UNIVERSITAT DE BARCELONA

Facultat de Matemàtiques
i Informàtica

GRAU DE MATEMÀTIQUES

Treball final de grau

Teoría de Autómatas

Autor: Alex Milesi Vidal

Director: Dr. Juan Carlos Martínez
Realitzat a: Departament de
Matemàtiques i Informàtica

Barcelona, 20 de junio de 2019

Abstract

This paper aims to approach automata theory which is the study of abstract computing devices.

In the first part we are going to study regular languages and finite automatas, the devices that allow to recognize regular languages.

In the last part we'll see context-free languages and the machines to recognize them, the pushdown automatas.

Resumen

Este trabajo es una introducción a la teoría de autómatas, que es el estudio de modelos de programación abstractos.

En la primera parte estudiaremos los lenguajes regulares y los autómatas finitos, que nos sirven para reconocer los lenguajes regulares.

En la última parte veremos los lenguajes incontextuales y las máquinas que nos permiten reconocerlos, los autómatas con pila.

2010 Mathematics Subject Classification. 03D05, 68Q70, 94A45

Agradecimientos

Agradezco ante todo a mi tutor, el Dr. Juan Carlos Martínez, por su ayuda a la hora de guiarme en este proyecto.

Agradezco también a mis padres, a mi familia, a mis amigos y a todas aquellas personas que han estado siempre conmigo en todos los momentos.

Índice

1. Introducción **1**

2. Preliminares **2**

2.1. Lenguajes formales 2

2.2. Representación de lenguajes 3

2.2.1. Expresiones regulares 4

2.2.2. Homomorfismos 5

3. Autómatas finitos **7**

3.1. Autómatas finitos deterministas 7

3.2. Autómatas finitos indeterministas 9

3.3. Operaciones para autómatas finitos 12

3.4. Autómatas finitos y expresiones regulares 13

3.5. Lenguajes no regulares 15

3.6. Aplicaciones 17

3.6.1. Programa para reconocer un patrón en un texto 17

3.6.2. Analizador léxico 18

4. Lenguajes incontextuales **20**

4.1. Autómatas con pila 20

4.2. Gramáticas incontextuales 22

4.3. Árboles de derivación 24

4.4. Autómatas con pila y gramáticas incontextuales 26

4.5. Lenguajes no incontextuales 30

4.6. Aplicaciones al análisis sintáctico 33

5. Conclusiones **41**

1. Introducción

Ya no nos sorprende que los ordenadores respondan a lo que escribimos en un teclado, o a los movimientos de nuestros dedos sobre una pantalla o incluso a breves órdenes vocales. Qué hay detrás? Este escrito pretende explicar, de forma concisa, cómo a mediados del siglo XX se formalizaron los lenguajes para comunicarnos con las máquinas que íbamos inventando para todo tipo de propósitos. Con anterioridad a esta fecha la humanidad ya había desarrollado máquinas, pero se limitaban a contar mediante mecanismos sencillos.

La teoría de autómatas es el estudio de modelos computacionales abstractos o "máquinas" para reconocer lenguajes y hoy en día tienen aplicaciones para el diseño de circuitos, el desarrollo de algoritmos de búsqueda, la verificación de sistemas informáticos y el diseño de compiladores.

En el contexto de la comunicación con ordenadores (compilación) debemos ser capaces de reconocer los distintos niveles gramaticales: el morfológico (las palabras construidas con letras) y el sintáctico (las sentencias construidas con palabras y símbolos). Para eso tenemos los autómatas finitos y los autómatas con pila, de los cuales hablaremos en los dos temas centrales de este trabajo. El uso de ellos nos permite crear lenguajes de programación.

Demostraremos (tema 3) que existe una equivalencia entre lenguajes regulares y autómatas finitos, de manera que para cada uno de los primeros existe uno entre los segundos que reconoce sus expresiones y viceversa. Probaremos incluso que esta equivalencia se cumple tanto con los autómatas deterministas (programables) como con los indeterministas (intuitivos, no programables). Veremos también (tema 4) la equivalencia entre gramáticas incontextuales y autómatas con pila. La pila añade una pequeña función de memoria al autómata que permite comprobar la estructura sintáctica del mensaje.

2. Preliminares

2.1. Lenguajes formales

En este primer apartado nos limitaremos a definir los conceptos más básicos, con el objetivo de llegar a conocer los lenguajes formales, ya que el propósito de los autómatas, que trabajaremos en los siguientes temas, es precisamente reconocer lenguajes.
Un **alfabeto** es un conjunto finito y no vacío de símbolos.
Una **palabra** sobre un alfabeto Σ es una secuencia, posiblemente vacía, de símbolos de Σ.
La **longitud** de una palabra x , que denotaremos como $|x|$, es el número de símbolos que la componen, contando repeticiones.
Si x es una palabra sobre un alfabeto Σ y $a \in \Sigma$ denotaremos por $n_a(x)$ el número de apariciones de a en x.
Escribiremos λ para denotar la palabra vacía, es decir, la palabra que no tiene ningún símbolo.

Sean x, y dos palabras sobre un alfabeto Σ. La **concatenación** de x e y es su yuxtaposición, que podemos escribir como $x \cdot y$, o más brevemente, xy
A partir de esta definición podemos definir la potencia de una palabra x como: $x^0 = \lambda$, $x^{i+1} = x^i x$ para un número natural i

Si Σ es un alfabeto, denotamos por Σ^* el conjunto de palabras de Σ.
Sobre Σ^* podemos definir el **orden lexicográfico** de la siguiente manera.
Supongamos que tenemos un alfabeto $\Sigma = \{a_1, ..., a_n\}$ donde $a_1 < a_2 < ... < a_n$.
Entonces el orden de Σ^* es:

$$\lambda, a_1, ..., a_n, a_1a_1, a_1a_2, ..., a_na_n, a_1a_1a_1, ...$$

Es decir, priorizamos la longitud de las palabras y luego, palabras de la misma longitud las ordenamos según el orden que determina el alfabeto. Notemos que si cogemos como alfabeto los dígitos del 0 al 9, el orden es el mismo que el de los números naturales.

Un **lenguaje** L es un conjunto de palabras sobre un alfabeto.
Si nuestro lenguaje es finito podemos representarlo simplemente listando todas sus palabras. Sin embargo, nos será más útil usar lenguajes infinitos, los cuales definiremos como las palabras de un alfabeto que cumplen una cierta propiedad.

Operaciones básicas para lenguajes
Sean L, L_1, L_2 lenguajes sobre un alfabeto Σ. Las cuatro primeras operaciones son equivalentes a las operaciones naturales entre conjuntos.
1. La **unión**. $L_1 \cup L_2$
2. La **intersección**. $L_1 \cap L_2$
3. La **diferencia**. $L_1 \backslash L_2$

4. La **complementación**. En este caso considerando el espacio total Σ^*, seria $\overline{L} = \{x \in \Sigma^* : x \notin L\}$
5. La **concatenación**. La definimos como $L_1L_2 = \{xy : x \in L_1, y \in L_2\}$. También podemos definir $L^n = \{x_1...x_n : x_i \in L \text{ para todo } i\}$
6. La **clausura**. Definida como $L^* = \bigcup_{n\geq 0} L^n = \{x_1x_2...x_n : n \geq 0; x_1, x_2, ..., x_n \in L\}$
7. La **clausura positiva**. $L^+ = \bigcup_{n\geq 1} L^n = \{x_1x_2...x_n : n \geq 1; x_1, x_2, ..., x_n \in L\}$

Ejemplo: Tomamos $\Sigma = \{0, 1\}, L_1 = \{x \in \Sigma^* : n_0(x) \geq n_1(x)\}, L_2 = \{x \in \Sigma^* : n_0(x) \leq n_1(x)\}$ y $L_3 = \{x \in \Sigma^* : n_0(x) = n_1(x)\}$.

Entonces tenemos $L_1L_1 = L_1, L_2L_2 = L_2, L_3L_3 = L_3, L_1L_2 = L_2L_1 = \Sigma^*, L_1L_3 = L_3L_1 = L_1, L_2L_3 = L_3L_2 = L_2$.

Veremos el caso $L_1L_2 = \Sigma^*$. Para demostrarlo, es evidente que $L_1L_2 \subseteq \Sigma^*$ y para ver que $\Sigma^* \subseteq L_1L_2$ tomamos un $x \in \Sigma^*$. Tenemos dos casos:
Caso 1: $n_0(x) \geq n_1(x)$. Tenemos $x = x\lambda \in L_1L_2$.
Caso 2: $n_0(x) \leq n_1(x)$. Tenemos $x = \lambda x \in L_1L_2$.

Ejemplo: Tomamos $\Sigma = \{0, 1\}, L_1 = \{x \in \Sigma^* : n_0(x) \text{ es impar}\}$ y $L_2 = \{x \in \Sigma^* : n_0(x) \text{ es par}\}$. Entonces $L_1^* = \{x \in \Sigma^* : n_0(x) > 0\} \cup \{\lambda\}$ y $L_2^* = L_2$.
Para demostrar la primera condición es evidente la inclusión

$$L_1^* = \bigcup_{n\geq 0} L_1^n = \{x_1x_2...x_n : n \geq 0; x_1, x_2, ..., x_n \in L_1\} \subseteq \{x \in \Sigma^* : n_0(x) > 0\} \cup \{\lambda\}$$

Para la otra inclusión, todo $x \in \Sigma^*$ tal que $n_0(x) > 0$ puede escribirse como $x = x_1x_2...x_n$ donde x_1 es la subpalabra de x desde el principio hasta el primer cero (incluido), x_2 la subpalabra de x del primer cero (no incluido) al segundo cero (incluido) y así sucesivamente, de lo que se desprende que

$$n_0(x_i) = 1, \forall i \Rightarrow x_i \in L, \forall i \Rightarrow x = x_1x_2...x_n \in L^n \subseteq L^*$$

Para demostrar que $L_2^* = L_2$ notemos que la concatenación de palabras, cada una con un número par de ceros, seguirá teniendo un número par de ceros y por tanto el lenguaje de la clausura es el mismo que el lenguaje original.

2.2. Representación de lenguajes

En este apartado veremos dos formas de representar lenguajes, las expresiones regulares y los homomorfismos.

2.2.1. Expresiones regulares

En muchas ocasiones, podemos representar un lenguaje utilizando únicamente los símbolos de un alfabeto y las operaciones básicas para lenguajes. Para ello tenemos las expresiones regulares, que nos permiten definir lenguajes, en general infinitos, mediante una sola palabra.

Definición: Una **expresión regular** sobre un alfabeto Σ es una palabra sobre el alfabeto $\Sigma \cup \{(,), \emptyset, \lambda, \cup, \cdot, *\}$ generada por las siguientes reglas:
1. $\emptyset$ y λ son expresiones regulares.
2. Para todo $a \in \Sigma$, a es una expresión regular.
3. Si α, β son expresiones regulares, también lo son $(\alpha \cup \beta)$ y $\alpha\beta$.
4. Si α es expresión regular, también lo es α^*.

Ahora podemos definir el lenguaje $L(\alpha)$ que describe una expresión regular α por:
1. $L(\emptyset) = \emptyset$, $L(\lambda) = \{\lambda\}$, $L(a) = \{a\}$ $\forall a \in \Sigma$.
2. $L(\alpha \cup \beta) = L(\alpha) \cup L(\beta)$, $L(\alpha\beta) = L(\alpha)L(\beta)$, $L(\alpha^*) = (L(\alpha))^*$.

A partir de esta definición decimos que un lenguaje L es **regular** si existe una expresión regular α tal que $L = L(\alpha)$

Ejemplo: Vamos a comprobar que los números enteros (tipo entero) y los números decimales (tipo float) se pueden representar mediante una expresión regular. El alfabeto que utilizaremos para los enteros es el compuesto por las diez cifras y los signos positivo y negativo

$$\Sigma_N = \{0, 1, 2, 3, \ldots 8, 9, +, -\}$$

Para los números decimales necesitaremos los doce símbolos anteriores y también el punto decimal

$$\Sigma_Q = \{0, 1, 2, 3, \ldots 8, 9, +, -, .\}$$

La expresión de un número entero es la siguiente

$$(\lambda \cup + \cup -) \cdot (1 \cup 2 \cup 3 \cup \cdots \cup 8 \cup 9) \cdot (0 \cup 1 \cup 2 \cup 3 \cup \cdots \cup 8 \cup 9)^*$$

Un número decimal debe contener el punto decimal, para distinguirlo de un entero. Delante del punto puede haber un entero o nada; detrás cualquier secuencia de cifras o incluso nada. No es válida la secuencia constituida por el punto y nada más. Formalmente tendremos

$$(\lambda \cup + \cup -)[(1 \cup 2 \cup \cdots \cup 9)(0 \cup 1 \cup 2 \cup \cdots \cup 9)^*.(0 \cup 1 \cup 2 \cup \cdots \cup 9)^* \cup$$
$$\cup .(0 \cup 1 \cup 2 \cup \cdots \cup 9)(0 \cup 1 \cup 2 \cup \cdots \cup 9)^*]$$

2.2.2. Homomorfismos

Un **homomorfismo** es una función $h : \Sigma_1^* \to \Sigma_2^*$, donde Σ_1, Σ_2 son alfabetos, tal que:
$h(xy) = h(x)h(y)\ \forall x, y \in \Sigma_1^*$
$h(\lambda) = \lambda$

Por tanto si h es un homomorfismo sobre un alfabeto Σ y cogemos una palabra $x = a_1a_2...a_n$ en Σ entonces $h(x) = h(a_1)h(a_2)...h(a_n)$.

Teorema 2.1. *Si L es un lenguaje regular sobre un alfabeto Σ_1 y $h : \Sigma_1^* \to \Sigma_2^*$ un homomorfismo, entonces $h(L)$ también es regular.*

Demostración. Suponemos que $L = L(\alpha)$ para alguna expresión regular α. En términos generales, para una expresión regular α , $h(\alpha)$ será la expresión resultante de sustiuir todos sus símbolos $a_i \in \Sigma$ por los correspondientes $h(a_i)$. Por construcción $h(\alpha)$ es también una expresión regular. Se puede entonces comprobar fácilmente que $h(\alpha) = h(L)$ □

La demostración del siguiente teorema la realizaremos en el Tema 3.

Teorema 2.2. *Si h es un homomorfismo entre un alfabeto Σ_1 y un alfabeto Σ_2, y L es regular sobre Σ_2, entonces $h^{-1}(L)$ también es regular.*

Ahora daremos un ejemplo de un lenguaje que podemos crear a partir de homomorfismos.

