpha = a$, $a \in \Sigma$, $L(\alpha) = \{a\}$

Si α y β son dos expresiones regulares entonces:

$$L(\alpha+\beta) = L(\alpha) \cup L(\beta)$$

$$L(\alpha\beta) = L(\alpha)L(\beta)$$

$$L(\alpha^*) = L(\alpha)^*$$

Definiciones regulares

Son nombres dados a las expresiones regulares.

Una definición regular es una secuencia de definiciones de la forma:

$$d_1 \rightarrow r_1$$

$$d_2 \rightarrow r_2$$

.....

$$d_n \rightarrow r_n$$

UNIDAD III • CONCEPTOS DE GRAMÁTICAS

Donde cada d_i es un nombre distinto y cada r_i una expresión regular sobre los símbolos del alfabeto $\Sigma \cup \{d_1, d_2, \dots, d_n\}$

Ejemplo: definición regular para identificadores en Pascal:

letra $\rightarrow A|B|\dots|Z|a|b|\dots|z$

digito $\rightarrow 0|1|\dots|9$

id $\rightarrow \text{letra}(\text{letra}| \text{digito})^*$

Abreviaturas en la notación

Uno o más casos: operador unitario posfijo: **+**

Cero o un caso: operador unitario posfijo: **?**

Clases de caracteres: **[abc]**

Ejemplos:

digitos $\rightarrow \text{digito}^+$

Exponente_opcional $\rightarrow (E(+|-)?\text{digitos})?$

Ejemplo:

Sobre el alfabeto $\Sigma = \{a, b\}$, podemos definir las siguientes expresiones regulares:

Expresión Regular	Lenguaje
ab	$\{ab\}$
$a + \emptyset$	$\{a\}$
$(a+b)^*$	$\Sigma^*\{a\}$
$a + \epsilon$	$\{\epsilon, a\}$
b^*ab^*	$\{b^nab^m n, m \in \mathbb{N}\}$
a^*	$\{\lambda, a, aa, aaa, \dots\}$

Propiedades de la Unión:

Asociativo: $\alpha + (\beta + \epsilon) = (\alpha + \beta) + \epsilon$

Comutativo: $\alpha + \beta = \beta + \alpha$

Elemento neutro la expresión vacía: $\emptyset + \alpha = \alpha + \emptyset = \alpha$

Idempotente: $\alpha + \alpha = \alpha$

Propiedades de la concatenación:Asociativo: $\alpha(\beta\epsilon) = (\alpha\beta)\epsilon$ No es conmutativo: $\alpha\beta \neq \beta\alpha$ Elemento neutro la expresión lambda: $\epsilon\alpha = \alpha\epsilon = \alpha$ Tiene como elemento anulador la expresión vacía: $\varnothing\alpha = \alpha\varnothing = \varnothing$ Distributivo respecto al operador de unión: $\alpha(\beta + \epsilon) = \alpha\beta + \alpha\epsilon$ **Propiedades de la clausura:**

$$\epsilon^* = \epsilon$$

$$\varnothing^* = \epsilon$$

$$\alpha^* = \epsilon + \alpha\alpha^*$$

$$\alpha\alpha^* = \alpha^*\alpha$$

$$(\alpha^* + \beta^*)^* = (\alpha + \beta)^*$$

$$(\alpha + \beta)^* = (\alpha^*\beta^*)^*$$

El Teorema servirá para describir mediante una expresión regular el lenguaje reconocido por un autómata.

- El objetivo del teorema fundamental es resolver sistemas de ecuaciones en el dominio de las expresiones regulares que tienen la forma siguiente: $X = \alpha X + \beta$
- El teorema se divide en dos partes que nos permiten obtener la solución a esta ecuación en los casos en que $\epsilon \in L(\alpha)$ y aquellos en los que $\epsilon \notin L(\alpha)$, respectivamente.

Regla General:

Teorema. Sean α y β dos expresiones regulares de forma que $\epsilon \in L(\alpha)$. En estas condiciones, la solución a la ecuación

$X = \alpha X + \beta$ es de la forma $X = \alpha^*(\beta + \epsilon)$ en donde γ representa cualquier expresión regular.

Regla de Inferencia:

Teorema. Sean α y β dos expresiones regulares de forma que $\epsilon \notin L(\alpha)$. En estas condiciones,

$$X = \alpha X + \beta \iff X = \alpha^*\beta$$

UNIDAD III • CONCEPTOS DE GRAMÁTICAS

Todo lenguaje definido por un autómata finito (AFD, AFND) es también definido por una expresión regular.

Todo lenguaje definido por una expresión regular es definido por un autómata finito según se muestra en la **figura 3.9**.

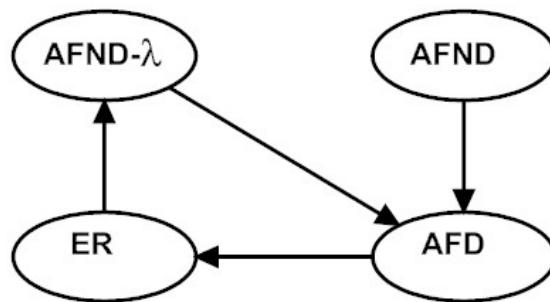


Figura 3.9 Relación de autómata finito NO determinístico con épsilon transiciones, y sin épsilon transiciones a otro Determinístico sin épsilon transiciones y este a una expresión regular.

Cada lenguaje definido por una expresión regular es también definido por un autómata finito.

La demostración es inductiva sobre el número de operadores de α (+, ., *).

Base.- (cero operadores)

A puede ser ϵ , \emptyset , a , donde $a \in \Sigma_T \dots \alpha$ Esto se muestra en la **figura 3.10**

Expresión regular	Autómata
ϵ	<p>Inicio \rightarrow $\xrightarrow{\epsilon}$ </p>
\emptyset	<p>Inicio \rightarrow $\xrightarrow{\emptyset}$ </p>
a	<p>Inicio \rightarrow \xrightarrow{a} </p>

Figura 3.10 Tabla de expresión regular de un símbolo a un autómata

AFN asociado a una expresión regular**Inducción.** (uno o más operadores en α)

Suponemos que se cumple la hipótesis para expresiones regulares de menos de n operadores.

Por hipótesis existen dos AF $M1$ y $M2$ tal que acepta el mismo lenguaje $L(M1)=L(\alpha_1)$ y $L(M2)=L(\alpha_2)$ donde los autómatas se muestran en la **figura 3.11**. Suponemos que tenemos una expresión regular α con “ n ” operadores.

Vamos a construir el autómata M tal que $L(M)=L(\alpha)$. Distinguimos tres casos correspondientes a las tres formas posibles de expresar α en función de otras expresiones regulares con menos de n operadores:

1. $\alpha = \alpha_1 + \alpha_2$ tal que $op(\alpha_1), op(\alpha_2) < n$
2. $\alpha = \alpha_1 \cdot \alpha_2$ tal que $op(\alpha_1), op(\alpha_2) < n$
3. $\alpha = (\alpha_1)^*$ tal que $op(\alpha_1) = n-1$ $\alpha = \alpha_1 + \alpha_2$ tal que $op(\alpha_1), op(\alpha_2) < n$

A partir de **M1** y **M2** construimos otro autómata **M** (la unión) que acepta el mismo lenguaje que es el que se muestra en la **figura 3.12**.

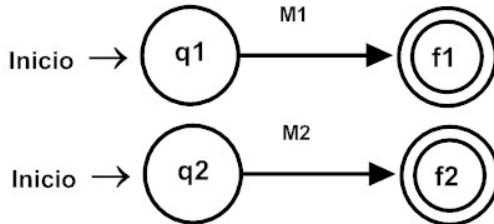


Figura 3.11 Autómatas de expresiones regulares M1 y M2

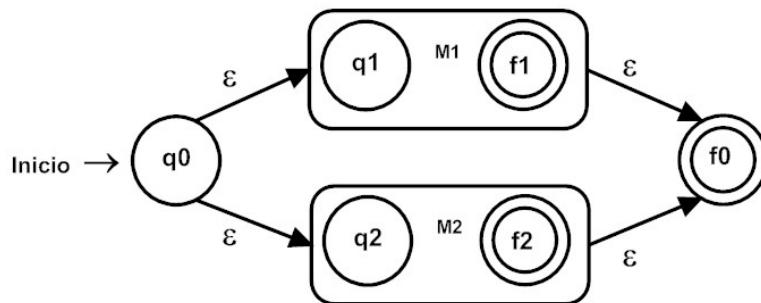


Figura 3.12 Autómatas de expresiones regulares M1 + M2

UNIDAD III • CONCEPTOS DE GRAMÁTICAS

$$\alpha = \alpha_1 \cdot \alpha_2$$

. α_2 tal que $op(\alpha_1), op(\alpha_2) < n$

A partir de **M1** y **M2** construimos otro autómata **M** (la concatenación), el autómata que acepta el mismo lenguaje es el que se muestra en la **figura 3.13**

$$\alpha = (\alpha_1)^* \quad \text{tal que } op(\alpha_1) = n-1$$

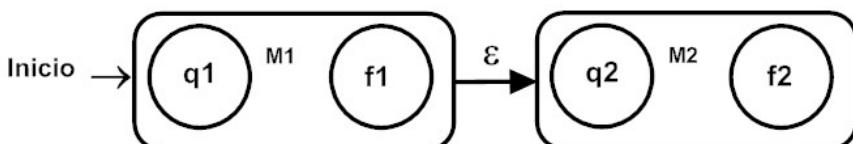


Figura 3.13 Autómatas de expresiones regulares $M1M2$

A partir de **M1** construimos otro autómata **M** (la clausura), el autómata que acepta el mismo lenguaje es el mostrado en la **figura 3.14**:

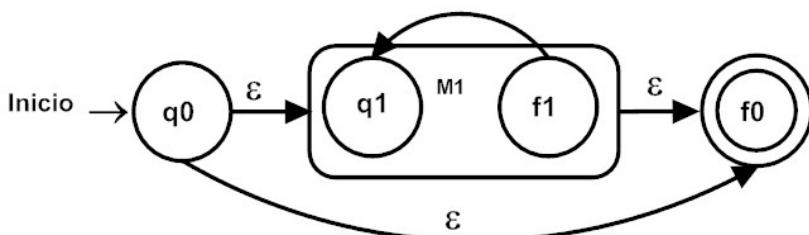


Figura 3.14 Autómatas de expresiones regulares $M1^*$

Ejemplo: AF construido para la expresión regular $01^* + 1$:

M1 representa el autómata para la expresión regular 0

M2 representa el autómata para la expresión regular 1^*

M3 representa el autómata para la expresión regular 1

En el Autómata final se integran simultáneamente los autómatas para la concatenación (0 con 1^*) y la suma de expresiones regulares $01^* + 1$ Misma que se muestra en la **figura 3.15**

Si L es un lenguaje aceptado por un autómata finito M entonces existe una expresión regular α tal que $L = L(M) = L(\alpha)$.

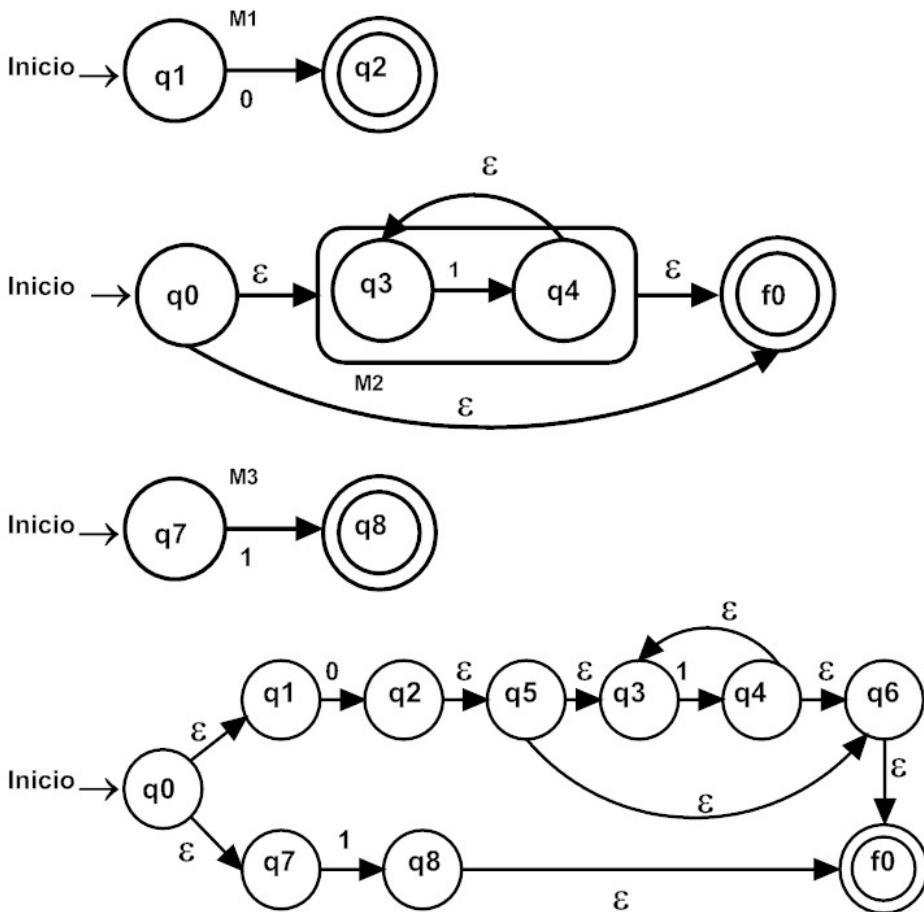


Figura 3.15

Podemos suponer que el autómata finito M no tiene ϵ -transiciones. Si las tuviera, podemos encontrar autómata equivalente sin ϵ -transiciones

Sea $M = (Q, \Sigma, f, q_0, F)$. A partir de su diagrama de transición podemos obtener un sistema de ecuaciones de expresiones regulares (ecuaciones características del autómata):

A cada nodo q_i le corresponde una ecuación y cada estado se puede considerar como una incógnita x_i de la ecuación. La ecuación para el estado q_i tiene en el primer

UNIDAD III • CONCEPTOS DE GRAMÁTICAS

miembro el estado q_i, x_i , y en el segundo miembro una suma de términos, de forma que por cada arco del diagrama de la forma $q_i \rightarrow q_j$ tenemos un término ax_j .

Si el estado q_i es final, añadimos el término α al segundo miembro.

Si el estado q_0 es final, añadimos el término ϵ al segundo miembro para x_0

Cada incógnita para q_i, x_i , representa el conjunto de palabras que llevan de q_i a un estado final.

Resolviendo el sistema de las ecuaciones tendremos soluciones de la forma $x_i = \alpha_i$, donde α_i es una expresión regular sobre el alfabeto Σ

El lenguaje descrito por esta expresión regular es:

$$L(\alpha_i) = \{w \in \Sigma^* \mid (q_i, w) \vdash^* (q_F, \epsilon), q_F \in F\}$$

El método para obtener una expresión regular α a partir de un AF es el siguiente:

1. Obtener las ecuaciones características del autómata.
2. Resolver el sistema de ecuaciones
3. $\alpha \leftarrow$ solución para el estado inicial.

Para comprobar que es válido hay que probar que se cumple (1) para toda solución $x_i = \alpha_i$ del sistema de ecuaciones, y en particular la solución para el estado inicial es la expresión regular correspondiente al autómata. No es necesario resolver todas las incógnitas, sólo necesitamos despejar la incógnita correspondiente al estado inicial x_0

Ejemplo. Sea el AF (Autómata Finito) según la **figura 3.16**

Ecuaciones características:

$$1. x_0 = 0x_0 + 1x_1 + 1$$

$$2. x_1 = 0x_0 + 1x_2$$

$$3. x_2 = 0x_2 + 1x_1 + 1$$

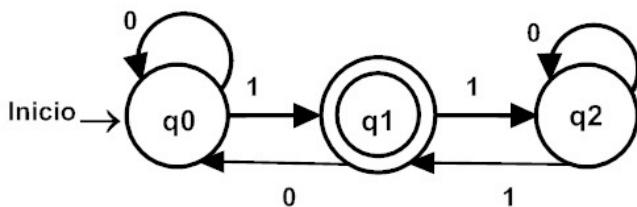


Figura 3.16

Resolvemos aplicando la regla de inferencia $X = \alpha X + \beta \leftrightarrow X = \alpha^* \beta$

Comenzando por la ecuación (3): $x_2 = 0^*(1x_1 + 1) = 0^*1x_1 + 0^*1$

Sustituyendo en (2): $x_1 = 0x_0 + 10^*1x_1 + 10^*1 = (10^*1)*(0x_0 + 10^*1) = (10^*1)*0x_0 + (10^*1)*10^*1$

Sustituyendo en (1):

$$\begin{aligned}
 x_0 &= 0x_0 + 1[(10^*1)*0x_0 + (10^*1)*10^*1] + 1 = 0x_0 + 1(10^*1)*0x_0 + \\
 &1(10^*1)*10^*1 + 1 = (0 + 1(10^*1)*0)x_0 + 1(10^*1)*10^*1 + 1 = \\
 &(0 + 1(10^*1)*0)*(1(10^*1)*10^*1 + 1) = \\
 &= (0 + 1(10^*1)*0)*1[(10^*1)*(10^*1) + \epsilon] = \\
 &= (0 + 1(10^*1)*0)*1(10^*1)^* \text{ expresión regular que describe el lenguaje } L(M).
 \end{aligned}$$

(Aplicando las propiedades: $(ab^*b + a) = a(b^*b + \epsilon) = ab^*$)

3.5 Otros conceptos relacionados

Gramáticas afijas

Están relacionadas con las gramáticas con atributos por la izquierda. Se describe la construcción mecánica de un traductor predictivo similar al que construye el algoritmo. La impresión de que el análisis sintáctico descendente permite una mayor flexibilidad en la traducción resultó ser falsa, ya que se demostró que un esquema de traducción basado en una gramática LL (Left to Left, de izquierda a izquierda, análisis que se describirá con más detalle en la siguiente unidad), se puede simular durante el análisis sintáctico LR. (Left to Right, de izquierda a derecha).

Independientemente, se han utilizado terminales marcadores para garantizar que los valores de los atributos heredados aparezcan en una pila durante el análisis sintáctico ascendente. Las posiciones en los lados derechos de las producciones donde se puede insertar los no terminales marcadores sin perder la propiedad LR.

Gramática aumentada

Si G es una gramática con símbolo inicial S , entonces G' , la gramática aumentada para G , es G con un nuevo símbolo inicial $S' \rightarrow S$. El propósito de esta nueva producción inicial es indicar al analizador cuando debe detener el análisis sintáctico y anunciar la aceptación de la cadena. La aceptación se produce cuando, y solo cuando, el analizador está a punto de reducir por $S' \rightarrow S$.

Ejemplo: considérese la gramática de expresiones aumentada:

$$\begin{aligned} E &\rightarrow E \\ E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow (E) \mid id \end{aligned}$$

Si A es el conjunto de un elemento $\{ [E \rightarrow .E] \}$, entonces cerradura (A) contiene los elementos

$$\begin{array}{ll} E \rightarrow .E & T \rightarrow .F \\ E \rightarrow .E + T & F \rightarrow .(E) \\ E \rightarrow .T & F \rightarrow .id \\ T \rightarrow .T * F & \end{array}$$

Gramáticas con atributos

Es una definición dirigida por la sintaxis en la que las funciones en las reglas semánticas no pueden tener efectos colaterales. En una gramática independiente del contexto, cada símbolo gramatical tiene un conjunto de atributos asociados, dividido en dos subconjuntos llamados atributos sintetizados y heredados de dicho símbolo gramatical.

Un atributo puede representar cualquier cosa: una cadena, un número, un tipo, una posición de memoria. El valor de un atributo en un nodo de árbol de análisis sintáctico se define mediante una regla semántica asociada a la producción utilizada en dicho nodo. El valor de un atributo sintetizado en un nodo, se calcula a partir de los valores de los atributos de los hijos de dicho nodo en el árbol de análisis sintáctico; el valor de un atributo heredado se calcula a partir de los valores en los atributos de los hermanos y el padre de dicho nodo.

Un árbol de análisis sintáctico que muestre los valores de los atributos en cada nodo se denomina un árbol de análisis sintáctico con anotaciones. El proceso de calcular los valores de los atributos en los nodos se denomina anotar o decorar el árbol de análisis sintáctico.

En una definición dirigida por la sintaxis, cada producción gramatical $A \rightarrow a$ tiene asociado un conjuntos de reglas semánticas de la forma $b := f(c_1, c_2, \dots, c_k)$, donde f es una función, y, o bien

1. b es un atributo sintetizado de A y c_1, c_2, \dots, c_k son atributos que pertenecen a los símbolos gramaticales de la producción, o bien
2. b es un atributo heredado de alguno de los símbolos gramaticales del lado derecho de la producción, y c_1, c_2, \dots, c_k son atributos que pertenecen a los símbolos gramaticales de la producción.

UNIDAD III • CONCEPTOS DE GRAMÁTICAS

Ejemplo: programa para calculadora de escritorio

PRODUCCIÓN	REGLAS SEMANTICAS
$L \rightarrow E_n$	Print (E.val)
$E \rightarrow E_1 + T$	$E.val ::= E_1.val + T.val$
$E \rightarrow T$	$E.val ::= T.val$
$T \rightarrow T_1 * F$	$T.val ::= T_1.val * F.val$
$T \rightarrow F$	$T.val ::= F.val$
$F \rightarrow (E)$	$F.val ::= E.val$
$F \rightarrow \text{digito}$	$F.val ::= \text{digito}.valex$

Gramáticas de operadores

Estas gramáticas tienen la propiedad de que de que ningún lado derecho de la producción es ϵ ni tiene dos no terminales adyacentes.

Ejemplo: la siguiente gramática

$$\begin{aligned} E &\rightarrow EAE \mid (E) \mid -E \mid \text{Id} \\ A &\rightarrow + \mid - \mid * \mid / \mid \uparrow \end{aligned}$$

No es una gramática de operadores porque el lado derecho EAE tiene dos no terminales consecutivos, sin embargo si se sustituye cada una de sus alternativas por A, se obtiene la siguiente gramática de operadores:

$$E \rightarrow E + E \mid E - E \mid E^* E \mid E / E \mid E \uparrow E \mid (E) \mid - E \mid \text{id}_b$$

Precedencia de Operador

Inconvenientes

- ✓ Es difícil de manejar componentes léxicos con dos precedencias distintas, como el signo menos (unario y binario)
- ✓ No se puede tener la seguridad de que el analizador acepta exactamente el lenguaje deseado
- ✓ Sólo una pequeña clase de gramáticas puede analizarse

Ventajas

- ✓ Sencillez
- ✓ Se pueden establecer relaciones de precedencia (* precede a +)

UNIDAD III • CONCEPTOS DE GRAMÁTICAS

Se aplican con otros analizadores para la parte que no son de operador.

El análisis recorre la entrada de izquierda a derecha y se encuentra en dos posibles estados: “Esperando un operador” y “Esperando un operando”

El análisis mantiene dos pilas

- ✓ Pila de Operadores
- ✓ Pila de Operandos

Cuando un operador en la cima de su pila es de más prioridad que el siguiente de la pila, entonces el pivote consiste en ese operador junto a los dos operandos situados más arriba de la pila de operandos.

Entrada: $\text{Id} \rightarrow \cdot \text{Id} * \cdot \text{Id}$

Gramática $E := E + E | E * E | (E) | \text{Id}$

La gramática es ambigua pero este tipo de análisis proporciona una única derivación.

Entrada	Pila de operadores	Pila de operandos
$\text{Id}_a + \cdot \text{Id}_b * \cdot \text{Id}_c$	\emptyset	\emptyset
$+ \cdot \text{Id}_b * \cdot \text{Id}_c$	\emptyset	Id_a
$\text{Id}_b * \cdot \text{Id}_c$	$+$	Id_a
$* \text{Id}_c$	$+$	$\text{Id}_b \text{Id}_a$
Id_c	$*+$	$\text{Id}_b \text{Id}_a$
\emptyset	$*+$	$\text{Id}_c \text{Id}_b \text{Id}_a$

Se definen tres relaciones de precedencia disjuntas

- ✓ $a < b$ si a tiene menos precedencia que b
- ✓ $a = b$ si a tiene igual precedencia que b
- ✓ $a > b$ si a tiene más precedencia que b

Algoritmo

- ✓ Sustituir todos los símbolos no terminales por un único símbolo
- ✓ Insertar $\$$ al principio y al final de la cadena de entrada
- ✓ Insertar las relaciones de precedencia en la cadena de entrada
- ✓ Mientras entrada diferente de $\$\$$ hacer
- ✓ Recorrer entrada desde la izquierda hasta encontrar $>$
- ✓ Buscar a la izquierda, a partir de ese punto, el primer $<$
- ✓ Reducir el pivote que se encuentra en el medio
- ✓ Reinsertar las relaciones de precedencia, ignorando los no terminales

UNIDAD III • CONCEPTOS DE GRAMÁTICAS

Entrada: \$ (·Id + ·Id ·) · \$

Gramática: E ::= E + E | E * E | (·E) | Id

Tabla de precedencia en la **figura 3.17**

	(Id	*	+)	\$
)			•>	•>	•>	•>
Id			•>	•>	•>	•>
*	<•	<•	•>	•>	•>	•>
+	<•	<•	<•	•>	•>	•>
(<•	<•	<•	<•	≡	
\$	<•	<•	<•	<•		≡

Figura 3.17

Análisis

Entrada

\$ <• (<• Id •> + <• Id •>) •>\$
\$ <• (<• E + <• Id •>) •> \$
\$ <• (<• E + E •>) •> \$
\$ <• (E =) •> \$

Derivación

\$ · (·E + ·Id) \$
\$ · (·E + E) \$
\$ · (E) \$
\$ · E \$

Obtención de las relaciones de precedencia:

X ± Y si existe: A ::= ...xBy... B ∈ {N ∪ λ}

X <• Y si existe: A ::= ...xB... C ∈ {N ∪ λ}
B ::= + Cy...

X •> Y si existe: A ::= ...By... C ∈ {N ∪ λ}
B ::= + ...xC

Si el operador Θ₁ tiene mayor precedencia que Θ₂ entonces hacer Θ₁•> Θ₂ y Θ₂ <• Θ₁

Si los operadores Θ₁ y Θ₂ son de igual precedencia (por ejemplo el mismo operador), entonces hacer:

Θ₁•> Θ y Θ₂•> Θ₁ si son asociativos por la izquierda
Θ₁<• Θ y Θ₂<• Θ₁ si son asociativos por la derecha

C-gramática.

Una gramática categorial o C-gramática es una basada en categorías gramaticales. Las formas léxicas y secuencias formadas a partir de ellas están etiquetadas con categorías que indican el tipo de entidad formada y sus posibilidades combinatorias (por ejemplo en una lengua nominal una secuencia de palabras puede constituir un sintagma nominal lo cual especifica con qué otro tipo de categorías puede combinarse este sintagma para formar otro sintagma mayor).

Las gramáticas categoriales se pueden definir como una estructura formal algebraica. Una gramática categorial es una quíntupla (W, C, L, X, E, CE) con las siguientes propiedades:

1. W (words) es el conjunto no vacío de formas bien formadas de la lengua (en una lengua natural W podría interpretarse como secuencias de fonemas que forman expresiones, irrespectivamente de su categoría gramatical).
2. C (categories) es el conjunto no vacío de categorías posibles. Para que este conjunto sea un conjunto de categorías aceptable se exige que si $X, Y \in C$ entonces también existan las categorías $XY \in C$ (frecuentemente denotada también como Y/X) y $YX \in C$ (frecuentemente denotada también como YX). Nótese que de lo anterior se desprende la existencia de las categorías $XY \in C$ y $YX \in C$ (sin más que intercambiar el papel de X e Y).
3. El conjunto LX (lexicón) es un conjunto $LX \subset W \times C$, este conjunto es algo diferente del lexicón convencional ya que incluye tanto palabras atómicas inanalizables como expresiones formadas a partir de ellas.
4. El conjunto R (rules) es un conjunto de reglas, generalmente formado por las siguientes dos reglas:
 - a. $\infty XY \circ B_Y \rightarrow \infty B_X$
 - b. $B_Y \circ \infty YX \rightarrow \infty B_X$

Los anteriores se aplican a cualesquiera categorías y se interpretan así: si en un lenguaje formal los elementos a la izquierda de la regla pertenecen al lexicón LX , entonces la expresión a la derecha de la regla también es parte del lexicón (es decir, del conjunto de expresiones posibles en dicho lenguaje). Se comprende que puesto que la composición puede ser por la izquierda (regla 1) o por la derecha (regla 2) se haya requerido que el conjunto C admita además de categorías X e Y las categorías XY y YX .

UNIDAD III • CONCEPTOS DE GRAMÁTICAS

El conjunto CE (*complete expresions*). Expresiones completas

Una gramática formal es un modelo matemático (más exactamente una estructura algebraica) compuesto por una serie de categorías sintácticas que se combinan entre sí por medio de unas reglas sintácticas que definen cómo se crea una categoría sintáctica por medio de otras o símbolos de la gramática. Existen varios tipos de gramáticas formales históricamente importantes:

- Las gramáticas formales categoriales (C-gramáticas) que usan un análisis de abajo a arriba y requieren el uso de etiquetas de categoría para cada secuencia formada o constituyente sintáctico propiamente dicho. Existe una única categoría superior que denota cadenas completas y válidas.
- Las gramáticas de estructuras sintagmáticas (ES-gramáticas, en inglés PS-grammars) basadas en reglas de reescritura y con un análisis de arriba abajo. Al igual que las C-gramáticas se basan en la noción de constituyente sintáctico.
- Las gramáticas asociativas (por la izquierda A-gramáticas, en inglés LA-grammars), que usa un análisis de abajo a arriba, que permiten un análisis en de complejidad lineal, aunque ignoran el concepto de constituyente sintáctico.

