LENGUAJES REGULARES

Abril de 2023

Introducción

En la clase de hoy, empezamos la parte de curso sobre teoría de autómatas.

Los autómatas son modelos computacionales que se utilizan en el diseño de software para sistemas informáticos.

Entre sus principales aplicaciones, podemos mencionar las siguientes:

Descripción básica

(1) Diseño de analizadores léxicos y procesadores de textos.
(2) Diseño de programas para buscar palabras o frases en textos amplios.
(3) Diseño de software para crear protocolos de comunicación o protocolos de intercambio de información.
(4) Diseño de compiladores.

Descripción básica

Los autómatas son reconocedores de lenguajes.

Dicho de manera intuitiva, un autómata M para un lenguaje L es un modelo computacional que recibe como entrada una tira de símbolos x y entonces determina si x pertenece o no al lenguaje L.

Hay diversos tipos de autómatas que se utilizan en el diseño de software de sistemas. En esta parte del curso, estudiaremos los llamados autómatas finitos y autómatas con pila, que son los tipos de autómatas que se utilizan en el diseño de compiladores.

Antes de definir el concepto de autómata, necesitamos introducir algunos conceptos básicos sobre lenguajes formales, que veremos en la clase de hoy.

Alfabetos y palabras

Un alfabeto es un conjunto no vacío de símbolos.

Una palabra sobre un alfabeto Σ es una secuencia finita de símbolos de Σ.

Denotamos por λ a la palabra vacía, es decir, a la palabra que no tiene ningún símbolo.

La longitud de una palabra x es el número de símbolos que la componen, contando repeticiones. Denotamos a la longitud de una palabra x por $|x|$.

Por ejemplo, consideremos las palabras $x = 1024$ y $z = 1509450753$ sobre el alfabeto $\{0, 1, \ldots, 9\}$. Entonces, $|x| = 4$ y $|z| = 10$.

Alfabetos y palabras

Si x es una palabra sobre un alfabeto Σ y a es un símbolo de Σ, denotamos por $n_a(x)$ al número de apariciones del símbolo a en x.

Por ejemplo, si consideramos la palabra $z = 1509450753$ sobre el alfabeto $\{0, 1, \ldots, 9\}$, tenemos que
$n_0(z) = 2, n_1(z) = 1, n_2(z) = 0, n_3(z) = 1, n_4(z) = 1, n_5(z) = 3,$
$n_6(z) = 0, n_7(z) = 1, n_8(z) = 0, n_9(z) = 1.$

Si Σ es un alfabeto, denotamos por Σ^* al conjunto de todas las palabras sobre Σ.

El concepto de lenguaje

Un lenguaje es un conjunto de palabras sobre un alfabeto.

Ejemplos de lenguajes:

(1) El conjunto $\{x \in \{a, b, c\}^* : n_a(x) = n_b(x) = n_c(x)\}$.

(2) El conjunto de identificadores de un lenguaje de programación.

En el caso del lenguaje Java, el lenguaje de los identificadores está definido en el alfabeto
$\Sigma = \{0, \ldots, 9\} \cup \{a, b, \ldots, z\} \cup \{A, B, \ldots, Z\} \cup \{\_, \$\}$.

(3) El conjunto de programas de un lenguaje de programación.

Este lenguaje está definido en el alfabeto
$\Sigma = \{0, \ldots, 9\} \cup \{a, b, \ldots, z\} \cup \{A, B, \ldots, Z\} \cup \{$ símbolos de puntuación del lenguaje de programación $\} \cup \{$ palabras reservadas del lenguaje de programación $\}$.

Operaciones básicas para palabras

La operación más importante es la concatenación, que consiste en la yuxtaposición de una palabra x con una palabra y, que se representa por $x \cdot y$, o más abreviadamente por xy. Por tanto, la palabra $x \cdot y$ consiste en poner los símbolos de x y a continuación los símbolos de y.

Por ejemplo, si $x = 100$ e $y = 01$, tenemos que $x \cdot y = xy = 10001$ e $y \cdot x = yx = 01100$.