Ejemplo Sea Σ_2 un alfabeto cualquiera y un lenguaje $L \subseteq \Sigma_2^*$. Estamos interesados en la transformación que se queda solamente con las palabras que resultan de borrar los símbolos que están en las posiciones pares de las palabras de longitud par de L. Es decir que, para cada lenguaje $L \subseteq \Sigma_2^*$ queremos obtener el lenguaje L' tal que

$$L' = \{a_1a_3 \dots a_{2n-1} | a_1a_2a_3 \dots a_{2n-1}a_{2n} \in L\}$$

Para concretar las ideas, supongamos que Σ_2 es un alfabeto de tres símbolos, $\Sigma_2 = \{a, b, c\}$. Construimos un alfabeto de nueve símbolos (en general el número de símbolos de Σ_1 será el cuadrado del número de símbolos de Σ_2), que representaremos como $\Sigma_1 = \{1, 2 \dots 9\}$. Definimos ahora los dos homomorfismos mediante

la tabla siguiente.

Σ_1	$h_1(\Sigma_1)$	$h_2(\Sigma_1)$
1	aa	a
2	ab	a
3	ac	a
4	ba	b
5	bb	b
6	bc	b
7	ca	c
8	cb	c
9	cc	c

Es facil constatar, por aplicación directa de las transformaciones indicadas, que

$$L' = h_2(h_1^{-1}(L))$$

Por tanto L' es regular.

3. Autómatas finitos

El objetivo de ahora en adelante será el de definir modelos computacionales para reconocer lenguajes. En este capítulo introduciremos los autómatas finitos, los cuales nos van a ser muy útiles para el diseño de analizadores léxicos, cosa que veremos al final del tema. Un autómata finito tiene asociada una cinta de lectura, la cual está dividida en celdas. La información de la cinta se encuentra entonces almacenada en celdas, y el autómata accede a dicha información mediante un puntero, inicialmente apuntando a la celda situada más a la izquierda de la cinta, que puede leer en un instante dado el contenido de la celda a la que apunta. El autómata tiene además asociada una "unidad de control ", que en un paso de cómputo se encuentra en un cierto estado. Al realizar el siguiente paso de cómputo el estado puede variar.

3.1. Autómatas finitos deterministas

Empezaremos por introducir el modelo de autómata finito determinista. Como bién dice su nombre este autómata es determinista, es decir, que cuando leemos un símbolo estando en un estado en concreto, el paso de cómputo a realizar está únivocamente determinado. Veamos la definición formal.

Definición: Un **autómata finito determinista** es una estructura $M = (K, \Sigma, \delta, q_0, F)$ donde:

K es un conjunto finito de estados,
Σ es el alfabeto de entrada,
q_0 es el estado inicial,
$F \subseteq K$ es el conjunto de estados aceptadores, y
δ es una función de $K \times \Sigma$ a K, a la que llamamos función de transición.

Llamamos **símbolo actual** al símbolo accesible a través del puntero en un momento concreto, es decir el símbolo que estamos leyendo.
La idea que comentábamos anteriormente consiste en aplicar la función δ para realizar los pasos de cómputo. Inicialmente el autómata se encuentra en el estado q_0 con una entrada $x \in \Sigma^*$ escrita en la cinta. El argumento de δ es el par (q,a) donde q es el estado actual y a el símbolo actual. El autómata se sitúa en el estado $p = \delta(q, a)$.

Definiciones: Si $M = (K, \Sigma, \delta, q_0, F)$ es un autómata determinista definimos los siguientes conceptos:
Una **configuración** de M es una palabra $qx \in K\Sigma^*$. Si en un paso de cómputo estamos en la configuración px significa que nos encontramos en el estado p y x es la subpalabra que nos falta por leer.
Si px, qy son configuraciones, decimos que px **produce** qy **en un paso de cómputo** si $x = ay$ para algún $a \in \Sigma$ y $\delta(p, a) = q$. Lo denotaremos como $px \vdash_M qy$.
Sean px, qy configuraciones. Decimos que px **produce** qy si existen configuraciones $c_0, ..., c_n$ tales que $px = c_0 \vdash_M c_1 ... \vdash_M c_n = qy$. Lo escribimos como $px \vdash_M^* qy$.

Una palabra $x \in \Sigma^*$ es **reconocida o aceptada** por M si existe $q \in F$ tal que $q_0 x \vdash_M^* q$.
Definimos el **lenguaje asociado a** M como $L(M) = \{x \in \Sigma^* : x \text{ es reconocida por } M\}$.

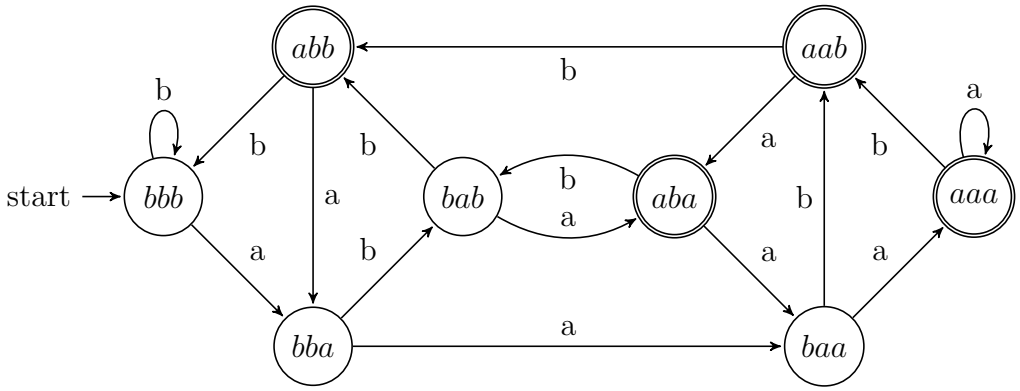
Ahora antes de mostrar un ejemplo, vamos a ver como se puede representar un autómata determinista mediante un grafo:
(1) Los nodos del grafo corresponden a los estados del autómata.
(2) Si $\delta(q, a) = p$ dibujamos un arco en el grafo de q a p i lo etiquetamos con a.
(3) Marcamos con una flecha el estado inicial y con un doble círculo los estados aceptadores.

Ejemplo: Definimos un autómata determinista $M = (K, \{a, b\}, \delta, q_0, F)$ tal que $L(M) = \{x \in \{a, b\}^* : \text{hay una } a \text{ en la antepenúltima posición de } x\}$.

Para construir el autómata identificaremos sus estados con los tres últimos símbolos leídos. Representaremos los estados por xyz donde x será el antepenúltimo carácter leído, y el penúltimo, y z el último. Entonces nuestra función de transición será:

$$\delta(xyz, a) = yza,\ \delta(xyz, b) = yzb$$

Finalmente solo nos queda identificar los estados en los cuales aún no se han leído tres símbolos con estados con b's a la izquierda. En concreto el estado inicial es bbb. Los estados aceptadores, son los cuatro que tienen una a en la antepenúltima posición.



Programación de autómatas deterministas
Nos va a ser muy útil poder programar los autómatas deterministas, lo cual como veremos más adelante será necesario para construir compiladores para los lenguajes de programación. En nuestro programa haremos lo siguiente:
(1)Representaremos los estados por números naturales, donde el cero hará referencia al estado inicial.
(2)Representaremos los símbolos del alfabeto por carácteres. Representaremos por \$ el final de la palabra de entrada
(3)El cálculo del autómata lo realizaremos mediante un bucle "while", en el que

iremos leyendo los carácteres y realizaremos las transiciones del autómata mediante una instrucción "switch-case".
(4)Al salir del bucle comprobamos si estamos en un estado aceptador.

3.2. Autómatas finitos indeterministas

Ahora pasaremos a estudiar una modificación de los autómatas anteriormente descritos, los autómatas finitos indeterministas. La principal diferencia, como bién dice su nombre, es que los cómputos de este autómata no estarán únivocamente determinados, es decir que desde un estado podremos aplicar cero, una o más transiciones del autómata para cada símbolo del alfabeto de entrada. Estos autómatas no van a ser en general programables, pero normalmente son mucho más simples que los autómatas deterministas ya que suelen tener menos estados y menos transiciones. También veremos la equivalencia entre los autómatas indeterministas y deterministas.

Definición: Un **autómata finito indeterminista** es una estructura
$M = (K, \Sigma, \Delta, q_0, F)$ donde:

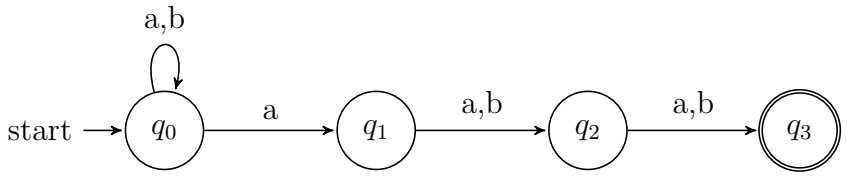
K es un conjunto finito de estados,
Σ es el alfabeto de entrada,
q_0 es el estado inicial,
$F \subseteq K$ es el conjunto de estados aceptadores, y
Δ es un subconjunto de $K \times (\Sigma \cup \{\lambda\}) \times K$.

Observamos, como comentábamos anteriormente, que Δ no es una función, por lo cual puede haber indeterminismos y transiciones con λ.

Definiciones: Si $M = (K, \Sigma, \Delta, q_0, F)$ es un autómata indeterminista definimos los siguientes conceptos:
Una **transición** es un elemento de Δ.
Una **configuración** de M es una palabra $qx \in K\Sigma^*$.
Si px, qy son configuraciones de M, decimos que px **produce** qy **en un paso de cómputo** si $x = uy$ para algún $u \in \Sigma \cup \{\lambda\}$ y $(p, u, q) \in \Delta$. Lo denotaremos como $px \vdash_M qy$.
Si px, qy son configuraciones de M, px **produce** qy si existen configuraciones $c_0, ..., c_n$ tales que $px = c_0 \vdash_M c_1 ... \vdash_M c_n = qy$. Lo escribimos como $px \vdash_M^* qy$.
Una palabra $x \in \Sigma^*$ es **reconocida o aceptada** por M si existe $q \in F$ tal que $q_0 x \vdash_M^* q$.
Definimos el **lenguaje asociado a** M como $L(M) = \{x \in \Sigma^* : x$ es reconocida por $M\}$.

Ahora vamos a ver la ventaja de los autómatas indeterministas haciendo el mis-

mo ejemplo que hemos realizado antes con un autómata determinista. Recordemos que queremos definir un autómata indeterminista $M = (K, \{a, b\}, \Delta, q_0, F)$ tal que $L(M) = \{x \in \{a, b\}^* : \text{hay una } a \text{ en la antepenúltima posición de } x\}$. Lo haremos de una manera mucho más simple a la anterior.



En este autómata nos aprovechamos del indeterminismo al entrar una a en el estado inicial y no definimos ninguna transición desde q_3. Esto nos permite pasar de los ocho estados anteriores a solo cuatro, con un grafo mucho más sencillo. Notemos que este grafo es correcto ya que si una palabra contiene una a en la antepenúltima posición el cómputo que debemos considerar para reconocer la palabra es el de mantenernos en el estado inicial hasta las últimas tres letras, y entonces hacer los tres pasos de cómputo que nos lleven al estado q_3.

Ahora pasamos a ver la equivalencia entre los autómatas deterministas y los autómatas indeterministas. Observemos que todo autómata determinista es también un autómata indeterminista. Evidentemente el recíproco no es cierto pero el siguiente teorema nos va a permitir construir un autómata determinista equivalente a un autómata indeterminista.

Teorema 3.1. *Para todo autómata finito indeterminista M existe un autómata finito determinista M' tal que $L(M) = L(M')$.*

Para demostrar este teorema necesitaremos definir la clausura de un estado.
Si $M = (K, \Sigma, \Delta, q_0, F)$ es un autómata indeterminista y $p \in K$, definimos la clausura de p como $\Lambda(p) = \{q \in K : p\lambda \vdash_M^* q\}$, es decir el conjunto de estados accesibles desde ese estado con λ-transiciones.

Demostración. Sea $M = (K, \Sigma, \Delta, q_0, F)$ un autómata indeterminista. Queremos construir un autómata determinista $M' = (K', \Sigma, \delta', q_0', F')$ equivalente a M. La idea para construirlo, es ver nuestro autómata indeterminista ocupando, en un momento dado, no un solo estado sino un conjunto de estados de M. Concretamente este conjunto será el conjunto de estados a los que se puede llegar desde el estado inicial dada un entrada determinada y habiendo realizando un número de cómputos en concreto.
Ahora ya podemos especificar los elementos de M'.
$K' = P(K)$, que es el conjunto formado por todos los subconjuntos de K.

$q'_0 = \Lambda(q_0)$, es decir el estado inicial es el conjunto de estados formado por los estados de M a los que se puede acceder sin leer ningún símbolo.
$F' = \{Q \in K' : Q \cap F \neq \emptyset\}$, es decir cualquier conjunto que contenga algún estado aceptador de M será un estado aceptador de M'.
δ' definida por

$$\delta'(Q, a) = \cup\{\Lambda(p) : p \in K \text{ y } (q, a, p) \in \Delta \text{ para algún } q \in Q\}$$

para todo $Q \in K', a \in \Sigma$.

Remarcamos que el autómata que acabamos de definir es efectivamente un autómata determinista ya que la función δ' esta únivocamente determinada y que no existen transiciones con λ. Ahora debemos ver que $L(M) = L(M')$. Para hacerlo demostraremos que para qualquier palabra $w \in \Sigma^*$ y dos estados $p, q \in K$ tenemos:

$$qw \vdash^*_M p \iff \Lambda(q)w \vdash^*_{M'} P \text{ para algún } P \text{ que contenga } p$$

.

Notemos que si demostramos esto la demostración concluye ya que $w \in L(M) \Leftrightarrow q_0 w \vdash^*_M q$ para algún $q \in F$ si y solo si $\Lambda(q_0)w \vdash^*_{M'} Q$ para algún Q que contenga q, que es lo mismo que decir $q'_0 w \vdash^*_{M'} Q$ para algún $Q \in F' \Leftrightarrow w \in L(M')$. Procedemos a demostrar la proposición por inducción sobre la longitud de w.
Caso básico: $|w| = 0$: Entonces $w = \lambda$. Hay que ver que

$$q \vdash^*_M p \Leftrightarrow \Lambda(q) \vdash^*_M P$$

para algún P que contenga a p.
Por un lado tenemos que $q \vdash^*_M p \Leftrightarrow p \in \Lambda(q) \Leftrightarrow \Lambda(q) \vdash^*_M P$.
Por otra parte tenemos que $\Lambda(q) \vdash^*_M P \Leftrightarrow P = \Lambda(q) \Leftrightarrow q \vdash^*_M p$. Lo que concluye el caso básico.
Caso inductivo: Suponemos que es cierto para palabras w tales que $|w| \leq n$.
Tomamos entonces w tal que $|w| = n + 1$. Podemos expresar $w = va$ donde $a \in \Sigma$ y v es una palabra tal que $|v| = n$.