Los dos primeros tipos tienen puntos de conexión obvia con la noción de consistencia sintáctica y el análisis mediante árboles sintácticos. Sin embargo, los analizadores sintácticos para las oraciones formadas según ellas no pueden basarse en las reglas de generación (asimetría hablante-oyente), lo cual sugiere que no puedan ser buenos modelos de la intuición de los hablantes. Además, los modelos de lengua natural basados en ellas parecen tener una complejidad polinómica o exponencial, lo cual no parece avenirse con la velocidad con que los hablantes procesan las lenguas naturales. Por contra las A-gramáticas en general tienen complejidad lineal, simetría entre hablantes y oyentes, sin embargo, ignoran los constituyentes clásicos del análisis sintáctico. Sin embargo, siguen siendo usadas para los analizadores sintácticos usados en computación.

Por medio de estos elementos constituyentes se define un mecanismo de especificación consistente en repetir el mecanismo de sustitución de una categoría por sus constituyentes en función de las reglas comenzando por la categoría superior y

UNIDAD III • CONCEPTOS DE GRAMÁTICAS

finalizando cuando la oración ya no contiene ninguna categoría. De esta forma, la gramática puede generar o producir cada una de las cadenas del lenguaje correspondiente y solo estas cadenas.

Definición de una ES-gramática

En la definición clásica que dio Noam Chomsky en la década de 1950, una gramática formal de estructura sintagmática (*ES-gramática*) es una cuádrupla $G = (N, T, S, P)$ donde:

- N es un conjunto finito de símbolos no terminales (variables).
- T es un conjunto finito de símbolos terminales (constantes), disjunto con N .
- S es un símbolo distinguido de N , el *símbolo inicial*.
- P es un conjunto finito de reglas de producción, cada una de la forma:
$$(N \cup T)^* N (N \cup T)^* \rightarrow (N \cup T)^*$$

Donde $*$ es la clausura de Kleene. Esto es, cada regla de producción mapea de una cadena de símbolos a otra, donde la primera cadena contiene al menos un símbolo no terminal. En el caso de que la segunda cadena sea la cadena vacía, para evitar confusión se la denota con una notación especial (usualmente ϵ , \emptyset ó bien λ).

El alfabeto de la gramática es entonces el conjunto $\Sigma = N \cup T$

EJERCICIO 22.

Sea G una gramática con las siguientes producciones:

S	$\rightarrow ASB \mid \epsilon$
A	$\rightarrow aAb \mid \epsilon$
B	$\rightarrow bBa \mid ba$

- Da una derivación a la izquierda de la palabra aabbba.
 - Da una derivación a la derecha de la misma palabra del apartado (a).
 - Da una derivación que no sea ni a la derecha ni a la izquierda de la misma palabra del apartado (a).
 - Describe $L(G)$.
-

UNIDAD III • CONCEPTOS DE GRAMÁTICAS

EJERCICIO 23.

Sea G una gramática independiente del contexto cuyo conjunto de reglas es el siguiente:

$$\begin{array}{ll} S & \rightarrow ASB \mid \epsilon \\ A & \rightarrow ab \mid \epsilon \\ B & \rightarrow bB \mid \epsilon \end{array}$$

- a) Da una derivación a la izquierda y una derivación a la derecha de la palabra aaabb.
 - b) Construye el árbol de derivación de alguna de las derivaciones anteriores.
 - c) Demuestra que G es ambigua.
 - d) Construye una gramática no ambigua equivalente a G .
 - e) Describe $L(G)$. ¿Es regular este lenguaje?
-

UNIDAD IV

“Las matemáticas son el alfabeto con el cual Dios ha escrito el Universo”, Frase de Galileo Galilei pero, para muchos las matemáticas suelen ser complejas por ejemplo: El chiste en que niño le dice a su papá: ¡Papá, papá! ¡Me haces el problema de matemáticas?

No hijo, no estaría bien.

Bueno, intétalo de todas formas.

NORMALIZACIÓN DE GRAMÁTICAS

“Normaliza una Gramática Libre de Contexto generando su autómata relacionado, por medio de operaciones; para la creación de traductores en empresas de software de base específico”

TEMAS:

- 4.1 Obtener la forma positiva de una gramática.
- 4.2 Hacer admisible y arborescente una gramática.
- 4.3 Obtener la Forma Normal de Chomsky de una gramática libre de contexto.
- 4.4 Obtener la Forma Normal de Greibach de una gramática libre de contexto y generar su Autómata de Pila.

Introducción a la normalización de las gramáticas:

Recordemos que en el mundo de las gramáticas, se utilizan varios conceptos, el concepto de “gramática positiva” que es el hecho de analizar las reglas de producción que definen a una gramática, para determinar cuáles de esas reglas no son necesarias en su definición, eliminándolas y dejando una gramática equivalente, esto es, que acepta el mismo lenguaje, pero definido de una forma más sencilla o simplificada y a esto se le puede dar el nombre de gramática positiva, gramática normalizada y dependiendo de las reglas que se utilizan para lograr este fin, se les conoce como normalización según quedó definido por **Chomsky**, también se le conoce a esa gramática como forma normal de **Chomsky**.

Con el mismo procedimiento de eliminar reglas de producción, es posible definir una forma normal de **Greibach**, siendo este un proceso de llevar una gramática con “n” numero de reglas de producción a otra gramática equivalente con “n-x” reglas de producción donde “n” es el número de reglas de producción de la gramática originalmente definida. “x” representa las reglas de producción que vamos a eliminar por ser innecesarias, inaccesibles o inútiles y el resultado será “n-x” reglas y serán el número menor en “x” de las reglas de producción que son las mínimas y necesarias para generar una gramática equivalente, bien definida, positiva o normalizada.

Por ejemplo, supongamos que una gramática libre de contexto originalmente esta definía con “n” reglas de producción, donde **n = 10**, esto significa que originalmente se definió con diez reglas de producción de las cuales existen dos reglas innecesarias, una inaccesible y otra regla de producción inútil, esto es, la suma de estos tres tipos de reglas, que podemos eliminar será el valor de x, de donde: $x = 2$ (reglas innecesarias) + 1 (regla inaccesible) + 1 (regla inútil), en total, **x = 4**. La gramática normalizada tendrá **10 – 4 (n – x)** cuyo resultado es: 6 reglas de producción que serán las que definan a la misma gramática de 10 reglas en sólo seis, al eliminar las cuatro reglas de producción se dice que se simplifica su definición y se hace más operable, simple y bien definida.

4.1 Obtener la forma positiva de una gramática.

“Una gramática es positiva si ninguna de sus producciones es de la forma $A \rightarrow \varepsilon$; a estas producciones se les conoce como ε -producciones.

Se demostrará que, si G es una Gramática Libre de Contexto (GLC), existe una gramática positiva G' tal que $L(G') = L(G) - \{\varepsilon\}$. Para esto, se define el núcleo de una GLC G , escrito como Kernel de G o $\text{Ker}(G)$, como el conjunto de variables A tales que $A \Rightarrow \varepsilon$. El siguiente lema proporciona un algoritmo para determinar $\text{ker}(G)$.

Lema 4.1. Sea G una GLC. Defínase, recursivamente, los siguientes conjuntos: VN_i (i Variables No terminales) como sigue:

- i) $VN_0 = \{A | A \rightarrow \varepsilon \text{ es una producción de } G\}$ (Desde la Variable No terminal cero VN_0 que es la primera, A representa otra Variable No terminal o VN o solo V que produce una variable terminal que produce la cadena vacía ε)
- ii) $VN_{i+1} = VN_i \cup \{A | A \rightarrow \infty \text{ es una producción de } G, \infty \in VN_i^*\}$ (Todas y cada una de las Variables No terminales que se unen a las variables A que producen la cadena vacía hasta el infinito)

Entonces, $\text{ker}(G) = VN_k$, si $VN_{k+1} = VN_k^*$

Demostración

Es claro que, por definición, $VN_i \subseteq \text{Ker}(G)$ para toda i . Recíprocamente, se demostrará que, si $A \in \text{Ker}(G)$, entonces $A \in VN_k$. Se hará esta demostración por inducción en el número de pasos en una derivación de ε en G .

Base ($j=1$). En este caso, $A \Rightarrow \varepsilon$, así que $A \rightarrow \varepsilon$ es una producción de G y $A \in VN_0$. Paso de inducción ($j > 1$). Supóngase que el resultado es válido para derivación con menos de j pasos.

Sea $A \Rightarrow \infty_1 \Rightarrow \infty_2 \Rightarrow \dots \Rightarrow \infty_{j-1} \Rightarrow \varepsilon$

Lo anterior es una derivación de j pasos en G . Las palabras $\infty_1, \infty_2, \dots, \infty_{j-1}$ deben consistir, únicamente, de variables, ya que no es posible eliminar terminales con las producciones de una GLC.

UNIDAD IV • NORMALIZACIÓN DE GRAMÁTICAS

Sea $\infty_1 = A_1 A_2 \dots A_r$; entonces $A_i \Rightarrow \epsilon$, $i = 1, 2, \dots, r$, en menos de j pasos. Por hipótesis de inducción, cada A_r pertenece a VN_k . Dado que G tiene entre sus producciones a $A \rightarrow A_1 A_2 \dots A_r$, y $\infty_1 \in VN_k$, se tiene que:

$A \in VN_{k+1} = VN_k$. Lo que queda demostrado (QED).

A continuación, se presenta el teorema que da la equivalencia, salvo por la palabra vacía, entre gramáticas libres de contexto y gramáticas positivas.

Teorema 4.1 Sea $L = L(G)$ para una GLC $G = (VN, VT; P; S)$, entonces existe una gramática positiva G' tal que: $L(G') = L - \{ \epsilon \}$.

Demostración

Se determina, primeramente, $\text{ker}(G)$ con ayuda del Lema 4.1. Se construye un conjunto de producciones P' como sigue. Si $A \rightarrow X_1 X_2 \dots X_n \in P$, entonces, se incluyen en P' todas las producciones $A \rightarrow \infty_1 \infty_2 \dots \infty_n$ tales que:

- i) Si X_i no pertenece a $\text{ker}(G)$, entonces $\infty_i = X_i$
- ii) Si X_i pertenece a $\text{ker}(G)$, entonces ∞_i es, o bien X_i , o bien ϵ
- iii) No todas las ∞_i son ϵ

Sea $G' = (VN, VT, P', S)$. Se afirma que, para toda $A \in VN$ y $w \in VT$, $A \Rightarrow w$ si, y sólo si, $w \neq \epsilon$ y $A \Rightarrow w$.

Suficiencia. Supóngase que $A \Rightarrow w$ y $w \neq \epsilon$. Se prueba, por inducción sobre i , que $A \Rightarrow w$. La base, $i = 1$, es trivial porque $A \Rightarrow w$ debe ser una producción de P . Dado que $w \neq \epsilon$, también es una producción de P' , por construcción de este último conjunto. Para el paso de inducción, sea $i > 1$.

Entonces $A \Rightarrow X_1 X_2 \dots X_n \Rightarrow w$. Escríbase $w = w_1 w_2 \dots w_n$ tal que, para cada j , $X_j \Rightarrow w_j$ en menos de i pasos. Si $w_j \neq \epsilon$ y X_j es una variable, entonces, por la hipótesis de inducción, se tiene $X_j \Rightarrow w_j$. Si $w_j = \epsilon$, entonces X_j pertenece a $\text{ker}(G)$.

Así, $A \Rightarrow \beta_1 \beta_2 \dots \beta_n$ es una producción en P' , donde $\beta_j = X_j$. Si $w \neq \epsilon$ y $\beta_j = \epsilon$ si $w_j = \epsilon$. Dado que $w \neq \epsilon$, no todas las β_j son ϵ . Por tanto se tiene una derivación:

$$A \Rightarrow \beta_1 \beta_2 \dots \beta_n \Rightarrow w_1 \beta_2 \dots \beta_n \Rightarrow w_1 w_2 \dots w_n, \text{ en } G'.$$

UNIDAD IV • NORMALIZACIÓN DE GRAMÁTICAS

Necesidad. Supóngase que $A \Rightarrow w$. Seguramente $w \neq \epsilon$ ya que G' no tiene ϵ -producciones. Se prueba, por inducción sobre i , que $A \Rightarrow w$. Para la base, $i = 1$, obsérvese que $A \rightarrow w$ es una producción en G' . Debe haber una producción $A \rightarrow^\infty$ en P tal que, si se cancelan, en ∞ , algunas variables pertenecientes a $\text{ker}(G)$, queda w . Entonces, hay una derivación $A \Rightarrow^\infty \Rightarrow w$, donde la derivación $\infty \Rightarrow w$, involucra derivar ϵ a partir de los símbolos de ∞ , pertenecientes a $\text{ker}(G)$, que se cancelaron para obtener w .

Para el paso inductivo, sea $i > 1$. Entonces $A \Rightarrow X_1 X_2 \dots X_n \Rightarrow w$. Debe haber una producción $A \rightarrow^\infty$ en P tal que $X_1 X_2 \dots X_n$ se encuentra con la cancelación de algunos símbolos de β pertenecientes a $\text{ker}(G)$. Así $A \Rightarrow X_1 X_2 \dots X_n$. Escríbase $w = w_1 w_2 \dots w_n$ tal que, para cada j , $X_j \Rightarrow w_j$ con menos de i pasos. Por la hipótesis de inducción, $X_j \Rightarrow w_j$. Si X_j es una variable. Ciertamente, si X_j es una terminal, entonces $w_j = X_j$, y $X_j = w_j$ es trivialmente cierta. Así, $A \Rightarrow w$.

Dado que el resultado fue probado para cualquier variable de G , es válido, en particular, para el símbolo inicial de la gramática, S . Por tanto, $S \Rightarrow w$ si, y sólo si, $S \Rightarrow w$ y $w \neq \epsilon$; por ejemplo, $L(G') = L(G) - \{\epsilon\}$. Lo que queda demostrado (QED). En vista del resultado probado en el teorema anterior, se supondrá, de aquí en adelante, que todas las gramáticas son positivas.

Ejemplo 4.3 Sea G la gramática cuyas producciones son:

$$S \rightarrow ABBA|aA$$

$$A \rightarrow \epsilon$$

$$B \rightarrow \epsilon$$

Se encontrará una gramática positiva G' equivalente a G . Para ello, se determinará, primeramente, $\text{ker}(G)$. En este caso, $V_0 = \{A, B\}$. Como $ABBA$ pertenece a V_0^* y $S \rightarrow ABBA$ es una producción de G , se tiene que $V_1 = V_0 \cup \{S\}$. Dado que V_1 incluye todas las variables de G , se tiene que $V_2 = V_1$ y, por tanto, $\text{ker}(G) = \{S, A, B\}$. Se aplica, ahora, el algoritmo del Teorema 4.1 y se obtienen las siguientes producciones para G' .

$$S \rightarrow ABBA|BBA|ABA|ABB|BA|AA|AB|BB|A|B$$

$$S \rightarrow aA|a$$

La gramática G' , así obtenida, es una gramática positiva equivalente a G'' .

UNIDAD IV • NORMALIZACIÓN DE GRAMÁTICAS

EJERCICIO 24.

Obtener la forma positiva de la siguiente gramática: (Recuerde eliminar las producciones con ϵ)

$$S \rightarrow AbC|Cd$$

$$A \rightarrow BCd|a$$

$$C \rightarrow Mq|a$$

$$M \rightarrow Qr|b$$

$$B \rightarrow b|\epsilon$$

$$Q \rightarrow Ql$$

4.2 Hacer admisible y arborescente una gramática.

GRAMÁTICAS ADMISIBLES

Se dice que una gramática positiva $G = (VN, VT, P, S)$ es admisible si:

1. Cada $A \in VN$ deriva alguna cadena terminal
2. Cada $a \in VT$ aparece en alguna cadena terminal
3. Cada derivación de una forma oracional ∞ puede completarse hasta obtener una cadena terminal.

Los siguientes lemas y teoremas prueban que siempre es posible encontrar una gramática positiva admisible equivalente a una gramática positiva dada.

Lema 4.2 Sean $G = (VN, VT, P, S)$ una gramática positiva y VN' el conjunto de variables de G que derivan alguna cadena terminal en G . Defínanse, recursivamente, los conjuntos VN_i como sigue:

1. $VN_0 = \{A | A \rightarrow w, \text{ para alguna } w \in VT'\}$
2. $VN_{i+1} = VN_i \cup \{A | A \xrightarrow{\infty}, \text{ para alguna } \infty \in (VN_i \cup VT)^*\}$

Entonces, $VN' = VN_k$, si $VN_{k+1} = VN_k$, para alguna k .

Demostración

Es claro que, por definición, $VN_i \subseteq VN'$ para cada i . Se probará, entonces, por inducción sobre el número de pasos de una derivación $A \Rightarrow w$ que $VN' \subseteq VN_k$, si $VN_{k+1} = VN_k$, para alguna k .

Base ($j=1$). Si la derivación consta, únicamente, de un paso, entonces $A \rightarrow w$ es una producción de G y A pertenecen a $VN_0 \subseteq VN_k$.

Paso de inducción ($j > 1$). Sea $A \rightarrow X_1 X_2 \dots X_n \Rightarrow w$ una derivación de j pasos. Puede escribirse, entonces, $w = w_1 w_2 \dots w_n$, donde $X_i \rightarrow w_i$, $i = 1, 2, \dots, n$, en menos de j pasos.

Por hipótesis por inducción, aquellas X_i que son variables pertenecen a VN_k . Dado que $A \rightarrow X_1 X_2 \dots X_n$, es una producción de G y $X_1 X_2 \dots X_n \in (VN_k \cup VT)^*$, se tiene que $A \in VN_{k+1} = VN_k$. Así, $VN' \subseteq VN_k$ y, por tanto, $VN' = VN_k$. Lo que queda demostrado (QED).

UNIDAD IV • NORMALIZACIÓN DE GRAMÁTICAS

Lema 4.3 Sea $G = (VN, VT, P, S)$ una gramática positiva con $L(G) \neq \emptyset$. Es posible encontrar, entonces, una gramática positiva $G' = (VN', VT, P', S')$ tal que $L(G) = L(G')$ y para cada $A \in VN'$ existe $w \in VT'$ para la cual $A \Rightarrow w$.

Demostración

Calcúlese VN' con el algoritmo del lema anterior y sea P' el conjunto de producciones cuyos símbolos pertenecen a $VN' \cup VT'$. Es claro, por esta construcción, que si $A \in VN'$, entonces $A \Rightarrow w$, para alguna w . Como toda derivación en G' es una derivación de G , se tiene que $L(G') \subseteq L(G)$. Supóngase, ahora, que existe alguna variable en $VN-VN'$ o una producción en $P-P'$. Pero, entonces, hay una variable en $VN-VN'$ que deriva una cadena terminal, lo cual contradice la definición de VN' . Por tanto, no existe tal w y $L(G) = L(G')$. Lo que queda demostrado.

Lema 4.4 Dada una gramática positiva $G = (VN, VT, P, S)$, puede encontrarse una gramática positiva equivalente $G' = (VN', VT', P', S)$ tal que, para cada $X \in VN' \cup VT'$, existen $\infty, \beta \in (VN' \cup VT')^*$ para las cuales $S \Rightarrow \infty X \beta$

Demostración

El conjunto $VN' \cup VT'$ de símbolos que aparecen en formas oracionales de G , se construye con el siguiente proceso iterativo. Colóquese S en VN' . Si A es una variable ya colocada en VN' y sus producciones son $A \rightarrow \infty_1 | \infty_2 | \dots | \infty_k$, entonces colóquense en VN' las variables que aparecen en las ∞_i y en VT' , los terminales que aparecen en ellas. El conjunto P' de producciones de G' consta de aquellas producciones de P que involucran, únicamente, a símbolos en $VN' \cup VT'$. Lo que queda demostrado.

Si se aplica, primeramente, el Lema 4.2, después el Lema 4.3 y, por último, el Lema 4.4 es posible transformar una gramática positiva en una gramática positiva admisible. Vale la pena hacer la observación de que, si se aplica, en primer lugar, el lema 4.4 y, a continuación, los lemas 4.2 y 4.3, la gramática obtenida puede no ser admisible.

Teorema 4.3 Todo lenguaje libre de contexto (LLC) sin ϵ es generado por una gramática positiva admisible.

Demostración

Sea $L = L(G)$ un LLC no vacío sin ε . Por el Teorema 4.1, puede suponerse que $G = (VN, VT, P, S)$ es positiva. Sea $G_1 = (VN_1, VT, P_1, S)$ el resultado de aplicar los Lemas 4.2 y 4.3 a G y sea $G_2 = (VN_2, VT_2, P_2, S)$, el resultado de aplicar el lema 4.4 a G_1 . Se probará que G_2 es admisible y positiva.

- i) Sea $A \in NV_2$. Ya que la construcción del lema 4.4 no agrega variables nuevas a VN_1 , se tiene que $VN_2 \subseteq VN_1$ y, por tanto, $A \in NV_1$. Por los lemas 4.2 y 4.3 se tiene que $A \rightarrow w$, para alguna $w \in VT_2$.
- ii) Sea $a \in VT_2$. Por el lema 4.4, existen $\infty, \beta \in (VN_2 \cup VT_2)^*$ tales que $S \Rightarrow^\infty a\beta$. Si ∞, β involucran variables, por los lemas 4.2 y 4.3, éstas derivan cadenas terminales, así que existen $w_1, w_2 \in VT_2^*$ tales que $\infty \Rightarrow^\infty w_1, \beta \Rightarrow^\infty w_2$, en G_2 . Por tanto, $S \Rightarrow^\infty a\beta \Rightarrow^\infty w_1aw_2$, en G_2 , y aparece en una cadena terminal. Si ∞, β no involucran variables, claramente a aparece en una cadena terminal.
- iii) Sea $\infty \in (VN_2 \cup VT_2)^*$ tal que $S \Rightarrow^\infty \infty$. Por los argumentos usados en el inciso anterior, debe existir $w \in VT_2^*$ tal que $\infty \Rightarrow^\infty w$, en G_2 . Por tanto, cada derivación de una forma oracional en G_2 puede completarse a una derivación de una cadena terminal en G_2 .

De i), ii), iii) se sigue que G_2 es admisible y, como las construcciones de los Lemas 4.2, 4.3 y 4.4 no agregan producciones, se tiene que $\text{ker}(G_2) = \emptyset$ por ejemplo G_2 es positiva. Lo que queda demostrado.

Ejemplo 4.6 Considérese la gramática cuyas producciones son

$$S \rightarrow BA|a$$

$$A \rightarrow a$$

Si se aplican los lemas 4.2 y 4.3 se obtiene lo siguiente

$$VN_0 = \{S, A\} \quad (VN_0 \cup VT)^* = \{S, A, a\}^*$$

$VN_1 = \{S, A\} \cup VN_0 = VN_0$, ya que las únicas producciones en la gramática cuyo lado derecho es una cadena de $\{S, A, a\}^*$ son aquéllas que tienen a “S” o bien a “A” en su lado izquierdo.

Así, el nuevo conjunto de producciones queda como sigue:

$$S \rightarrow a$$

$$A \rightarrow a$$

UNIDAD IV • NORMALIZACIÓN DE GRAMÁTICAS

Si se aplica el Lema 4.4 a la gramática obtenida arriba, se llega a

$$S \rightarrow a$$

Si, por el contrario, se aplica, en primer lugar, el Lema 4.4, se obtiene el siguiente conjunto de producciones:

$$S \rightarrow BA|a \quad A \rightarrow a$$

Aplicando, ahora, a esta gramática (que es la dada originalmente) los Lemas 4.2 y 4.3 se obtiene como nuevo conjunto de producciones a

$$S \rightarrow a \quad A \rightarrow a$$

Como puede verse, A no aparece en ninguna forma oracional de la gramática; por tanto, ésta no es admisible.

GRAMÁTICAS ARBORESCENTES.

Sea $G = (VN, VT, P, S)$ una gramática; se dice que G es arborescente si no contiene producciones de la forma $A \rightarrow B$. A este tipo de producciones se les llama producciones unitarias.

El siguiente teorema demuestra que todo lenguaje libre de contexto puede generarse a partir de una gramática positiva, admisible y arborescente.

Teorema 4.4 Todo LLC sin ϵ es generado por una gramática positiva, admisible y arborescente.

Demostración

Sea L un LLC sin ϵ generado por una gramática $G = (VN, VT, P, S)$; por el Teorema 4.1 puede suponerse que G es positiva.

Supónganse que $A \Rightarrow B$, para algunas $A, B \in VN$. Esto puede verificarse fácilmente, ya que G es una gramática positiva, y sucede que:

$A \Rightarrow B_1 \Rightarrow B_2 \Rightarrow \dots \Rightarrow B_n \Rightarrow B$, en G, y alguna variable aparece dos veces en la sucesión, es posible encontrar una sucesión más corta de producciones unitarias que lleva $A \Rightarrow B$; así que basta con considerar, sólo, aquellas sucesiones de producciones unitarias que no repiten ninguna de las variables G.

UNIDAD IV • NORMALIZACIÓN DE GRAMÁTICAS

Constrúyase, ahora, un nuevo conjunto de P' de producciones a partir de P , incluyendo, en primer lugar, todas las producciones no unitarias de P , agréguese a P' todas las producciones de la forma: $A \Rightarrow \infty$

Se tiene, ahora, una gramática modificada $G' = (VN, VT, P', S)$. Es claro que, si $A \Rightarrow \infty$ es una producción en P' , entonces $A \Rightarrow \infty$, en G . Así que, si $S \Rightarrow w$, en G' , entonces $S \Rightarrow w$, en G ; por ejemplo, $L(G') \subseteq L(G)$.

Supónganse que $w \in L(G)$ y considérese una derivación izquierda de w en G , por ejemplo: $S = \infty_0 \Rightarrow \infty_1 \Rightarrow \dots \Rightarrow \infty_n = w$

Si, para $0 \leq i < n$, $\infty_i \Rightarrow \infty_{i+1}$ en G , por una producción no unitaria, entonces $\infty_i = \infty_{i+1}$ en G' . Supóngase, ahora, que: $\infty_i \Rightarrow \infty_{i+1} \Rightarrow \dots \infty_j$, en G ,

Por producciones unitarias y que $\infty_{i-1} \Rightarrow \infty_i$ y $\infty_j \Rightarrow \infty_{j+1}$, en G , por producciones no unitarias, o que $i=0$. Entonces, $\infty_i, \infty_{i-1}, \dots, \infty_j$ son todas de la misma longitud y, dado que la derivación izquierda, el símbolo reescrito en cada una de éstas debe ocupar, siempre, la misma posición. Pero, en este caso, se tiene que $\infty_i = \infty_{j+1}$, en G' , por una producción en $P'-P$; así que, $S \Rightarrow w$, en G' y, por tanto, $L(G) = L(G')$. Claramente, G' es positiva y arborescente. Si se usan los Lemas 4.2, 4.3 y 4.4, para transformar a G' en una gramática admisible, no se agregan producciones, así que, el resultado de aplicar estas construcciones a G' es una gramática que satisface el teorema. QLD.

EJERCICIO 25.

De la gramática definida en el ejercicio anterior (24), obtener la forma arborescente de la gramática obtenida como positiva.

4.3 Obtener la Forma Normal de Chomsky de una gramática libre de contexto.

Cuando se obtiene la forma normal de una gramática se le conoce como normalización de gramáticas. Aun cuando las gramáticas libres de contexto tienen una forma muy simple, es deseable normalizarlas, entre otras razones, para facilitar el análisis sintáctico en el diseño de compiladores. En esta parte, se verán dos de las formas normales, para GLC, más importantes.

Forma Normal de Chomsky

Se dice que una gramática está en forma normal de Chomsky (FNC). Si todas sus producciones son de la forma $A \rightarrow a$ o bien $A \rightarrow BC$. El siguiente teorema demuestra que siempre es posible encontrar, para un LLC (Lenguaje Libres de contexto) sin ε , una gramática en FNC que lo genera.

Teorema 4.5 Forma Normal de Chomsky (FNC), todo LLC sin ε es generado por una gramática positiva en la cual todas las producciones son de la forma $A \rightarrow a$ o $A \rightarrow BC$.

Demostración

Sea $G = (VN, VT, P, S)$ una GLC que genera un LLC sin ε . Por el Teorema 4.4 puede suponerse que G es positiva y arborescente; de este modo, si alguna producción tiene un único símbolo en el lado derecho, ese símbolo es un terminal y la producción está ya en una forma permitida.

Ahora, considérese una producción en P , de la forma $A \rightarrow X_1 X_2 \dots X_m$, $m \geq 2$. Si X_i es un terminal, i intodúzcase una nueva variable C_a y una producción $C_a \rightarrow a$, que está en una forma permitida; entonces, reemplácese X_i por C_a . Sean VN' el nuevo conjunto de variables y P' el nuevo conjunto de producciones; considérese la gramática $G_1 = (VN', VT, P', S)$. Es claro que si $\alpha \Rightarrow \beta$ en G , entonces $\alpha \Rightarrow \beta$ en G_1 ; así, $L(G) \subseteq L(G_1)$. Ahora, se prueba, por inducción sobre el número de pasos en una derivación, que si $A \Rightarrow w$, en G_1 , para $A \in V$, $w \in T^*$, entonces $A \Rightarrow w$, en G .
Base ($k = 1$). En este caso, el resultado es trivial.