Utilizaremos la notación exponencial para representar la concatenación repetida de una palabra con ella misma. Si x es una palabra, definimos

$$x^0 = \lambda,$$
$$x^{n+1} = x^n \cdot x.$$

Por ejemplo, si $x = 011$, tenemos que $x^0 = \lambda$, $x^1 = x = 011$, $x^2 = 011011$, $x^3 = 011011011$, ...

Operaciones básicas para palabras

Otra operación importante es la inversa de una palabra x. Si $x = a_1 \dots a_n$, definimos la inversa de x por la palabra $x^I = a_n \dots a_1$.

Recordemos que λ representa la palabra vacía, es decir, la palabra que no tiene ningún símbolo. Tenemos entonces:

(1) Para toda palabra x, se tiene que $x \cdot \lambda = \lambda \cdot x = x$.

(2) Para cualesquiera palabras x, y, se tiene que $(x \cdot y)^I = y^I \cdot x^I$.

Operaciones básicas para lenguajes

(1) La unión.

Si L_1 y L_2 son dos lenguajes sobre un alfabeto Σ, entonces $L_1 \cup L_2 = \{x \in \Sigma^* : x \in L_1 \text{ o } x \in L_2\}$.

(2) La intersección.

Si L_1 y L_2 son dos lenguajes sobre un alfabeto Σ, entonces $L_1 \cap L_2 = \{x \in \Sigma^* : x \in L_1 \text{ y } x \in L_2\}$.

(3) La complementación.

Si L es un lenguaje sobre un alfabeto Σ, definimos el complementario de L por $\overline{L} = \{x \in \Sigma^* : x \notin L\}$.

(4) La diferencia.

Si L_1 y L_2 son dos lenguajes, definimos el lenguaje $L_1 \setminus L_2 = \{x \in L_1 : x \notin L_2\}$.

Operaciones básicas para lenguajes

(5) La concatenación.

Si L_1 y L_2 son dos lenguajes, definimos la concatenación de L_1 con L_2 por $L_1 \cdot L_2 = L_1 L_2 = \{xy : x \in L_1, y \in L_2\}$.

Por ejemplo, si $L_1 = \{0, 01, 100\}$ y $L_2 = \{01, 10\}$, tenemos que $L_1 L_2 = \{001, 010, 0101, 0110, 10001, 10010\}$ y $L_2 L_1 = \{010, 0101, 01100, 100, 1001, 10100\}$.

Utilizaremos la notación exponencial para representar la concatenación repetida de un lenguaje consigo mismo. Si L es un lenguaje, definimos:

$$L^0 = \{\lambda\},$$
$$L^{n+1} = L^n \cdot L.$$

Por ejemplo, si $L = \{1, 00\}$, tenemos que $L^0 = \{\lambda\}$, $L^1 = L = \{1, 00\}$, $L^2 = L \cdot L = \{11, 100, 001, 0000\}$, $L^3 = L^2 \cdot L = \{111, 1100, 1001, 10000, 0011, 00100, 00001, 000000\}$, ...

Operaciones básicas para lenguajes

(6) La clausura de un lenguaje.

Si L es un lenguaje, definimos la clausura de L por $L^* = \{x_1 x_2 \dots x_n : n \geq 0, x_1, x_2, \dots, x_n \in L\}$. Es decir, L^* es el lenguaje formado por todas las posibles concatenaciones de palabras de L.

Se tiene que $L^* = \bigcup \{L^n : n \geq 0\}$.

Por ejemplo, consideremos $\Sigma = \{0, 1\}$. Si tomamos $L = \{0\}$, tenemos que $L^* = \{0^n : n \geq 0\}$.

Y si tomamos $L = \{00, 1\}$, entonces $L^* = \{x \in \{0, 1\}^* : \text{en } x$ todos los ceros aparecen en pares adyacentes$\}$.

Lenguajes regulares

Hay muchos lenguajes que admiten descripciones que utilizan únicamente los símbolos de un alfabeto y las operaciones básicas entre lenguajes. Son los llamados lenguajes regulares, los cuales se describen por las llamadas expresiones regulares, que definimos a continuación.