Para provar la implicación de izquierda a derecha suponemos $qw \vdash^*_M p$. Entonces existen dos estados intermedios r, s tales que

$$qw = q(va) \vdash^*_M ra \vdash_M s \vdash^*_M p$$

Es decir separamos el cómputo de leer w en dos pasos. El primero consiste en leer v y el segundo en leer a. Del primer paso deducimos que $qv \vdash^*_M r$ y, como $|v| = k$, por hipótesis de inducción, $\Lambda(q)v \vdash^*_{M'} R$ para algún R que contenga r. Como $ra \vdash_M s$ tenemos que $(r, a, s) \in \Delta$, y por construcción de M', $\Lambda(s) \subseteq \delta'(R, a)$.Como además tenemos que $s \vdash^*_M p$, en consecuencia $p \in \Lambda(s)$. Ahora tenemos que $p \in \delta'(R, a)$. Por consiguiente $Ra \vdash_{M'} P$ para algún P que contiene

a p, y por tanto $\lambda(q)(va) \vdash^*_{M'} Ra \vdash_{M'} P$.

Para la otra implicación suponemos que $\Lambda(q)w \vdash^*_M P$ para algún P que contenga p. Entonces existe un estado intermedio R tal que $\Lambda(q)(va) \vdash^*_{M'} Ra \vdash_{M'} P$ para algún P que contenga a p y algún R tal que $\delta'(R, a) = P$.
Recordemos que, por definición,$\delta'(R, a) = \cup\{\Lambda(s) : (r, a, s) \in \Delta \text{ para algún } r \in R\}$. Además, como $p \in P = \delta'(R, a)$ existe un s en concreto tal que $p \in \Lambda(s)$, y existe un $r \in R$ en concreto tal que (r, a, s) es una transición de M. Ahora como $\Lambda(q)v \vdash^*_M R$ aplicamos la hipótesis de inducción y tenemos que $qv \vdash^*_M r$. Por consiguiente, juntando las condiciones obtenidas, deducimos que $qv \vdash^*_M r \Rightarrow q(va) \vdash^*_M ra \vdash_M s \vdash^*_M p$. □

3.3. Operaciones para autómatas finitos

El objetivo de este apartado es demostrar el siguiente resultado.

Teorema 3.2. *La clase de lenguajes aceptada por un autómata finito es cerrada respecto de las siguientes operaciones:*
a) unión
b) concatenación
c) clausura
d) complementación
e) intersección

Demostración. Para demostrar este teorema, en cada apartado, tomaremos dos autómatas finitos M_1 y M_2 y construiremos un nuevo autómata finito M que reconozca el lenguaje deseado. Si K_1 y K_2 son los conjuntos de estados de M_1 y M_2 respectivamente, consideraremos que $K_1 \cap K_2 = \emptyset$, en caso contrario es solo cuestión de renombrar los estados.

a) Unión. Si $M_1 = (K_1, \Sigma, \Delta_1, q_1, F_1), M_2 = (K_2, \Sigma, \Delta_2, q_2, F_2)$ son autómatas indeterministas, queremos construir un autómata indeterminista M tal que $L(M) = L(M_1) \cup L(M_2)$. La idea en este caso es que nuestro autómata final M, al recibir una palabra w, pueda realizar tanto el cómputo que haría M_1 como el que haría M_2, de tal forma que el lenguaje reconocido por M será la unión de ambos. Para hacerlo, crearemos un nuevo estado inicial q_0 con dos $\lambda-$transiciones una a q_1 y otra a q_2 y los dos autómatas iniciales invariantes, de modo que M imite a M_1 o a M_2. Formalmente $M = (K, \Sigma, \Delta, q_0, F)$ donde:
$K = K_1 \cup K_2 \cup \{q_0\}$, $F = F_1 \cup F_2$, $\Delta = \Delta_1 \cup \Delta_2 \cup \{(q_0, \lambda, q_1), (q_0, \lambda, q_2)\}$

b) Concatenación: Si $M_1 = (K_1, \Sigma, \Delta_1, q_1, F_1), M_2 = (K_2, \Sigma, \Delta_2, q_2, F_2)$ son autómatas indeterministas, queremos construir un autómata indeterminista M tal que $L(M) = L(M_1)L(M_2)$. Aquí lo que queremos es que M, al recibir una palabra w, realice el cómputo de M_1, y seguidamente el de M_2. Para hacerlo, M iniciará

siendo idéntico a M_1 pero los estados aceptadores de M_1 estarán conectados al estado inicial de M_2 mediante transiciones con λ. De este modo una palabra será aceptada por M si inicia realizando unos pasos de cómputo para ser aceptada por M_1 y sigue realizando otros para ser aceptada por M_2 que es precisamente lo que queremos. Formalmente $M = (K, \Sigma, \Delta, q_0, F)$ donde:
$K = K_1 \cup K_2$, $F = F_2$, $q_0 = q_1$, $\Delta = \Delta_1 \cup \Delta_2 \cup F_1 \times \{\lambda\} \times \{q_2\}$

c) Clausura: Si $M_1 = (K_1, \Sigma, \Delta_1, q_1, F_1)$ es el autómata indeterminista inicial, queremos construir un autómata indeterminista M tal que $L(M) = L(M_1)^*$. Esta vez lo que queremos es que M, al recibir una palabra w, pueda realizar el cómputo que realizaría M_1 un número indefinido de veces (o ninguna). Lo que haremos será crear un nuevo estado inicial, que también sea aceptador, para asegurar que la palabra vacía sea reconocida. Además este estado tendrá una transición con λ al estado inicial de M_1. El resto se mantendrá como en M_1 pero añadiremos $\lambda-$transiciones de los estados aceptadores de M_1 al estado inicial de M_1, de esta manera una vez se ha leído una palabra en $L(M_1)$ se reinicia el cómputo, que es precisamente la idea de la clausura. Formalmente $M = (K, \Sigma, \Delta, q_0, F)$ donde:
$K = K \cup \{q_0\}$, $F = F_1 \cup \{q_0\}$, $\Delta = \Delta_1 \cup F_1 \times \{\lambda\} \times \{q_1\}$.

d) Complementación: Consideramos $M_1 = (K_1, \Sigma, \delta_1, q_1, F_1)$ un autómata determinista inicial. Queremos construir un autómata determinista M tal que $L(M) = \overline{L(M_1)}$. En este caso el autómata resultante es $M = (K, \Sigma, \delta, q_0, K \backslash F)$; es decir, consiste en cambiar los estados aceptadores.

e) Intersección: Si $M_1 = (K_1, \Sigma, \delta_1, q_1, F_1), M_2 = (K_2, \Sigma, \delta_2, q_2, F_2)$ son autómatas deterministas, queremos construir un autómata determinista M tal que $L(M) = L(M_1) \cap L(M_2)$. Aquí la idea va a ser generar el conjunto de estados como pares, donde el primer elemento es un estado de M_1 y el segundo de M_2. La idea entonces es realizar las transiciones elemento a elemento, tal y como sería en los autómatas originales y entonces los estados aceptadores son aquellos en que ambos elementos eran aceptadores en sus respectivos autómatas originales. Formalmente $M = (K, \Sigma, \Delta, q_0, F)$ donde:
$K = K_1 \times K_2$, $F = F_1 \times F_2$, $q_0 = (q_1, q_2)$, $\delta((q_1, q_2), a) = (\delta_1(q_1, a), \delta_2(q_2, a))$ para todo $q_1 \in K_1, q_2 \in K_2$.
Obsérvese que si $w \in \Sigma^*$ entonces

$w \in L(M) \Leftrightarrow q_0 w \vdash_M^* p$ para algún $p \in F \Leftrightarrow (q_1, q_2) w \vdash_M^* (p_1, p_2)$ donde $p = (p_1, p_2)$ $\Leftrightarrow q_1 \vdash_{M_1}^* p_1$ y $q_2 \vdash_{M_2}^* p_2 \Leftrightarrow w \in L(M_1)$ y $w \in L(M_2) \Leftrightarrow w \in L(M_1) \cap L(M_2)$. $\square$

3.4. Autómatas finitos y expresiones regulares

Nuestro objetivo en esta sección es ver la relación entre los autómatas finitos y las expresiones regulares.

Proposición 3.3. *Todo lenguaje finito es regular*

Demostración. Si tenemos un lenguaje finito $L = \{a_1, ..., a_n\}$ tomamos la expresión regular $\alpha = a_1 \cup ... \cup a_n$. □

Teorema 3.4. *Un lenguaje L es regular $\Leftrightarrow$ existe un autómata finito M tal que $L(M) = L$.*

Demostración. Empezamos con la implicación de izquierda a derecha ($\Rightarrow$). Recordemos que tal y como hemos definido las expresiones regulares, la clase de los lenguajes regulares es la clase más pequeña de lenguajes que contenga el conjunto vacío,el conjunto $\{\lambda\}$ los conjuntos de un solo elemento a, donde $a \in \Sigma$ y que sea cerrada respecto a la unión, la concatenación y la clausura. Trivialmente podemos construir autómatas que reconozcan los conjuntos de un solo elemento (sería un autómata con dos estados donde uno es el inicial, el otro el aceptador, y la única transición que existe es la de leer el símbolo en cuestión que nos lleva del estado inicial al aceptador). Y el teorema que acabamos de ver nos dice que los lenguajes generados por autómatas finitos son cerrados respecto a estas tres operaciones, con lo que cualquier lenguaje regular es aceptado para algún autómata finito.

Para la otra implicación($\Leftarrow$) consideraremos un autómata indeterminista $M = (K, \Sigma, \Delta, q_1, F)$ y construiremos una expresión regular α tal que $L(\alpha) = L(M)$. Lo que haremos será expresar $L(M)$ como la unión de un número finito de lenguajes más simples. Suponemos $K = \{q_1, ..., q_n\}$, entonces para $i, j, = 1, .., n$, y $k = 0, ..., n$ definimos $\alpha(i, j, k)$ como el conjunto de palabras de Σ^* que, en nuestro autómata, nos llevan del estado q_i al estado q_j, sin pasar por ningún estado q_l donde $l > k$. En particular si $k = n$ tenemos que

$$\alpha(i, j, n) = \{w \in \Sigma^* : q_i w \vdash_M^* q_j\}$$

De lo que deducimos que $L(M) = \cup\{\alpha(1, j, n) : q_j \in F\}$
Si todos los conjuntos $\alpha(i, j, k)$ son regulares, la unión que hemos descrito también lo será, y por tanto $L(M)$ es regular. Así que solo nos falta ver que $\alpha(i, j, k)$ es regular, lo cual lo haremos por inducción sobre k.
Si $k = 0$, $\alpha(i, j, 0)$ será un conjunto formado únicamente por símbolos $a \in \Sigma$ tales que $(q_i, a, q_j) \in \Delta$, (más, quizás, la palabra vacía). Como, en todo caso, $\alpha(i, j, 0)$ es finito, es regular.
En el paso inductivo suponemos que $\alpha(i, j, k - 1)$ es regular para cualquier i, j. Entonces definiremos $\alpha(i, j, k)$ a partir de lenguajes regulares utilizando las operaciones básicas de las expresiones regulares. Tenemos que

$$\alpha(i, j, k) = \alpha(i, j, k - 1) \cup \alpha(i, k, k - 1)\alpha(k, k, k - 1)^*\alpha(k, j, k - 1)$$

Es decir las palabras que van del estado q_i al estado q_j sin pasar por ningún estado con índice mayor que k realizan uno de los siguientes cómputos en M.
1)Van del estado q_i al estado q_j sin pasar por ningún estado de índices mayor que $k - 1$.

2)Van de q_i a q_k sin pasar por ningún estado con índice mayor que $k-1$, luego van zero o más veces de q_k a q_k, y finalmente van de q_k a q_j sin pasar por ningún estado de índice mayor que $k-1$.
Como, por hipótesis de inducción, $\alpha(i,j,k-1)$, $\alpha(i,k,k-1)$, $\alpha(k,k,k-1)$, $\alpha(k,j,k-1)$ son lenguajes regulares, y los lenguajes regulares son cerrados respecto a la clausura, la concatenación y la unión, deducimos que $\alpha(i,j,k)$ es regular. □

Ahora tras ver este resultado podemos demostrar el Teorema 2.2 por el que un homomorfismo inverso aplicado a un lenguaje regular sigue siendo regular.

Para ello necesitamos una nueva definición para poder aplicar la funcíon de transición δ de un autómata determinista $M = (K, \Sigma, \delta, q_0, F)$ sobre una palabra entera y no sobre un solo símbolo. Para ello, definiremos la función $\widehat{\delta}$ de $K \times \Sigma^*$ a K partiendo de la función δ y por inducción sobre la longitud de una palabra x.
$\widehat{\delta}(q, \lambda) = q$
$\widehat{\delta}(q, xa) = \delta(\widehat{\delta}(q, x), a)$ donde $a \in \Sigma$

Demostración. (del Teorema 2.2) Sea $M = (K, \Sigma_2, \delta, q_0, F)$ un autómata determinista que reconoce L. Construimos el autómata determinista de $h^{-1}(L)$ al que llamamos $M' = (K, \Sigma_1, \delta', q_0, F)$ de la siguiente manera. Los estados, el estado final y el estado aceptador son los mismos. que en M. El alfabeto de entrada es ahora Σ_2 y la función de transición δ' viene definida por
$\delta'(q, a) = \widehat{\delta}(q, h(a))$
Notemos que es necesario hacer uso de $\widehat{\delta}$ ya que $h(a)$ podría contener más de un símbolo. Ahora lo que queremos ver es que $\widehat{\delta'}(q, x) = \widehat{\delta}(q, h(x))$ para todo $x \in \Sigma^*$. Lo haremos por inducción en la longitud de x.
Caso básico: $x = \lambda$, entonces $\widehat{\delta'}(q, \lambda) = q = \widehat{\delta} = \widehat{\delta}(q, h(\lambda))$.
Caso inductivo: Suponemos $\widehat{\delta'}(q, x) = \widehat{\delta}(q, h(x))$ y tomamos $a \in \Sigma$.
Tenemos $\widehat{\delta'}(q, xa) = \delta'(\widehat{\delta'}(q, x), a)$ por definición de $\widehat{\delta'}$
$= \delta'(\widehat{\delta}(q, h(x)), a)$ por hipótesis de inducción
$= \widehat{\delta}(\widehat{\delta}(q, h(x)), h(a))$ por definición de δ'
$= \widehat{\delta}(q, h(x)h(a))$ por construcción de $\widehat{\delta}$
$= \widehat{\delta}(q, h(xa))$ por definición de un homomorfismo.
Ahora ya podemos probar que $L(M') = h^{-1}(L(M))$. Tomamos $x \in \Sigma^*$. Entonces

$$x \in L(M') \Leftrightarrow \widehat{\delta'}(q_0, x) \in F \Leftrightarrow \widehat{\delta}(q_0, h(x)) \in F \Leftrightarrow h(x) \in L(M) \Leftrightarrow x \in h^{-1}(L(M))$$

□

3.5. Lenguajes no regulares

En la última sección hemos visto todas las propiedades de los lenguajes regulares, sin embargo aún no hemos visto sus limitaciones. En esta sección el objetivo es

poder demostrar cuando un lenguaje no es regular. Para eso tenemos el teorema de bombeo para lenguajes regulares.