Paso de inducción ($k > 1$). Supóngase que el resultado es cierto para derivaciones de menos de k pasos y sea:

UNIDAD IV • NORMALIZACIÓN DE GRAMÁTICAS

$A \Rightarrow w$, en G_1 , una derivación de k pasos. El primer paso debe ser de la forma $A \rightarrow B_1B_2\dots B_m$, $m \geq 2$; puede escribirse, entonces, $w = w_1w_2\dots w_m$, donde $B_i \Rightarrow w_i$, en G_1 , $1 \leq i \leq m$. Si B_i es C_{ai} , para algún terminal a_i , entonces w_i debe ser a_i . Por la construcción de P' debe haber una producción $A \rightarrow X_1X_2\dots X_m$, en P , donde: $X_i = B_i$ si $B_i \in V$ y $X_i = a_i$ si $B_i \in V - V$. Para aquellas $B_i \in V$, se sabe que la derivación $B_i \Rightarrow w_i$, en G' se lleva menos de k pasos, así que, por hipótesis de inducción, $X_i = w_i$ en G . De aquí se sigue, entonces, que $A \Rightarrow w$ en G .

Se ha probado el resultado intermedio de que todo LLC puede ser generado por una GLC donde cada producción es, o bien de la forma $A \rightarrow a$, o bien de la forma $A \rightarrow B_1B_2\dots B_m$, $m \geq 2$. Nótese que esta gramática aún no está en FNC.

Sea $G_1 = (VN', T, P', S)$ una gramática con producciones de la forma indicada arriba. Se modifica G_1 , agregando algunos símbolos adicionales a V' y reemplazando algunas producciones de P' . Para cada producción $A \rightarrow B_1B_2\dots B_m$ en P' , con $m \geq 3$, se crean nuevas variables D_1, D_2, \dots, D_{m-2} y se reemplazan $A \rightarrow B_1B_2\dots B_m$ por el conjunto de producciones: $\{A \rightarrow B_1D_1, D_1 \rightarrow B_2D_2, \dots, D_{m-3} \rightarrow B_{m-2}D_{m-2}, D_{m-2} \rightarrow B_{m-1}B_m\}$

Sean V'' el nuevo conjunto de variables y P'' el nuevo conjunto de producciones. Sea $G_2 = (V'', T, P'', S)$; G_2 está en FNC. Es claro que si $A \Rightarrow w$, en G_1 , entonces $A \Rightarrow w$, en G_2 ; así que, $L(G_1) \subseteq L(G_2)$. Pero, también es cierto que $L(G_2) \subseteq L(G_1)$, como puede demostrarse en, esencialmente, la misma forma en que se probó que $L(G_1) \subseteq L(G)$, lo que queda demostrado.

Ejemplo 4.7 Sea G la gramática cuyas producciones son

$$S \rightarrow aAb|B$$

$$A \rightarrow aA|a$$

$$B \rightarrow bB|b$$

Se encontrará una gramática equivalente en FNC

Las producciones $A \rightarrow a$ y $B \rightarrow b$ se encuentran ya en una de las formas permitidas, así que se introducen en las nuevas variables C_a y C_b , y las nuevas producciones $C_a \rightarrow a$ y $C_b \rightarrow b$, y se sustituyen a y b por C_a y C_b , respectivamente, en los lados derechos de las producciones correspondientes.

UNIDAD IV • NORMALIZACIÓN DE GRAMÁTICAS

El nuevo conjunto de producciones es:

$$S \rightarrow C_a A C_b B$$

$$A \rightarrow C_a A | a$$

$$B \rightarrow C_b B | b$$

$$C_a \rightarrow a$$

$$C_b \rightarrow b$$

Por último, se introducen dos nuevas variables, D1 y D2, y tres nuevas producciones:

$$S \rightarrow C_a D_1$$

$$D_1 \rightarrow A D_2$$

$$D_2 \rightarrow C_b B$$

De este modo, el nuevo conjunto de producciones para la gramática inicial es:

$$S \rightarrow C_a D_1$$

$$D_1 \rightarrow A D_2$$

$$D_2 \rightarrow C_b B$$

$$A \rightarrow C_a A | a$$

$$B \rightarrow C_b B | b$$

$$C_a \rightarrow a$$

$$C_b \rightarrow b$$

Claramente, la gramática obtenida está en forma normal de Chomsky

EJERCICIO 26.

De la gramática definida en el ejercicio anterior (25), obtener la forma normal de Chomsky (FNC de la gramática obtenida como arborescente).

4.4 Forma Normal de Greibach de una gramática libre de contexto y generar su Autómata de pila.

Forma normal de Greibach.

Se dice que una gramática está en forma normal de Greibach (FNG) si todas sus producciones son de la forma:

$A \rightarrow a^\infty$, donde ∞ es una cadena, posiblemente vacía, de variables.

Los siguientes lemas y teoremas demuestran que siempre es posible encontrar una GLC en FNC que genere a una LLC dado, que no contenga a ϵ .

Lema 4.5 Sea $G = (VN, VT, P, S)$ una GLC. Sea $A \rightarrow \infty_1 B \infty_2$, una producción en P y sea $B \rightarrow \beta_1 | \beta_2 | \dots | \beta_r$ todas las B -producciones en P . Sea $G_1 = (VN, VT, P_1, S)$ la gramática obtenida, a partir de G , suprimiendo la producción $A \rightarrow \infty_1 B \infty_2$ de P y agregando las producciones $A \rightarrow \infty_1 B_1 \infty_2 | \infty_1 B_2 \infty_2 | \dots | \infty_1 B_r \infty_2$.

Entonces $L(G) = L(G_1)$.

Demostración

Es claro que si $A \rightarrow \infty_1 B \infty_2$ se usa en una derivación en G_1 , entonces $A \Rightarrow \infty_1 B \infty_2 \Rightarrow \infty_1 B_i \infty_2$ puede usarse en G ; así que $L(G_1) \subseteq L(G)$. Dado que $A \rightarrow \infty_1 B \infty_2$ es la única producción en G que no está en G_1 , si es usada en alguna derivación en G , la variable B debe reescribirse, en algún paso posterior, usando una producción de la forma $B \rightarrow \beta_i$; estos dos pasos pueden reemplazarse por el único paso $A \Rightarrow \infty_1 B_i \infty_2$, en G_1 . Lo que queda demostrado.

Lema 4.6 Sea $G = (VN, VT, P, S)$ una GLC. Sea $A \rightarrow A^\infty_1 | A^\infty_2 | \dots | A^\infty_r$ el conjunto de A -producciones cuyo lado derecho inicia con A y sean $A \rightarrow \beta_1 | \beta_2 | \dots | \beta_s$ las restantes A -producciones. Sea $G_1 = (VN \cup \{B\}, VT, P, S)$ la GLC obtenida al agregar la variable B a V y reemplazar todas las A -producciones por las producciones:

- | | | | |
|-----|--------------------------|----------------------------|-------------------|
| i) | $A \rightarrow \beta_i$ | $A \rightarrow \beta_i B$ | $1 \leq i \leq s$ |
| ii) | $B \rightarrow \infty_i$ | $B \rightarrow \infty_i B$ | $1 \leq i \leq r$ |

Entonces, $L(G) = L(G_1)$.

UNIDAD IV • NORMALIZACIÓN DE GRAMÁTICAS

Demostración

En una derivación izquierda, una sucesión de producciones de la forma $A \rightarrow A^\infty_i$ debe terminar, en algún momento, con una producción de la forma $A \rightarrow \beta_j$. La sucesión de reescrituras

$$A \Rightarrow A^\infty_{i1} \Rightarrow A^\infty_{i2} \dots \Rightarrow \dots \Rightarrow A^\infty_{ip} \dots \infty_{i2} \infty_{i1} \Rightarrow \beta_j \infty_{ip} \dots \infty_{i2} \infty_{i1}$$

en G , puede reemplazarse en G_1 por

$$A \Rightarrow \beta_j B \Rightarrow \beta_j \infty_{ip} B \Rightarrow \dots \Rightarrow \beta_j \infty_{ip} \dots \infty_{i2} \infty_{i1} B \Rightarrow \beta_j \infty_{ip} \dots \infty_{i2} \infty_{i1}$$

Claramente, la trasformación inversa también es posible. Por lo tanto, $L(G_1) = L(G)$. Lo que queda demostrado.

Teorema 4.6 (Forma normal de Greibach FNG) Todo LLC sin ϵ es generado por una gramática donde todas las producciones son de la forma $A \rightarrow a^\infty$, donde ∞ es una cadena de variables, posiblemente vacía.

Demostración

Sea $G = (VN, VT, P, S)$ una gramática en FNC que genera al LLC L y supóngase que $V = \{A_1, A_2, \dots, A_m\}$. El primer paso es modificar las producciones de tal modo que, si $A_i \rightarrow A_i \delta$ es una producción, entonces $j > i$. Para lograr esto, se procede, recursivamente, como sigue:

Supóngase que se han modificado las producciones de tal modo que, para $1 \leq i \leq k$,

$A_i \rightarrow A_i \delta$ es una producción sólo si $i < j$; se procede, ahora, a modificar las A_k -producciones.

Si $A_k \rightarrow A_k \delta$ es una producción con $k > 1$, se genera un nuevo conjunto de producciones sustituyendo A_j por el lado derecho de cada A_j -producción, de acuerdo con el Lema 4.5. Repitiendo este proceso, a lo más, $k-1$ veces, se obtienen producciones de la forma $A_k \rightarrow A_j \delta$, $1 \geq k$. Las producciones con $1 = k$ se reemplazan, de acuerdo con el **Lema 4.6**, introduciendo una nueva variable B_k .

UNIDAD IV • NORMALIZACIÓN DE GRAMÁTICAS

Repetiendo el proceso anterior para cada variable original, se tienen únicamente, producciones de las formas:

- i) $A_i \rightarrow A_i\delta$ $j > i$
- ii) $A_i \rightarrow a\delta$ $a \in T$
- iii) $B_i \rightarrow \delta$ $\delta \in (V \cup \{B_1, B_2, \dots, B_{i+1}\})^*$

Nótese que el lado derecho de cualquier producción para A_m , debe iniciar con un terminal, ya que A_m es la variable con mayor índice. El símbolo inicial en el lado derecho de cualquier A_{m-1} -producciones debe ser A_m o un terminal. Si es A_m , pueden generarse nuevas producciones, reemplazando A_m por el lado derecho de las A_m -producciones, de acuerdo con el Lema 4.5; estas producciones deben tener lados derechos que inician con un símbolo terminal.

Como último paso, se examinan las producciones para las nuevas variables B_1, B_2, \dots, B_m . Dado que se inició con una gramática en FNC, es fácil probar, por inducción sobre el número de aplicaciones de los Lemas 4.5 y 4.6, que el lado derecho de cada A_i -producciones, $i \leq i \leq m$, inicia con un terminal o con $A_j A_k$, para algunos j, k . Por tanto, en la producción $A_k \rightarrow A_k^\infty$, ∞ no puede ser vacía o iniciar consiguiente, todas las B_i -producciones tienen lados derechos que inician con terminales o con alguna A_i , y una aplicación más del Lema 4.5, para cada B_i -producción, completa la construcción.

Ejemplo 4.8 Sea G la gramática cuyas producciones son:

$$\begin{aligned} A_1 &\rightarrow A_2 A_1 | c \\ A_2 &\rightarrow A_3 A_2 | a \\ A_3 &\rightarrow A_1 A_3 | b \end{aligned}$$

Se construirá una gramática equivalente en FNG. La única producción cuyo lado derecho inicia con una variable de índice menor que la variable en el lado izquierdo es $A_3 \rightarrow A_1 A_3$. Como primer paso se le reemplaza por las producciones que se obtienen de sustituir A_1 por los lados derechos de sus producciones. El nuevo conjunto de producciones queda como sigue:

$$\begin{aligned} A_1 &\rightarrow A_2 A_1 | c \\ A_2 &\rightarrow A_3 A_2 | a \\ A_3 &\rightarrow A_2 A_1 A_3 | c A_3 | b \end{aligned}$$

UNIDAD IV • NORMALIZACIÓN DE GRAMÁTICAS

En el siguiente paso se reemplaza A_2 , en el lado derecho de la producción $A_3 \rightarrow A_2A_1A_3$, por los lados derechos de las A_2 -producciones y se elimina $A_3 \rightarrow A_2A_1A_3$. Se obtiene el siguiente conjunto de producciones:

$$\begin{aligned}A_1 &\rightarrow A_2A_1|c \\A_2 &\rightarrow A_3A_2|a \\A_3 &\rightarrow A_3A_2A_1A_3|aA_1A_3|cA_3|b\end{aligned}$$

A continuación, se introduce la nueva variable B_3 y se reemplazan las A_3 -producciones, utilizando el Lema 4.6, y se obtiene el nuevo conjunto de producciones

$$\begin{aligned}A_1 &\rightarrow A_2A_1|c \\A_2 &\rightarrow A_3A_2|a \\A_3 &\rightarrow aA_1A_3|cA_3|b \\A_3 &\rightarrow aA_1A_3B_3|cA_3B_3|bB_3 \\B_3 &\rightarrow A_2A_1A_3|A_2A_1A_3B_3\end{aligned}$$

Ahora, todos los lados derechos de las A_3 -producciones inician con un terminal; por tanto, se utilizan para transformar la producción $A_2 \rightarrow A_3A_2$ en seis nuevas producciones, que inician sus lados derechos con un terminal. A su vez, las nuevas A_2 -producciones. El conjunto de producciones queda, entonces, como sigue

$$\begin{aligned}A_1 &\rightarrow aA_1A_3A_2A_1|cA_3A_2A_1|bA_2A_1|aA_1|c \\A_1 &\rightarrow aA_1A_3B_3|cA_3B_3A_2A_1|bB_3A_2A_1 \\A_2 &\rightarrow aA_1A_3A_2|cA_3A_2|bA_2|a \\A_2 &\rightarrow aA_1A_3B_3A_2|cA_3B_3A_2|bB_3A_2 \\A_3 &\rightarrow aA_1A_3|cA_3|b \\A_3 &\rightarrow aA_1A_3B_3|cA_3B_3|bB_3 \\B_3 &\rightarrow aA_1A_3A_2A_1A_3|cA_3A_2A_1A_3|bA_2A_1A_3|aA_1A_3 \\B_3 &\rightarrow aA_1A_3B_3A_2A_1A_3|cA_3B_3A_2A_1A_3|bB_3A_2A_1A_3 \\B_3 &\rightarrow aA_1A_3A_2A_1A_3B_3|cA_3A_2A_1A_3B_3|bA_2A_1A_3B_3|aA_1A_3B_3 \\B_3 &\rightarrow aA_1A_3B_3A_2A_1A_3B_3|cA_3B_3A_2A_1A_3B_3|bB_3A_2A_1A_3B_3\end{aligned}$$

Autómatas de Pila (AP)

Los autómatas de pila que desde este momento los abreviamos como: AP son similares a los autómatas finitos ya que se tienen autómatas de pila determinísticos o no determinísticos, también se pueden utilizar para aceptar cadenas de un lenguaje definido sobre un alfabeto A.

Los AP pueden aceptar lenguajes que no pueden aceptar los autómatas finitos. Como ya se ha estudiado, los autómatas finitos no determinísticos (AFND) reconocen los mismos lenguajes que los autómatas finitos determinísticos (AFD). Sin embargo, no ocurre lo mismo con autómatas de pila no determinísticos (APND) y autómatas de pila determinísticos (APD). Algunos lenguajes sólo pueden ser reconocidos por un APND pero no por un APD.

Un AP cuenta con una cinta de entrada y un mecanismo de control que puede encontrarse en uno de entre un número finito de estados. Uno de estos estados se designa como estado inicial, y además algunos estados se llaman de aceptación o finales. A diferencia de los autómatas finitos, los autómatas de pila cuentan con una memoria auxiliar llamada pila. Los símbolos (llamados símbolos de pila) pueden ser insertados o extraídos de la pila, de acuerdo con el manejo el último en entrar es el primero en salir (Last-In-First-Out LIFO).

Las transiciones entre los estados que ejecutan los autómatas de pila dependen de los símbolos de entrada y de los símbolos de la pila. El autómata acepta una cadena X si la secuencia de transiciones, comenzando en estado inicial y con pila vacía, conduce a un estado final, después de leer toda la cadena X.

Autómata de pila reconocedor determinístico

$APD = \langle E, A, P, \delta, e_0, Z_0, F \rangle$ de donde:

E: Conjunto finito de estados $E = \{e_1, e_2, e_3, e_4, \dots, e_i, \dots, e_j\}$

A: Alfabeto o conjunto finito de símbolos de la cinta de entrada,

P: Alfabeto o conjunto finito de símbolos de la Pila. $P \cap A = \emptyset$

δ : función de transición de estados

e_0 : Estado inicial $e_0 \in E$.

Z_0 : Símbolo distinguido $Z_0 \in P$

F: Conjunto de estados finales o estados de aceptación. $F \subseteq E$.

UNIDAD IV • NORMALIZACIÓN DE GRAMÁTICAS

La función de transición definida como: $\delta: E \times (A \cup \{\epsilon\}) \times P \rightarrow E \times P^*$

- 1) $\delta(e_i, a, X) = (e_j, \alpha)$
- 2) $\delta(e_i, \epsilon, X) = (e_j, \alpha)$ donde $a \in A; X \in P; \alpha \in P^*; e_i, e_j \in E$.

Nota: Si existe transición de tipo (2), sólo se garantiza que AP es determinístico si $\forall s \in A, \delta(e_i, s, X)$ está indefinida.

Descripción instantánea:

Una configuración de un AP es una tripla $\langle e, \sigma, \pi \rangle$ donde
e: estado_actual, por ejemplo e_i, e_j ;
 σ : cadena de entrada a ser leída;
 π : contenido de la pila.

Luego, se define una relación de transición \rightarrow en el espacio de posibles configuraciones del AP, tanto si:

- (1) $\langle e_i, aw, X\beta \rangle \rightarrow \langle e_j, w, \alpha\beta \rangle$ Si existe la transición tipo (1), el AP pasa al estado por ejemplo, avanza la cabeza lectora y reemplaza el tope X por α .
- (2) $\langle e_i, aw, X\beta \rangle \rightarrow \langle e_i, aw, \alpha\beta \rangle$ Si existe la transición tipo (2), el AP pasa al estado por ejemplo, NO avanza la cabeza lectora y reemplaza el tope X por α .

Donde $a \in A; w \in A^*; X \in P; \alpha, \beta \in P^*; e_i, e_j \in E$

La función de transición de estados de un AP puede ser representada por un diagrama donde los nodos representan los estados y los arcos transiciones. Si existe transición tipo (1), el arco queda rotulado de la manera en lo que muestra la **figura 4.0**

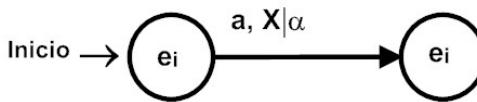


Figura 4.0

Si el estado actual es e_i y la cabeza lectora apunta un símbolo a , y el tope de la pila es X , entonces cambiar al nuevo estado e_j , avanzar la cabeza lectora, y sustituir el símbolo del tope X en la pila por la cadena α . Por ejemplo:

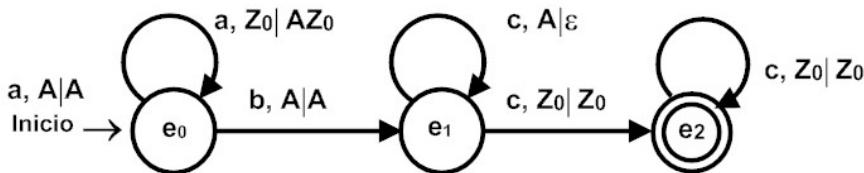


Figura 4.2

Ejemplo 4.11

$L_3 = \{a^i b c^k / i, k \geq 1 \text{ y } i \leq k\}$

$\text{APD}_3 = \langle \{e_0, e_1, e_2\}, \{a, b, c\}, \{A, Z_0\}, \delta_3, e_0, Z_0, \{e_2\} \rangle$ Se muestra en la figura 4.3
 $\delta_3:$

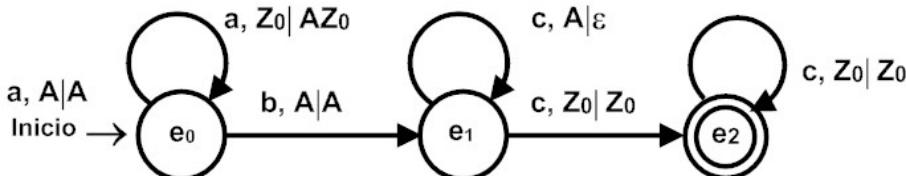


Figura 4.3

Ejemplo 4.12

$L_4 = \{a^i b c^k / i, k \geq 1 \text{ y } i > k\}$

$\text{APD}_{41} = \langle \{e_0, e_1, e_2, e_3\}, \{a, b, c\}, \{A, Z_0\}, \delta_{41}, e_0, Z_0, \{e_3\} \rangle$ Se muestra en la figura 4.4
 $\delta_{41}:$

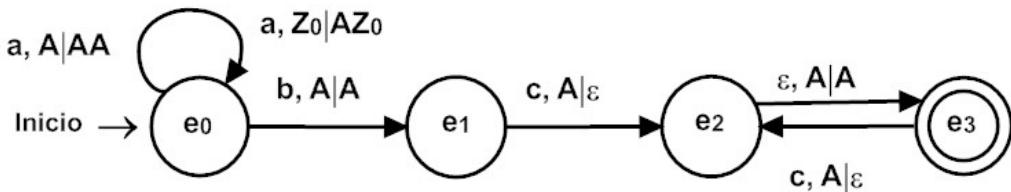


Figura 4.4

$\text{APD}_{42} = \langle \{e_0, e_1, e_2, e_3\}, \{a, b, c\}, \{A, Z_0\}, \delta_{42}, e_0, Z_0, \{e_3\} \rangle$ Se muestra en la figura 4.5

$\text{PD}_{42} = \langle \{e_0, e_1, e_2, e_3\}, \{a, b, c\}, \{A, Z_0\}, \delta_{42}, e_0, Z_0, \{e_3\} \rangle$

δ42:

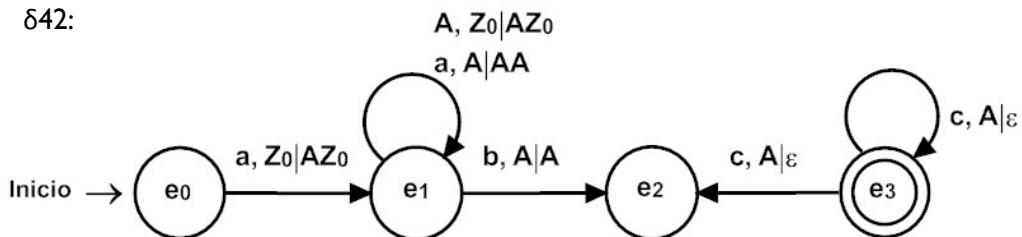


Figura 4.5

Ejemplo 4.13

$$L_5 = \{a^i b c^k / i, k \geq 1 \text{ y } i \geq k\}$$

APD₅ = <{e₀, e₁, e₂}, {a, b, c}, {A, Z₀}, δ₅, e₀, Z₀, {e₂}> Ver la figura 4.6

δ5:

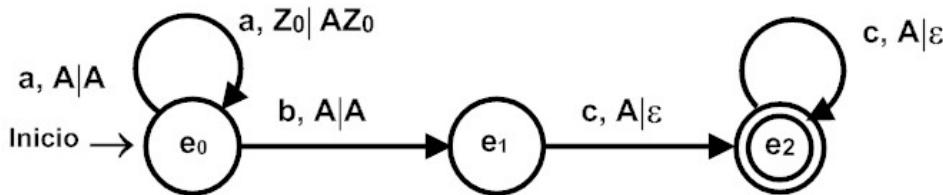


Figura 4.6

Ejemplo 4.14

$$L_6 = \{0^i 1^{i+k} 2^k 3^{n+1} / i, k, n \geq 0\}$$

APD₆ = <{e₀, e₁, e₂, e₃, e₄}, {0, 1, 2, 3}, {A, B, Z₀}, δ₆, e₀, Z₀, {e₄}> Ver figura 4.7

δ6:

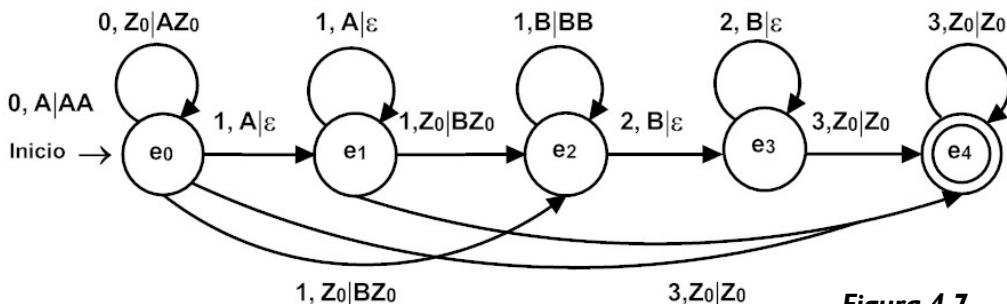


Figura 4.7

Ejemplo 4.15

UNIDAD IV • NORMALIZACIÓN DE GRAMÁTICAS

$$L_7 = \{h^n g^j e^{2n} d^{3i} / i, j, n \geq 0\}$$

APD₇ = < {e₀, e₁, e₂, e₃, e₄, e₅, e₆, e₇, e₈, e₉}, {h, e, g, d}, {H, Z₀}, δ₇, e₀, Z₀, {e₀, e₄, e₈, e₉}>

CASOS

- Si n, i, j > 0
- Si n = 0 y i, j > 0
- Si i = 0 y i, j > 0
- Si j = 0 y n, i > 0
- Si n, i = 0 y j > 0
- Si n, j = 0 y i > 0
- Si n, j = 0 y n > 0
- Si n, i, j = 0

CADENAS DE L₇

- hⁿ g^j e²ⁿ d³ⁱ
- g^j d³ⁱ
- hⁿ g^j e²ⁿ
- hⁿ e²ⁿ d³ⁱ
- g^j
- d³ⁱ
- hⁿ e²ⁿ
- ε

Ver la **figura 4.8** Autómata de pila no determinístico (APND)

67:

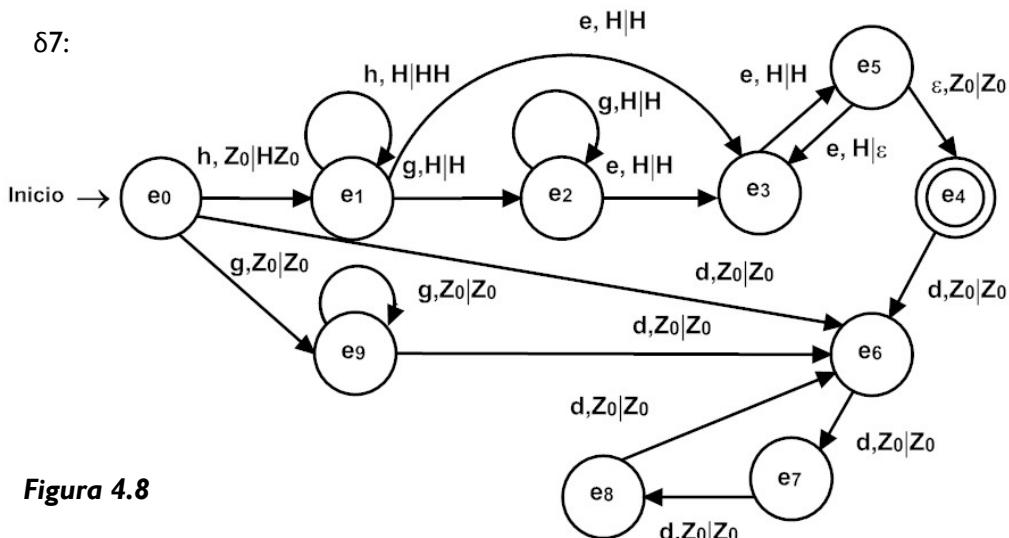


Figura 4.8

Descripción.

UNIDAD IV • NORMALIZACIÓN DE GRAMÁTICAS

APND= $\langle E, A, P, \delta, e_0, Z_0, F \rangle$, donde las componentes E, A, P, e_0, Z_0, F quedó definido como:

E: Conjunto finito de estados $E = \{e_1, e_2, e_3, e_4, \dots, e_i, \dots, e_j\}$

A: Alfabeto o conjunto finito de símbolos de la cinta de entrada,

P: Alfabeto o conjunto finito de símbolos de la Pila. $P \cap A = \emptyset$

δ : función de transición de estados

e_0 : Estado inicial $e_0 \in E$.

Z_0 : Símbolo distinguido $Z_0 \in P$

F: Conjunto de estados finales o estados de aceptación. $F \subseteq E$.

(Repetición de la definición de autómata de pila).

La función de transición se define como:

$\delta: E \times (A \cup \{\epsilon\}) \times P \rightarrow P_f (E \times P^*)$

$$1) \quad \delta(e_i, a, X) = \{(e_j, \alpha_1), (e_k, \alpha_2), \dots\}$$

$$2) \quad \delta(e_i, \epsilon, X) = \{(e_j, \alpha_1), (e_k, \alpha_2), \dots\} \text{ donde } a \in A; X \in P; \alpha_1, \alpha_2 \in P^*; e_i, e_j, e_k \in E.$$

Como ya se ha estudiado, repetimos, los autómatas finitos no determinísticos (AFND) reconocen los mismos lenguajes que los autómatas finitos determinísticos (AFD). Sin embargo, no ocurre lo mismo con autómatas de pila no determinísticos (APND) y autómatas de pila determinísticos (APD). Algunos lenguajes sólo pueden ser reconocidos por un APND, pero no por un APD.