Definición de expresión regular

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 y β son expresiones regulares, también lo son $(\alpha \cup \beta)$ y $(\alpha \cdot \beta)$.

(4) Si α es una expresión regular, también lo es α^*.

Lenguaje asociado a una expresión regular

Si α es una expresión regular, definimos el lenguaje $L(\alpha)$ asociado a α por las siguientes reglas:

(1) Definimos $L(\emptyset) = \emptyset$ y $L(\lambda) = \{\lambda\}$.

(2) Si $a \in \Sigma$, definimos $L(a) = \{a\}$.

(3) Definimos $L(\alpha \cup \beta) = L(\alpha) \cup L(\beta)$.

(4) Definimos $L(\alpha \cdot \beta) = L(\alpha) \cdot L(\beta)$.

(5) Definimos $L(\alpha^*) = L(\alpha)^*$.

Se dice entonces que un lenguaje L es regular, si hay una expresión regular α tal que $L = L(\alpha)$.

En ocasiones, para simplificar la notación, escribiremos α en lugar de $L(\alpha)$.

Ejemplos

(1) Si $\alpha = (0 \cup 1)^*$, $L(\alpha) = \{0,1\}^*$, es decir, es el lenguaje de todas las palabras de bits, ya que aplicando la definición anterior tenemos:
$L((0 \cup 1)^*) = (L(0 \cup 1))^* = (L(0) \cup L(1))^* = (\{0\} \cup \{1\})^* = \{0,1\}^*$.

(2) Si $\alpha = (0 \cup 1)^*1$, $L(\alpha) = \{x \in \{0,1\}^* : x$ acaba en $1\}$.

(3) Si $\alpha = ((0 \cup 1) \cdot (0 \cup 1) \cdot (0 \cup 1))^*$, entonces $L(\alpha)$ es el conjunto de las palabras de bits cuya longitud es un múltiplo de 3.

(4) Si $\alpha = (1 \cup 2 \cup 0(1 \cup 2)^*0)^*$, entonces
$L(\alpha) = \{x \in \{0,1,2\}^* : x$ tiene un número par de ceros $\}$.

Ejemplos

(5) Es bien sabido que todos los tipos de datos de los lenguajes de programación se pueden representar mediante expresiones regulares. Por ejemplo, si $\alpha = (+ \cup - \cup \lambda) \cdot (0 \cup 1 \cup \cdots \cup 9) \cdot (0 \cup 1 \cup \cdots \cup 9)^*$, entonces $L(\alpha)$ es el tipo entero.

Expresiones regulares equivalentes

Dos expresiones regulares α, β son equivalentes, si $L(\alpha) = L(\beta)$. Escribiremos entonces $\alpha \equiv \beta$.

Veamos algunos ejemplos:

(1) Las expresiones regulares $\alpha = 0^*1^*$ y $\beta = (01)^*$ no son equivalentes, ya que por ejemplo la palabra $0101 \in L(\beta)$ pero $0101 \notin L(\alpha)$, ya que las palabras de $L(\alpha)$ están formadas por un bloque de ceros seguido de un bloque de unos, por lo que es imposible que en una palabra de $L(\alpha)$ aparezca un 1 antes que un 0.

Expresiones regulares equivalentes

(2) Veamos ahora un ejemplo de dos expresiones regulares distintas que son equivalentes. Consideremos $\alpha = (0 \cup 1)^*$ y $\beta = (0^*1)^*0^*$. Como antes hemos visto, tenemos que $L((0 \cup 1)^*) = \{0,1\}^*$, es decir, es el lenguaje de todas las palabras de bits. Por tanto, $L(\beta) \subseteq L(\alpha)$. Demostramos ahora que $L(\alpha) \subseteq L(\beta)$. Para ello, observamos que toda palabra x de $\{0,1\}^*$ se puede representar agrupando cada 1 que aparece en la palabra con los ceros que le preceden; cada uno de estos grupos es una palabra de 0^*1. Por tanto, todos los grupos juntos forman una palabra $(0^*1)^*$. Además, hay que añadir los ceros que eventualmente pueda haber al final de la palabra x y que no vayan seguidos de ningún uno, los cuales pertenecen a 0^*. Por tanto, $x \in L(\beta)$.