Teorema 3.5. *Sea L un lenguaje regular. Entonces existe un entero $n \geq 1$ tal que para cualquier palabra $w \in L$, $|w| \geq n$ existe una representación $w = xyz$ donde*
(1) $y \neq \lambda$,
(2) $|xy| \leq n$ *y*
(3) $xy^iz \in L$, $\forall i \geq 0$.

Demostración. Sabemos que si L es regular, dicho lenguaje es aceptado por un autómata finito M. Suponemos que n es el número de estados de M y cogemos una palabra w de longitud n o mayor. Entonces consideramos los primeros n pasos de cómputo de esta palabra.

$$q_0(w_1w_2...w_n) \vdash_M q_1(w_2w_3...w_n) \vdash_M ... \vdash_M q_n$$

donde q_0 es el estado inicial. Recordemos que nuestro autómata tiene n estados y estamos realizando $n+1$ configuraciones, por tanto existen i, j, $0 \leq i < j \leq n$ tales que $q_i = q_j$. Por tanto la subpalabra $y = w_iw_{i+1}...w_j$ inicia en un estado, realiza unos pasos de cómputo y vuelve a el mismo estado; es decir que esta subpalabra no influye al determinar si la palabra original es aceptada o no por el autómata. Por tanto podemos generar una nueva palabra donde repetimos esta subpalabra indefinidamente, ya que esta nueva palabra seguirá siendo acceptada por M. Finalmente notemos que efectivamente $|xy| \leq n$ ya que solo hemos tomado los primeros n símbolos de w. □

Veamos un ejemplo donde aplicar este teorema.

Ejemplo: El lenguaje $L = \{0^n1^n : n \geq 0\}$ no es regular. Supongamos que si lo fuera y aplicamos el teorema anterior para un cierto n. Cogemos la palabra $w = 0^n1^n$, su representación $w = xyz$ debe cumplir que $y \neq \lambda$ y $|xy| \leq n$ de lo que deducimos que $y = 0^i$ para algun $i > 0$. Finalmente debe cumplirse que $xy^iz \in L$, lo qual claramente no es cierto, ya que esta nueva palabra sigue teniendo el mismo número de unos que la palabra original, mientras el número de ceros aumenta si $i > 1$, por tanto no pertenece a la palabra original.

Ejemplo: El lenguaje $L = \{x \in \{0,1\}^* : n_0(x) = n_1(x)\}$ no es regular. Suponemos que L es regular. Consideramos L_1 el lenguaje generado por la expresión regular 0^*1^*. Como los lenguajes regulares son cerrados respecto a la intersección, $L \cap L_1$ es regular. Sin embargo la intersección de estos dos lenguajes es $L_2 = \{0^n1^n : n \geq 0\}$, que ya hemos visto que no es regular. Por tanto L no es regular.

Ejemplo: El lenguaje $L = \{a^n : n$ es primo$\}$ no es regular. Como antes, supongamos que si lo fuese, y tomamos, cumpliendo las condiciones del teorema,

$x = a^p, y = a^q, z = a^r$ donde $p, r \geq 0$ y $q > 0$. Aplicando el teorema $xy^iz \in L$ para todo $i \geq 0$, es decir $p + iq + r$ es primo para todo $i \geq 0$. Esto, sin embargo, no es posible ya que si tomamos $i = p+2q+r+2$, entonces $p+iq+r = (q+1)(p+2q+r)$, que es el producto de dos números naturales mayores que 1, y por tanto no es primo.

3.6. Aplicaciones

3.6.1. Programa para reconocer un patrón en un texto

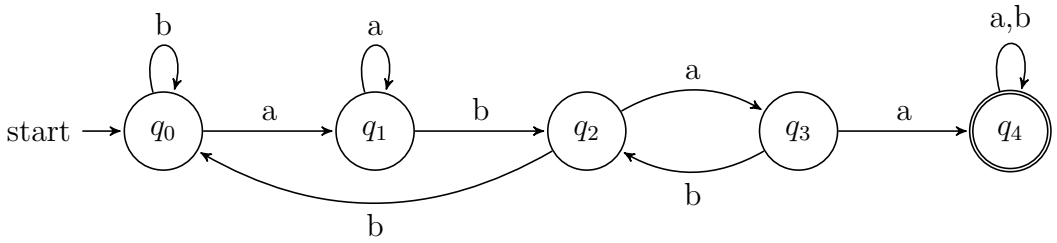
Consideremos el problema clásico de buscar en un texto una secuencia de caracteres, que suele denominarse *patrón*: tenemos un texto t, que suponemos largo en relación al patrón, escrito sobre un cierto alfabeto Σ, y un patrón p, que suele ser una palabra o un prefijo, y queremos determinar si el patrón aparece en algún lugar del texto. Podemos crear un algoritmo "ingenuo", que se limita a buscar la primera aparición del patrón.

Esta seria la programción en lenguaje C del programa "ingenuo".

```
int buscar (char texto [], char patron[], n,m){
//n y m son las longitudes del texto y patrón respectivamente
        int i,j;
        for(i = 0; i < n - m + 1; i++){
                j=0;
                while(texto[i+j] == patron[j]  &&  j < m){
                        j++;
                }
                if ( j == m);
                        return 1;
        }
        return 0;
}
```

Se trata de situarse en una posición del texto y comparar uno a uno los sucesivos símbolos del patrón con los correspondientes del texto a partir de la posición considerada, interrumpiendo la confrontación a la primera discrepancia, para pasar a la posición siguiente en el texto. El proceso finaliza en cuanto se encuentra la primera aparición del patrón. Una cota superior del número de comparaciones entre pares de símbolos efectuadas por este algoritmo es $n(m+1)/2$, donde n y m son las longitudes del texto y del patrón, respectivamente. Hemos calificado el algoritmo como ingenuo porque no saca provecho alguno de las informaciones que va recogiendo a medida que explora el texto. Veremos como se puede mejorar sustancialmente el tiempo de proceso utilizando un autómata finito. Para explicitar un caso concreto, supongamos $\Sigma = \{a, b\}$ y $p = abaa$. El grafo de abajo muestra un autómata finito determinista de cuatro estados que acepta exactamente los textos que contienen es-

te patrón. El autómata arranca en el estado q_0 i va leyendo el texto t de izquierda a derecha, según le indica el símbolo que lee en cada instante. Si el texto t contiene el patrón p como subpalabra, el autómata entra en el estado q_4, que será denominado estado de aceptación, justo cuando acaba de leer por primera vez dicha subpalabra.

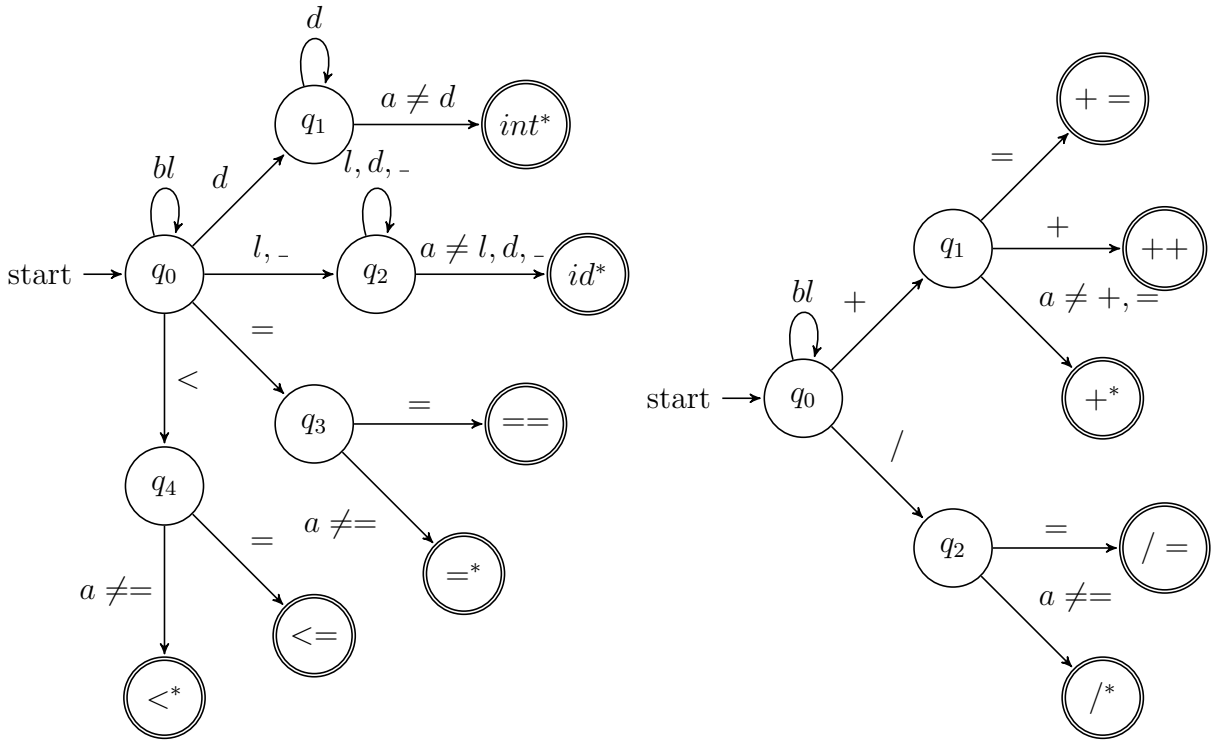


Es fácil demostrar que, sea cual sea el patrón p considerado, si $|p| = m$ existe un autómata finito determinista de $m + 1$ estados que efectúa el proceso de búsqueda. Este autómata ejecuta exactamente n pasos para leer un texto de longitud n. En consecuencia, la tarea total de construir el autómata a partir del patrón i procesar a continuación el texto con él, requiere $O(m + n)$ pasos, contra los $O(m \cdot n)$ del algoritmo ingenuo.

3.6.2. Analizador léxico

Como sabemos un compilador es el encargado de transformar un archivo de entrada en código ejecutable. Para eso hay tres fases: el análisis léxico, el análisis sintáctico y el análisis semántico.
El trabajo del analizador léxico es leer los caráctares del código fuente y transformarlos en unidades lógicas (como identificadores, enteros, palabras reservadas...) para que lo aborden las partes siguientes del compilador. El diseño de un analizador léxico puede hacerse mediante autómatas finitos.
Para simplicar, consideraremos que nuestro lenguaje de programación tiene solo las siguientes unidades lógicas (que llamaremos tokens):
(1) Los identificadores, palabras formadas por letras o dígitos, que pueden contener el subrayado, pero no pueden empezar por un dígito.
(2) El tipo entero, formado por uno o más dígitos
(3) Cada palabra reservada (for, while, int ...)
(4) Símbolos relacionales (=, ==, <, <=, >, >= ...).
(5) Operadores aritméticos, +, −, ∗, /.
(6) Símbolos de puntuación ((,), ;, :)

Ahora para construir nuestro autómata usaremos l para referirnos a cualquier letra, d para referirnos a cualquier dígito, a para cualquier símbolo del lenguaje y bl para referirnos al espacio en blanco.



En este grafo faltan el caso de $>, >=$ que es análogo al de $<, <=$, y los casos $*$, $-$ que son análogos al caso $+$.
Ahora para programar el autómata resultante hacemos las siguientes modificaciones:
(1) Cuando se llega a un estado marcado con $*$ no se lee el siguiente carácter.
(2) Cuando se llega a un estado aceptador se devuelve el token asociado a ese estado y volvemos al estado inicial.
(3) Cuando se llega a *id* se explora la tabla de las palabras reservadas del lenguaje. Si la unidad reconocida no está en la tabla se da como salida "identificador", si no la palabra reservada.
(4) Cuando llega a un comentario el analizador léxico lo saltará, ya que no tiene interés para el análisis sintáctico.

4. Lenguajes incontextuales

En el tema anterior hemos visto como podemos usar los autómatas finitos para reconocer lenguajes que pueden ser generados por una expresión regular. En este tema, definiremos un nuevo tipo de autómata, los autómatas con pila, los cuales se utilizan para el diseño del analizador sintáctico y semántico de un compilador.

4.1. Autómatas con pila

Los autómatas con pila tienen una cinta, un puntero y un conjunto de estados como en los autómatas finitos, pero esta vez también tenemos una pila. Cada vez que realizamos un paso de cómputo podemos añadir o quitar elementos de la pila. Esta pila funcionará igual a las pilas que pueden usarse en los lenguajes de programación, es decir, cuando añadimos un elemento, éste se pone en la cima de la pila y si queremos eliminar un elemento debe ser el elemento de la cima. Entonces, una palabra será reconocida si existe un cómputo que lee toda la palabra de entrada, termina en un estado aceptador y deja la pila vacía.

Definición: Un **autómata con pila** es una estructura $M = (K, \Sigma, \Gamma, \Delta, q_0, F)$ donde:

K es un conjunto finito de estados,
Σ es el alfabeto de entrada,
Γ es el alfabeto de la pila,
q_0 es el estado inicial,
$F \subseteq K$ es el conjunto de estados aceptadores, y
Δ es un subconjunto finito de $(K \times (\Sigma \cup \{\lambda\}) \times \Gamma^*) \times (K \times \Gamma^*)$.

Formalizando las ideas que comentábamos anteriormente, si en un paso de cómputo se aplica $((p, a, b), (q, x)) \in \Delta$ entonces:
(1) Se pasa del estado p al estado q.
(2) Si $a \neq \lambda$, se lee a en la cinta de entrada.
(3) En la pila se reemplaza b por x. Si $b \neq \lambda$, entonces b debe estar en la cima de la pila.

Definiciones: Si $M = (K, \Sigma, \Gamma, \Delta, q_0, F)$ es un autómata con pila definimos los siguientes conceptos:
Una **transición** es un elemento de Δ.
Una **configuración** de M es una palabra $\alpha q x \in \Gamma^* K \Sigma^*$, donde α es el contenido de la pila en ese momento, q el estado actual, y x la subpalabra de entrada que falta por leer.
Si $\alpha px, \beta qy$ son configuraciones de M, decimos que αpx **produce** βqy **en un paso de cómputo** si podemos pasar de αpx a βqy aplicando una transición de Δ. Lo denotaremos como $\alpha px \vdash_M \beta qy$.
Si $\alpha px, \beta qy$ son configuraciones de M, αpx **produce** βqy si podemos pasar de

αpx a βqy aplicando un número finito de transiciones de Δ Lo escribimos como $\alpha px \vdash_M^* \beta qy$.
Una palabra $x \in \Sigma^*$ es **reconocida o aceptada** por M si existe $q \in F$ tal que $\lambda q_0 x \vdash_M^* \lambda q \lambda$.
Definimos el **lenguaje asociado a** M como $L(M) = \{x \in \Sigma^* : x \text{ es reconocida por } M\}$.