Ejemplo 4.16

$$L_8 = \{a^m b^p c^{p+m} / m, p \geq 1\} \cup \{a^{2i} b^i / i \geq 1\}$$

L_8 sólo se puede reconocer con un APND

APND₈ = $\langle \{e_0, e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8, e_9\}, \{a, b, c\}, \{A, B, Z_0\}, \delta_8, e_0, Z_0, \{e_9\} \rangle$ Ver

Figura 4.9

UNIDAD IV • NORMALIZACIÓN DE GRAMÁTICAS

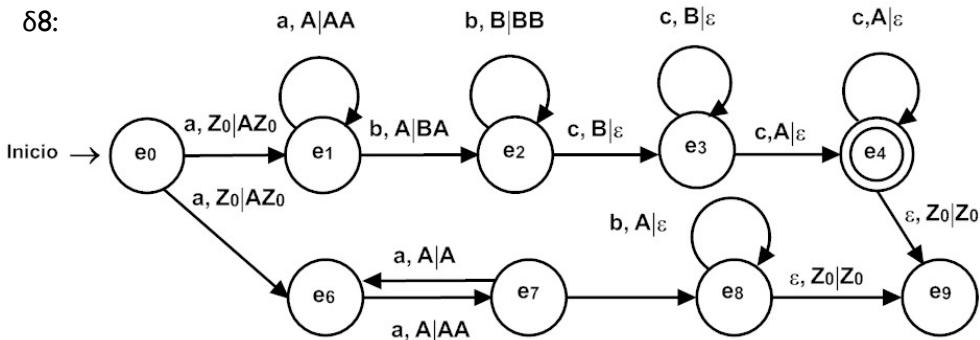


Figura 4.9

APND₈ es no determinístico ya que en el caso de:

$$\delta(e_0, a, Z_0) = \{(e_1, AZ_0), (e_6, AZ_0)\}$$

Ejemplo 4.17

$$L_9 = \{a^m b^p c^{p+m} / m, p \geq 1\} \cup \{a^{2i} b^i / i \geq 1\}$$

L₉ sólo se puede reconocer con un APND

APND₉ = <{e₀, e₁, e₂, e₃, e₄, e₅, e₆, e₇}, {a, b, c}, {A, B, Z₀}, δ₉, e₀, Z₀, {e₇}> Ver Figura 4.10

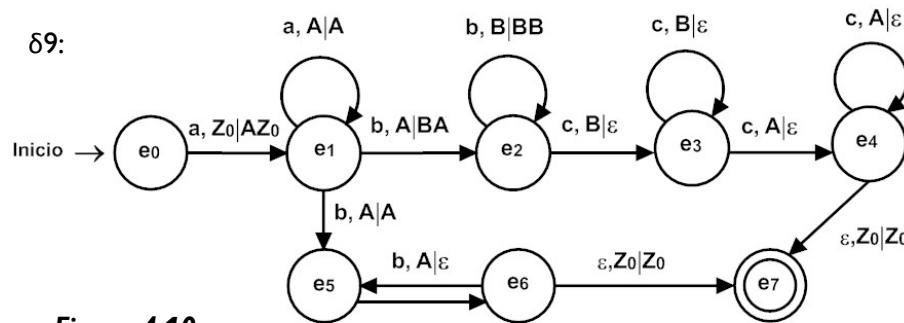


Figura 4.10

Lenguajes aceptados por los autómatas de pila

Una cadena $w \in A^*$ es aceptado por AP = <E, A, P, δ, e₀, Z₀, F>

Si y sólo si $\langle e_0, w, Z_0 \rangle \rightarrow \langle e_x, \epsilon, \alpha \rangle$

UNIDAD IV • NORMALIZACIÓN DE GRAMÁTICAS

El autómata de pila AP, comienza en el estado e_0 , con pila vacía luego de leer toda la cadena w , llega a un estado $e_r \in F$, y en la pila queda cualquier cadena $\alpha \in P^*$

El lenguaje aceptado por AP, es el conjunto de todas las cadenas que son aceptadas por AP.

$$L(AP) = \{w | \langle e_0, w, Z_0 \rangle \rightarrow \langle e_r, \epsilon, \alpha \rangle \text{ y } w \in A^* \text{ y } e_r \in F \text{ y } \alpha \in P^*\}$$

Los lenguajes aceptados por los AP* se denominan lenguajes libres de contexto.

Autómata de pila traductor AP_T

Un AP_T es simplemente como una tupla de 9 elementos.

$$AP_T = \langle E, A, P, \delta, e_0, Z_0, F, \lambda, S \rangle$$

Donde E, A, P, δ, e₀, Z₀, F ya se definieron anteriormente y se agregan los componentes.

S: Alfabeto o conjunto finito como $\lambda: E \times (A \cup \{\epsilon\}) \times P \rightarrow S^*$

La λ está definida siempre que δ está definida. En el diagrama de transición de AP_T puede describirse como una extensión de la notación usada para AP. Si existe $\delta(e_i, a, x) = t$; luego el arco queda rotulado según se observa en la **figura 4.11**

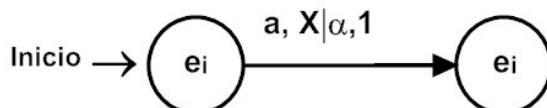


Figura 4.11

Donde $e_i, e_j \in E; a \in A; x \in P; \alpha \in P^*; t \in S^*$

Función de transición para cadenas

El autómata solo define la traducción, si el autómata AP subyacente “acepta”. Es decir, la traducción $T(w): A^* \rightarrow S^*$ asociado a AP, está definida como $T(w)$ es válida $\Leftrightarrow \langle e_0, w, Z_0 \rangle$

Ejemplo

$$L = \{0^i 1^{i+k} 2^k 3^{n+1} / i, k, n \geq 0\}$$

Traducir las cadenas de L $0^i 1^{i+k} 2^k 3^{n+1}$ como $a^{i+k} b^{2k} c^{3n}$

L, sólo se puede reconocer con un APD_T

$$APD_T = \langle \{e_0, e_1, e_2, e_3, e_4\}, \{0, 1, 2, 3\}, \{A, B, Z_0\}, \delta, e_0, Z_0, \{e_4\}, \lambda, \{a, b, c\} \rangle$$

Ver figura 4.12

UNIDAD IV • NORMALIZACIÓN DE GRAMÁTICAS

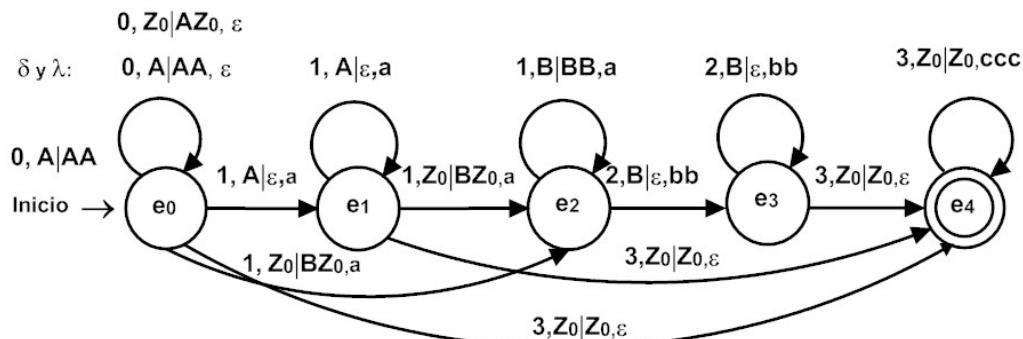


Figura 4.12

Ejemplo

$$L = \{h^n g^i e^{2n} d^{3i} / i, n \geq 0\}$$

Traducir las cadenas de L $h^n g^i e^{2n} d^{3i}$ como $1^{2i} 0^n 2^i$

APD_T = <{ $e_0, e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8, e_9$ }, { h, e, g, d }, { H, Z_0 }, δ , $e_0, Z_0, \{e_0, e_4, e_8, e_9\}, \lambda, \{0, 1, 2\}$ >

Ver figura 4.13

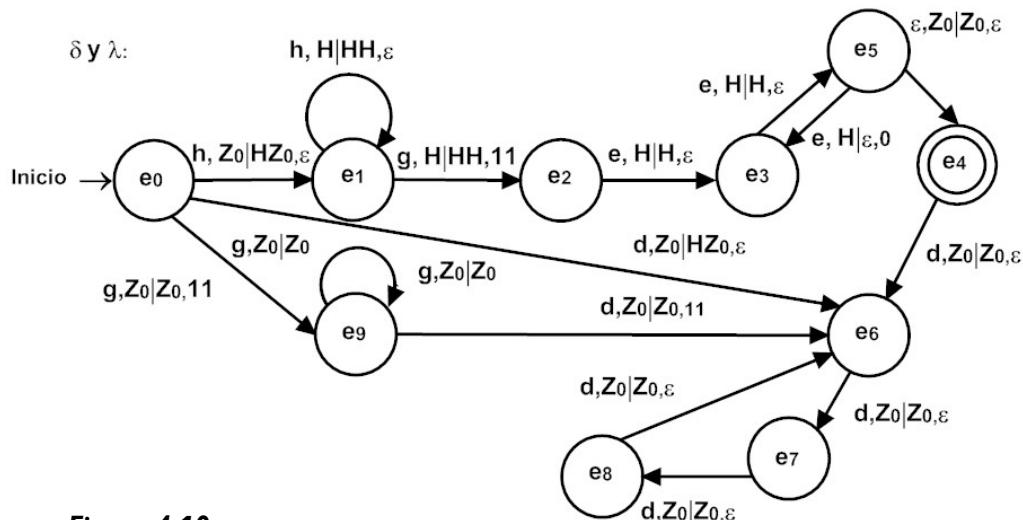


Figura 4.10

UNIDAD V

“Solo podemos ver poco del futuro, pero lo suficiente para darnos cuenta de que hay mucho que hacer” Alan Turing.

EQUIVALENCIAS Y MÁQUINA DE TURING

“Usa Equivalencias y Máquina de Turing, mediante métodos (Myhill-Nerode, entre otros); para la simplificación de autómatas y generación de modelos en aplicaciones de diseño asistido por computadora.”

TEMAS:

- 5.1 Equivalencias de una Gramática Regular Lineal por la derecha.
 - 5.1.1 Método para pasar de una Gramática Regular Lineal por la Derecha a un AFN con Épsilon Transiciones
 - 5.1.2 Método para pasar de un AFN con Épsilon Transiciones a una Gramática Regular Lineal por la Derecha.
- 5.2 Equivalencias de una Gramática Regular Lineal por la izquierda.
 - 5.2.1 Método para pasar de una Gramática Regular Lineal por la Izquierda a un AFN con Épsilon Transiciones
 - 5.2.2 Método para pasar de un AFN con Épsilon Transiciones a una Gramática Regular Lineal por la Izquierda.
- 5.3 Generación de los ‘Items’ de un Gramática
- 5.4 Simplificación de un AFD
- 5.5 Máquina de Turing.

Ejercicios para adquirir la competencia de esta unidad.

Recordando de forma simplificada una gramática regular también es una gramática formal definida como un conjunto de elementos, según la teoría de conjuntos, una gramática formal queda comprendida como una “tupla” o tabla de al menos cuatro elementos, dependiendo del tipo de gramática: N, Σ , P, S. Recordamos que N significa el conjunto de elementos variables No terminales, como VN o V o N (las tres abreviaturas significan variables no terminales); Σ representa el alfabeto o conjunto de símbolos terminales definidos dentro del conjunto Σ , para con estos símbolos integrar las palabras o cadenas del lenguaje, un ejemplo de Σ puede ser $\Sigma = \{0, 1\}$ en este ejemplo se tiene que el lenguaje está integrado, por los símbolos 0 y 1.

Por **ejemplo**, puede ser $\Sigma = \{a, b\}$ definiendo un lenguaje solo con los símbolos a y b. La letra P mayúscula representa las reglas de producción del lenguaje o relaciones entre variables no terminales y terminales (definidas por letras minúsculas). Por último, S es el símbolo inicial no terminal, definido por lo general con letras mayúsculas. Esta descripción, en forma de notación matemática queda definida como: G_R o Gramática regular de la siguiente forma:

$$G_R = \{VN, \Sigma, P, S\} \quad VN = \{L\} \quad \Sigma = \{0, 1\} \quad S = \{q_0\}$$

P=Reglas de producción y L = Elementos inicial No terminal o primer variable no terminal (VN), también puede ser S

$$P \rightarrow 0L1 \quad L \rightarrow 01$$

Ejemplo de palabra aceptada por esta gramática regular es: 0011 (Esto es

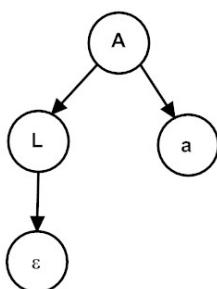


Figura 5.0

que en lugar de 0L1, reemplazamos la L por 01). Una gramática también puede ser clasificada como regular izquierda o regular derecha. Las gramáticas regulares sólo pueden generar a los lenguajes regulares de manera similar a los autómatas finitos y las expresiones regulares. Dos gramáticas regulares que generan el mismo lenguaje regular se denominan equivalentes. Toda gramática regular es una gramática libre de contexto. Una gramática regular derecha es aquella cuyas reglas de producción P son de la siguiente forma: $A \rightarrow aL$ $L \rightarrow \epsilon$

Para ilustrar estas dos reglas de producción, las esquematizamos en el árbol de la **figura 5.0**

UNIDAD V • EQUIVALENCIAS Y MÁQUINA DE TURING

Análogamente, en una gramática regular izquierda, las reglas son de la siguiente forma:

$$A \rightarrow La$$

$$L \rightarrow \epsilon$$

Para ilustrar estas dos reglas de producción anteriores, las esquematisamos en el árbol de la **figura 5.1**

Una definición equivalente evita la siguiente regla $A \rightarrow a$. En el caso de las gramáticas regulares derechas y por el caso de las izquierdas, algunos autores alternativamente no permiten el uso de la regla $A \rightarrow a$ suponiendo que la cadena vacía no pertenece al lenguaje.

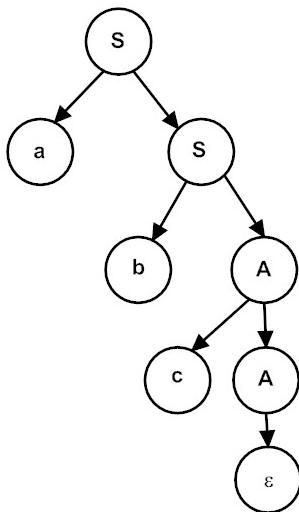


Figura 5.2

Un ejemplo de una gramática regular G_R con $N = \{S, A\}$, $\Sigma = \{a, b, c\}$, P se define mediante las siguientes reglas:

$$S \rightarrow aS \qquad \qquad S \rightarrow bA$$

$$A \rightarrow cA \qquad \qquad A \rightarrow \epsilon$$

El diagrama de árbol de las anteriores reglas de producción se muestra en la **figura 5.2**

Donde S es el símbolo inicial. Esta gramática describe el mismo lenguaje representado mediante la expresión regular a^*bc^* misma que queda también expresada en el siguiente

diagrama de estados finitos: $Q = \{e_0, e_1\}$, $S = \{e_0\}$, $F = \{e_1\}$

El autómata que define esta expresión: a^*bc^* se

muestra en la **figura 5.3**

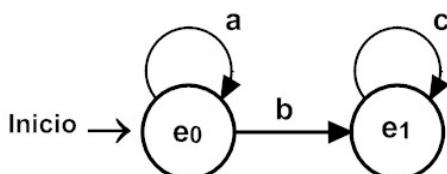


Figura 5.3

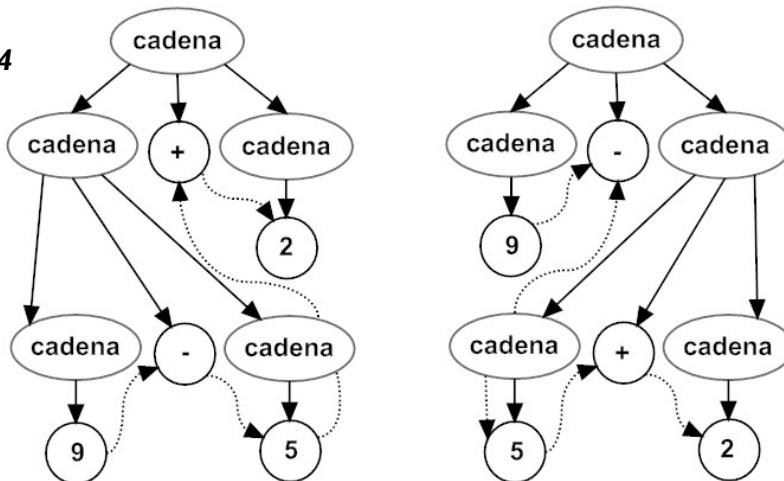
5.1 Equivalencias de una Gramática Regular Lineal por la derecha.

Una gramática regular lineal por la derecha es aquella cuyas reglas se denominan equivalentes. Son reglas equivalentes las que definen las producciones de una gramática de igual forma que podemos obtener el mismo lenguaje o conjunto de cadenas iguales no importando si derivamos o las obtenemos de un árbol que se desarrolla del lado derecho o del lado izquierdo, siendo esto también un concepto que genera una gramática ambigua. El concepto de ambigüedad lo tratamos de clarificar con el siguiente ejemplo que muestra dos árboles de análisis sintáctico por la izquierda y derecha:

La cadena de símbolos: $9 - 5 + 2$ se muestran en la **figura 5.4**

Cadena de símbolos: $9 - 5 + 2$

Figura 5.4



Recorrido izquierdo:

En este árbol, el recorrido es desde el nodo inferior izquierdo 9, luego el símbolo $-$ con el otro nodo inferior izquierdo 5, sube al símbolo $+$ y termina con el nodo derecho 2.

Recorrido derecho

En este otro árbol, el recorrido es desde el nodo superior izquierdo 9, luego el símbolo $-$ arriba también la izquierda con el otro nodo inferior derecho 5, sube al símbolo $+$ y termina con el nodo derecho 2.

UNIDAD V • EQUIVALENCIAS Y MÁQUINA DE TURING

El tener mediante dos recorridos diferentes la generación de la misma cadena, estamos describiendo una gramática ambigua, cuyas reglas de producción son las dos siguientes:

- 1) $\langle \text{cadena} \rangle ::= \langle \text{cadena} \rangle - \langle \text{cadena} \rangle + \langle \text{cadena} \rangle$
- 2) $\langle \text{cadena} \rangle ::= 9|5|2$

Supóngase que, en cada paso de una derivación, la variable que se reescribe es aquélla que aparece más a la izquierda de la forma oracional; si éste es el caso, se dice de la derivación que es una derivación izquierda. Si, por el contrario, la variable que se reescribe en cada paso es aquélla que aparece más a la derecha, se dice que es una derivación derecha.

Claramente, una palabra w puede tener varias derivaciones izquierdas o derechas, ya que puede haber más de un árbol de derivación para w . Sin embargo, es fácil demostrar que, a partir de un árbol de derivación dado, pueden encontrarse una única derivación izquierda y una única derivación derecha.

Si en una gramática positiva G , alguna palabra tiene más de un árbol de derivación o, equivalentemente, más de una derivación de izquierda o derecha, se dice de la gramática que es ambigua.

Si un Lenguaje Libre de Contexto (LLC) es tal que toda gramática que lo genera es ambigua, entonces se dice de él que es inherentemente ambiguo.

Ejemplo. Sea G gramática cuyas producciones son:

$$\begin{aligned} S &\rightarrow AB|BA \\ A &\rightarrow Aa|aA|a \\ B &\rightarrow Bb|bB|b \end{aligned}$$

Sea $w = aaab$; se mostrará que w tiene, al menos, dos derivaciones izquierdas distintas y dos derivaciones derechas distintas:

- i) Derivaciones izquierdas:
 - a) $S \Rightarrow AB \Rightarrow AaB \Rightarrow aAaB \Rightarrow aaaB \Rightarrow aaab$
 - b) $S \Rightarrow AB \Rightarrow aAB \Rightarrow aaAB \Rightarrow aaaB \Rightarrow aaab$

ii) Derivaciones derechas:

- $S \Rightarrow AB \Rightarrow Ab \Rightarrow aAb \Rightarrow aAab \Rightarrow aaab$
- $S \Rightarrow AB \Rightarrow Ab \Rightarrow Aab \Rightarrow Aaab \Rightarrow aaab$

La variable no terminal “A” la corresponde el árbol de derivación izquierda, de los dos árboles mostrados arriba y, a la ii)a). El de la derivación derecha. De los resultados obtenidos puede inferirse, entonces que G es una gramática ambigua:

Derivación i) a) $S \rightarrow AB \quad A \rightarrow aA \quad A \rightarrow a \quad B \rightarrow b$

En figura 5.5

Derivación ii) a) $S \rightarrow AB \quad A \rightarrow aA \quad A \rightarrow a \quad B \rightarrow b$

En figura 5.6

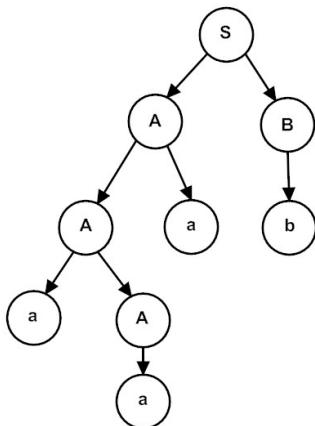


Figura 5.5

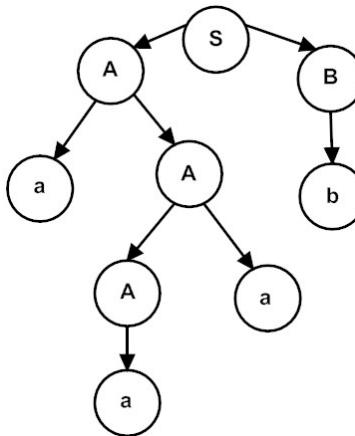


Figura 5.6

Toda gramática regular es una gramática libre de contexto. Una gramática regular lineal por la derecha es aquella cuyas reglas de producción P son de la siguiente forma:

1. $A \rightarrow a$, donde A es un símbolo no-terminal en N y a un elemento terminal en Σ
2. $A \rightarrow aB$, donde A y B pertenecen a N y a pertenece a Σ
3. $A \rightarrow \epsilon$, donde A pertenece a N.

UNIDAD V • EQUIVALENCIAS Y MÁQUINA DE TURING

Análogamente, en una gramática regular a la izquierda, las reglas son de la siguiente forma:

1. $A \rightarrow a$, donde A es un símbolo no-terminal en N y a un elemento terminal en Σ
2. $A \rightarrow Ba$, donde A y B pertenecen a N y a pertenece a Σ
3. $A \rightarrow \epsilon$, donde A pertenece a N.

Una definición equivalente evita la regla 1. ($A \rightarrow a$) ya que es sustituible por:
 $A \rightarrow aLL \rightarrow \epsilon$ en el caso de las gramáticas regulares derechas y por: $A \rightarrow LaL \rightarrow \epsilon$ en el caso de las izquierdas. Algunos autores alternativamente no permiten el uso de la regla 3 suponiendo que la cadena vacía no pertenece al lenguaje.

Dada una gramática regular izquierda es posible convertirla, mediante un **algoritmo** en una derecha y viceversa. Aquí en este último capítulo, donde se toca el tema de métodos:

- 1) Para pasar de una Gramática Regular Lineal por la Derecha a un AFN con Épsilon Transiciones;
- 2) Para pasar de un AFN con Épsilon Transiciones a una Gramática Regular Lineal por la Derecha;
- 3) Para pasar de una Gramática Regular Lineal por la Izquierda a un AFN con Épsilon Transiciones
- 4) para pasar de un AFN con Épsilon Transiciones a una Gramática Regular Lineal por la Izquierda.

Estos cuatro métodos y más pueden ser programados de diferentes formas o mediante el uso o aplicación de lenguajes de programación existentes, mismos que incluyen métodos particulares dentro de cada lenguaje como parte del mismo lenguaje. El tema de los lenguajes de programación ya quedó descrito en el primer capítulo de este material.

Los métodos son intuitivos, creados, originados de las fuentes de información bibliografía anotadas al final del desarrollo de este capítulo conjuntamente con la experiencia, esto los hace también empíricos. Se hace esta aclaración porque en fuentes de información bibliográfica consultadas para desarrollar este tema, se encuentran básicamente métodos relacionados con los utilizados dentro de los lenguajes de programación como C, Java, sistemas operativos como UNIX, LINUX, etc.

UNIDAD V • EQUIVALENCIAS Y MÁQUINA DE TURING

Sin embargo, se presentarán algunos ejemplos de un lenguaje de programación conocidos como AWK el cual se describirá brevemente en la generación de “Items” y de momento sólo lo describimos como un lenguaje creado con lenguaje C y además acepta de forma “formal” expresiones regulares que son definidas para filtrar los elementos terminales y no terminales de la gramática definida. Misma que tiene como principal utilidad el generar cadenas, palabras o conjuntos de símbolos aceptados y rechazados, según se mostrará al final de este capítulo.

Un ejemplo en C, Java o AWK es utilizar un método y un constructor (elementos agregados al método) que permiten definir la materia prima o elementos que alimentan el procesamiento de la gramática mediante los lenguajes de programación descritos o cualquier otro.

Además del constructor por defecto que como su nombre lo indica, se encarga de construir una gramática vacía, habrá que incluir un constructor que reciba todos los elementos de la gramática, alfabeto de variables, alfabeto de terminales y conjunto de reglas para abrir el archivo. Con la definición de una gramática hay que tener en cuenta cómo se especificarán sus elementos en el mismo.

Por ejemplo, la gramática $G = (\{S,A\}, \{a,b\}, \{SaA, A, aAb|b\}, S)$ se representará en un archivo de texto de la manera siguiente: **SAab3S aAA aAbA b** Cada una de las filas de este archivo representa:

- Primera fila.- **SAab3S**, representa una cadena en la que se especifican, ordenadamente, los símbolos que forman el alfabeto de las variables terminales y no terminales.
- Segunda fila.- **aAA**, representa una cadena en la que se especifican, ordenadamente, los símbolos que forman el alfabeto de los símbolos terminales.
- Tercera fila.- **aAbA**, representa el número de reglas de producción que componen la gramática.
- Cuarta fila.- **b**, Primera regla de producción. Se especifica primero la parte izquierda de la regla y, después, la parte derecha ambas separadas por espacios en blanco. Cuando se trate de una producción nula, A, la cadena vacía se representará por ‘e’.
- Siguientes filas.- Resto de reglas de producción donde u T^* y A, B V.

UNIDAD V • EQUIVALENCIAS Y MÁQUINA DE TURING

Esta comprobación la llevará a cabo el **método Boolean es Regular** (Dentro de la clase Gramática). Una vez, finalizada la implementación de esta clase, dispondremos de todos los elementos necesarios para llevar a la práctica la equivalencia entre AFD's y gramáticas regulares, utilizando un lenguaje de programación.

Con base a las aclaraciones realizadas para las palabras algoritmo y método, a continuación, se describen métodos o descripción simplificada de los pasos que se pueden realizar para realizar las siguientes conversiones.

En teoría de la computación existen algunos algoritmos relacionados con las propiedades de los lenguajes regulares y estos son:

Algoritmos de decisión

Son procedimientos aplicados a toda instancia del problema, efectivo y que siempre termina dando un resultado y estos son:

- **Finitud** (Que tiene fin, término o límite)
- **Vacuidad** (Que tiene vacío, falso de contenido)
- **Membresía** (Que pertenece a un conjunto)
- **Inclusión** (Que está dentro de un conjunto)
- **Equidad** (Igualdad)

Los algoritmos de decisión son diferentes a los problemas de decisión

- Algoritmos de decisión \neq Problema de decisión

Los problemas aplicados a lenguajes regulares son diferentes a los problemas de decisión

- Problemas aplicados a lenguajes regulares

- **Finitud: ¿L1 es finito?**

- El lenguaje común es infinito

Por el lema del bombeo \rightarrow Si existen ciclos es infinito.

Autómata Finito (AF): Algoritmo para hallar ciclos en un dígrafo desde el estado inicial “ q_0 ” a algún estado final “ q_f ”. En **figura 5.7**

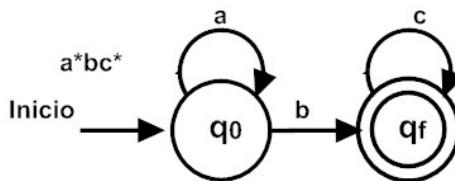


Figura 5.7

ER: ¿Si existe clausura de Kleene?

Vacuidad: ¿ L_1 es vacío?

AF: por accesibilidad de un dígrafo, existe un camino desde el estado inicial a algunos de los estados finales.

ER: propiedades algebraicas de ER.

- $R + S \not\in R + \emptyset = R$, R y S deben ser \emptyset
- $RS \not\in R \emptyset = \emptyset$, R o S deben ser \emptyset
- $R^* \not\in \emptyset$ porque la clausura contiene a la cadena vacía ϵ

Membresía: Dado w en Σ^* , ¿Si w está en L_1 ?

AF: recorrido de un dígrafo o simular un AFD
ER: convertir a un AFN- ϵ , convertir a un AFD y simular.

Equidad: ¿ L_1 y L_2 son iguales?

Existe un algoritmo para determinar si dos AFD aceptan el mismo lenguaje.

Demostración. Sean M_1 y M_2 dos AFD, entonces de forma algorítmica se puede construir al AFD M que acepte el lenguaje:

$L(M) = (L(M_1) \cap L(M_2)^c) \cup (L(M_1)^c \cap L(M_2))$ Entonces si $L(M) \neq \emptyset$ (vacuidad).