Autómatas finitos

Los lenguajes regulares se pueden representar mediante los llamados autómatas finitos, que son los autómatas más simples.

Un autómata finito tiene asociada una cinta de entrada, la cual es una cinta de lectura que está dividida en celdas. La información de la cinta se encuentra entonces almacenada en las celdas, y el autómata accede a dicha información mediante un puntero que puede leer en un instante dado el contenido de una celda de la cinta. Al producirse un paso de cómputo, el puntero se mueve a la siguiente celda a la derecha en la cinta de entrada.

El autómata tiene además asociada una "unidad de control", que en un paso de cómputo se encuentra en un cierto estado. Al pasar entonces al siguiente paso de cómputo, el estado puede variar.

Definición de autómata determinista

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

(1) K es un conjunto finito y no vacío de estados.

(2) Σ es un alfabeto finito, el alfabeto de entrada.

(3) δ es una función de $K \times \Sigma$ en K, a la que se denomina función de transición.

(4) $q_0 \in K$ es el estado inicial.

(5) $F \subseteq K$ es el conjunto de estados aceptadores (o estados finales).

Observaciones

Llamaremos símbolo actual al carácter accesible a través del puntero en un instante dado.

Inicialmente, el autómata se encuentra en el estado inicial q_0 con una entrada $x \in \Sigma^*$ escrita al comienzo de la cinta (de manera que el puntero señala a la primera celda).

Los cómputos del autómata se realizan por medio de la función δ. Es decir, para pasar de un paso de cómputo al siguiente se aplica la función δ. El argumento de δ es el par (q, a) donde q es el estado actual del autómata y a es el símbolo actual. La imagen de la función δ es un estado p, que corresponde al estado del autómata en el siguiente paso de cómputo.

Representación de un autómata mediante un grafo

Consideremos un autómata determinista $M = (K, \Sigma, \delta, q_0, F)$. Para representar M mediante un grafo, seguimos el siguiente algoritmo:

(1) Los nodos del grafo son los estados del autómata.

(2) Se marca el estado inicial con una flecha, y se marca cada estado aceptador con un doble círculo o con una cruz.

(3) Si $\delta(q, a) = p$, se dibuja un arco en el grafo de q a p con etiqueta a.

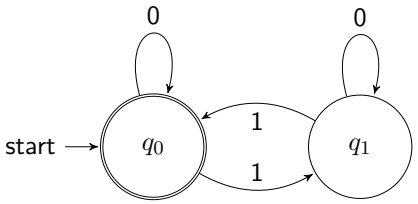
Ejemplo

Consideremos el autómata determinista $M = (K, \Sigma, \delta, q_0, F)$ donde $K = \{q_0, q_1\}$, $\Sigma = \{0, 1\}$, $F = \{q_0\}$ y δ está definida por la siguiente tabla:

q	σ	$\delta(q, \sigma)$
q_0	0	q_0
q_0	1	q_1
q_1	0	q_1
q_1	1	q_0

Grafo del autómata

El autómata anterior se puede representar mediante el siguiente grafo:



Nociones básicas para autómatas deterministas

Si $M = (K, \Sigma, \delta, q_0, F)$ es un autómata determinista, definimos una configuración de M como una palabra $px \in K\Sigma^*$.

Si en un paso de cómputo la configuración de M es px, esto significa que el autómata M se encuentra en el estado p y x es la parte de la palabra de entrada que todavía no se ha leído.

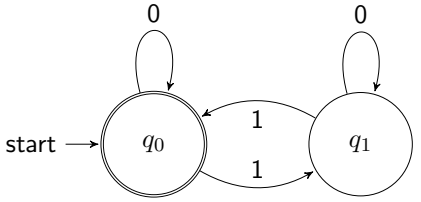
Nociones básicas para autómatas deterministas

Si px y qy son configuraciones de M, decimos que px produce qy en un paso de cómputo, lo que representamos por $px \vdash_M qy$, si en M podemos pasar de px a qy aplicando una vez la función de transición.