Al igual que en los autómatas finitos, solo podemos programar los autómatas con pila que no contienen indeterminismos, por eso nos va a ser útil definir los autómatas con pila deterministas.

Definición: Sea $M = (K, \Sigma, \Gamma, \Delta, q_0, F)$ un autómata con pila
(a) Decimos que dos transiciones $((p_1, x_1, \alpha_1), (q_1, \beta_1))$, $((p_2, x_2, \alpha_2), (q_2, \beta_2))$ son compatibles si:
(1) $p_1 = p_2$
(2) $\alpha_1 = \alpha_2$ o $\alpha_1 = \lambda$ o $\alpha_2 = \lambda$
(3) $x_1 = x_2$ o $x_1 = \lambda$ o $x_2 = \lambda$

(b) Decimos que M es **determinista**, si en Δ no hay dos transiciones compatibles distintas

Ejemplo: En el tema anterior hemos visto que el lenguaje $L = \{x \in \{0,1\}^* : n_0(x) = n_1(x)\}$ no es regular y por tanto no puede expresarse con un autómata finito. Sin embargo si podemos hacerlo mediante un autómata con pila. Definimos el autómata con pila $M = (\{q_0, q_1, q_2\}, \{0,1\}, \{0,1,2\}, \Delta, q_0, \{q_2\})$ donde Δ tiene las siguientes transiciones:

1.$((q_0, \lambda, \lambda), (q_1, 2))$
2.$((q_1, 0, 2), (q_1, 02))$
3.$((q_1, 0, 0), (q_1, 00))$
4.$((q_1, 0, 1), (q_1, \lambda))$
5.$((q_1, 1, 2), (q_1, 12))$
6.$((q_1, 1, 1), (q_1, 11))$
7.$((q_1, 1, 0), (q_1, \lambda))$
8.$((q_1, \lambda, 2), (q_2, \lambda))$

La idea es utilizar la pila como un contador, si hay n ceros en la pila quiere decir que en ese momento se han leído n ceros más que unos. Para eso cada vez que leemos un 0 aplicamos una de las transiciones 2-4 mientras que si leemos un 1 aplicamos una de las transiciones 5-7. Al aplicar estas transiciones tenemos dos opciones, eliminar el símbolo de la pila en caso de que sea el opuesto al que estamos leyendo, o añadir el símbolo que leemos en la pila. El 2 solo nos sirve para asegurar que la pila no esta vacía y se va a eliminar en el último paso de cómputo. Veamos como sería el cómputo de la palabra $x = 001110$.

Estado	Cinta	Pila	Transición
q_0	001110	λ	–
q_1	001110	2	1
q_1	01110	02	2
q_1	1110	002	3
q_1	110	02	7
q_1	10	2	7
q_1	0	12	5
q_1	λ	2	4
q_2	λ	λ	8

4.2. Gramáticas incontextuales

En el primer tema vimos como podíamos representar lenguajes mediante una sola palabra gracias a las expresiones regulares. Una gramática incontextual nos va a permitir generar un lenguaje mediante un conjunto de reglas.

Definición: Una **gramática incontextual** es una estructura $G = (V, \Sigma, P, S)$ donde:
V es un alfabeto.
Σ es un subconjunto V a cuyos elementos se les llama símbolos terminales. A los elementos de $V \backslash \Sigma$ se les llama variables.
P es un subconjunto finito de $(V \backslash \Sigma) \times V^*$ a cuyos elementos se les llama reglas o producciones.
S es la variable inicial.

Definiciones: Sea $G = (V, \Sigma, P, S)$ una gramática incontextual:
Si $(A, x) \in P$, escribiremos $A \to x$.
Si $(A, x_1), ...(A, x_n) \in P$, escribiremos $A \to x_1|...|x_n$.
Sean $u, v \in V^*$.
(a) v **se deriva de** u **en un paso** si existen $x, y, w \in V^*$, $A \in (V \backslash \Sigma)$ tales que $u = xAy$, $v = xwy$ y $(A, w) \in P$. Lo escribiremos como $u \Rightarrow_G v$.
(b) v **se deriva de** u si existen $u_0, u_1, ..., u_n \in V^*$, tales que $u = u_0 \Rightarrow_G u_1... \Rightarrow_G u_n = v$. Lo escribiremos como $u \Rightarrow_G^* v$.
Similarmente podemos definir el concepto de **derivación por la izquierda** (respectivamente **derivación por la derecha**) que consiste en sustituir la primera variable, es decir la situada más a la izquierda de la palabra (resp. la última). Formalmente $u \overset{L}{\Rightarrow} v$ (resp. $u \overset{R}{\Rightarrow} v$) si existen $x \in \Sigma^*$, $w, y \in V^*$ (resp. $x, w \in V^*, y \in \Sigma^*$) y $A \in (V \backslash \Sigma)$ tales que $u = xAy$, $v = xwy$ y $(A, w) \in P$.
Escrbimos $u \overset{L}{\Rightarrow}^* v$ (resp. $u \overset{R}{\Rightarrow}^* v$) si existen $u_0, u_1, ..., u_n \in V^*$, tales que $u = u_0 \overset{L}{\Rightarrow} u_1... \overset{L}{\Rightarrow} u_n = v$ (resp. $u = u_0 \overset{R}{\Rightarrow} u_1... \overset{R}{\Rightarrow} u_n = v$).
Definimos el **lenguaje generado por G** como $L(G) = \{x \in \Sigma^* : S \Rightarrow_G^* x\}$.

Decimos que un lenguaje L es **incontextual** si existe una gramática incontextual G tal que $L = L(G)$

Ejemplo: Como antes, queremos generar $L = \{x \in \{0,1\}^* : n_0(x) = n_1(x)\}$. Definimos una gramática incontextual $G = (\{S, 0, 1\}, \{0, 1\}, P, S)$ donde P consta de:

1. $S \to 0S1S$,
2. $S \to 1S0S$,
3. $S \to \lambda$

Solo tenemos tres producciones , las dos primeras nos permiten poner ceros y unos libremente, pero siempre poniendo el mismo número de ceros y unos. La otra es simplemente eliminar la variable para finalizar la derivación. Claramente todas las palabras generadas de esta manera pertenecerán al lenguaje que queremos y todas las palabras del lenguaje pueden generarse según estas reglas.

Ahora veremos que si restringimos las gramáticas incontextuales a unas ciertas condiciones, podemos generar únicamente los lenguajes regulares.

Definición: Una **gramática regular** es una estructura $G = (V, \Sigma, P, S)$, donde V, Σ y S son como en la definición de gramática incontextual, pero todas sus producciones son de la forma

$$A \to xB \text{ o } A \to x$$

donde $A, B \in V \backslash \Sigma$ y $x \in \Sigma^*$.
Si $x = x_1...x_n$ donde $x_i \in \Sigma$ podemos sustituir las regla $A \to xB$ por las reglas

$$A \to x_1A_1, A_1 \to x_2A_2, ..., A_{n-2} \to x_{n-1}A_{n-1}, A_{n-1} \to x_nB$$

y la producción $A \to x$ por las producciones

$$A \to x_1A_1, A_1 \to x_2A_2, ..., A_{n-2} \to x_{n-1}A_{n-1}, A_{n-1} \to x_n$$

De tal modo podemos simplificar la definición anterior, imponiendo que $x \in \Sigma \cup \{\lambda\}$ lo cual nos va a simplificar la demostración del teorema siguiente.

Ahora juntando los resultados del tema anterior obtenemos el siguiente teorema.

Teorema 4.1. *Para todo lenguaje L son equivalentes:*
(1) Existe un autómata finito determinista M tal que $L(M) = L$.
(2) Existe un autómata finito indeterminista M tal que $L(M) = L$.
(3) Existe una expresión regular α tal que $L(\alpha) = L$.
(4) Existe una gramática regular G tal que $L(G) = L$.

Demostración. Por los teoremas 3.1 y 3.4, sabemos que los tres primeros puntos son equivalentes. Por tanto nos limitaremos a demostrar que (4) $\Rightarrow$ (2) y (1) $\Rightarrow$ (4). Empezamos demostrando que (4) $\Rightarrow$ (2).
Suponemos pues que tenemos una gramática regular $G = (V, \Sigma, P, S)$ donde las producciones de G son de la forma anteriormente descrita. Definimos un autómata finito $M = (K, \Sigma, \Delta, q_0, F)$ donde:
$K = V \backslash \Sigma \cup \{f\}$, donde f es un nuevo símbolo
$F = \{f\}$
$q_0 = S$
$\Delta = \{(A, x, B) : A \to xB \in P\} \cup \{(A, x, f) : A \to x\}$
Es fácil comprobar que $L(G) = L(M)$.

Para demostrar que (1) $\Rightarrow$ (4) tomamos un autómata finito determinista $M = (K, \Sigma, \delta, q_0, F)$, definimos una gramática regular $G = (V, \Sigma, P, S)$ donde
$V = K \cup \Sigma$,
$S = \{q_0\}$,
$P = \{q \to ap : \delta(q, a) = p\} \cup \{q \to \lambda : q \in F\}$.
Se comprueba fácilmente que $L(M) = L(G)$. □

4.3. Árboles de derivación

En esta sección veremos como expresar las derivaciones en una gramática incontextual mediante un árboles de derivación. El concepto de árbol de derivación es esencial en el diseño de compiladores.

Definición: Si $G = (V, \Sigma, P, S)$ es una gramática incontextual, asignamos a una derivación $S \Rightarrow_G^* \alpha$ de G el siguiente **árbol de de derivación**:
(1) La raíz del árbol es S.
(2) Cada nodo interno del árbol corresponde a una variable sustituida en el proceso de derivación.
(3) La palabra formada por las hojas del árbol en sentido de izquierda a derecha es α.

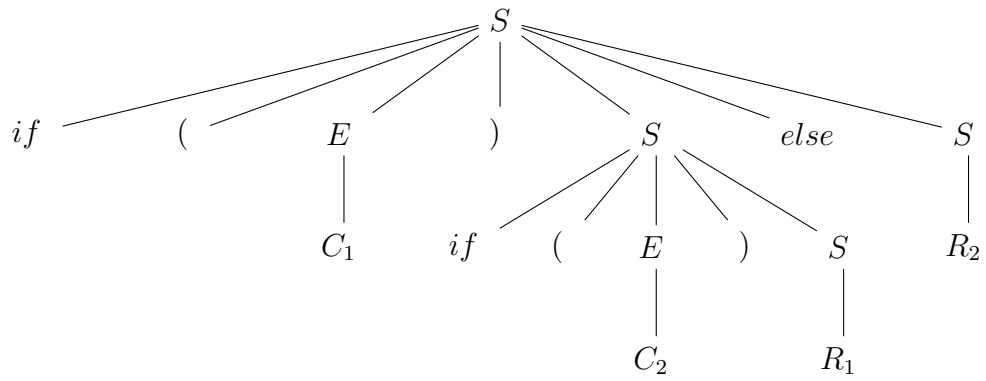
Deficinición: Decimos que una gramática incontextual G es **ambigua** si existe $x \in L(G)$ que tiene más de un árbol de derivación.

Ejemplo: Veremos un ejemplo muy útil para lenguajes de programación. Definimos la gramática incontextual $G = (V, \Sigma, P, S)$ donde $V \backslash \Sigma = \{S, E\}$ y P consta de:

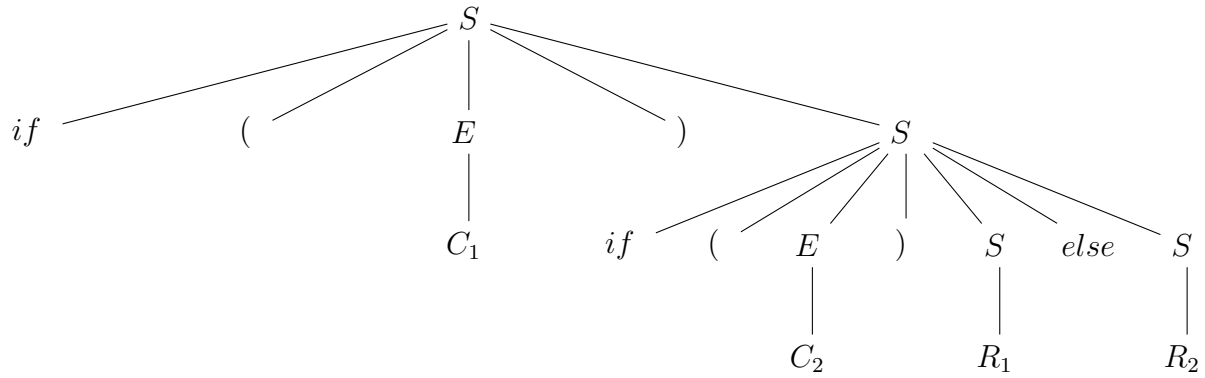
1.$S \to if(E)\ S\ else\ S$
2.$S \to if(E)\ S$
Donde S y E pueden ser sustituidas por condiciones.

Vamos a ver que G es ambigua.
Consideramos que queremos realizar una instrucción del tipo $if(C_1)\ if(C_2)\ R_1\ else\ R_2$.

Podemos observar que el siguiente árbol deriva la instrucción deseada



Sin embargo, este árbol también deriva la misma instrucción



Tenemos dos árboles para la misma palabra, entonces la gramática definida es ambigua.
Podemos observar que el primer árbol de derivación es erróneo ya que el "else" hace referencia al "if" externo y no al interno como debería ser. En este caso si añadimos la variable S_1 y sustituimos las dos primeras reglas por las tres siguientes

$S \to if(E)\ S_1\ else\ S$
$S \to if(E)\ S$
$S_1 \to if(E)\ S_1\ else\ S_1$

eliminamos la ambiguedad de la gramática ya que añadiendo S_1 evitamos que la gramática genere instrucciones condicionales simples. Claramente el lenguaje generado es el mismo.

En general siempre va a ser preferible sustituir una gramática ambigua por una no ambigua. Como ya hemos visto en el ejemplo anterior las gramáticas ambiguas pueden contener instrucciones con diferentes interpretaciones, algunas de las cuales pueden ser incorrectas y, por tanto, dan resultado a diferentes ejecuciones. Sin embargo, existen lenguajes que solo puede generarse mediante gramáticas ambiguas, estos lenguajes se llaman **inherentemente ambiguos**. Por suerte, no es el caso de los lenguajes de programación.