Teorema de Myhill-Nerode para minimización de autómatas

Sea L un subconjunto de A^* un lenguaje arbitrario. Asociado a este lenguaje L se puede definir una relación de equivalencia RL en el conjunto A , de la siguiente forma:

Si x, y están en A^* , entonces $(xRLy)$ si y solo si (Para todo z en A^* ; (xz en L , si y solo si, yz en L)). Esta relación de equivalencia dividirá el conjunto A^* en clases de equivalencia.

El número de clases de equivalencia se llama índice de la relación. También se puede definir una relación de equivalencia, RM , en A^* asociada a AFD $M = (Q; A; \delta; q_0; F)$

Si u, v en A^* , entonces $uRMv$ si y solo si $(\delta(q_0; u) = \delta(q_0; v))$

UNIDAD V • EQUIVALENCIAS Y MÁQUINA DE TURING

La relación de equivalencia divide también el lenguaje A en clases de equivalencia.

- Si L es subconjunto de A^* entonces:
 1. L es aceptado por un autómata finito
 2. L es la unión de algunas de las clases de equivalencia de una relación de equivalencia en A^* de índice finito que sea invariante por la derecha.
 3. La relación de equivalencia RL es de índice finito.
- Si L es un conjunto regular y RL la relación de equivalencia asociada, entonces el autómata construido en el teorema anterior es “minimal” o mínimo y único salvo isomorfismos.

Existe un método simple para encontrar el AFD con número mínimo de estados M' equivalente a un AFD.

$$M = (Q, A, \delta, q_0, F).$$

Sea \equiv la relación de equivalencia de los estados de M tal que $p \equiv q$ si y sólo si para cada entrada x , $\delta^*(p, x)$ es un estado de aceptación si y sólo si $\delta^*(q, x)$ es un estado de aceptación. Si $p \equiv q$, decimos que p es equivalente a q .

Decimos que p es distingible de q si existe un x tal que $\delta^*(p, x) \in F$ y $\delta^*(q, x)$ no está en F , o viceversa.

Ejemplo

Sea M el siguiente autómata o máquina de estados finitos en la **figura 5.8**

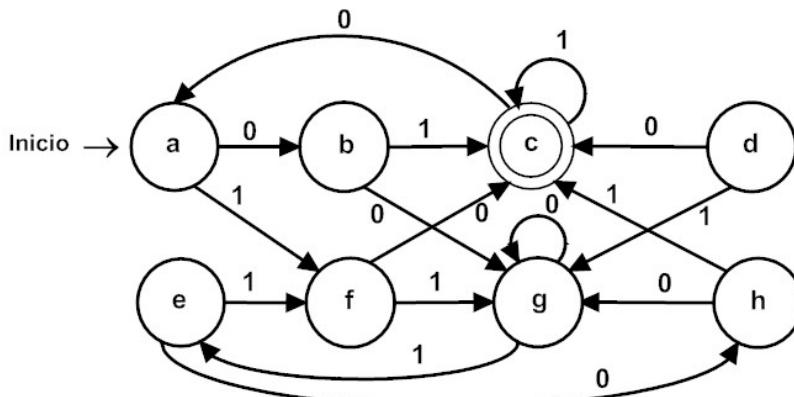


Figura 5.8

UNIDAD V • EQUIVALENCIAS Y MÁQUINA DE TURING

Lenguaje regular aceptado: **01((0(01))⁺)^{*}1^{*}**

- Se tiene que construir una tabla con una entrada para cada par de estados.
 - Se coloca una X en la tabla cada vez que un par de estados son distinguibles. Inicialmente se coloca una X en cada entrada correspondiente a un estado final y un estado no final. En el ejemplo, $Q - F = \{a, b, d, e, f, g, h\}$, esto es $Q = \text{conjunto de todos los estados} = \{a, b, c, d, e, f, g, h\}$ y $F = \{c\}$, recordando que F es el estado final marcado con doble circulo.
 - Colocamos una X en las entradas (a, c) , (b, c) , (c, d) , (c, e) , (c, f) , (c, g) y (c, h) . Esto es, todos los estados involucrados con el estado final c.
 - Para cada par de estados p y q que no se han identificado como distinguibles, consideramos el par de estados (r, s) , $r = \delta(p, a)$ y $s = \delta(q, a)$ para cada entrada a.

Ejemplo

Si se demuestra que los estados s y r son distinguibles para alguna cadena x entonces p y q son distinguibles para cualquier cadena ax.

Así si la entrada (r, s) en la tabla tiene una X, se coloca una X en la entrada (p, q) . Si la entrada (r, s) no tiene X, entonces el par (p, q) es colocado en una lista asociada con la entrada (r, s) .

Continuando se tiene que si la entrada (r, s) recibe una X entonces cada par en la lista asociada con la entrada (r, s) también recibe una X.

Ejemplo

En el ejemplo, colocamos una X en la entrada (r, s) , porque la entrada $(\delta(b,1), \delta(a,1)) = (c, f)$ ya tiene una X. Similarmente, la entrada (a, d) recibe una X.

Ahora consideramos la entrada (a, e) que con la entrada 0 va a dar el par (b, h) , así (a, e) es colocado en la lista asociada con (b, h) . Observe que con la entrada 1, a y e van al mismo estado f y por lo tanto no hay cadena con 1 que pueda distinguir a de e.

Tabla de estados distinguibles según método del Teorema de Myhill-Nerode para minimización de autómatas

B	X						
C	X	X					
D	X	X	X				
E		X	X	X			
F	X	X	X		X		
G	X	X	X	X	X	X	
H	X		X	X	X	X	X
	A	B	C	D	E	F	G

El autómata mínimo queda según la **figura 5.9**

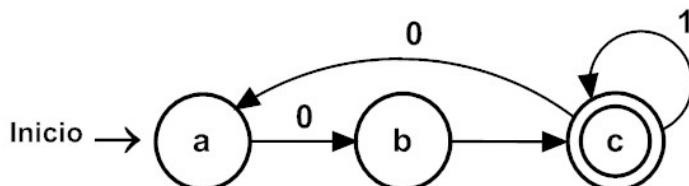


Figura 5.9

Lenguaje Regular aceptado: $01((0(01))^*)^*1^*$

5.1.1 Método para pasar de una Gramática Regular Lineal por la Derecha a un AFN con Épsilon Transiciones.

Recordemos que una gramática regular lineal por la derecha es aquella cuyas reglas de producción nos permiten decidir que las cadenas generadas serán tomadas de derecha a izquierda, el resto de reglas de producción donde se especifica la gramática son del siguiente tipo:

- 1) $A \rightarrow aB$
- 2) $A \rightarrow a$

Donde A, B son elementos de los símbolos no terminales y a (en minúscula) pertenece al conjunto de elementos terminales esto es: $A, B \in VN$ y $a \in VT$
Las dos reglas de producción anteriores con respecto a las variables A, a, y B. Se pueden integrar en una sola:

$$A \rightarrow aB|a$$

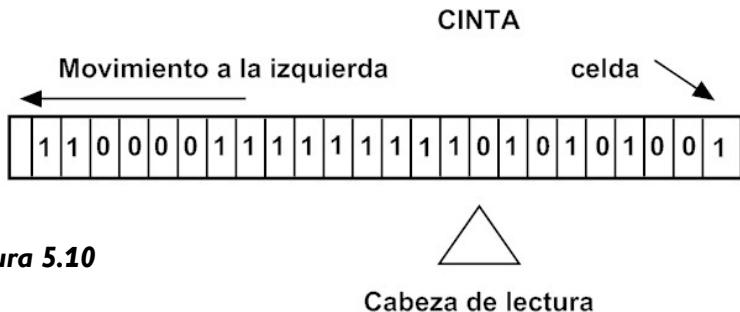
Se pueden decir también, que según la jerarquía de Chomsky, una gramática regular por la derecha corresponden al tipo 3 o gramáticas regulares o de estado finito, mismas gramáticas que generan lenguajes regulares son fácilmente transformadas a autómatas finitos no deterministas, esto es, que no necesitan una transición por cada símbolo definido por el lenguaje, para efectos de exemplificar esta posibilidad, recordemos que un Autómata Finito No determinista (AFN) puede ser generado por una expresión regular (ER) que a su vez es generada por una gramática regular G_R y ésta a su vez generada por una expresión regular (ER) la cual para la representación de la pequeña gramática anterior, tenemos lo siguiente Lenguaje (LR) reconocido por un autómata finito no determinista y este se puede hacer determinista, AFD, según se mostró en capítulos anteriores.

$AFN \rightarrow ER \rightarrow G_R \rightarrow LR \rightarrow AFN \rightarrow AFD$ (con épsilon y sin épsilon transiciones)
Cuando un lenguaje transita a una configuración final partiendo de la configuración inicial, en varios movimientos, se dice que se ha producido aceptación o reconocimiento de la cadena de entrada. Es decir, los autómatas finitos reconocen los lenguajes regulares, o de tipo 3 según jerarquía de Chomsky y se pueden representar intuitivamente como una máquina de Turing, por una cinta y una cabeza de lectura que

UNIDAD V • EQUIVALENCIAS Y MÁQUINA DE TURING

para este caso solo realiza movimientos a la izquierda y cuenta con una cabeza de lectura sobre cada una de las celdas de la CINTA.

Según se muestra en la **figura 5.10**



La cinta de entrada, solo contiene símbolos de un determinado alfabeto y en este caso es: $\Sigma = \{0,1\}$, y se mueve en una sola dirección.

El control de estados, determina el comportamiento del autómata, en este caso está determinado con el movimiento de izquierda a derecha, conjuntamente con la cabeza lectora y los símbolos de la cinta, que para este ejemplo en particular es finita, contrariamente a la máquina de Turing que se diseñó con un número infinito de símbolos planteando el problema como de la parada de la Máquina de Turing es “indecible” y en matemática quedó demostrado, así como la complejidad computacional.

La característica más importante de un autómata finito es el hecho de que una sentencia de un lenguaje determinado, colocada en cinta, y leída por el autómata finito, es reconocida por éste, si el control de estados llega a un estado final.

La definición formal de la gramática lineal por la derecha de la gramática mostrada en el autómata finito dibujado es la siguiente: $M = \{Q, \Sigma, \delta, q_0, F\}$

Donde M representa: máquina de estados finitos o autómata finito.

Q representa en conjunto de estados del autómata.

Σ representa: El alfabeto que acepta el autómata que es este caso es $\Sigma = \{0,1\}$

δ representa: Función de transición, esto es, el símbolo que permite cambiar de un estado a otro. Ampliando esta función de transición, queda mejor representado como un eje cartesiano donde $\delta = Q \times \Sigma$ o bien la siguiente tabla.

UNIDAD V • EQUIVALENCIAS Y MÁQUINA DE TURING

Función de transición δ

Q = conjunto de estados del autómata que, para efectos del diagrama, pueden ser cada una de las celdas, que si las contamos son 24, esto significa que para nuestro ejemplo tenemos:

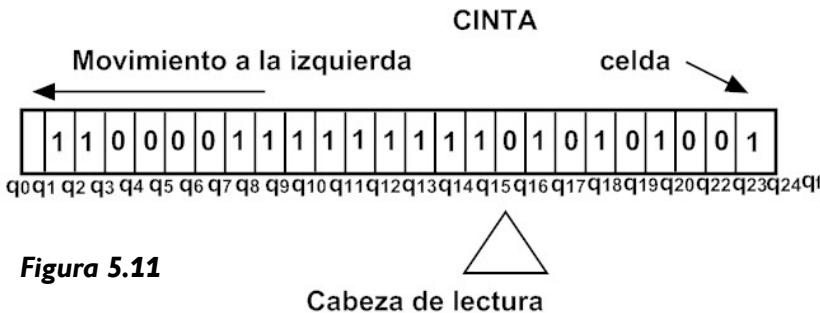
$$Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_8, q_9, q_{10}, q_{11}, q_{12}, q_{13}, q_{14}, q_{15}, q_{16}, q_{17}, q_{18}, q_{19}, q_{20}, q_{21}, q_{22}, q_{23}, q_{24}\}$$

Según se muestra la **figura 5.11**

El lenguaje regular aceptado por este diagrama y el autómata finito no determinístico es: 11000011111111010101001

Este es un lenguaje regular L_R el cual también se puede expresar como:

$L_R = \{1^i 0^j 1^k (01)^l 0^m 1^n / i=2, j=4, k=9, l=3\}$ (Esto significa que el 1 se repite dos veces ($i=2$), el 0 se repite cuatro veces ($j=4$), el 1 se repite nueve veces ($k=9$), el 01 se repite tres veces ($l=3$), nuevamente el 0 se repite dos veces ($i=2$) y al último 1.



Para cada uno de estos estados se puede tener cada uno de los símbolos del alfabeto, esto es, 0 y 1

$S = \{q_0\}$ = representación del estado inicial

Según la tabla de transiciones de la **figura 5.12**

UNIDAD V • EQUIVALENCIAS Y MÁQUINA DE TURING

Estados Q	Entradas		Estados Finales
	0	Σ	
q_0	\emptyset	q_1	0 q_0 con un 1 en la celda pasa a q_1
q_1	\emptyset	q_2	0
q_2	q_3	\emptyset	0
q_3	q_4	\emptyset	0
q_4	q_5	\emptyset	0
q_5	q_6	\emptyset	0
q_6	\emptyset	q_7	0
q_7	\emptyset	q_8	0
q_8	\emptyset	q_9	0
q_9	\emptyset	q_{10}	0
q_{10}	\emptyset	q_{11}	0
q_{11}	\emptyset	q_{12}	0
q_{12}	\emptyset	q_{13}	0
q_{13}	\emptyset	q_{14}	0
q_{14}	\emptyset	q_{15}	0
q_{15}	q_{16}	\emptyset	
q_{16}	\emptyset	q_{17}	
q_{17}	q_{18}	\emptyset	
q_{18}	\emptyset	q_{19}	
q_{19}	q_{20}	\emptyset	
q_{20}	\emptyset	q_{21}	
q_{21}	q_{22}	\emptyset	
q_{22}	\emptyset	q_{23}	
q_{23}	\emptyset	q_{24}	
q_{24}	q_f	\emptyset	
q_f	\emptyset	\emptyset	

Figura 5.12

UNIDAD V • EQUIVALENCIAS Y MÁQUINA DE TURING

$$F = \{q_f\}$$

Considerando el L_R como un ejemplo de lo que es un lenguaje regular:

$$L_R = 11000011111111010101001$$

Nos conviene recordar que existe el exponencial cerradura positiva $+$ y cerradura de Kleene $*$

Según se muestra en la **figura 5.13**

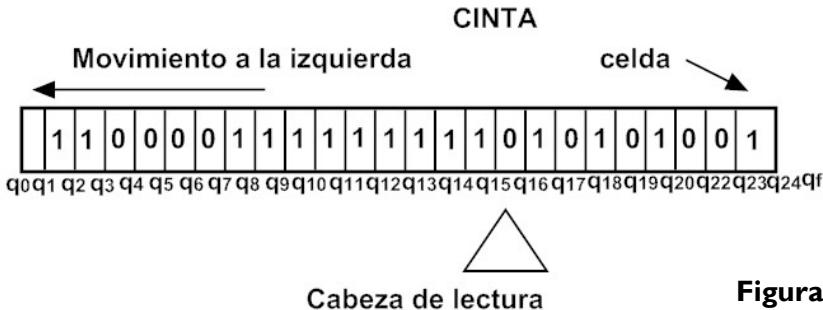


Figura 5.13

Estas **configuraciones** se describen con más detalle a continuación. Se entiende por configuración de un autómata finito, a un par de la forma (q, w) , donde q , es el estado actual, y w la cadena que queda por leer en ese instante. Según la definición anterior, se puede afirmar que la configuración inicial de un autómata finito es el par (q_i, t) siendo t la sentencia o cadena de entrada a reconocer. La configuración final se representa por el par (q_i, ϵ) donde $q_i \in F$, y ϵ indica que no queda nada por entrar de la cinta.

Un movimiento de un autómata finito, puede definirse como el tránsito entre dos configuraciones, y se representa por $(q, aW) \rightarrow (q', W)$ y se debe cumplir que $f(q, a) = q'$.

Lenguaje reconocido por un autómata finito.

Cuando un autómata transita a una configuración final partiendo de la configuración inicial, en varios movimientos, se dice que se ha producido aceptación o reconocimiento de la cadena de entrada. Es decir que dicha cadena, pertenece al lenguaje reconocido por el autómata.

UNIDAD V • EQUIVALENCIAS Y MÁQUINA DE TURING

Por el contrario, cuando el autómata finito no es capaz de llegar a un estado final, se dice que el autómata no reconoce dicha cadena y que por tanto no pertenece al lenguaje.

El lenguaje reconocido por un autómata finito es:

$$L(AF) = \{t | t \in E^*, (q_i, t) \rightarrow (q_i, \epsilon), q_i \in F\}$$

Teoremas

1. Para toda gramática regular G , existe un autómata finito AF , tal que el lenguaje reconocido por el autómata finito es igual al lenguaje generado por la gramática.
 $L(AF) = L(G)$
2. Para todo autómata finito AF , existe una gramática regular G , tal que el lenguaje generado por la gramática es igual al lenguaje reconocido por el autómata finito
 $L(G) = L(AF)$

Corolario

Según el teorema 1), se tiene que $\{L(G)\} \subseteq \{L(AF)\}$ y por el teorema 2) $\{L(AF)\} \subseteq \{L(G)\}$, luego $\{L_R\} = \{L(AF)\} = \{L(G)\}$

La forma habitual de representar los autómatas finitos es mediante un grafo dirigido o diagrama de estados, donde los nodos son los estados (Q) y las aristas (flechas dirigidas) son los símbolos del alfabeto de entrada Σ . Las aristas se construyen según la función de transición, así debe de cumplir $f(q_1, a) = \delta(q_1, a) = q_2$. La representación de esta única función de transición se utiliza usando dos nodos (círculos) etiquetados como q_1, q_2 , el nodo q_1 , se etiqueta como inicial y el nodo q_2 , como final definiendo un doble círculo en el de la **figura 5.14**:

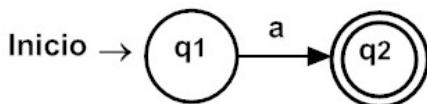
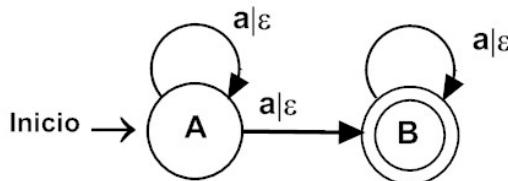


Figura 5.14

Gramática regular lineal por la derecha

1. $A \rightarrow aAB$ (esto equivale a que A como elemento no terminal o nodo puede producir tres elementos: a, A, B)
2. $A \rightarrow a|\epsilon$ (aquí equivale a que A solo puede ser una a o nada o cadena vacía ϵ)
3. $B \rightarrow a|\epsilon$ (aquí al igual que la A, B equivale a que B solo puede ser una a o nada o cadena vacía ϵ)

Esta gramática en forma de autómata quedan dos estados A y B que también son variable no terminales VN, las transiciones pueden ser a minúscula y la cadena vacía ϵ , según se muestra en la **figura 5.15**

**Figura 5.15**

Este es un autómata finito NO determinista porque por cada símbolo, que en este caso son dos a y ϵ se tienen transiciones a: A y B y también se repite la misma transición con los mismos símbolos. La representación formal es:

$$\{Q, \Sigma, \delta, q_0, F\} = M = \{Q, \Sigma, \delta, A, B\}$$

$$Q = \{A, B\} \quad \Sigma = \{a, \epsilon\} \quad \delta(A, a) = A \quad \delta(A, \epsilon) = A \quad \delta(A, a) = B \quad \delta(A, \epsilon) = B$$

En este caso particular el estado inicial q_0 equivale a A($q_0 = A$) y el estado final $F = B$. Las cuatro transiciones antes descritas con δ , se puede expresar en la siguiente tabla de transiciones:

Estados Q	Entradas Σ		Finales
	ϵ	a	
A	{A, B}	{A, B}	0
B	{A, B}	{A, B}	0

UNIDAD V • EQUIVALENCIAS Y MÁQUINA DE TURING

Este es el método más simple que podemos describir de como la expresión regular o lenguaje regular $L_R = a^*$ se puede pasar de una expresión regular lineal por la derecha a un autómata finito no determinista.

Otro ejemplo es: $L_R = \{ab, aab, \dots, abbb, aabb, \dots\} = L_R = \{a^n b^m | n \geq 1 m \geq 1\} = a^+ b^+ = aa^* bb^*$

Ejemplo

Se solicita, pasar ésta expresión regular: $aa^* bb^*$ a un autómata finito no determinista también conocido como diagrama de Moore.

Solución:

Paso 1. Se construye el diagrama de Moore, colocando en primer lugar todos los estados dentro de círculos, marcando con doble círculo el estado final.

Paso 2. El estado inicial se indica con una flecha que lo señala con la palabra INICIO encima.

Paso 3. Para construir las ramas, nos situamos en el primer estado de la tabla de transiciones y se observa que $\delta(q_1, a) = q_2$, entonces se traza una flecha entre q_1 y q_2 , apuntando a q_2 y se coloca encima de la flecha el símbolo del vocabulario o alfabeto de entrada a.

De igual forma se recorres la siguiente tabla de transiciones para cada estado y entrada completándose el diagrama de Moore.

$$\delta = Q \times \Sigma$$

Tabla de transiciones:

		Entradas $\Sigma = \{a, b\}$		Estados finales
Estados $Q = \{q_1, q_2, q_3, q_4\}$		a	b	
q_1	q_2	q_4	0	
q_2	q_2	q_3	0	
q_3	q_4	q_3	0	
q_4	q_4	q_4	1	

UNIDAD V • EQUIVALENCIAS Y MÁQUINA DE TURING

Autómata Finito No determinista con épsilon transiciones (ϵ -transiciones). Según se muestra en la **figura 5.16**

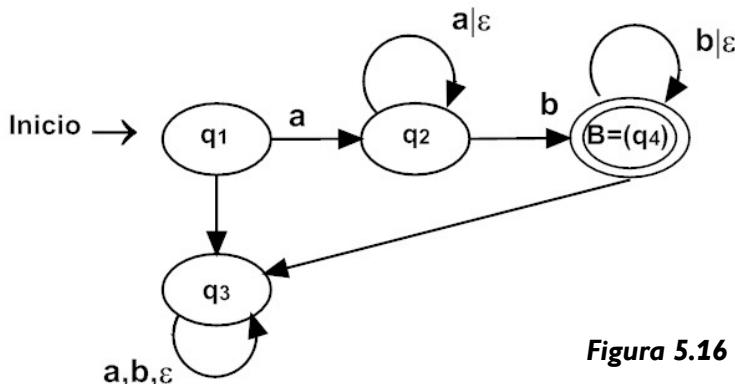


Figura 5.16

Definición formal:

$$M = \{Q, \Sigma, q_1, \delta, F\}$$

Donde $Q = \{q_1, q_2, q_3, q_4\}$, $\Sigma = \{a, b, \epsilon\}$, $\delta = Q \times \Sigma$

Desarrollando cada una de las transiciones nos queda:

$\delta = (q_1, a) = \{q_2, q_3\}$ (Aquí queda claro que es un autómata finito NO determinista)

$\delta = (q_2, a|\epsilon) = \{q_2\}$ (Aquí $a|\epsilon$ significa que puede tenerse un símbolo a o nada, esto es, la cadena vacía)

$\delta = (q_2, b) = \{q_4\}$

$\delta = (q_3, a|b|\epsilon) = \{q_3\}$ (Aquí $a|b|\epsilon$ significa que puede tenerse un símbolo a o bien b, o nada, esto es, la cadena vacía)

$\delta = (q_4, a) = \{q_3\}$

$\delta = (q_4, b|\epsilon) = \{q_4\}$

$S = B = \{q_4\}$

$F = \{q_4\}$

5.1.2 Método para pasar de un AFN con Épsilon Transiciones a una Gramática Regular Lineal por la Derecha.

Recordando que un autómata finito es una la representación gráfica del comportamiento de cualquier sistema que sea posible abstraer como un conjunto de elementos que recibe símbolos de entrada, mismos que alimentan transiciones que pueden integrarse como pasos de un proceso y se genera una salida que son un conjunto de cadenas que conforman un lenguaje. Todos los elementos de un autómata se pueden integrar en una definición formal, misma que puede ser una cuádrupla o una quíntupla de elementos, dependiendo del tipo de gramática. En la siguiente quíntupla de elementos definimos, a un Autómata Finito NO determinístico con Épsilon Transiciones con la siguiente definición formal, donde:

$M = \text{Máquina de Estados Finitos.}$

$M = \{Q, \Sigma, q_0, \delta, F\}$ Para este ejemplo sustituimos el estado inicial q_0 por la letra S, ($S = \{q_0\}$), también sustituimos el estado final F por la letra B, o bien, $F = B$ y, tenemos:
 $M = \{Q, \Sigma, S, \delta, B\}$

Donde $Q = \{S, B\}$ En este caso, en vez de etiquetar los estados q_0, q_1, \dots, q_f como S y B donde el alfabeto tiene:

$$\Sigma = \{a, b, \epsilon\}$$

$$\delta = Q \times \Sigma$$

Se puede seguir una descripción de la siguiente forma:

Entradas o Estados $Q=\{S, B\}$	$\Sigma = \{si, A\}$	no, b	$\epsilon\}$ ϵ	Estados finales
S	A	S	S	0
A	A	\emptyset	A	1

Reglas de producción de una gramática lineal por la derecha

- 1) $S \rightarrow bS$
- 2) $S \rightarrow aA$
- 3) $A \rightarrow aS$
- 4) $A \rightarrow a$

UNIDAD V • EQUIVALENCIAS Y MÁQUINA DE TURING

En la **figura 5.17** se muestra un Autómata finito No determinístico con épsilon transiciones con estas cuatro reglas de producción:

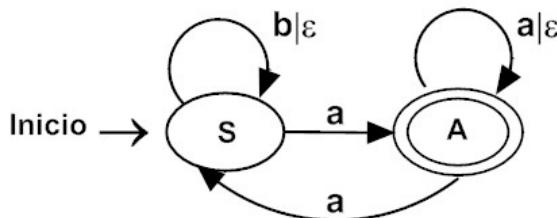


Figura 5.17

Corresponde a la siguiente descripción de un Lenguaje regular (L_R)

$$L_R = b^*aa^*$$

Ejemplo

Con los datos anteriores que describen un autómata finito NO determinístico, se pretende lograr la Gramática regular por la derecha.

Solución:

- **Paso 1.** Con base al diagrama de Moore, colocados en primer lugar todos los estados dentro de círculos (S, A), marcando con doble círculo el estado final (A).
- **Paso 2.** El estado inicial (S) se indica con una flecha que lo señala con la palabra INICIO encima. Esto significa que el símbolo o símbolos etiquetados de la letra S hacia el estado final A, son símbolos requeridos para el lenguaje regular, en este caso son tres símbolos (a, b, ε). En el caso del símbolo a, se tiene una sola vez en la transición de S hacia A, por lo que se coloca el símbolo a. Los símbolos b y ε están dentro de un ciclo, esto significa que puede haber una b o ninguna, por ε que significa la cadena vacía o nada. Para esto, tenemos la cerradura de Kleene y puede ser identificada simplemente como b^* esto significa un símbolo b o nada.
- **Paso 3.** Para terminar el lenguaje regular, tenemos la posibilidad de un símbolo a o nada y este es el lenguaje aceptado, en función de que llega al estado final A.
- **Conclusión:** El lenguaje regular lineal por la derecha queda como: $LR = ab^*a^*$

NOTA: La última a del estado A hacia S ya no es necesaria debido a que no tiene dirección al estado final sino al contrario.

Ejemplo

Con base a la siguiente gramática lineal regular con producciones épsilon por la derecha, genere el diagrama de autómata finito No determinístico correspondiente y desarrolle el lenguaje regular (L_R) con base a las siguientes reglas de producción:

$$S \rightarrow aB|bA|\epsilon$$

$$A \rightarrow aA|bB|\epsilon$$

$$B \rightarrow b|\epsilon$$

Solución.

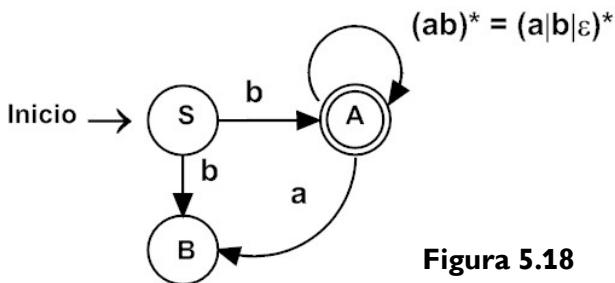
Primero vamos a dibujar el autómata finito determinístico con épsilon transiciones correspondientes a las tres reglas de producción de S, A, B. Para hacer la conversión más evidente u objetiva, se muestra a continuación un árbol de derivación de las tres reglas de producción para identificar los ciclos que en un diagrama de autómatas corresponde a la cerradura positiva (cuando no se tiene a la cadena vacía ϵ) o bien cerradura de Kleene cuando si incluye a la cadena vacía ϵ .

En este árbol de derivación obtenemos la cadena a (por la regla aB), al sustituir la B, tenemos b (con minúscula o símbolo terminal), después nuevamente una a (por la regla aA que se repite debido a que en A tenemos la producción $A \rightarrow aA$ y esto crea un ciclo al igual que $B \rightarrow b$ y de ahí queda $(ab)^*$ ya que en ambas (a, b) se tiene a la cadena vacía épsilon (ϵ).