Observación: Si $G = (V, \Sigma, P, S)$ es una gramática incontextual, $A \in V \backslash \Sigma$ y $w \in \Sigma^*$, entonces las siguientes condiciones son equivalentes:

(a) $A \Rightarrow_G^* w$
(b) Existe un árbol de derivación asociado a $A \Rightarrow_G^* w$
(c) Existe una derivación por la izquierda tal que $A \overset{L}{\Rightarrow}^* w$
(d) Existe una derivación por la derecha tal que $A \overset{R}{\Rightarrow}^* w$

4.4. Autómatas con pila y gramáticas incontextuales

El objetivo de este tema es demostrar que los autómatas con pila son el módelo computacional para reconocer los lenguajes que generan las gramáticas incontextuales.

Teorema 4.2. *Para todo lenguaje L, existe un autómata con pila M tal que $L(M) = L$ si y solo si existe una gramática incontextual G tal que $L(G) = L$.*

Demostración. Demostraremos las dos implicaciones por separado.

Empezaremos demostrando que cualquier lenguaje incontextual es reconocido por algún autómata con pila. Tomamos una gramática incontextual $G = (V, \Sigma, P, S)$; queremos construir un autómata con pila M tal que $L(M) = L(G)$. Este autómata tendrá solamente dos estados, q_0 (el inicial) y q_1 (el aceptador). El cómputo va estar permanente en q_1 después de su primer paso de cómputo. Además M usará V como el alfabeto de la pila. Formalmente $M = (\{q_0, q_1\}, \Sigma, V, \Delta, q_0, \{q_1\})$ donde Δ consta de las siguientes transiciones:

(1) $((q_0, \lambda, \lambda), (q_1, S))$
(2) $((q_1, \lambda, A), (q_1, x))$ para cualquier producción $A \to x$ en P.
(3) $((q_1, a, a), (q_1, \lambda))$ para todo $a \in \Sigma$.

Nuestro autómata empieza realizando la transición (1), que escribe S en la pila inicialmente vacía y nos sitúa en el estado q_1. En los siguientes pasos, o bien sustituye el elemento de la cima de la pila $A \in V \backslash \Sigma$ por un x tal que $(A, x) \in P$ (transición (2)), o bien elimina el primer elemento de la pila a, siempre y cuando coincida con el símbolo a leer en la cinta (transición (3)). La idea de este autómata es que los elementos de la pila coincidan con una supalabra (empezando por el inicio) de la cinta de entrada. Para ello cada vez que aplicamos una transición del tipo (2) y sustituimos en la pila una variable por una palabra de V^*, haremos la misma sustitución que haríamos en la gramática incontextual. Ahora debemos probar que efectivamente $L(M) = L(G)$, lo que haremos demostrando la siguiente condición.

(∗) Sea $w \in \Sigma^*$ y $\alpha \in (V \backslash \Sigma)V^* \cup \{\lambda\}$. Entonces $S \overset{L}{\Rightarrow}^* w\alpha \Leftrightarrow Sq_1w \vdash_M^* \alpha q_1$.

Observemos que por la transición (1) $\lambda q_0 w \vdash_M Sq_1w$, por tanto $\lambda q_0 w \vdash_M^* \lambda q_1$. Con la condición (∗) la demostración de esta implicación concluye ya que tomando $\alpha = \lambda$, tenemos que $S \overset{L}{\Rightarrow}^* w\alpha$ si y solo si $Sq_1w \vdash_M^* \lambda q_1$. Por tanto
$w \in L(G) \Leftrightarrow S \Rightarrow^* w \Leftrightarrow S \overset{L}{\Rightarrow}^* w \Leftrightarrow S \overset{L}{\Rightarrow}^* w\lambda \Leftrightarrow Sq_1w \vdash_M^* \lambda q_1 \Leftrightarrow \lambda q_0 w \vdash_M^* \lambda q_1 \Leftrightarrow w \in L(M)$.
Entonces vamos a demostrar (∗) empezando por la implicación de izquierda a derecha. Suponemos que $S \overset{L}{\Rightarrow}^* w\alpha$, donde $w \in \Sigma^*$ y $\alpha \in (V \backslash \Sigma)V^* \cup \{\lambda\}$. Ahora vamos a demostrar por inducción sobre la longitud de la derivación por la izquierda de w desde S que $Sq_1w \vdash_M^* \alpha q_1$.
Caso básico: Si la derivación es de longitud 0 entonces $w = \lambda$ y $\alpha = S$, por tanto $Sq_1w \vdash_M^* \alpha q_1$.
Caso inductivo: Si $S \overset{L}{\Rightarrow}^* w\alpha$ por una derivación de longitud $\leq n$, entonces $Sq_1w \vdash_M^* \alpha q_1$.
Tomamos $S = u_0 \overset{L}{\Rightarrow} u_1 \overset{L}{\Rightarrow} ... \overset{L}{\Rightarrow} u_n \overset{L}{\Rightarrow} u_{n+1} = w\alpha$ una derivación por la izquierda de $w\alpha$ desde S. Suponemos que A es la primera variable de u_n. Entonces, como $u_n \overset{L}{\Rightarrow} u_{n+1}$, tenemos que $u_n = xAy$ y $u_{n+1} = xzy$ donde $x \in \Sigma^*$, $y, z \in V^*$ y $(A, z) \in P$. Como existe una derivación por la izquierda de longitud n de $u_n = xAy$ desde S podemos aplicar la hipótesis de inducción, tomando $\alpha = Ay$ y tenemos que

$$Sq_1x \vdash_M^* (Ay)q_1$$

Como $A \to z$ es una producción de G tenemos $(Ay)q_1 \vdash_M (zy)q_1$ aplicando una transición del tipo (2). Fijémonos que $u_{n+1} = xzy = w\alpha$. Como $w, x \in \Sigma^*$ y α empieza con una variable existe un $\beta \in \Sigma^*$ tal que $w = x\beta$ y por tanto $\beta\alpha = zy$. Observemos que

$$(zy)q_1\beta = (\beta\alpha)q_1\beta \vdash_M^* \alpha q_1$$

al poder aplicar $|\beta|$ transiciones del tipo (3) ya que $\beta \in \Sigma^*$.
Ahora por tanto al juntar las condiciones obtenidas tenemos

$$Sq_1w = Sq_1x\beta \vdash_M^* (Ay)q_1\beta \vdash_M (zy)q_1\beta = (\beta\alpha)q_1\beta \vdash_M^* \alpha q_1$$

concluyendo así el paso inductivo.

Demostramos ahora la otra implicación de (∗).
Suponemos que $Sq_1w \vdash_M^* \alpha q_1$ donde $w \in \Sigma^*$ y $\alpha \in (V \backslash \Sigma)V^* \cup \{\lambda\}$, queremos ver que $S \overset{L}{\Rightarrow}^* w\alpha$.
Nuevamente haremos la demostración por inducción, esta vez por el número de transiciones del tipo (2) que realicemos en el cómputo de M.
Caso básico: Niguna transición del tipo (2). El primer paso de cómputo, tras haber realizado la transición (1), debe ser por fuerza del tipo (2), ya que en la pila no hay símbolo terminales. Por tanto si $Sq_1w \vdash_M^* \alpha q_1$ sin transiciones del tipo (2), entonces $w = \lambda$ y $\alpha = S$, por tanto la condición es cierta.
Caso inductivo: Si $Sq_1w \vdash_M^* \alpha q_1$ en un cómputo con n o menos transiciones del

tipo (2) entonces $S \overset{L}{\Rightarrow}^* w\alpha$. Suponemos que $Sq_1w \vdash_M^* \alpha q_1$ en $n+1$ transiciones de tipo (2) y separamos el cómputo en dos partes, en la primera parte realizaremos n transiciones de tipo (2), y la segunda será el resto del cómputo.

$$Sq_1(x\beta) \vdash_M^* (Ay)q_1\beta \vdash_M (zy)q_1\beta \vdash_M^* \alpha q_1$$

donde $w = x\beta$ para algún $x, \beta \in \Sigma^*$ y $(A, z) \in P$. Como $Sq_1(x\beta) \vdash_M^* (Ay)q_1\beta$ deducimos que $Sq_1x \vdash_M^* (Ay)q_1$ y por tanto podemos aplicar la hipótesis de inducción y tenemos que $S \overset{L}{\Rightarrow}^* xAy$ y aplicando la producción $A \to z$ tenemos que $S \overset{L}{\Rightarrow}^* xzy$. Como además tenemos $(zy)q_1\beta \vdash_M^* \alpha q_1$ y para realizar este cómputo solo se realizan transiciones del tipo (3) tenemos que $\beta\alpha = zy$ y por tanto siguiendo desde la condición anterior tenemos que $S \overset{L}{\Rightarrow}^* xzy = x\beta\alpha = w\alpha$.
Esto concluye la demostración de $(*)$ y consecuentemente la primera implicación del teorema.

Procedamos con la otra implicación del teorema. Queremos ver que cualquier lenguaje asociado a un autómata con pila es un lenguaje incontextual.
Vamos a definir un autómata con pila simple, ya que así podremos construir una gramática incontextual de forma más sencilla.
Cualquier transición $((q, a, \beta), (p, \gamma))$ del autómata con pila simple donde q no es el estado inicial, satisface que $\beta \in \Gamma$ (alfabeto de la pila) y $|\gamma| \leq 2$.
Es decir, este autómata consulta siempre la cima de la pila (ningún símbolo más) y o bien lo elimina o bien lo sustituye por uno o dos símbolos. Evidentemente la pila empieza estando vacía, con lo que si únicamente hubiera transiciones de este tipo no podría realizar ningún cómputo, por eso estas transiciones no tienen esta limitación cuando q es el estado inicial.
Ahora lo que queremos ver es que si un lenguaje es aceptado por un autómata con pila cualquiera, entonces es aceptado por un autómata con pila simple. Para verlo tomamos un autómata con pila $M = (K, \Sigma, \Gamma, \Delta, q_0, F)$ y construiremos un autómata con pila simple $M' = (K', \Sigma, \Gamma \cup Z, \Delta', q_0', \{f'\})$ que reconozca $L(M)$. En este autómata consideramos un nuevo estado inicial q_0' y un nuevo y único estado aceptador f'. También añadimos Z al alfabeto de la pila, símbolo que usaremos para denotar el último símbolo de la pila. En Δ' añadimos la transición $((q_0', \lambda, \lambda), (q_0, Z))$ que coloca el símbolo Z en la pila, el cual permanecerá en la pila hasta el último paso cómputo, en el cual eliminaremos Z para dejar la pila vacía. Ningún otro cómputo pondrá Z en la pila. También añadimos la transición $((f, \lambda, Z), (f', Z))$ para todo $f \in F$. Al realizar una de estas transiciones eliminaremos Z de la pila y el cómputo finalizará.
Inicialmente, Δ' consiste de las dos transiciones que hemos comentado (que serán el primer y último paso de cómputo) y todas las transiciones de Δ. Pero debemos sustituir todas las transiciones de Δ' que infrinjan la norma de simplicidad. Lo haremos en tres pasos, primero sustituiremos aquellas transiciones donde $|\beta| \geq 2$ (considerando una transición $((q, a, \beta), (p, \gamma))$), luego modificar aquellas en que $\gamma \geq 2$ y finalmente aquellas donde $\beta = \lambda$. En cada paso debemos procurar no crear una nueva excepción de las que ya hemos solucionado en el paso anterior.

Empecemos pues considerando una transición $((q, a, \beta), (p, \gamma))$ donde $\beta = B_1B_2...B_n$ por $B_i \in \Gamma$ y $n > 1$. Lo que haremos será eliminar β símbolo a símbolo y no todo de golpe. Añadiremos a Δ' las siguientes transiciones:

$((q, \lambda, B_1), (q_{B_1}, \lambda))$,
$((q_{B_1}, \lambda, B_2), (q_{B_1B_2}, \lambda))$,
...
$((q_{B_1...B_{n-2}}, \lambda, B_{n-1},), (q_{B_1...B_{n-1}}, \lambda))$
$((q_{B_1...B_{n-1}}, a, B_n,), (p, \gamma))$

donde $q_{B_1...B_i}$ son nuevos estados que nos sirven para saber cual ha sido el último símbolo leído. De este modo, nos aseguramos que estas transiciones se realizan en el orden correcto y por tanto son equivalentes a realizar directamente la transición inicial. Repetimos este proceso con todas las transiciones $((q, a, \beta), (p, \gamma))$ donde $\beta > 2$.
Similarmente reemplazamos las transiciones $((q, a, \beta), (p, \gamma))$ donde $\gamma = C_1C_2...C_n$ con $n > 1$ por las transiciones

$((q, a, \beta), (r_1, C_m))$,
$((r_1, \lambda, \lambda), (r_2, C_{m-1}))$,
...
$((r_{m-2}, \lambda, \lambda), (r_{m-1}, C_2))$
$((r_{m-1}, \lambda, \lambda), (p, C_1))$

donde r_i son nuevos estados que nos sirven para saber cual ha sido el último símbolo añadido a la pila. Recordemos que por el funcionamiento de la pila debemos añadir los símbolos de γ del último al primero. Notemos también que estas transiciones cumplen que $\gamma \leq 1$ lo cual es más fuerte que la condición de simplicidad que queríamos imponer, volverán a ser $\gamma \leq 2$ en el siguiente paso. En este paso no hemos añadido ninguna transición de las modificadas en el paso 1.
Finalmente queremos reemplazar las transiciones del tipo $((q, a, \lambda), (p, \gamma))$ donde $q \neq q'_0$. Las reemplazaremos por transciones $((q, a, A), (p, A\gamma))$ para todo $A \in \Gamma \cup \{Z\}$, es decir en vez de añadir directamente un símbolo en la pila, quitamos el símbolo de la cima y lo volvemos a añadir junto con el símbolo que queríamos. Recordemos que gracias al elemento Z sabemos que siempre hay almenos un símbolo en la pila (salvo al final del cómputo). Notemos que las nuevas transiciones introducen $\gamma's$ de longitud 2, lo cual no es un problema para la condición de simplicidad y es necesario para no perder la generalidad de los automátas con pila. Estas modifaciones no añaden ninguna de las dos excepciones anteriores.
Por construcción, todas las modificaciones que hemos hecho son equivalentes a la transición original, por eso el autómata con pila M' satisface que $L(M) = L(M')$. Con lo que ahora podemos simplificar la demostración a encontrar una gramática incontextual G tal que $L(G) = L(M')$.
Tomamos $G = (V, \Sigma, P, S)$, donde V contiene, además de un nuevo símbolo para la variable inicial S y los símbolos de Σ, una nueva variable $[qAp]$ para todo $p, q \in K'$ y para todo $A \in \Gamma \cup \{\lambda, Z\}$. Para entender el significado de estas nuevas variables

recordemos que queremos que G genere el lenguaje que reconoce M'. Entonces la variable $[qAp]$ hará referencia a una subpalabra de la palabra de entrada que puede ser leída desde un instante en que M' esta en el estado q con A en la cima de la pila, hasta otro en el que se elimina A y se pasa al estado p. Observemos pues que si $A = \lambda$, entonces $[q\lambda p]$ hace referencia a una supalabra de la palabra de entrada que puede ser leída partiendo de un instante en el que nos encontramos en el estado q hasta otro en el que pasamos al estado p sin variar la pila.
Ahora definimos las producciones de G que serán de cuatro tipos.
(1) $S \to [q_0 Z f']$, donde q_0 es el estado inicial de M y f' el estado aceptador de M'.
(2) Para toda transición de M' $((q, a, B), (r, C))$, donde $q, r \in K'$, $a \in \Sigma \cup \{\lambda\}$, $B, C \in \Gamma \cup \{\lambda\}$, y para todo $p \in K'$ añadimos la regla $[qBp] \to a[rCp]$.
(3) Para toda transición de M' $((q, a, B), (r, C_1C_2))$, donde $q, r \in K'$, $a \in \Sigma \cup \{\lambda\}$, $B \in \Gamma \cup \{\lambda\}$, $C_1, C_2 \in \Gamma$, y para todos los $p, p' \in K'$ añadimos la producción $[qBp] \to a[rC_1p'][p'C_2p]$.
(4) Para todo $q \in K'$ añadimos la producción $[q\lambda q] \to \lambda$.

Como M' es un autómata con pila simple todas sus transiciones están relacionadas con alguna producción del tipo (2) o (3) en G. La regla del tipo (1) es la primera regla que se aplica en la gramática, que generará las palabras que reconoce M' entre el estado q_0 y f' antes de eliminar el último símbolo de la pila, es decir $L(M')$. Las reglas del tipo (4) nos dicen que para mantenernos en el mismo estado no hace falta cambiar la pila. Finalmente las reglas del tipo (2) o (3) dicen que si $((q, a, B), (p, \gamma))$ es una transición de Δ', uno de los posibles pasos de cómputos que nos llevan del estado q al estado p eliminando B de la cima de la pila, consiste en leer a, sustituir B por γ, pasando a un estado intermedio r y finalmente eliminar γ y finalizar en p. Para ver que efectivamente este gramática genera el lenguaje que queremos enunciamos la siguiente condición.

$(**)$Para todo $p, q \in K'$, $A \in \Gamma \cup \{\lambda\}$, y $w \in \Sigma^*$

$$[qAp] \Rightarrow_G^* w \;\Leftrightarrow\; Aqw \vdash_{M'}^* \lambda p \lambda$$

Con esta condición la demostración de la implicación de derecha a izquierda concluye, ya que tomamos $q = q_0, A = \lambda, p = f'$, y obtenemos que
$w \in L(G) \Leftrightarrow \lambda q w \vdash_{M'}^* \lambda f' \lambda \Leftrightarrow [q_0 \lambda f'] \Rightarrow_G^* w \Leftrightarrow w \in L(M')$.
La condición $(**)$ se demuestra por inducción sobre la longitud de una derivación en G o de la longitud de un cómputo en M' de forma similar a como hemos demostrado la condición $(*)$ que nos servía para demostrar la otra implicación del teorema.

Así concluimos la demostración del Teorema. □

4.5. Lenguajes no incontextuales

El objetivo de esta sección será demostrar sobre que operaciones es cerrado un lenguaje incontextual y cuando un lenguaje no es incontextual.

Teorema 4.3. *Los lenguajes incontextuales son cerrados respecto a la unión, concatenación y clausura.*

Demostración. Para demostrar todos los casos consideraremos dos gramáticas incontextuales $G_1 = (V_1, \Sigma_1, R_1, S_1)$ y $G_2 = (V_2, \Sigma_2, R_2, S_2)$. Supondremos que $V_1 \backslash \Sigma_1 \cap V_2 \backslash \Sigma_2 = \emptyset$. En caso contrario renombramos las variables.

a) Unión: Queremos construir una gramática incontextual G tal que $L(G) = L(G_1) \cup L(G_2)$. La idea va a ser crear una nueva variable inicial S y dos producciones que sustituyen la variable inicial de G por la de G_1 y G_2 respectivamente. Definimos pues $G = (V_1 \cup V_2 \cup \{S\}, \Sigma_1 \cup \Sigma_2, R, S)$ donde
$R = R_1 \cup R_2 \cup \{S \to S_1, S \to S_2\}$.

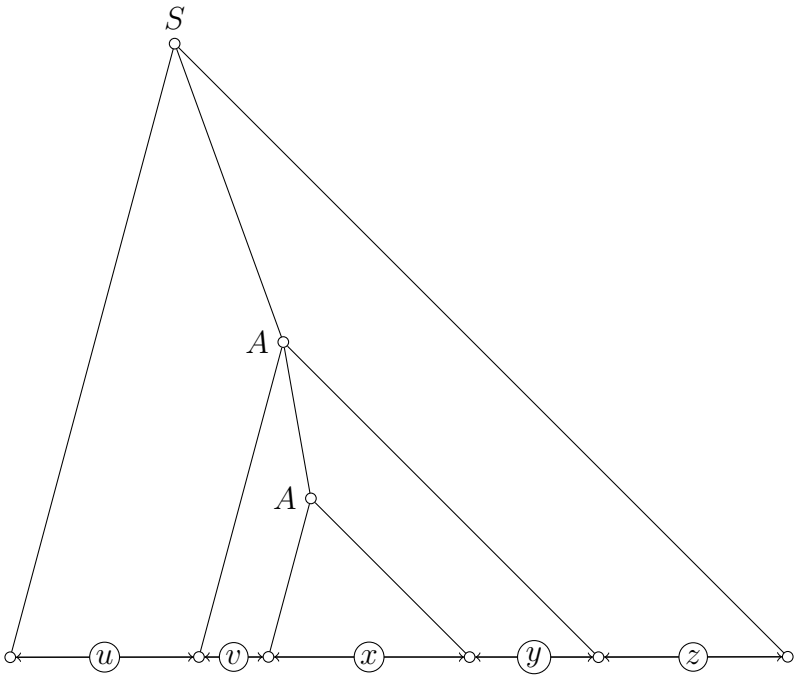
b)Concatenación: Queremos construir una gramática incontextual G tal que $L(G) = L(G_1)L(G_2)$. En este caso definimos $G = (V_1 \cup V_2 \cup \{S\}, \Sigma_1 \cup \Sigma_2, R, S)$ donde
$R = R_1 \cup R_2 \cup \{S \to S_1 S_2\}$.

c) Clausura: Queremos construir una gramática incontextual G tal que $L(G) = L(G_1)^*$. Definimos $G = (V_1 \cup \{S\}, \Sigma_1 \cup \Sigma_2, R, S)$ donde $R = R_1 \cup \{S \to \lambda, S \to SS_1\}$. Hemos añadido dos producciones, una para generar la palabra vacía y otra para permitir concatenar palabras generadas por G_1. □

Ahora demostraremos el teorema de bombeo para poder demostrar cuando un lenguaje no es incontextual.

Teorema 4.4. *Para toda gramática incontextual $G = (V, \Sigma, P, S)$, existe un natural $n \geq 1$ tal que , para toda palabra $w \in L(G)$, $|w| > n$, existe una representación $w = uvxyz$ tal que*
(1) $vy \neq \lambda$
(2) Para todo natural i, $uv^i xy^i z \in L(G)$

Demostración. Definimos $k = max\{|\alpha| : X \to \alpha \in P\}$ y tomamos h el número de variables de G. Entonces consideramos $n = k^h$. Ahora tomamos una palabra $w \in L(G)$ tal que $|w| > n$ y consideramos el árbol de derivación de w con el mínimo número de hojas de todos los posibles árboles que derivan w.
Obsérvese que si h es la altura del árbol y k es la longitud máxima de las partes derechas de las derivaciones de G, entonces dicho árbol tiene como máximo k^h hojas. Esto implica que el árbol que hemos considerado tiene como mínimo altura $h + 1$, ya que $|w| > k^h$, por tanto existe un camino de longitud $h + 1$, es decir, con $h + 2$ nodos. Solo el último nodo de este camino será un símbolo terminal, por tanto hay $h + 1$ variables en este camino, lo que implica que por lo menos una variable esta repetida. Llamemos A a la variable repetida y consideramos el árbol siguiente.



Llamamos por tanto u, v, x, y, z a las partes del árbol de derivación tal y como se muestra en el gráfico.
Claramente el subárbol que tiene por raíz la primera aparición de A sin el subárbol que tiene como raíz la segunda aparición de A puede repetirse todas las veces que se quiera, y esto generará otro árbol de derivación de G pudiendo escribir la palabra que se genera de la forma uv^nxy^nz, para todo $n \geq 0$. Finalmente $vy \neq \lambda$ ya que si $vy = \lambda$, no existiría una rama en el árbol con una variable repetida. □

Veamos aplicaciones de este teorema

Ejemplo: El lenguaje $L = \{a^kb^kc^k : k \geq 0\}$ no es incontextual. Supongamos que si lo fuera, entonces existe un $n \geq 1$ tal que toda palabra $w \in L$ de longitud $> n$ cumple las hipótesis del teorema. Entonces tomamos $w = a^nb^nc^n \in L$ que tiene una representación $w = uvxyz$ donde v o y no es la palabra vacía. Ahora tenemos dos opciones.
(1) vy contienen todos los símbolos (a,b,c). Esto implica que v o y contienen al menos dos símbolos distintos. Por tanto la palabra uv^2xy^2z contiene o bien una b delante de una a o una c delante de una a o una b. Por lo cual uv^2xy^2z no pertenece a L.
(2) vy no contienen todos los símbolos. Como $vy \neq \lambda$, uv^2xy^2z tendrá un número diferente de apariciones en alguno de los símbolos. Por tanto uv^2xy^2z no pertenece a L.
Lo que implica que L no es incontextual.

Ejemplo: El lenguaje $L = \{a^k : k \geq 1 \text{ es primo}\}$ no es incontextual. Nuevamente supongamos que lo fuera. Entonces existiría un $n \geq 1$ tal que toda palabra $w \in L$ de longitud $> n$ cumple las hipótesis del teorema. Tomamos $w = a^p$ que tiene una representación $w = uvxyz$. Suponemos que $vy = a^q$ y $uxz = a^r$, donde q y r son

naturales y $q > 0$. Entonces por el teorema de bombeo tenemos que $uv^ixy^iz \in L$ para todo natural i. Lo que implica que $r + iq$ es primo para todo $i \geq 0$. Esto no es posible como ya vimos al demostrar que este lenguaje no es regular. Por tanto L no es incontextual.

Teorema 4.5. *Los lenguajes incontextuales no son cerrados respecto a la intersección o complementación.*

Demostración. El lenguaje $L_1 = \{a^nb^nc^m : n, m \geq 0\}$ es incontextual. Es fácil verlo definiendo una gramática incontextual $G = \{\{S, R, a, b, c\}, \{a, b, c\}, P, \{S\}\}$ donde P consta de
1. $S \to aSbR$
2. $S \to \lambda$
3. $R \to cR$
4. $R \to \lambda$

Análogamente el lenguaje $L_2 = \{a^mb^nc^n : n, m \geq 0\}$ es también incontextual. La intersección de estos dos lenguajes es $L = \{a^nb^nc^n : n \geq 0\}$ que acabamos de demostrar que no es incontextual. Por tanto los lenguajes incontextuales no son cerrrados respecto a la intersección.
Por igualdad de conjuntos tenemos que

$$L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$$

por tanto los lenguajes incontextuales tampoco son cerrados respecto a la complementación, ya que si lo fueran serían cerrados respecto a la intersección. □

4.6. Aplicaciones al análisis sintáctico

La aplicación principal de los autómatas con pila es el diseño del analizador sintáctico de compiladores.
El trabajo del análizador sintáctico es determinar si el programa que recibe como entrada satisface las reglas del lenguaje de programación. Si es así, proporciona como representación del análisis el árbol de derivación de la entrada. Para ello vamos a ver el llamado análisis sintáctico descendente

Análisis sintáctico descendente
La idea es construir el árbol que proporciona el analizador de la raíz a las hojas.
En este caso el algoritmo es el siguiente
(1) Definir una gramática incontextual no ambigua que genere el lenguaje
(2) Transformar la gramática incontextual de (1) en una gramática incontextual cuyo autómata con pila asociado sea determinista.
(3) Programar el autómata con pila asociado a la gramática incontextual de (2)

Pese a que hemos visto que existen lenguajes inherentemente ambiguos, y por tanto no pueden generarse con una gramática incontextual no ambigua, no es el caso de los lenguajes de programción, por lo cual el punto (1) no conlleva problemas. Para realizar el punto (2) sin embargo necesitamos un nuevo concepto de gramática incontextual que definiremos a continuación.

Definiciones: Sea $G = (V, \Sigma, P, S)$ una gramática incontextual
(a) Para todo $\alpha \in V$ definimos

$$Primeros(\alpha) = \begin{cases} \{a \in \Sigma : \alpha \Rightarrow_G^* ap \text{ donde } p \in V^*\} \text{ si } \alpha \not\Rightarrow_G^* \lambda \\ \{a \in \Sigma : \alpha \Rightarrow_G^* ap \text{ donde } p \in V^*\} \cup \{\lambda\} \text{ si } \alpha \Rightarrow_G^* \lambda \end{cases}$$

Obsérvese que si α empieza por un símbolo terminal entonces $Primeros(\alpha)$ es ese símbolo terminal.

(b) Para todo $A \in V \backslash \Sigma$ definimos

$$Siguientes(A) = \{a \in \Sigma : S \Rightarrow_G^* \alpha A a \beta \text{ donde } \alpha, \beta \in V^*\}$$

(c) Para todo $A \in V \backslash \Sigma$ y $a \in \Sigma$ definimos:

$$Tabla[A, a] = \{A \Rightarrow \beta \in P : a \in Primeros(\beta \cdot Siguientes(A))\}$$

(d) G es $LL(1)$, si para todo $A \in V \backslash \Sigma$, $a \in \Sigma$

$$Tabla[A, a] \text{ tiene a lo sumo un elemento}$$

Las siguientes reglas de eliminación del indeterminismo nos permiten transformar en muchos casos una gramática no ambigua en una gramática $LL(1)$

Consideramos $G = (V, \Sigma, P, S)$ una gramática no ambigua.

Regla de factorización Si $A \to \alpha\beta_1|...|\alpha\beta_n$ donde $\alpha \neq \lambda$ y $n \geq 2$, reemplazamos estas producciones por $A \to \alpha A'$, $A' \to \beta_1|...|\beta_n$, donde A' es una nueva variable.
Es trivial ver que esta regla no varía el lenguaje generado por la gramática.