Con la derivación de $babbab\dots\infty$ generamos el lenguaje regular $L_R = b(ab)^*$

Siguiendo los tres pasos para construir un autómata finito no determinístico o diagrama de Moore descritos en el punto 5.1.2 Tenemos el autómata de la **figura 5.18**.

Este autómata finito No determinístico genera el lenguaje regular $L_R = b(ab)^*$
Definido por una gramática lineal regular por la derecha.
(No se incluye en el lenguaje regular la a que va de A para S porque S no es estado final, tampoco se incluye el símbolo terminal b de S a B porque tampoco B es un estado final).

**Figura 5.18**

5.2.1 Método para pasar de una Gramática Regular Lineal por la Izquierda a un AFN con Épsilon Transiciones.

De la misma forma que realizamos la conversión para una gramática regular lineal por la derecha, aplicaremos los mismos pasos para ésta conversión, solo que ahora partiendo de una gramática cuyas derivaciones son de la forma descrita en el punto V.1 de este material que es una gramática regular lineal por la izquierda, por ejemplo.

$$G_R = (\{0, 1\}, \{A, B\}, A, \{A ::= B1 | 1, B ::= A0\})$$

Tomamos esta forma de representar una gramática regular lineal por la izquierda, para realizar paso a paso una transformación a:

$M = \{Q, \Sigma, q_0, \delta, F\}$ que es la representación formal de una Autómata Finito No determinístico (AFN).

Para este ejemplo sustituimos $\{0, 1\}$ como Σ el cual se define como: $\Sigma = \{0, 1\}$.

Después sustituimos $\{A, B\}$ como el conjunto de estados o bien Q , donde $Q = \{A, B\}$. La variable no terminal A , corresponde al estado inicial q_0 , esto es: ($A = \{q_0\}$), también sustituimos el estado final F por la letra B y $B1$.

Por último, δ también definida como las transiciones, queda definida como las reglas de producción:

$\{A ::= B1 | 1, B ::= A0\}$ que también se pueden representar como:

- 1) $A \rightarrow B1|1$
- 2) $B \rightarrow A0$

$F = B$ y, que se lee como F o estado final es la variable no terminal B .

La anterior gramática regular se puede representar también en un árbol según se muestra en la **figura 5.19**

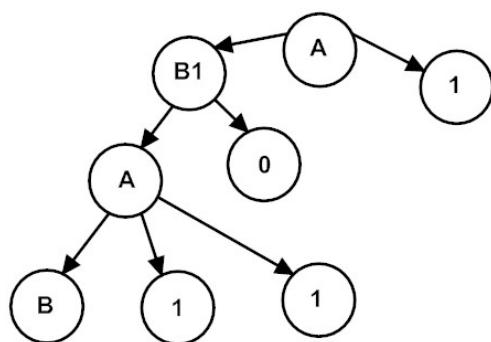


Figura 5.19

Como resultado de un recorrido del árbol de izquierda a derecha tenemos las siguientes cadenas como parte de un lenguaje regular LR. En vez de la B de la segunda regla de producción, (B1101), la sustituimos por A0 y en A, por regla uno la sustituimos por el símbolo terminal 1 y queda 10, luego tomamos los símbolos terminales 1101 y queda: LR = 101101

Dado que la B está produciendo A0 y A produce un 1, podemos tener que $B \rightarrow 10$ y también $A \rightarrow B1$

Esto es 101 (Sustituyendo la B por 10) tenemos cadenas infinitas en la función de A que produce B y B que produce A de tal forma que: $B \rightarrow 10$ y $A \rightarrow 101$ por lo tanto $A \rightarrow (10)1 \rightarrow B1|1$

Formalmente tenemos: $M = \{Q, \Sigma, S, \delta, B\}$

Se transforma la gramática para que no haya regla como:

- 1) $B ::= A0$. Se crea un nuevo A.
- 2) Se crean las reglas $A ::= B1|1$
- 3) Se crea la regla $B ::= 0A'0$
- 4) Se borra la regla $B ::= A0$

La Gramática resultante es: $G_R = (\{0, 1\}, \{A, B, A'\}, A, \{A ::= B1|1, A' ::= B1 | 1, B ::= A' 0\})$

Este autómata finito No determinístico genera el lenguaje regular $L_R = b(ab)^*$ Definido por una gramática lineal regular, donde $b = 1$ y $a = 0$ de lo cual queda $L_R = 1(01)^*$

Se transforma en las reglas de la Gramática lineal por la derecha equivalente:

$G_{R2} = (\{0, 1\}, \{A, B, A'\}, A, \{A ::= 1 A' | 1, A' ::= 0B, B ::= 1 A' | 1\})$ y obtenemos el autómata de la **figura 5.20**

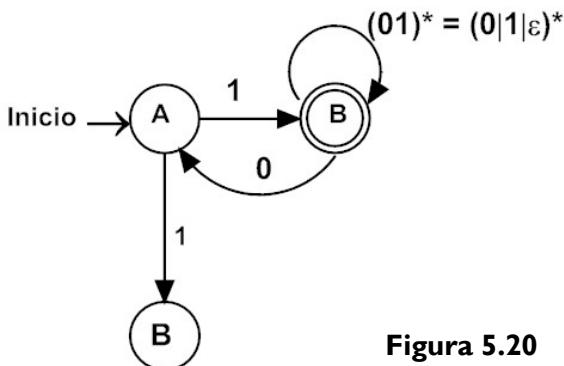


Figura 5.20

Ejemplo

A continuación, se demostrará la equivalencia entre ambas representaciones: Lineal izquierda y Lineal derecha. Gracias a que, para cada gramática lineal derecha existe una gramática lineal izquierda equivalente y viceversa. El algoritmo para convertir una gramática lineal derecha en otra equivalente lineal izquierda tiene cuatro pasos.

1. Se transforma la gramática de forma que no haya ninguna regla en cuya parte derecha este el axioma de la gramática, cuando esto ocurre se denomina axioma inducido, para ello se aplica el siguiente proceso.
 - Se crea un nuevo símbolo no terminal inicial S'
 - Para cada regla de la forma $S ::= x$, donde S es el axioma y $x \in \Sigma^*$, se crea una nueva regla $S' ::= x$.
 - Cada regla de la forma de $A ::= xSy$ donde $A \in \Sigma_N$ y, $x, y \in \Sigma^*$, se transforma en $A ::= x S'y$
2. Se crea un grafo G dirigido:
 - Para cada $A \in \Sigma_N \cup \{\lambda\}$ se crea un nodo λ .
 - Para cada producción $(A ::= aB) \in P(A, B \in \Sigma_N, a \in \Sigma_T)$, se crea un arco etiquetado con a que va del nodo A al nodo B .
 - Para cada producción $(A ::= a) \in P(a \in \Sigma_T)$, se crea un nodo etiquetado con a que va del nodo A al nodo λ .
 - Si existe una regla $S ::= \lambda$, se crea un arco sin etiqueta desde el nodo del axioma al nodo λ .

UNIDAD V • EQUIVALENCIAS Y MÁQUINA DE TURING

3. Se crea otro grafo G' a partir de G :
 - Se intercambian las etiquetas del axioma S y λ .
 - Se invierte la dirección de todos los arcos.
4. Se transforma en un conjunto de reglas.
 - Para cada nodo, se crea un símbolo no terminal de la gramática, excepto para el etiquetado como λ (cuyo símbolo puede ser épsilon ϵ).
 - Para cada arco etiquetado con $a \in \Sigma_T$ que va del nodo $A \in \Sigma_N \cup \{\lambda\}$, se crea una producción $A ::= Ba$
 - Si existe un arco del nodo del axioma al nodo de λ , se crea una regla $S ::= \lambda$.

El algoritmo para realizar el paso inverso es similar al descrito.

Supóngase la siguiente gramática lineal derecha:

$$G = (\{0,1\}, \{A, B\}, A, P)$$

$$P = \{(A ::= 1B), (A ::= \lambda), (B ::= 0A), (B ::= 0)\}$$

La gramática lineal izquierda equivalente se formará en los siguientes cuatro pasos.

1. Se trasforma la gramática de forma que no aparezca la A en la parte derecha de la producción $B ::= 0A$
 - Se crea un nuevo símbolo A'
 - Se crea las reglas $A' ::= 1B$ y $A' ::= \lambda$
 - Se crea la regla $B ::= 0A'$
 - Se borra la regla $B ::= 0A$

La gramática queda como: $G' = (\{0,1\}, \{A, B, A'\}, A, P)$

$$P' = \{(A ::= 1B), (A ::= \lambda), (A' ::= 1B), (A' ::= \lambda), (B ::= 0A'), (B ::= 0)\}$$

Como la gramática por definición no puede tener la regla $A' ::= \lambda$, ya que A' en todas las producciones en las que aparezca en la parte derecha quedando la gramática como: $G'' = (\{0,1\}, \{A, B, A'\}, A, P)$

$$P'' = \{(A ::= 1B), (A ::= \lambda), (A' ::= 1B), (B ::= 0A'), (B ::= 0)\}$$

2. Se crea un grafo que represente las transiciones de esta gramática.

3. Se invierte el grafo

Se transforma el grafo en la siguiente gramática: $G''' = (\{0,1\}, \{A, B, A'\}, A, P)$

$$P''' = \{(A ::= B0), (A ::= \lambda), (A' ::= B0), (B ::= A'1), (B ::= 1)\}$$

UNIDAD V • EQUIVALENCIAS Y MÁQUINA DE TURING

En la **figura 5.21** se muestra el Autómata con la gramática lineal izquierda

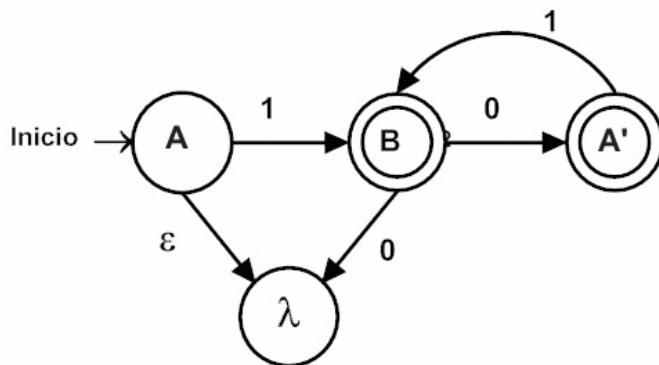


Figura 5.21 Diagrama de estados para la expresión regular $1|10|(101)^*$

El Autómata con la gramática lineal derecha siguiendo los cuatro pasos ya descritos se muestra en la **figura 5.22**

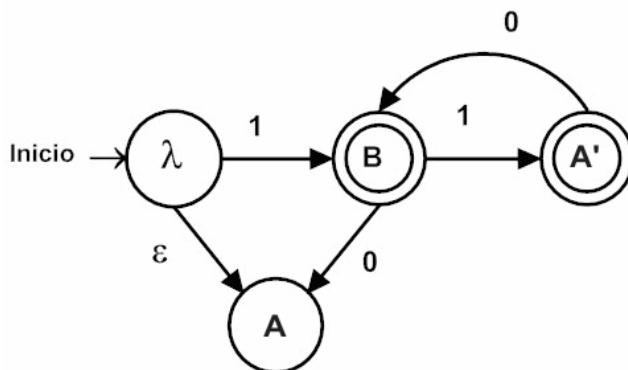


Figura 5.22 Diagrama de estados inverso para la expresión regular $1|10|(101)^*$

5.2.2 Método para pasar de un AFN con Épsilon Transiciones a una Gramática Regular Lineal por la Izquierda.

Tomando como base el siguiente autómata finito No determinista con épsilon transiciones el cual corresponde a una gramática lineal regular, se va a describir un método simple para obtener la comprobación de que corresponde a una gramática lineal regular por la izquierda.

$G_R = (\{ a, b \}, \{A, B, C, D\}, A, S ::= A, F ::= D, P)$ El diagrama para esta gramática está en la figura 5.23

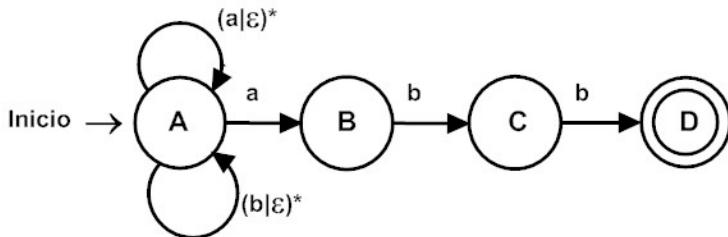


Figura 5.23 Diagrama de transiciones para la gramática
 $G_R = (\{ a, b \}, \{A, B, C, D\}, A, S ::= A, F ::= D, P)$

Las reglas de producción P de la gramática regular son:

$$\begin{aligned} A &::= A | aB | a | b | \epsilon \\ B &::= bC \\ C &::= bD \\ D &::= \epsilon \end{aligned}$$

El árbol correspondiente para visualizar el tipo de gramática es el siguiente:
La anterior gramática regular se puede representar también en un árbol en la figura 5.24

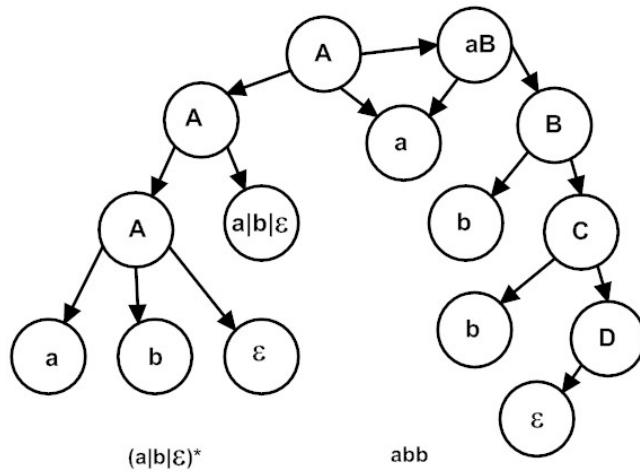


Figura 5.24 Árbol de derivación para la gramática anterior
 $G_R = (\{a, b\}, \{A, B, C, D\}, A, S ::= A, F ::= D, P)$

Este árbol muestra un ciclo en A , que representa la posibilidad de que se repita **a** o **b** o ϵ , esto es, $(a|b)^*$ y corresponde a la cerradura de Kleene, también observamos que además del ciclo en A , tenemos la cadena **abb**. Uniendo todos los ítems tenemos: $(a|b)^*abb$

Aplicando los cuatro pasos descritos a continuación, cambiamos esta gramática lineal o regular por la derecha a la izquierda.

1. Se crea el símbolo A'
2. Se crea las reglas $A' ::= A'$, $A' ::= a|b|\epsilon$ $A' ::= aD$, $A' ::= \lambda$
3. Se crea la regla $D ::= bC$, $C ::= bB$, $B ::= \lambda$
4. Se borra la regla $A ::= A|aB|a|b|\epsilon$

Se tienen las siguientes reglas de producción P de la gramática:

$$\begin{aligned}
 A' &::= A'|a|b|\epsilon|aD|\lambda \\
 D &::= bC \\
 C &::= bB \\
 B &::= \lambda \\
 G_R &= (\{a, b\}, \{A, B, C, D\}, A, S ::= A, F ::= D, P)
 \end{aligned}$$

Correspondiente al autómata que se muestra en la **figura 5.25**

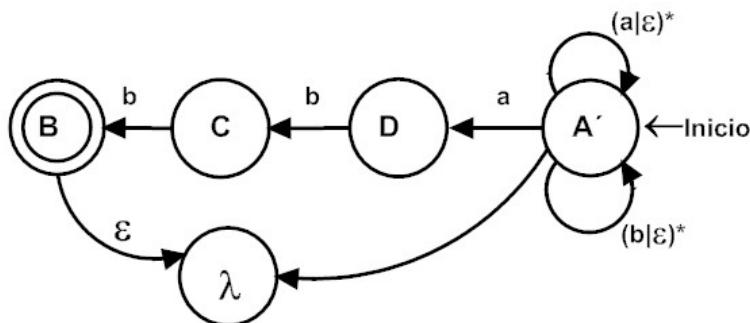


Figura 5.25 Diagrama de transiciones para la expresión regular $a^*|b^*abb$

En la **figura 5.26** se muestra el árbol correspondiente para visualizar éste tipo de gramática

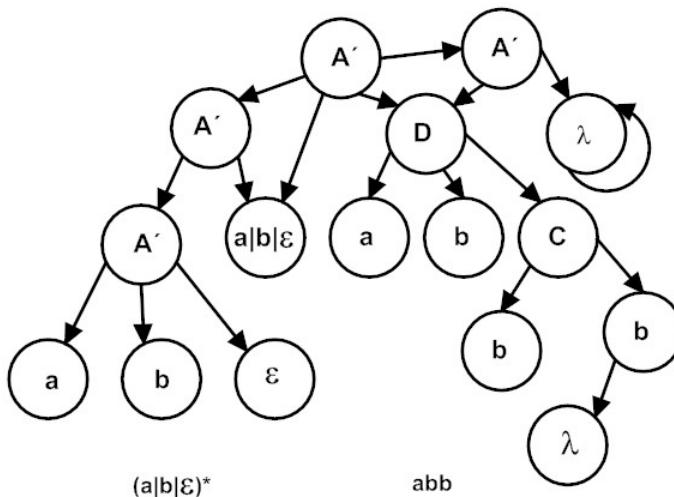


Figura 5.26 Árbol de derivación para $a^*|b^*abb$

5.3 Generación de los ‘Items’ de un Gramática

La generación de los “Items” de una gramática los podemos describir como el recorrido que se realiza a través de todos y cada uno de los elementos léxicos (simbólicos), sintácticos (de orden), mismos que quedan definidos con las reglas de producción de la gramática. Éstas objetivamente se siguen en el recorrido de un árbol de análisis sintáctico, tal como se demostró en el subtema anterior, en el “item” $(a|b|\epsilon)^*abb$.

La generación de “Items” se puede realizar de dos formas:

- 1) Manual, observando los árboles de derivación, los diagramas de autómatas donde se determinan el conjunto de elementos simbólicos que integrarán cadenas o bien palabras que serán parte del lenguaje regular aceptado. Esto se basa en la definición formal del lenguaje con el tipo de gramática. Esta definición, también se puede programar en diferentes lenguajes como se muestra en la simulación donde que se comenta en la siguiente forma de generar “Items”.
- 2) Automática o programada, esto se permite gracias a que varios lenguajes de programación como C, JAVA, awk, etc., aceptan expresiones regulares que permite separar “Items” en dos tipos:

Cadenas rechazadas y Cadenas aceptadas por el lenguaje, según las reglas de producción que se programan tal como ya se explicó.

En este subtema se muestra la simulación de un autómata finito determinista codificado en lenguaje C:

```
s = s0;
c = sigCar();
while (c!= eof) {
    s = mover(s, c);
    c = sigCar();
}
If (s está en F) return "si";
else return "no";
```

UNIDAD V • EQUIVALENCIAS Y MÁQUINA DE TURING

Entrada: Una cadena de entrada x , que se termina con un carácter de fin de archivo **EOF**. Un AFD D con el estado inicial s_0 , que acepta estados F , y la función de transición mover.

Salida: Responde “sí” en caso de que D acepte a x , “no” en caso contrario.

Método: Aplicar el algoritmo arriba transcrita del libro Aho, Alfred V., “Compiladores, principios, técnicas y herramientas, segunda edición, Edit. Pearson, México 2008. La cadena de entrada x , La función $mover(s, c)$ proporciona el estado para el cual hay un flanco desde el estado s sobre la entrada c . La función $sigCar$ devuelve el siguiente carácter de la cadena de entrada x .

Ejemplos:

- 1) Genere el autómata y la expresión regular que acepte caracteres alfabéticos de la letra o símbolo **a**, **b**, **c**, **d**, ..., **z** (esto es, de la letra **a** la letra **z** en minúsculas).

Respuesta: Tenemos el siguiente autómata.

Escribir dentro de corchetes la **a-z** significa abarcar un rango de caracteres o símbolos de la letra **a** hasta la letra **z** en minúsculas. En awk la expresión regular es: **/[a-z]/** misma que corresponde a la gramática regular:

$G_R = (\{a-z\}, \{q_0, q_f\}, P \{ q_0 ::= a-z \})$ Según se muestra en la **figura 5.27**

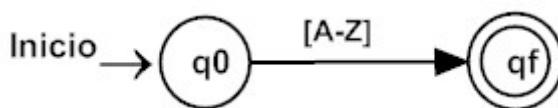


Figura 5.27 Diagrama de transiciones para los símbolos alfabéticos con minúsculas [a-z]

La expresión regular correspondiente en una expresión regular que acepta directamente el lenguaje intérprete conocido como awk el cual recibe su nombre por sus creadores, a de Alfred Aho, w de Peter Weinberger y k de Brian Kernighan.

- 2) Genere el autómata y la expresión regular que acepte caracteres alfabéticos de la letra o símbolo **a, b, c, d, ..., z** (esto es, de la letra **a** la letra **z** en minúsculas). Además, que acepte cualquier combinación de estos caracteres en cualquier orden. Esto significa que se dará la posibilidad de que se tengan como cadenas aceptadas las palabras que se forman con letras minúsculas de cualquier tamaño o número de caracteres y se rechazan cadenas que incluyan letras mayúsculas.

Respuesta:

Tenemos el siguiente autómata.

$G_R = (\{a-z\}, \{ q_0, q_f \}, P\{ q_0 ::= a-z \})$ que se muestra en la **figura 5.28**

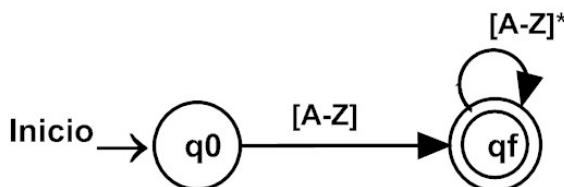


Figura 5.28 Diagrama de transiciones para los símbolos alfabéticos con minúsculas $[a-z]^*$

La expresión regular correspondiente en awk queda simplemente como: $/[a-z]^*/$ y con este autómata, se tiene que para que se acepte una cadena deberá tener al menos un símbolo que puede ser una a, b, c, d, e, f, ..., z.

- 3) Genere el autómata y la expresión regular que acepte exclusivamente caracteres numéricos con o sin signo. Esto significa que se dará la posibilidad de que se tengan como cadenas aceptadas las palabras que se forman con números del 0 al 9 positivos, sin signo y cantidades negativas.

Respuesta:

Tenemos el siguiente autómata.

$G_{R2} = (\{0-9\}, \{ q_0, q_f \}, P\{ q_0 ::= [+] - | \epsilon | [0-9]^* \})$ según la **figura 5.29**

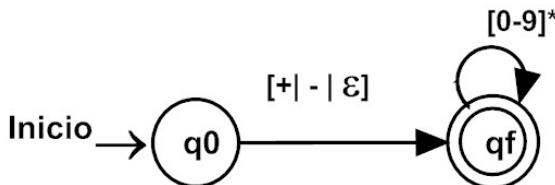


Figura 5.29 Diagrama de transiciones para los símbolos numéricos con o sin signo, positivo o negativo $[+|-|\epsilon|][0-9]^*$

La expresión en awk para aceptar cadenas exclusivamente numéricas del 0 al 9 en números enteros positivos, negativos o sin signo es la siguiente: `/^[-+]?[0-9]*/`. Esto significa para el lenguaje que al dar como circunflejo, tendrá la posibilidad de los símbolos + o - o nada que es lo que significa el signo de interrogación ?, hace que el contenido de los corchetes sea opcional y al final la expresión regular que ya hemos utilizado en el ejemplo anterior que es el aceptar números del 0 al 9 en cualquiera de sus combinaciones que es la representación del asterisco * como cerradora de Kleene, cerrando la expresión con otro slash /.

Como estos tres ejemplos de generación de “Items” con expresiones regulares podemos obtener cualquier cantidad de cadenas aceptadas o rechazadas, dependiendo de lo que necesitamos definir dentro de las reglas de producción, esto es una ventaja significativa de los lenguajes regulares.

5.4 Simplificación de un AFD

La simplificación de un Autómata Finito Determinístico es obtener otro autómata equivalente, esto es, otro autómata que acepte el mismo lenguaje que acepta el autómata original para que ambos (el original y el obtenido se puedan considerar equivalentes).

El Teorema de Myhill-Nerode permite la minimización de Autómatas Finitos desde uno que puede tener un número mayor de estados al que se obtiene de otro autómata equivalente con un menor número de estados del autómata original, como ya se vio anteriormente.

Ahora vamos a explicar otra forma de simplificar o reducir el número de estados de un autómata comprobando que ambos son equivalentes, esto es, aceptan las mismas cadenas a pesar de que vean diferentes y tengan diferente número de estados. A este método lo conocemos como conjuntos de estados y se explica a continuación.

Paso 1. Empezamos con elaborar una tabla de transiciones del autómata, la cual contiene en la primera columna el conjunto de todos los estados del autómata, el alfabeto y sus transiciones a cada uno de los estados. Esto es, $Q \times \Sigma$ y marcamos una última columna donde colocamos un 0 si el estado no es de aceptación y un 1 si es un estado final o de aceptación (dibujado por lo general con doble círculo).

Paso 2. Organizamos los estados del autómata en los dos conjuntos cuyos elementos son:

- 1) Un conjunto con los estados de aceptación, según nuestros diagramas, éstos son todos aquellos estados que tienen doble círculo, esto significa que, si se llega a ellos mediante símbolos que le preceden, las cadenas que se forman son aceptadas. Estos estados también se les conoce como estados de equivalencia de longitud uno o bien de clase cero, abreviando C0 significa el conjunto cociente de la relación de equivalencia de la longitud que es cero, porque en éstos estados, cualquiera de ellos, se llega a un estado de equivalencia de aceptación cero.
- 2) El otro conjunto se integra con los estados que no son de aceptación, esto es, estados que se les conoce como clase uno, abreviando C1.

¿Por qué se dice que hay estados equivalentes de longitud cero?

Respuesta: Porque si estamos en algún estado de partida (del lado de inicio) y se recibe una cadena de longitud cero. Si se llega inmediatamente a un estado de aceptación, se dice que esos estados son equivalentes de longitud cero.

Paso 3. Se toman parejas de estados de la misma clase C_0 y C_1 observando en el diagrama de autómata para cada pareja, ¿A dónde nos vamos con cada uno de los símbolos del alfabeto? Esto para determinar:

¿Cuáles de los estados que tenemos dentro de cada una de las clases siguen siendo equivalentes cuando reciben una cadena de longitud una unidad mayor? Para esto, tomamos parejas de estados de una misma clase y veremos a qué estado se llega para cada una de las posibles entradas. Si los estados destino para cada entrada pertenecen a la misma clase, los estados de partida seguirán siendo equivalentes.

Paso 4. Se toman parejas de estados de la misma clase C_0, C_1, C_2 , etc. Esto hasta que no tengamos más clases. Con cada uno de los grupos por clases, se agrupan en un nuevo estado y se construye el nuevo autómata.

Paso 5. Este paso ya se puede considerar opcional pero muy útil, ya que se generan “ítems” del autómata original y se verifican con los que se pueden generar con el nuevo autómata mínimo o reducido si las cadenas generadas son iguales podemos comprobar que se tratan de autómatas equivalentes.

Ejemplo:

Con el autómata de la **figura 5.30** realice la reducción a su forma mínima equivalente

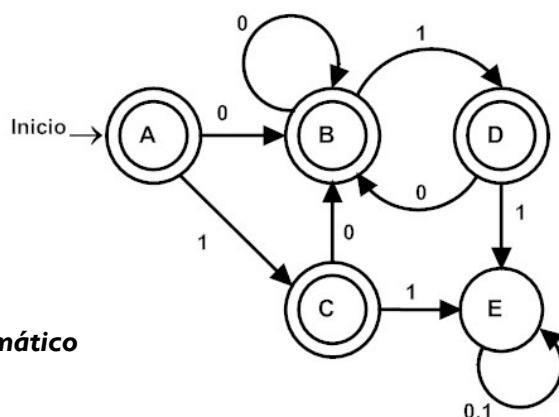


Figura 5.30. Diagrama esquemático para mostrar un AFD de $[0|1|\epsilon]$

UNIDAD V • EQUIVALENCIAS Y MÁQUINA DE TURING

Solución.

Paso 1. Empezamos con elaborar una tabla de transiciones del autómata. $Q \times \Sigma = \{0,1\}$

ESTADOS Q	Alfabeto $\Sigma = 0$	Alfabeto $\Sigma = 1$	Estados Finales NO=0 SI = 1	¿A dónde voy con cada uno de los símbolos del alfabeto? 0 o 1
A	B	C	1	Siguiendo el diagrama, estoy en A y con 0 voy a B y con 1 voy a C. En tercera columna A es estado final y coloco un 1
B	B	D	1	Siguiendo el diagrama, estoy en B y con 0 me quedo en B y con 1 voy a D. En tercer columna B si es estado final y coloco un 1
C	B	E	1	Siguiendo el diagrama, estoy en C y con 0 me voy a B y con 1 voy a E. En tercer columna C si es estado final y coloco un 1
D	B	E	1	Siguiendo el diagrama, estoy en D y con 0 me voy a B y con 1 voy a E. En tercer columna D si es estado final y coloco un 1
E	E	E	0	Siguiendo el diagrama, estoy en C y con 0 me voy a E y con 1 voy a E. En tercera columna E no es estado final y coloco un 0

UNIDAD V • EQUIVALENCIAS Y MÁQUINA DE TURING

Paso 2. Organizamos los estados del autómata en los dos conjuntos.

- 1) Clase cero con los estados de aceptación (estados con doble círculo)
 $C_0 = \{A, B, C, D\}$
- 2) Clase uno con los estados que no son de aceptación $C_1 = \{E\}$

Antes de continuar al siguiente paso, es importante dar respuesta a la siguiente cuestión:

¿Por qué se dice que hay estados equivalentes de longitud cero?