Regla de recursión Si $A \to A\alpha_1|...|A\alpha_n|\beta_1|...|\beta_m$ donde los β_i no empiezan por A y $n > 0$. Entonces sustituimos estas producciones por $A \to \beta_1 A'|...|\beta_m A'$ y $A' \to \alpha_1 A'|...|\alpha_n A'|\lambda$, donde A' es una nueva variable.
Nuevamenta el lenguaje generado por la gramática no varía ya que en ambos casos, los bloques de producciones generan el lenguaje $(\beta_1 \cup ... \cup \beta_m)(\alpha_1 \cup ...\alpha_n)^*$.

Ahora vamos a ver como podemos programar los autómatas deterministas, para poder diseñar el analizador sintáctico.

Recordemos, por el Teorema 4.2, que el autómata con pila asociado a una gramática incontextual tiene transiciones de 3 tipos.
(1) $((q_0, \lambda, \lambda), (q_1, S))$
(2) $((q_1, \lambda, A), (q_1, x))$ para cualquier producción $A \to x$ en P.
(3) $((q_1, a, a), (q_1, \lambda))$ para todo $a \in \Sigma$.
De éstas, solo las del tipo (2) pueden ser compatibles. En el caso de que esto sucediera, cuando leemos el carácter a, y una variable A es el elemento de la cima de la pila, entonces aplicamos la transición correspondiente a la producción de $Tabla[A, a]$ (que es única ya que la gramática es LL(1)).

Algoritmo para programar un autómata con pila asociado a una gramática incontextual LL(1)

(1) Representamos los símbolos de V por carácteres. Definimos una pila y empezamos poniendo un \$ en la pila. También definimos una variable booleana b inicializada en 0, que nos servirá para saber si no se puede realizar ninguna transición.
(2) Realizamos la transición del tipo (1), que pone S en la pila.
(3) Realizamos un bucle "while"mientras la pila o la entrada no estén vacías y b sea 0, es decir mientras podamos realizar alguna transición. Dentro del while aplicaremos la transción correspondiente.

(3,1) Primero comprobamos si el carácter de entrada coincide con el de la cima de la pila, y entonces aplicamos la transición del tipo (3) correspondiente y leemos el siguiente carácter.

(3,2) En caso de que haya una variable en la cima de la pila realizamos una instrucción "switch-case", donde cada caso es una variable de la gramática incontextual. Suponiendo que A es la variable de la pila y a el carácter de entrada, comprobamos si $Tabla[A, a]$ existe, y en tal caso realizamos la transición correspondiente a la producción de $Tabla[A, a]$.

(3,3) Si no se ha podido realizar ninguna transición ponemos $b = 1$ para salir del bucle.

(4) Al salir del bucle comprobamos si la pila esta vacía (hay solo el \$) y hemos leído toda la entrada. En caso afirmativo retornamos 1 y si no retornamos 0.

Para entender el concepto de los *Primeros* y los *Siguientes* debemos recordar la idea del autómata con pila asociado a una gramática incontextual. Lo que hacíamos para construir el autómata era añadir transiciones que sustituyeran las variables de la pila de la misma forma que lo harían las producciones de la gramática, y cada vez que el carácter de entrada coincidía con el carácter de la cima de la pila, lo leíamos y eliminábamos el carácter de la cima de la pila. Por eso siempre que tenemos una variable en la cima de la pila, debemos realizar transiciones tales que el símbolo que quede en la cima de la pila sea el mismo que el carácter de entrada, siempre que sea posible. Para ello solo hay dos opciones.
(1) Que el símbolo del carácter de entrada se encuentre en *Primeros* de la variable de la cima de la pila.
(2) Que el símbolo del carácter de entrada se encuentre en *Siguientes* de la variable

de la cima de la pila, y la variable pueda ser anulada (mediante una producción que la sustituye por λ).
La $Tabla$ nos indica que transición debemos realizar, en el caso que sea posible, para poder sustituir la variable de la cima de la pila por el símbolo terminal deseado.

Por tanto una gramática es $LL(1)$ si podemos realizar las derivaciones por la izquierda que acabamos de comentar únivocamente.

Vamos a ver un ejemplo de un analizador sintáctico.

Ejemplo: Queremos diseñar un programa que reconozca expresiones aritméticas. Para simplificar, supondremos que en las operaciones aritméticas solo aparecen variables (id) y las operaciones $+, *$.

El primer paso es definir una gramática no ambigua que genere este lenguaje. Definimos pues $G = (\{E, F, T, id, +, *, (,)\}, \{id, +, *, (,)\}, P, \{E\})$ donde P consta de

1. $E \to E + T$
2. $E \to T$
3. $T \to T * F$
4. $T \to F$
5. $F \to id$
6. $F \to id(E)$
7. $F \to (E)$

Se puede comprobar que esta gramática no es ambigua, ya que la variable E genera una suma de términos de manera únivoca, la variable T genera un producto de factores de manera únivoca, y la variable F se sustituye por átomos $(id, id(E), (E))$. Ahora podemos observar que

$$Primeros(E) = Primeros(T) = Primeros(F) = \{(, id\}$$

y que

$$Siguientes(E) = \{+,)\}, Siguientes(T) = \{+, *,)\}, Siguientes(F) = \{+, *,)\lambda\}$$

Para obtener los $Primeros$ y los $Siguientes$ debemos aplicar todas las transiciones posibles y ver que símbolos terminales cumplen la definición. Veamos como sería el caso de la variable E. Escribiremos $\overset{i}{\Rightarrow}$ para denotar que estamos aplicando la producción i

Para obtener los $Primeros$ podemos realizar las derivaciones $E \overset{2}{\Rightarrow} T \overset{4}{\Rightarrow} F \overset{5}{\Rightarrow} id$ obteniendo el símbolo id. También podemos realizar la cadena de derivaciones $E \overset{2}{\Rightarrow} T \overset{4}{\Rightarrow} F \overset{7}{\Rightarrow} (,$ por tanto (también pertenece a $Primeros(E)$. Realizando cualquier otra derivación no obtendremos ningún símbolo nuevo.

En cuanto a los $Siguientes$ observemos que al aplicar la regla 1 la variable E

aparece seguida del símbolo +, y al aplicar las producciones 6 o 7 la variable E aparece seguida del símbolo). Al ser estas las únicas reglas en las que aparece la variable E, deducimos que $Siguientes(E) = \{+,)\}$

Análogamente deducimos los $Primeros$ y $Siguientes$ de T y F.

Por tanto ahora podemos construir la $Tabla$ aplicando la defínción para cada producción.

1. $Primeros(E + T \cdot Siguientes(E)) = Primeros(E) = \{id, (\}$
2. $Primeros(T \cdot Siguientes(E)) = Primeros(T) = \{id, (\}$
3. $Primeros(T * F \cdot Siguientes(T)) = Primeros(T) = \{id, (\}$
4. $Primeros(F \cdot Siguientes(T)) = Primeros(F) = \{id, (\}$
5. $Primeros(id \cdot Siguientes(F)) = \{id\}$
6. $Primeros(id(E) \cdot Siguientes(F)) = \{id\}$
7. $Primeros((E) \cdot Siguientes(E)) = \{(\}$

Obteniendo la siguiente $Tabla$

$Tabla$	$+$	$*$	id	$($	$)$
E	$-$	$-$	$1, 2$	$1, 2$	$-$
T	$-$	$-$	$3, 4$	$3, 4$	$-$
F	$-$	$-$	$5, 6$	7	$-$

Es decir que G no es $LL(1)$ ya que hay conflictos en la $Tabla$.

Procedemos entonces con el segundo paso del diseño del analizador sintáctico. Debemos sustituir las producciones según las reglas descritas anteriormente.

1. Aplicando la regla de factorización, sustituimos $F \to id|id(E)$ por $F \to F'$ y $F' \to \lambda|(E)$, añadiendo la variable F' a la gramática.

2. Aplicando la regla de recursión, transformamos $E \to E + T|T$ en $E \to TE'$ y $E' \to +TE'|\lambda$, donde E' es una nueva variable.

3. Aplicando de nuevo la regla de recursión reemplazamos $T \to T*F|F$ por $T \to FT'$ y $T' \to *FT'|\lambda$, añadiendo la variable T' a la gramática.

Por tanto ahora la nueva gramática es
$G = (\{E, F, T, E', F', T', id, +, *, (,)\}, \{id, +, *, (,)\}, P, \{E\})$ donde P consta de

1. $E \to TE'$
2. $E' \to +TE'$
3. $E' \to \lambda$
4. $T \to FT'$
5. $T' \to *FT'$
6. $T' \to \lambda$

7. $F \to (E)$
8. $F \to idF'$
9. $F' \to (E)$
10. $F' \to \lambda$

Ahora podemos ver que

$$Primeros(E) = Primeros(T) = Primeros(F) = \{id, (\},$$

$$Primeros(E') = \{+, \lambda\}, Primeros(T') = \{*, \lambda\}, Primeros(F') = \{(, \lambda\}$$

y que

$$Siguientes(E) = Siguientes(E') = \{)\}$$

$$Siguientes(T) = Siguientes(T') = \{), +\}$$

$$Siguientes(F) = Siguientes(F') = \{), +, *\}$$

Por tanto podemos construir la $Tabla$ aplicando la fórmula vista anteriormente

1. $Primeros(TE' \cdot Siguientes(E)) = Primeros(T) = \{id, (\}$
2. $Primeros(+TE' \cdot Siguientes(E')) = \{+\}$
3. $Primeros(Siguientes(E')) = Siguientes(E') = \{)\}$
4. $Primeros(FT' \cdot Siguientes(T)) = Primeros(F) = \{id, (\}$
5. $Primeros(*FT' \cdot Siguientes(T')) = \{*\}$
6. $Primeros(Siguientes(T')) = Siguientes(T') = \{), +\}$
7. $Primeros((E) \cdot Siguientes(F)) = \{(\}$
8. $Primeros(idF' \cdot Siguientes(F)) = \{id\}$
9. $Primeros((E) \cdot Siguientes(F')) = \{(\}$
10. $Primeros(Siguientes(F')) = Siguientes(F') = \{), +, *\}$

Tenemos entonces la siguiente $Tabla$ de análisis.

$Tabla$	$+$	$*$	id	$($	$)$
E	$-$	$-$	1	1	$-$
E'	2	$-$	$-$	$-$	3
T	$-$	$-$	4	4	$-$
T'	6	5	$-$	$-$	6
F	$-$	$-$	8	7	$-$
F'	10	10	$-$	9	10

Con lo cual G es LL(1) ya que no hay conflictos en la $Tabla$.

Por tanto ya podemos programar el autómata con pila asociado a esta gramática. Ahora para realizar código en C utilizamos i, X, Y, Z en vez de id, E', T', F' para así usar solamente un carácter. El código sería el siguiente

```
int expresiones_aritmeticas (char entrada, int n){
```

```
//n es el número máximo de elementos de la pila
   char pila[n];
   int puntero = 0, i = 0, b = 0;
   char c = entrada[0];
   pila[puntero] = '$';
   pila[++puntero] = S;
   while((c != '$' || pila[puntero] != '$') && (b ==0){
        if(c == pila[puntero]){
           pila[puntero] = ' ';
           puntero --;
           c = entrada[++i];
        }
        else{
           switch(pila[puntero]){
                case 'E':
                                if (c == 'i' || c  == '('){
                                        pila[puntero] = 'X';
                                        pila[++puntero] = 'T';
                                }
                                else b = 1;
                                break;
                case 'X':
                                if (c == '+'){
                                        pila[puntero] = 'X';
                                        pila[++puntero] = 'T';
                                        pila[++puntero] = '+';
                                }
                                else if (c == ')'){
                                        pila[puntero] = ' ';
                                        puntero --;
                                else b = 1;
                                break;
                case 'T':
                                if (c == 'i' || c == '('){
                                        pila[puntero] = 'Y';
                                        pila[++puntero] = 'F';
                                }
                                else b = 1;
                                break;
                case 'Y':
                                if (c == '*'){
                                        pila[puntero] = 'Y';
                                        pila[++puntero] = 'T';
                                        pila[++puntero] = '*';
                                }
                                else if (c == '+' || c == ')'){
```

```
                                        pila[puntero] = ' ';
                                        puntero --;
                                    else b = 1;
                                    break;
                        case 'F':
                                    if (c == 'id'){
                                        pila[puntero] = 'Z';
                                        pila[++puntero] = 'i';
                                    }
                                    else if (c == '('){
                                        pila[puntero] = ')';
                                        pila[++puntero] = 'E';
                                        pila[++puntero] = '(';
                                    else b = 1;
                                    break;
                        case 'Z':
                                    if (c == '+' || c == '*' || c == ')'){
                                        pila[puntero] = ' ';
                                        puntero --;
                                    }
                                    else if (c == '('){
                                        pila[puntero] = ')';
                                        pila[++puntero] = 'E';
                                        pila[++puntero] = '(';
                                    else b = 1;
                                    break;
                        default: b = 1;
                    }
                }
            }
            if (pila == '$' && c == '$')
                return 1;
            return 0;
        }
```

5. Conclusiones

En este trabajo hemos visto una introducción a la teoría de autómatas. Hemos trabajado los reconocedores de lenguajes (autómatas finitos y autómatas con pila) y los generadores (expresiones regulares y gramáticas incontextuales).

En el Tema de autómatas finitos remarcamos como teoremas más importantes la equivalencia entre autómatas finitos deterministas y autómatas finitos indeterministas y la equivalencia entre lenguajes regulares y autómatas finitos. El primer resultado nos permite plantear los problemas de una forma más intuitiva (usando un autómata indeterminista) para después transformar esa solución permitiendo que ésta sea programable (autómata determinista). El segundo resultado nos determina cuales son los lenguajes que podemos reconocer con autómatas finitos. Como principal aplicación destacamos el diseño de un analizador léxico (la primera fase del diseño de un compilador).

En el Tema de lenguajes incontextuales hemos visto la equivalencia entre gramáticas incontextuales y autómatas con pila. Esto nos permite generar lenguajes mediante una gramática incontextual y luego reconocer ese lenguaje mediante al autómata con pila correspondiente, como hemos visto en el caso del analizador sintáctico.

Finalmente comentar, que una continuación de este trabajo podría ser profundizar sobre las "máquinas de Turing", consideradas como los primeros ordenadores. Para desarollar este nuevo concepto, es fundamental la teoría de autómatas descrita en este trabajo.

Referencias

[1] Lewis, H.R.; Papadimitrou, C,H.: Elements of the theory of computation. Prentice Hall, 1998.

[2] Hopcroft, J.E.; Motwani, R.; Ullman, J.D.: Introduction to automata theory, languages and computation. Addison Weasley, 2001.

[3] Kozen, D.C.: Automata and computability. Springer, 1997.

[4] Louden, K.C.: Construcción de compiladores. Thomson, 2004.

[5] Cases, R.; Màrquez, L; Llenguatjes, gramàtiques i autòmats. Edicions UPC, 2000.