Respuesta: Porque si estamos en algún estado de partida (del lado de inicio) y se recibe una cadena de longitud cero. Si se llega a un estado de aceptación, se dice que esos estados son equivalentes de longitud cero.

Paso 3. Se toman parejas de estados de la misma clase C_0 y C_1 observando en el diagrama de autómata para cada pareja, ¿A dónde nos vamos con cada uno de los símbolos del alfabeto? Esto para determinar:

¿Cuáles de los estados que tenemos dentro de cada una de las clases siguen siendo equivalentes cuando reciben una cadena de longitud una unidad mayor? Para esto, tomamos parejas de estados de una misma clase y veremos a qué estado se llega para cada una de las posibles entradas. Si los estados destino para cada entrada pertenecen a la misma clase, los estados de partida seguirán siendo equivalentes.

E_0 nos permite definir al conjunto de estados equivalentes de longitud cero.

A E_0 B Esto significa que la pareja de estados A y B son estados equivalentes de longitud cero y de lo que se trata en este paso, con el ejemplo del autómata es verificar que estados siguen siendo equivalentes, pero ahora con longitud uno, esto es, E_1

Q/ E_0 Esto significa estados de equivalencia de longitud cero y Q/ E_1 es un estado de equivalencia de longitud uno.

Q/ E_0 = $\{C_0 = \{A, B, C, D\}, C_1 = \{E\}\}$ Estados de equivalencia de longitud cero dividida en dos clases, C_0 para estados de aceptación y C_1 para estados de no aceptación.

Paso 4. Se toman parejas de estados de la misma clase C_0, C_1, C_2 , etc. Esto hasta que no tengamos más clases. Con cada uno de los grupos por clases, se agrupan en un nuevo estado y se construye el nuevo autómata, como se muestra en la **figura 5.31**.

UNIDAD V • EQUIVALENCIAS Y MÁQUINA DE TURING

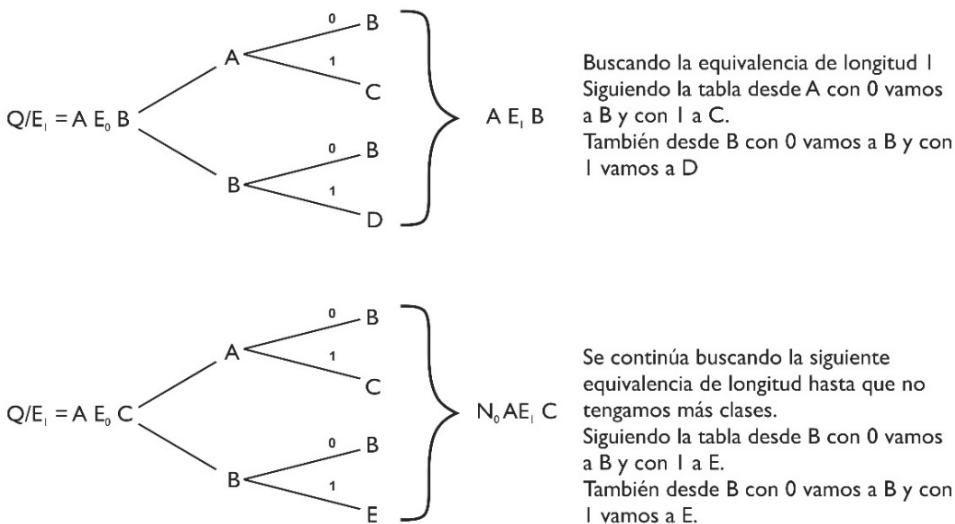


Figura 5.31 Diagrama de transiciones simplificado con diferentes longitudes y clases

Para este ejemplo en particular tenemos tres clases de equivalencia C_0, C_1 y C_2 . Éstas las representamos de la siguiente forma:

$$Q/E_1 = 4 = \{C_0 = \{A, B\}, C_1 = \{E\}, C_2 = \{C, D\}\}$$

Cada clase de equivalencia representará un nuevo estado, esto es, C_0 es A, C_1 es B y C_2 es C y el nuevo autómata queda en la **figura 5.32**

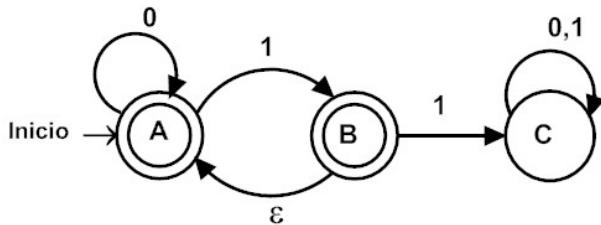


Figura 5.32 Diagrama de transiciones para la expresión regular 0^*1^*

Paso 5. El lenguaje o cadenas aceptadas que aceptan los dos autómatas es el mismo y este es: $L = \{\epsilon, 0, 1, 0^*, 100^*, 0^*1, 0^*10, 0^*(10)^*, 10, 1\}$ Lo que queda demostrado.

5.5 Máquina de Turing.

La máquina de Turing (abreviado MT desde aquí en adelante), recibe su nombre por su creador, el visionario matemático inglés Alan Mathison Turing quién en 1937 publicó un famoso artículo, donde demostró el teorema de “incompletitud” de Kurt Gödel quién lo desarrolló en 1931. La MT puede considerarse como el origen oficial de la Informática Teórica.

Turing también es conocido como el padre de la inteligencia artificial, Turing inició con su máquina, sencilla pero ingeniosa emplea un mecanismo que formaliza el concepto de algoritmo, se introduce para demostrar la validez de los postulados de Gödel. Es decir, Turing demostró que existen problemas irresolvibles, inasequibles para cualquier máquina de Turing, y por ende, actualmente, para cualquier computadora.

Al ser la máquina de Turing un modelo ideal de máquina capaz de adoptar una infinidad de estados posibles, resulta obvio pensar que su realización está fuera del alcance práctico. Sin embargo, debido a la gran capacidad de las computadoras actuales, la máquina de Turing resulta ser un modelo adecuado de su funcionamiento.

¿Qué es y cómo se comporta la MT (Máquina de Turing)?

Consideramos importante para este material el recordar que dentro de un dispositivo electrónico como puede ser una computadora, se tienen dos elementos básicos, el primero es su funcionamiento que para esta descripción se refiere a los elementos físicos o hardware con su interacción, esto corresponde al ámbito del diseño digital, circuitería, alimentación, etc. El segundo elemento es lo que corresponde a su comportamiento, que es lo referente a lo lógico, también conocido como elementos de software, aquí se incluyen los elementos como el lenguaje y las decisiones que se toman para el procesamiento de los estímulos de entrada que generan salidas o resultados en base a la lógica programada o descrita en algún lenguaje de programación.

La MT es un dispositivo de reconocimiento de lenguajes formales o regulares, es más general que cualquier autómata finito y cualquier autómata de pila, debido a que ellas pueden reconocer tanto los lenguajes regulares, como los lenguajes independientes de contexto y además otros tipos de lenguajes, como los lenguajes de programación.

UNIDAD V • EQUIVALENCIAS Y MÁQUINA DE TURING

La MT se comporta de una forma muy simple comparada con las modernas computadoras actuales, sin embargo, su importancia es gracias a que con un mecanismo muy sencillo que consiste en: un control finito, una cabeza lectora y una cinta donde puede haber caracteres o símbolos, mismos que eventualmente tienen la palabra de entrada. La cinta es de longitud infinita hacia la derecha, hacia donde se extiende indefinidamente, llenándose los espacios con el carácter blanco (que, para este ejemplo, lo representaremos con un símbolo “**t**”). La cinta no es infinita hacia la izquierda, por lo que hay un cuadro de la cinta que es el extremo izquierdo, la MT la cabeza lectora es de lectura y escritura, por lo que la cinta puede ser modificada en curso de ejecución.

Además, en la MT la cabeza se mueve bidireccionalmente (izquierda y derecha), por lo que puede pasar repetidas veces sobre un mismo segmento de la cinta. Estos últimos elementos descritos que son de funcionamiento fueron simplificados para dejar muy claro su importancia y consiste es haber sido la primera máquina que procesa el primer lenguaje formal que dio origen a muchos otros lenguajes de programación.

La MT es un modelo que está conformado por un alfabeto de entrada y uno de salida, un símbolo especial llamado blanco (normalmente **b**, Δ o **0**), un conjunto de estados finitos y un conjunto de transiciones entre dichos estados. Su funcionamiento se basa en una función de transición, que recibe un estado inicial y una cadena de caracteres (la cinta, la cual es finita por la izquierda) pertenecientes al alfabeto de entrada. Luego va leyendo una celda de la cinta, borrando el símbolo, escribir el nuevo símbolo perteneciente al alfabeto de salida y finalmente avanza a la izquierda o a la derecha (solo una celda a la vez), repitiendo esto según se indique en la función de transición, para finalmente detenerse en un estado final o de aceptación, representando así la salida. Todo esto ya se esquematizo con anterioridad en el presente capítulo.

La MT consta de un cabezal lector y escritor, así como de la cinta infinita del lado izquierdo y derecho, mismo que pasa por el cabezal de lectura-escritura, también es posible borrar el contenido anterior y escribe un nuevo valor. Las operaciones que se pueden realizar en esta máquina se limitan a avanzar el cabezal de lectura-escritura hacia la derecha o izquierda. El cálculo o proceso es determinado a partir de una tabla de estados de la forma: (estado, valor, nuevo estado, nuevo valor y dirección. Esta tabla toma como parámetros el estado actual de la máquina y el carácter leído de la cinta, dando la dirección para mover el cabezal, el nuevo estado de la máquina y el valor a ser escrito en la cinta. Con este aparato extremadamente sencillo es posible realizar cualquier cómputo o calculo que una computadora digital sea capaz de realizar.

UNIDAD V • EQUIVALENCIAS Y MÁQUINA DE TURING

Mediante este modelo teórico y el análisis de complejidad de algoritmos, fue posible la categorización de problemas computacionales de acuerdo a su comportamiento, apareciendo así, el conjunto de problemas denominados P y NP, cuyas soluciones en tiempo polinómico (P) son encontradas según el determinismo y no determinismo o No polinómico (NP), respectivamente de la MT. Se puede probar matemáticamente que para cualquier programa de computadora es posible crear una MT equivalente. Esta prueba resulta de la Tesis de Church-Turing, formulada por Alan Turing y Alonzo Church, de forma independiente a mediados del siglo XX.

¿Cómo funciona una MT?

Una MT es un dispositivo que transforma un INPUT (entrada) en un OUTPUT (salida) después de algunos pasos. Tanto el INPUT como el OUPUT constan de números en código binario (ceros y unos). En su versión original la máquina de Turing consiste en una cinta infinitamente larga con unos y ceros que pasa a través de una caja. La caja es tan fina que solo el trozo de cinta que ocupa un bit (0 ó 1) está en su interior. Tiene una serie de estados internos finitos que también se pueden numerar en binario. Para llevar a cabo algún algoritmo, la máquina se inicializa en algún estado interno arbitrario. A continuación, se pone en marcha y la máquina lee el bit que se encuentra en ese momento en su interior y ejecuta alguna operación con ese bit (lo cambia o no, dependiendo de su estado interno). Después se mueve hacia la derecha o hacia la izquierda, y vuelve a procesar el siguiente bit de la misma manera. Al final se para, dejando el resultado al lado izquierdo, por ejemplo.

Una instrucción típica podría ser: $01 \rightarrow 110111$ La traducción es como sigue: si la máquina se encuentra en el estado interno 0 y lee 1 en la cinta, entonces pasará al estado interno 1101 (equivale al 13 en decimal), escribirá 1 y se moverá hacia la izquierda un paso (la cinta se moverá hacia la derecha). A continuación es conveniente inventar una notación para la secuencia del INPUT. Esta notación se llama notación binaria expandida. Consiste en cambiar la secuencia original binaria por otra construida de la siguiente forma: el 0 se cambia por 0 y el 1 por 10 y se ponen un cero a la izquierda y/o a la derecha del resultado si empieza o acaba en 1 respectivamente. Así por ejemplo, el número 13 que en binario es 1101 es en binario expandido 1010010 con un cero delante por esta última regla 01010010. Para volver al original hay que contraer el binario expandido con la siguiente regla: Empezamos a leer por la izquierda el binario expandido. Cuando encontramos un 0 tomamos nota de cuántos 1 hay hasta llegar al siguiente 0 y lo escribimos. Si encontramos que hay dos 0 seguidos, apuntaríamos un 0

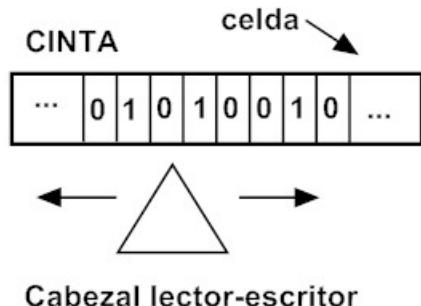
UNIDAD V • EQUIVALENCIAS Y MÁQUINA DE TURING

porque no habría ningún 1. Veamos con el 13 cómo se haría. El primer 0 se encuentra en la primera posición y el siguiente 0 está en la posición 3. Entre los dos solo hay un 1. Lo anotamos. Seguidamente hay un 1, y después un 0, entonces apuntamos 1 porque hay un 1 entre medias de ellos. Esto es lo que se hace sucesivamente y encontramos: **1101** que es el número original.

La MT consta de un cabezal lector/escritor y una cinta infinita en la que el cabezal lee el contenido, borra el contenido anterior y escribe un nuevo valor. Las operaciones que se pueden realizar en esta máquina se limitan a: avanzar el cabezal lector/escritor hacia la derecha como se muestra en la **figura 5.33**.

Visualización de una máquina de Turing en la que se ve el cabezal y la cinta que se lee. Avanzar el cabezal lector-escritor hacia la derecha-izquierda.

El cómputo es determinado a partir de una tabla de estados de la forma: (estado, valor) (nuevo estado, nuevo valor, dirección). Esta tabla toma como parámetros el estado actual de la máquina y el carácter leído de la cinta, dando la dirección para mover el cabezal, el nuevo estado de la máquina y el valor a ser escrito en la cinta. La memoria será la cinta la cual se divide en espacios de trabajo denominados celdas, donde se pueden escribir y leer símbolos. Inicialmente todas las celdas contienen un símbolo especial denominado “blanco”. Las instrucciones que determinan el funcionamiento de la máquina tienen la forma, “si estamos en el estado x leyendo la posición y , donde hay escrito el símbolo z , entonces este símbolo debe ser reemplazado por este otro símbolo, y pasar a leer la celda siguiente, bien a la izquierda o bien a la derecha”. La MT puede considerarse como un autómata capaz de reconocer lenguajes formales. En ese sentido es capaz de reconocer los lenguajes recursivamente enumerables, de acuerdo a la jerarquía de Chomsky. Su potencia es, por tanto, superior a otros tipos de autómatas, como el autómata finito, o el autómata con pila, o igual a otros modelos con la misma potencia computacional.



Cabezal lector-escritor

Figura 5.33 Diagrama esquemático de la máquina de Turing para la cadena 01010010

UNIDAD V • EQUIVALENCIAS Y MÁQUINA DE TURING

Construcción modular y descripción formal de las máquinas de Turing. Las máquinas de Turing se pueden representar mediante grafos particulares, también llamados diagramas de estados finitos o autómatas de los cuales se muestra en la **figura 5.34**

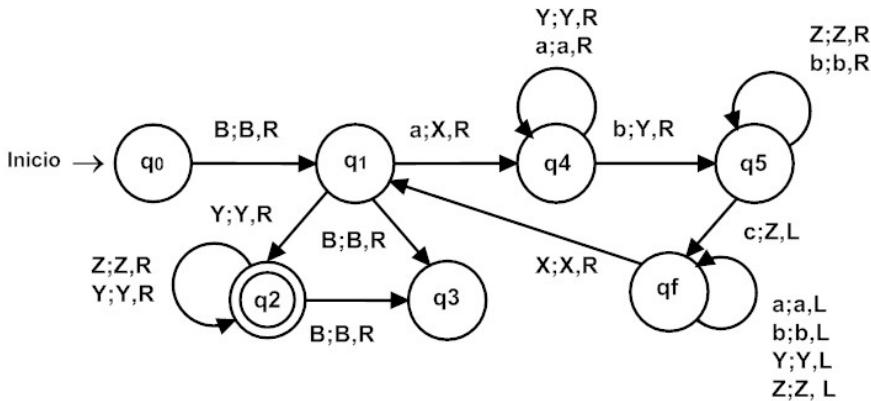


Figura 5.34 Autómata de pila de la máquina de Turing

Esta Máquina de Turing está definido sobre el alfabeto $\Sigma = \{a, b, c\}$, posee el conjunto de estados $Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6\}$, con las transiciones que se pueden ver. Su estado inicial es q_1 y el estado final es q_2 , con doble circulo, el lenguaje de salida $= \{X, Y, Z, B\}$ siendo B el símbolo denominado Blanco . Esta Máquina reconoce la expresión regular de la forma $\{a^n b^n c^n, n \geq 0\}$

Los estados se representan como vértices, al igual que otros autómatas, etiquetados con su nombre en el interior. Una transición desde un estado a otro, se representa mediante una arista dirigida que une a estos vértices, y esta rotulada por símbolo que lee el cabezal o el símbolo que escribirá el cabezal, movimiento del cabezal. El estado inicial se caracteriza por tener una arista que llega a él, proveniente de ningún otro vértice.

El o los estados finales se representan mediante vértices que están encerrados a su vez por otra circunferencia. Descripción instantánea (abreviada a partir de aquí como DI) Secuencia de la forma $\alpha_1 q \alpha_2$ donde α_1, α_2 y q Describe la situación de una MT La cinta contiene la cadena $\alpha_1 \alpha_2$ seguida de infinitos blancos. El cabezal señala el primer símbolo de α_2 .

UNIDAD V • EQUIVALENCIAS Y MÁQUINA DE TURING

Ejemplo:

Para la MT= $(\{p, q\}, \{0, 1\}, \{0, 1, x\}, \delta, p, \Delta, \{q\})$

Con las transiciones

$$\delta(p, 1) = (p, x, D)$$

$$\delta(p, 0) = (p, 0, D)$$

$$\delta(p, \Delta) = (q, \Delta, D)$$

Realizaremos la DI para la cinta 1011

Ejemplo

p1011ΔΔ... → xp011ΔΔ... → x0p11ΔΔ... → x0xp1ΔΔ... → x0xxpΔΔ... → x0xxqΔΔ...

Definimos una máquina de Turing sobre el alfabeto $\{0, 1\}$, donde 0 representa el símbolo blanco. La máquina comenzará su proceso situada sobre un símbolo "1" de una serie. La máquina de Turing copiará el número de símbolos "1" que encuentre hasta el primer blanco detrás de dicho símbolo blanco. Es decir, situada sobre el 1 situado en el extremo izquierdo, doblará el número de símbolos 1, con un 0 en medio. Así, si tenemos la entrada "111" devolverá "1110111", con "1111" devolverá "111101111", y sucesivamente.

El conjunto de estados es Q y el estado inicial es q_0 . La tabla que describe la función de transición es la siguiente tabla también conocida como de transiciones δ :

Estado	Símbolo leído	Símbolo escrito	Movimiento	Estado siguiente
q_1	1	0	R	q_2
q_2	1	1	R	q_2
q_2	0	0	R	q_3
q_3	0	1	L	q_4
q_3	1	1	R	q_3
q_4	1	1	L	q_4
q_4	0	0	L	q_5
q_5	1	1	L	q_5
q_5	0	1	R	q_1

UNIDAD V • EQUIVALENCIAS Y MÁQUINA DE TURING

Definición formal de una Máquina de Turing (MT),
 $M = (Q, \Sigma, \Gamma, q_0, T, B, \delta)$

Dónde:

1. Q es un conjunto finito de estados.
2. Σ es el alfabeto de entrada.
3. Γ es el alfabeto de la cinta, que incluye: $\Sigma, \Sigma \subseteq \Gamma$
4. $q_0 \in Q$ es el estado inicial.
5. $B \in \Gamma$ es el símbolo blanco (el símbolo B no puede hacer parte de Σ) aparece en todas las casillas excepto en aquéllas que contienen los símbolos de entrada.
6. $T \subseteq Q$ conjunto de estados finales.
7. δ es la función de transición tal que: $\delta: Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, D\}$
$$\delta(q, X) = (p, Y, \{L, D\})$$

δ es una función parcial, es decir, No puede estar definida en algunos elementos del dominio.

Ejemplo:

Queremos construir una máquina de Turing que verifique si el número de 0s en una palabra es par:

$M = (Q, \Sigma, \Gamma, q_0, \delta, F)$

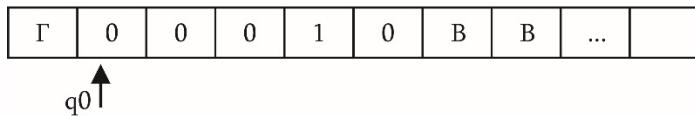
- $Q = \{q_0, q1\}$.
- $\Sigma = \{0, 1\}$.
- $\Gamma = \{0, 1, \text{' }, B\}$.
- $F = \{q0\}$.
- δ es definida como:
 - $\delta(q_0, 0) = (q_1, B, D)$
 - $\delta(q_0, 1) = (q_0, B, D)$
 - $\delta(q_1, 0) = (q_0, B, D)$
 - $\delta(q_1, 1) = (q_1, B, D)$

Supongamos que $w = 00010$: Se realiza un proceso con estos valores en una máquina de Turing que inicia y termina en la **figura 5.35**

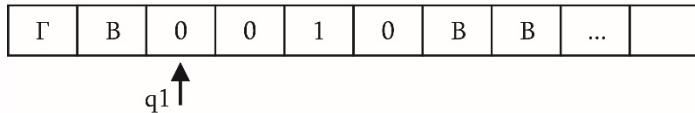
Supongamos que $w = 00010$:

UNIDAD V • EQUIVALENCIAS Y MÁQUINA DE TURING

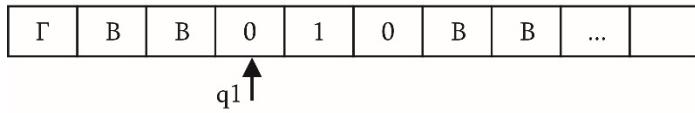
Inicio:



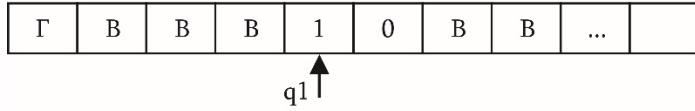
Paso 1.



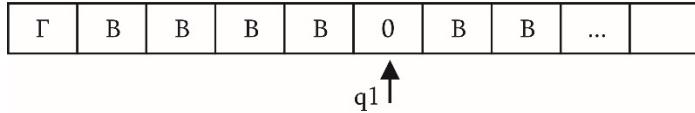
Paso 2.



Paso 3.



Paso 4.



Paso 5.

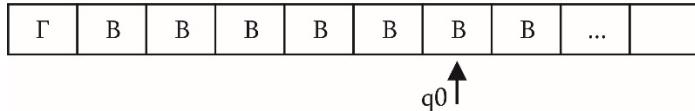


Figura 5.35

Conclusión: La máquina acepta $w = 00010$.

¿QUÉ OPERACIONES PUEDE REALIZAR UNA MÁQUINA DE TURING?

- 1) Parar la computación.
- 2) Moverse un cuadrado a la derecha.
- 3) Moverse un cuadrado a la izquierda.
- 4) Escribir el símbolo S0 en lugar de cualquier otro que este en el cuadrado examinado.
- 5) Escribir el símbolo S1 en lugar de cualquier otro que este en el cuadrado examinado.

¿Cómo se lleva a cabo el proceso de reconocimiento de una cadena con una MÁQUINA DE TURING?

- Registrando la cadena a partir de la segunda celda de su cinta
- Situando su cabeza de lectura en el extremo izquierdo

Arrancando la máquina desde el estado inicial hasta alcanzar un estado de parada una Máquina de Turing acepta una cadena con dos posibles criterios:

1. Cuando se detiene en la configuración.
2. Cuando simplemente se detiene.

¿De qué depende el movimiento de una MT?

Cada movimiento de la MT con varias cintas depende tanto de su estado, como del símbolo leído por cada una de sus cabezales de cinta.

¿Cuáles son los procesos que puede realizar una MT?

Definimos una máquina de Turing sobre el alfabeto $\{0,1\}$, donde 0 representa el símbolo blanco. La máquina comenzará su proceso situada sobre un símbolo "1" de una serie. La máquina de Turing copiará el número de símbolos "1" que encuentre hasta el primer blanco detrás de dicho símbolo blanco. Es decir, situada sobre el 1 situado en el extremo izquierdo, doblará el número de símbolos 1, con un 0 en medio. Así, si tenemos la entrada "111" devolverá "1110111", con "1111" devolverá "111101111", y sucesivamente.

UNIDAD V • EQUIVALENCIAS Y MÁQUINA DE TURING

¿Cómo decide una MT determinar qué acción debe de tomar?

La siguiente lista resume las diferencias entre un autómata finito y una máquina de Turing:

1. Una máquina de Turing puede escribir a una cinta y leer de ella
2. La cabeza de la cinta puede moverse a la izquierda y a la derecha
3. La cinta es infinita
4. Los estados especiales para aceptar y rechazar tienen efecto inmediato

Consideremos una máquina de Turing M1 para probar la membresía en el lenguaje $B = \{w\#w \mid w \in \{0, 1\}^*\}$. Es decir, queremos diseñar M1 que acepte la cadena si es miembro de B.

M1 realiza múltiples pasos con la cabeza de la cinta sobre la cadena de entrada. En cada pase se compara un carácter a cada lado del símbolo #. Para tener control sobre los símbolos previamente comparados, M1 marca cada símbolo ya examinado. Si se marcan todos los símbolos, esto significa que todo se comparó exitosamente y M1 va a un estado de aceptación. Si se descubre alguna diferencia, M1 va a un estado de rechazo. El siguiente algoritmo resume el funcionamiento de M1.

M1 = “Sobre la cadena de entrada w:

1. Barrer la entrada para asegurar que contiene sólo un símbolo #. Si no, rechazar.
2. Moverse en zig-zag sobre la cinta en las posiciones correspondientes a cada lado del símbolo # para verificar que estas posiciones contienen el mismo símbolo. Si no, rechazar. Marcar los símbolos según se van verificando para seguir la pista de la correspondencia de símbolos.
3. Cuando todos los símbolos a la izquierda del # se han marcado, verificar que no queden símbolos sin revisar. Si quedan símbolos, rechazar; de otro modo aceptar.”

CLASIFICACIÓN DE MÁQUINA TURING

Máquina de Turing Multi-cinta.

En este modelo, la máquina de Turing tiene k cintas, infinitas en ambos sentidos, y k cabezales de L/E. Sólo hay una entrada de información, en la primera cinta. Los tres pasos asociados a cada transición son ahora:

- 1) Transición de estado,
- 2) Escribir un símbolo en cada una de las celdas sobre las que están los cabezales de L/E.
- 3) El movimiento de cada cabezal es independiente y será R, L ó NADA (Z).

Máquina de Turing No Determinista.

Es una Máquina de Turing con cinta limitada a la izquierda, que se caracteriza por que a partir de un estado y un símbolo puede haber diferentes transiciones, el número de transiciones asociado a cada par ya sea estado o bien símbolo que siempre es finito.

Máquina de Turing Multidimensional.

En este modelo la cinta es un “array” o bien arreglo de k dimensiones de celdas, infinito en las dos mil direcciones posibles. Dependiendo del estado y del símbolo leído, hay una transición que difiere de las de la Máquina de Turing unidimensional en que el movimiento puede ser en cualquiera de las dos mil direcciones existentes. Se considera que la entrada está sobre un eje, y que la posición inicial del cabezal está ajustada a la izquierda de esa entrada.

Máquina de Turing con Múltiples Cabezales

Tiene k cabezales de L/E, como la multi-cinta, pero con una sola cinta. Los cabezales operan todos de forma independiente. Como en las Máquinas de Turing multi-cinta, se admiten movimientos L, R ó Z.

Máquina de Turing Offline.

Es un caso particular de las Máquinas de Turing multi-cinta: tienen una cinta especial de sólo lectura en la que el cabezal, que sólo puede moverse hacia la derecha, no puede moverse de la zona delimitada por un par de símbolos especiales.

UNIDAD V • EQUIVALENCIAS Y MÁQUINA DE TURING

Máquina de Turing con movimiento "stay" o "esperar"

La función de transición de la MT sencilla está definida por $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$, la cual puede ser modificada como $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R, S\}$. Donde S significa "permanecer" o "esperar", es decir no mover el cabezal de lectura/escritura. Por lo tanto $\delta(q, \sigma) = (p, \sigma', S)$ significa que se pasa del estado q al p, se escribe σ' en la celda actual y la cabeza se queda sobre la celda actual, tal como lo muestra la **figura 5.36**, en donde Δ significa cabezal y tenemos que en la cinta, el contenido es 1010

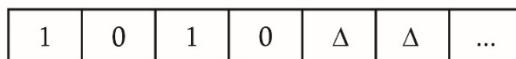


Figura 5.36 Diagrama de transiciones de las marcas que realiza el cabezal de lectura y escritura de la máquina de Turing

Máquina de Turing con cinta infinita a ambos lados.

Esta modificación se denota al igual que una MT sencilla, lo que la hace diferente es que la cinta es infinita tanto por la derecha como por la izquierda lo cual permite realizar transiciones iniciales como $\delta(q_0, x) = (q_1, y, L)$. Según se muestra en la **figura 5.37**

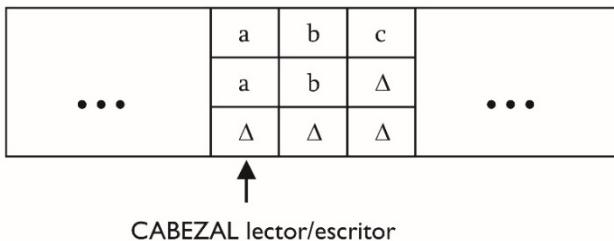


Figura 5.37
Diagrama de transiciones del cabezal de lectura y escritura de la máquina de Turing

Máquina de Turing con cinta multi-pista

Es aquella que mediante la cual cada celda de la cinta de una máquina sencilla se divide en sub-celdas. Cada sub-celda es capaz de contener símbolos de la cinta. La cinta tiene cada celda subdividida en tres sub-celdas. Se dice que esta cinta tiene múltiples pistas puesto que cada celda de esta máquina de Turing contiene múltiples caracteres, el contenido de las celdas de la cinta puede ser representado mediante n-tuplas ordenadas. Los movimientos que realice esta máquina dependerán de su estado actual y de la n-tupla que represente el contenido de la celda actual. Cabe mencionar que posee un solo cabezal al igual que una MT sencilla.

Máquinas de Turing multidimensionales

Una MT multidimensional es aquella cuya cinta puede verse como extendiéndose infinitamente en más de una dirección, el ejemplo más básico sería el de una máquina bidimensional cuya cinta se extendería infinitamente hacia arriba, abajo, derecha e izquierda según se muestra en la **figura 5.38**.

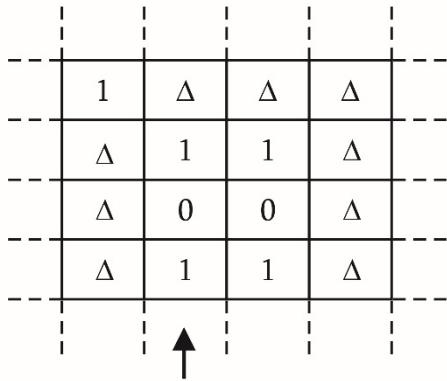


Figura 5.38 Diagrama de transiciones del cabezal de la máquina de Turing

En esta modificación de la MT también se agrega dos nuevos movimientos del cabezal {U,D} (es decir arriba y abajo). De esta forma la definición de los movimientos que realiza el cabezal será {L, R, U, D}.

Máquina Universal de Turing

Una máquina de Turing computa una determinada función parcial de carácter definido, y unívoca, definida sobre las secuencias de posibles cadenas de símbolos de su alfabeto. En este sentido se puede considerar como equivalente a un programa de ordenador, o a un algoritmo. Sin embargo, es posible realizar una codificación de la tabla que representa a una máquina de Turing, a su vez, como una secuencia de

símbolos en un determinado alfabeto; por ello, podemos construir una máquina de Turing que acepte como entrada la tabla que representa a otra máquina de Turing, y, de esta manera, simule su comportamiento.

En 1947, Turing indicó: Se puede demostrar que es posible construir una máquina especial de este tipo que pueda realizar el trabajo de todas las demás. Esta máquina especial puede ser denominada máquina universal.

Esta fue, posiblemente, la idea germinal del concepto de Sistema Operativo, un programa que puede, a su vez, ejecutar en el sentido de controlar otros programas, demostrando su existencia, y abriendo camino para su construcción real. Con esta codificación de tablas como cadenas, se abre la posibilidad de que unas máquinas de Turing se comporten como otras máquinas de Turing. Sin embargo, muchas de sus posibilidades son indecidibles, pues no admiten una solución algorítmica. Por ejemplo,

UNIDAD V • EQUIVALENCIAS Y MÁQUINA DE TURING

un problema interesante es determinar si una máquina de Turing cualquiera se parará en un tiempo finito sobre una determinada entrada; problema conocido como Problema de la parada, y que Turing demostró que era indecidible. En general, se puede demostrar que cualquier cuestión no trivial sobre el comportamiento o la salida de una máquina de Turing es un problema indecidible.

Máquina de Turing Cuántica.

En 1985, Deutsch presentó el diseño de la primera Máquina Cuántica basada en una máquina de Turing. Con este fin enunció una nueva variante la tesis de Church dando lugar al denominado "Principio de Church-Turing-Deutsch" según se muestra en la **figura 5.39**

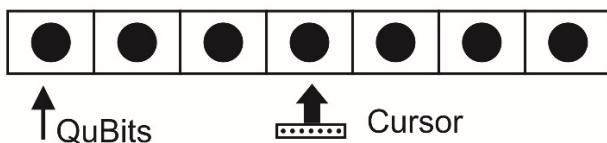


Figura 5.39 Diagrama esquemático del autómata de pila de la máquina de Turing

La estructura de una máquina de Turing cuántica es muy similar a la de una máquina de Turing clásica. Está compuesta por los tres elementos clásicos:

- Una cinta de memoria infinita en que cada elemento es un QuBit
- Un procesador finito.
- Un cursor

El procesador contiene el juego de instrucciones que se aplica sobre el elemento de la cinta señalado por el cursor. El resultado dependerá del **QuBit** de la cinta y del estado del procesador. El procesador ejecuta una instrucción por unidad de tiempo.

La cinta de memoria es similar a la de una máquina de Turing tradicional. La única diferencia es que cada elemento de la cinta de la máquina cuántica es un **QuBit**. El alfabeto de esta nueva máquina está formado por el espacio de valores del **QuBit**. El cursor es el elemento que comunica la unidad de memoria y el procesador. Su posición se representa con una variable entera.

LENGUAJE ACEPTADO POR UNA MT

Aceptan lenguajes formales que pueden ser generados por una gramática de tipo 0: recursivamente innumerable. Las máquinas de Turing son los reconocedores de lenguaje más poderosos que existen.

Lenguajes regulares.

Las gramáticas (de tipo 3) formales definen un lenguaje describiendo como se pueden generar las cadenas del lenguaje... Las gramáticas regulares (aquellos reconocidos por un autómata finito). Son las gramáticas más restrictivas. El lado derecho de una producción debe contener un símbolo Terminal y como máximo un símbolo no Terminal.

Máquinas de Turing Deterministas y no Deterministas

La entrada de una máquina de Turing viene determinada por el estado actual y el símbolo leído, un par [estado, símbolo], siendo el cambio de estado, la escritura de un nuevo símbolo y el movimiento las acciones a tomar en función de una entrada. En el caso de que para cada par estado y símbolo posible exista al sumo una posibilidad de ejecución, se dirá que es una máquina de Turing determinista, mientras que en el caso de que exista al menos un par [estado, símbolo] con más de una posible combinación de actuaciones se dirá que se trata de una máquina de Turing no determinista. La función de transición δ en el caso no determinista, queda definida como sigue:

¿Cómo sabe una máquina no determinista cuál de las varias actuaciones tomar? Hay dos formas de verlo: una es decir que la máquina es "el mejor adivino posible", esto es, que siempre elige la transición que eventualmente la llevará a un estado final de aceptación. La otra es imaginarse que la máquina se "clona", bifurcándose en varias copias, cada una de las cuales sigue una de las posibles transiciones. Mientras que una máquina determinista sigue un solo "camino computacional", una máquina no determinista tiene un "árbol computacional". Si cualquiera de las ramas del árbol finaliza en un estado de aceptación, se dice que la máquina acepta la entrada.

La capacidad de cómputo de ambas versiones es equivalente; se puede demostrar que dada una máquina de Turing no determinista existe otra máquina de Turing determinista equivalente, en el sentido de que reconoce el mismo lenguaje, y viceversa. No obstante, la velocidad de ejecución de ambos formalismos no es la misma, pues si una máquina no determinista M reconoce una cierta palabra de tamaño

UNIDAD V • EQUIVALENCIAS Y MÁQUINA DE TURING

n en un tiempo $O(t(n))$, la máquina determinista equivalente reconocerá la palabra en un tiempo $O(2t(n))$. Es decir, el no determinismo permitirá reducir la complejidad de la solución de los problemas, permitiendo resolver, por ejemplo, problemas de complejidad exponencial en un tiempo polinómico.

Lenguajes Libres de contexto.

Estas gramáticas conocidas también como gramáticas de tipo 2 o gramáticas independientes del contexto, son las que generan los lenguajes libres o independientes del contexto. Los lenguajes libres del contexto son aquellos que pueden ser reconocidos por un autómata de pila determinístico o no determinístico. Como toda gramática se definen mediante una cuadrupla ($G=N, T, S, P$), siendo N un conjunto finito de símbolos no terminales; T un conjunto de símbolos terminales; P un conjunto finito de producciones; S es el símbolo distinguido o axioma.

DEFINA MATEMÁTICAMENTE CUALES SON LOS LENGUAJES QUE ACEPTA UNA MT

La máquina de Turing es un modelo computacional publicado por la Sociedad Matemática de Londres en 1936, en el cual se estudiaba la cuestión planteada por David Hilbert sobre si las matemáticas son decidibles, es decir, si hay un método definido que pueda aplicarse a cualquier sentencia matemática y que nos diga si esa sentencia es cierta o no. Turing ideó un modelo formal de computador, la máquina de Turing, y demostró que existían problemas que una máquina no podía resolver. La máquina de Turing es un modelo matemático abstracto que formaliza el concepto de algoritmo.

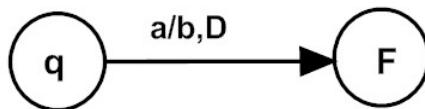
VARIACIONES DE MÁQUINAS DE TURING

- Más de una cinta de memoria
- Memoria ilimitada también por la izquierda
- Entrada en la cinta de memoria
- Memoria en varias dimensiones
- Máquina no determinista

DIAGRAMA DE TRANSICIÓN

Un diagrama de transición está formado por un conjunto de nodos que corresponden a los estados de la MT. La transición $\delta(q, a) = (p, b, D)$ se representa en la **figura 5.40**

Figura 5.40 Diagrama de estados y transiciones para el autómata de pila de $a|b, D$



Control finito estacionario.

El control finito estacionario es una transición que tiene la forma: $\delta(q, a) = (p, b, N)$ donde $a, b \in \Gamma$ y N representa un No desplazamiento.

MT con transiciones estacionarias.

Los MT con transiciones estacionarias $\delta(q, a) = (p, b, N)$, aceptan los mismos lenguajes que los MT estándares. La directiva N también se puede simular con un movimiento a la izquierda, seguido de un retorno a la derecha.

Ejemplo:

Se puede construir una MT que reconozca a^* sobre $\Sigma = \{a, b\}$.

Para $M = (Q, \Sigma, \Gamma, q_0, T, B, \delta)$ donde $Q = \{q_0, q_1\}$, $T = \{q_1\}$

$\delta(q_0, a) = (q_0, a, D)$

$\delta(q_0, B) = (q_0, a, D)$

Sea $w = aa$

$q_0 a a \leftarrow a q_0 a \leftarrow a a q_0 B \leftarrow a a B q_1 B$

Por tanto esta máquina para en el estado q_1 y reconoce la cadena que muestra la **figura 5.41**.

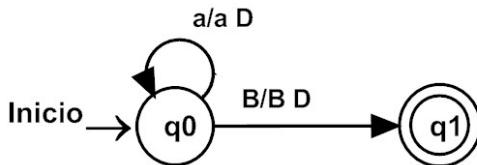


Figura 5.41. Diagrama de estados y transiciones para el autómata de pila.

MT COMO GENERADORAS DE LENGUAJE

Otra de las capacidades importantes es la de generar lenguajes, tarea en la cual los estados finales son necesarios. Concretamente una MT $M = (Q, \Sigma, \Gamma, q_0, B, \delta)$ genera un lenguaje $L \subseteq \Sigma^*$ si:

1. M comienza a operar con la cinta en blanco en el estado inicial q_0 .
2. Cada vez que M retorna al estado inicial q_0 , hay una cadena $u \in L$ escrita sobre la cinta.
3. Todas las cadenas de L son, eventualmente, generadas por M .

TÉCNICAS PARA LA CONSTRUCCIÓN DE UNA MT

Existen técnicas que facilitan la construcción de MT, pero no afectan la potencia computacional del modelo ya que siempre se puede simular la solución obtenida mediante el modelo estándar.

Almacenamiento en el control finito.

Se puede utilizar el estado de control para almacenar una cantidad finita de información.

$$\delta([q_i, \Gamma], \sigma) = ([q_b, \Gamma], \beta, \{l, D, N\})$$

Cada estado se representa como un par ordenado $[q_i, \Gamma]$ donde el primer elemento es el estado real y el segundo la información que se pretende almacenar, además $\sigma, \beta \in \Gamma$

Ejemplo:

Construir una MT que reconozca: $L = 01^* + 10^*$

Para la máquina $M = (Q, \Sigma, \Gamma, q_0, T, B, \delta)$:

$$Q = \{q_0, q_1\} \times \{0, 1, B\}$$

Estado inicial $[q_0, B]$

Estado final $[q_1, B]$

La función de transición δ está dada por:

$$\delta([q_0, B], 0) = ([q_1, 0], 0, D)$$

$$\delta([q_1, 0], 1) = ([q_1, 0], 1, D)$$

$$\delta([q_1, 0], B) = ([q_1, B], B, D)$$

$$\delta([q_0, B], 1) = ([q_1, 1], 1, D)$$

$$\delta([q_1, 1], 0) = ([q_1, 1], 0, D)$$

$$\delta([q_1, 1], B) = ([q_1, B], B, D)$$

El autómata de la máquina de Turing se muestra en la **figura 5.42**

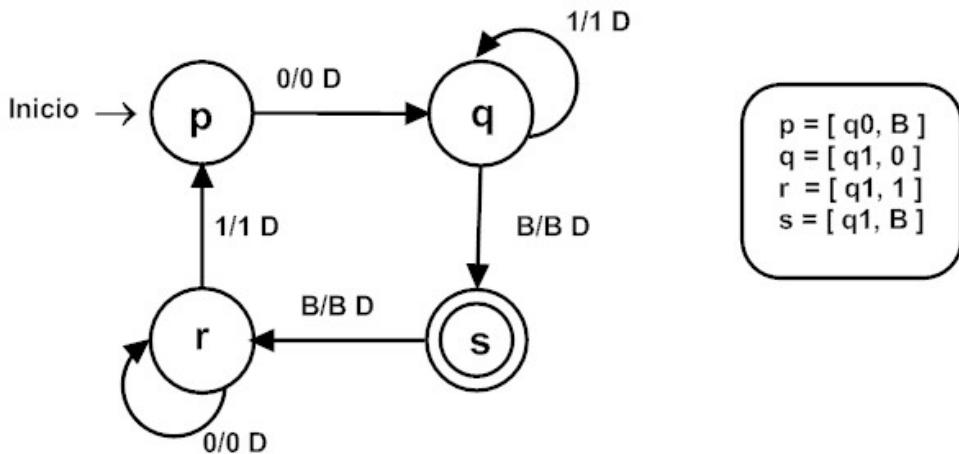


Figura 5.42 Diagrama de estados y transiciones y esquema de PILA para el autómata de Turing

Uso de subrutinas

Es la misma idea cuando se trabaja en un lenguaje de alto nivel, aprovechar las ventajas del diseño modular para facilitar el diseño de la MT.

En una tabla de transición se resuelve el problema de llamados:

- Habrá estados de “llamados a subrutina” q_i caracterizados por que suponen la transición al estado inicial de una “subrutina”.
- El estado final de la subrutina será realmente un estado de salida que permite transitar hacia un estado de return en la MT “principal” según se muestra en la figura 5.44.

Subrutina

Q	∞	β	...
q_i	$(q_{i...})$...
...			...
q_1		$(q_{1...})$...

MT principal

Q	\square	∞	β	...
q_i	$(q_{i...})$...
q_2				...
q_3	$(q_{3...})$...
q_4		$(q_{2...})$	$(q_{1...})$...

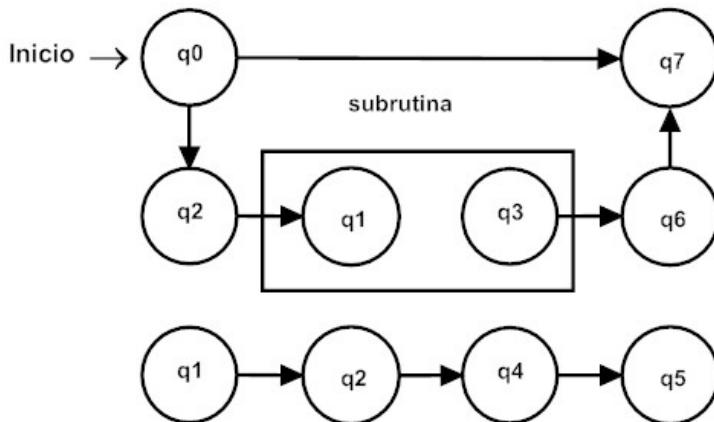


Figura 5.44 Diagrama esquemático de las transiciones y rutinas de la máquina de Turing

- ¿Por qué las máquinas de Turing son una buena formalización para un algoritmo?
- Porque cada programa de una máquina de Turing puede ser implementado.
- Porque todos los algoritmos conocidos han podido ser implementados en máquinas de Turing.
- Porque todos los otros intentos por formalizar este concepto fueron reducidos a las máquinas de Turing.
- Los mejores intentos resultaron ser equivalentes a las máquinas de Turing.
- Todos los intentos “razonables” fueron reducidos eficientemente.

FUNCIÓN COMPUTADA POR UNA MT

Las MT's pueden transformar entradas en salidas:

- La entrada son todos los símbolos no blancos en la cinta en el momento inicial
- El contenido de la cinta (los símbolos no blancos) al final de la computación (cuando la máquina se para en un estado final) se considera como salida.

En otras palabras, se puede considerar una MT como la implementación de una función $f: y=f(x)$ si para la configuración inicial q_{0x} la máquina M para en una configuración q_f , donde q_f es un estado final de M: $q_{0x} \vdash_M^* q_f$.

SIMULACIÓN DE UNA MT EN JAVA

Ejemplo: Simule una MT que muestre el complemento a 1 de una cadena.

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class Complemento1
{
    //declaracion de las variables
    public static String cadena ,cadena2, aux;
    public static BufferedReader reader;
    public static int longitud;
    public String complemento="";
    public String arreglo[];
    public int apuntador=1;

    //el constructor del objeto muestra la cadena ingresada
    public Complemento1()
    {
        System.out.println("Cadena: "+cadena);
        decodificar(cadena);
    }

    //metodo decodificar se encarga de crear el arreglo que contendra la
    cadena
    public void decodificar(String dec)
    {
        cadena2="#"+cadena+"#";
        longitud= cadena2.length();
        arreglo= new String[longitud];
        for(int i=0; i<longitud; i++)
        {
            arreglo[i]= ""+cadena2.charAt(i);
        }
        e0();
    }

    //el metodo e0 representa el estado 0 o estado inicial
    public void e0()
    {
```

UNIDAD V • EQUIVALENCIAS Y MÁQUINA DE TURING

```
aux= arreglo[apuntador];
if(aux.equals("0"))
{
    arreglo[apuntador]= "1";
    MoverDerecha();
    e1();
}
else if(aux.equals("1"))
{
    arreglo[apuntador]= "0";
    MoverDerecha();
    e1();
}
else if(aux.equals("#"))
{
    arreglo[apuntador]= "#";
    MoverIzquierda();
    System.out.println("Cadena vacia");
}
else
{
    rechazar(aux);
}

//el metodo e1 representa el estado 1
public void e1()
{
    aux= arreglo[apuntador];
    if(aux.equals("0"))
    {
        arreglo[apuntador]= "1";
        MoverDerecha();
        e1();
    }
    else if(aux.equals("1"))
    {
        arreglo[apuntador]= "0";
        MoverDerecha();
        e1();
    }
    else if(aux.equals("#"))
```

UNIDAD V • EQUIVALENCIAS Y MÁQUINA DE TURING

```
{  
    arreglo[apuntador]= "#";  
    MoverIzquierda();  
    e2();  
}  
else  
{  
    rechazar(aux);  
}  
}  
  
//el metodo e2 representa el estado 2  
public void e2()  
{  
    aux= arreglo[apuntador];  
    if(aux.equals("0"))  
    {  
        arreglo[apuntador]= "0";  
        MoverIzquierda();  
        e2();  
    }  
    else if(aux.equals("1"))  
    {  
        arreglo[apuntador]= "1";  
        MoverIzquierda();  
        e2();  
    }  
    else if(aux.equals("#"))  
    {  
        arreglo[apuntador]= "#";  
        MoverDerecha();  
        e3();  
    }  
    else  
{  
    rechazar(aux);  
}  
}  
  
//el metodo e3 representa el estado 3 o de aceptacion  
public void e3()  
{
```

UNIDAD V • EQUIVALENCIAS Y MÁQUINA DE TURING

```
for(int i=1; i<longitud-1; i++)
{
    complemeto= complemeto+arreglo[i];
}
System.out.println("El complemeto a 1 de: "+cadena+" es:
"+complemeto);
}

//mueve el puntero de la MT a la derecha
public void MoverDerecha()
{
    apuntador++;
}
//mueve el puntero de la MT a la izquierda
public void MoverIzquierda()
{
    apuntador= apuntador-1;
}

//si entra en este estado significa que encontro un caracter no valido
public void rechazar(String noval)
{
    System.out.println("Caracter no valido"+noval);
}

//pide un numero binario y lo guarda en la cadena
public static void main(String[] args)
{
    System.out.println("Ingrese numero binario:");
    try
    {
        reader= new BufferedReader(new InputStreamReader(System.in));
        cadena= reader.readLine();
    }catch(IOException e)
    {
        e.getMessage();
    }

    Complemento1 obj= new Complemento1();
}
```

GLOSARIO

Abstracción. Proceso de abstraer.

Abstraer. Separar por medio de una operación intelectual un rasgo o una cualidad de algo para analizarlos aisladamente o considerarlos en su pura esencia.

Álgebra de Boole, se le atribuye al matemático George Boole en el siglo XIX, el cual es creador de la lógica basada en dos símbolos; el cero y el uno desde 1849, simplificando el álgebra y creando las bases con las que se construyen las modernas computadoras actuales.

Análisis. es el estudio realizado para separar las distintas partes de un todo.

Árboles de análisis. El árbol desde el punto de vista de una estructura de datos, el árbol tiene una raíz que es el inicio de los elementos que a continuación se describen en los nodos hijos, padres, hojas, etc. El recorrido sobre sus elementos es lo que se denomina el análisis o derivación.

Auto-compilador. Es aquél que está escrito en el mismo lenguaje que se pretende compilar.

Autómata. Es un servomecanismo o mecanismo capaz de auto-gobernarse, el cual primero se diseñó con la posibilidad de que trabaje y se comporte de forma independiente a la manipulación del ser humano.

Autómatas finitos determinísticos AFD. Es un modelo matemático o diagrama dirigido (dígrafo) que permite esquematizar los elementos de un sistema el cual consta de un inicio y un FIN (de aquí el nombre de finito).

Autómatas finitos no determinísticos AFN. Es un modelo matemático o representación simbólica que muestra más flexibilidad en su creación al permitir que se utilicen símbolos los cuales pueden tener una transición o varias y no hay un equilibrio en cuanto a los símbolos empleados y el diseño de sus transiciones.

Bootstrapping. Es una técnica usada para el desarrollo de compiladores de lenguajes de alto nivel. Para describir el proceso de auto-compilación se emplea la notación en T que representa gráficamente los tres lenguajes implicados en el proceso de compilación (Fuente, generador y generado).

Código. Es un conjunto de reglas que proporcionan una correspondencia biunívoca que permite representar datos, programas y otras informaciones con vistas a facilitar su tratamiento automático o su transmisión.

Compilador. Es un traductor que convierte un texto escrito en un lenguaje fuente de alto nivel en un programa que puede ser objeto en código máquina, cuando se trata de un compilador completo, pero también existen pseudo-compiladores que en vez de generar código objeto, generan un código intermedio conocido como bytecode y también existen compiladores que incluyen un optimizador de código.

Compilador cruzado. Es el que genera un código objeto ejecutable en un ordenador distinto de aquél en el que se realiza la compilación.

Compilación-Montaje-Ejecución. En las aplicaciones grandes es conveniente fragmentar el programa a realizar en módulos que se compilan por separado, y una vez que estos estén compilados unirlos mediante un programa denominado montador para formar el módulo ejecutable. El montador se encarga, a su vez, de incluir las librerías donde están guardadas las funciones predefinidas de uso común.

Compilación en una o varias pasadas. Se llama pasada a cada lectura que hace el compilador del texto fuente.

Compilación incremental. Este compilador actúa de la siguiente manera. Compila un programa fuente. Caso de detectar errores al volver a compilar el programa corregido sólo compila las modificaciones que se han hecho respecto al primero.

Conversores Fuente-Fuente. Traducen un lenguaje fuente de alto nivel a otro (por ejemplo, PASCAL -> C).

De-compilador. Es el que traduce código máquina a lenguaje de alto nivel. Los de compiladores más usuales son los desensambladores, que traducen un programa en lenguaje máquina a otro en ensamblador.

Ensamblandores. Son compiladores cuyo lenguaje de entrada, llamado ensamblador, permite la traducción de cada sentencia fuente a una instrucción en código máquina. En este caso, se trata de un lenguaje pero también puede ser un programa o un proceso.

Intérprete. Es un traductor que realiza la operación de compilación paso a paso. Para cada sentencia que compone el texto de entrada, se realiza una traducción, ejecuta dicha sentencia y vuelve a iniciar el proceso con la sentencia siguiente.

Lenguaje. Es el medio de comunicación entre los seres humanos a través de signos orales y escritos que poseen un significado. En un sentido más amplio, es cualquier procedimiento que sirve para comunicar ideas o sentimientos.

Lenguajes artificiales. Este grupo ha sido creado por el mismo ser humano, con el objetivo de poder establecer una comunicación directa con diversos objetos, de entre los cuales encontramos a las computadoras u ordenadores.

Los lenguajes de alto nivel. Lenguaje simbólico en el que una instrucción de programa fuente da lugar a varias instrucciones de lenguaje máquina, debido a que es compilado o interpretado.

El lenguaje máquina. lenguaje específico de las computadoras, en el que las instrucciones se expresan en código binario directamente asimilable por la máquina.

Lingüística. Ciencia del lenguaje, estudio comparativo de las lenguas.

Matemáticas discretas. Es el área de las matemáticas en la cual se emplean dos símbolos exclusivamente el cero y el uno, de aquí surge el álgebra de Boole y la tecnología conocida como digital basada en compuertas lógicas donde se le agregan elementos que describen el funcionamiento de elementos numerables, finitos o infinitos pero de fácil modelado.

Meta-compilador. Es un traductor que tiene como entrada la definición de un lenguaje y como salida el compilador para dicho lenguaje.

Modelado. Acción de modelar o crear un modelo.

Modelar. Se refiere al proceso creativo de abstraer un problema y encontrar solución al mismo mediante elementos también abstractos como las matemáticas, los lenguajes de programación o en general el software y hardware.

Pre-compiladores. Son compiladores que toman un código X y generan otro código en función del primero X y se produce Y.

Preprocesadores. También conocidos como pre-compiladores, procesan un texto fuente modificándolo en cierta forma previamente a la compilación.

Programa. Es un conjunto de instrucciones escritas en un lenguaje de programación y también se llama programa fuente.

Pseudo-compiladores. Son compiladores que generan código intermedio, conocido como código de “bytes” o “bytecode”.

Rutinas de análisis de instrucciones. El conjunto de instrucciones del entorno de un sistema operativo constituye un lenguaje que debe ser analizado previamente para realizar las acciones oportunas. Igualmente, ciertos programas como editores de texto, sistemas de diseño asistido, etc., utilizan instrucciones complejas que deben interpretarse adecuadamente.

Síntesis. Es la unión de esas partes (o la composición) del todo.

Sistema. Es un conjunto de elementos independientes pero interrelacionados (recursos) que interactúan, directa o indirectamente, que forman un todo indivisible y organizado que afecta y se ve afectado por su medio ambiente y realiza actividades, para el cumplimiento de su propósito u objetivo, valiéndose de mecanismos de retroalimentación que le permiten efectuar su auto control para continuar su existencia.

Teoría de conjuntos. La teoría de conjuntos es una rama de las matemáticas que estudia las propiedades de los conjuntos o bien colecciones abstractas de objetos, consideradas como objetos en sí mismas.

Traductor. Es una máquina teórica que tiene como entrada un texto escrito en un lenguaje L1 y como salida un texto escrito en un lenguaje L2. Habitualmente se denomina a L1 lenguaje fuente y a L2 lenguaje objeto.

TEORÍA DE LA COMPUTACIÓN



Cascada de Cusarare de la Sierra Tarahumara en Chihuahua, México.

Sara Méndez García tutora desde hace 10 años en la Unidad Profesional Interdisciplinaria de Ingeniería y Ciencias Sociales y Administrativas (UPIICSA) del Instituto Politécnico Nacional (IPN).

De profesión egresada de la UPIICSA de la Licenciatura en Ciencias de la Informática. Generación 1979-1983 (se inició con el primer semestre completo en la licenciatura en Administración Industrial para lograr un cambio a la anhelada informática).

Actualmente, candidato a maestro en ciencias con especialidad en ciencias de la informática en la Sección de Estudios de Posgrado e Investigación (SEPI) de la UPIICSA.

Profesora en la UPIICSA desde 1991

ISBN 978-607-97197-1-5

A standard linear barcode representing the ISBN number 978-607-97197-1-5.

9 786079 719715