Y decimos que px produce qy, lo que representamos por $px \vdash_M^* qy$, si en M podemos pasar de px a qy aplicando un número finito de veces la función de transición.

Ejemplo

Consideremos el autómata M del ejemplo anterior:



Tenemos entonces el siguiente cómputo para la entrada $x = 00110$:

$q_000110 \vdash_M q_00110 \vdash_M q_0110 \vdash_M q_1110 \vdash_M q_010 \vdash_M q_0$.

Lenguaje asociado a un autómata determinista

Sea $M = (K, \Sigma, \delta, q_0, F)$ un autómata determinista.

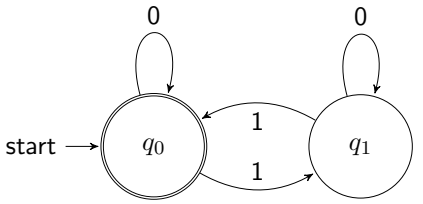
(a) Una palabra $x \in \Sigma^*$ es reconocida (o aceptada) por M, si existe un estado $q \in F$ tal que $q_0 x \vdash_M^* q$.

(b) Definimos el lenguaje reconocido por M por

$$L(M) = \{x \in \Sigma^* : x \text{ es reconocida por } M\}.$$

Ejemplo 1

Consideremos el autómata M visto anteriormente:

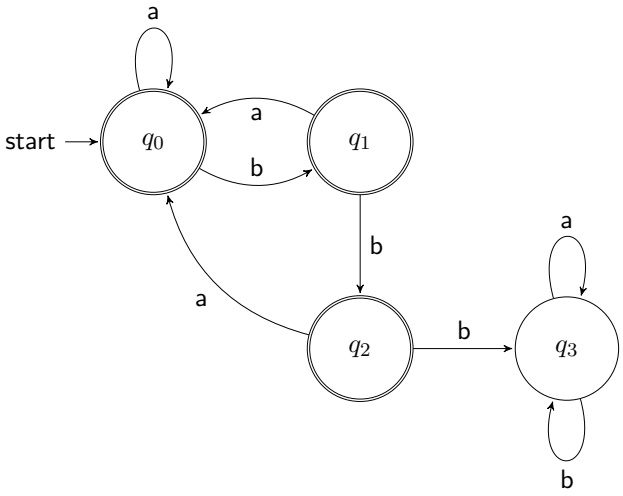


Observamos que si en un paso de cómputo del autómata estamos en el estado q_0, entonces el número de unos leídos en la entrada es par; y si estamos en el estado q_1, entonces el número de unos leídos en la entrada es impar. Así pues, como q_0 es el estado aceptador, tenemos que

$L(M) = \{x \in \{0,1\}^* : n_1(x) \text{ es par}\}$.

Ejemplo 2

M':



Ejemplo 2

Observamos que si entran tres b's seguidas, el autómata va al estado q_3, el cual no es aceptador y del cual ya no saldremos, ya que tanto si entra una a como una b seguiremos en q_3. Y si no entran tres b's seguidas, el autómata terminará el cómputo o bien en el estado q_0, o en el q_1, o en el q_2, los cuales son estados aceptadores. Por tanto,

$L(M') = \{x \in \{a,b\}^* : \text{en } x \text{ no aparecen tres b's consecutivas}\}$.

Equivalencia entre expresiones regulares y autómatas deterministas

La equivalencia viene dada por el siguiente teorema.

Teorema

Para todo lenguaje L, existe una expresión regular α tal que $L = L(\alpha)$ si y sólo si existe un autómata determinista M tal que $L(M) = L$.

Este teorema expresa entonces la equivalencia entre el concepto de expresión regular y el concepto de autómata determinista.

Programación de autómatas deterministas

Los autómatas deterministas son especificaciones de programas. Para escribir un programa en Java o en C correspondiente a un autómata determinista, podemos proceder de la siguiente manera:

(1) Representamos a los estados del autómata por números naturales, representando por el 0 al estado inicial.

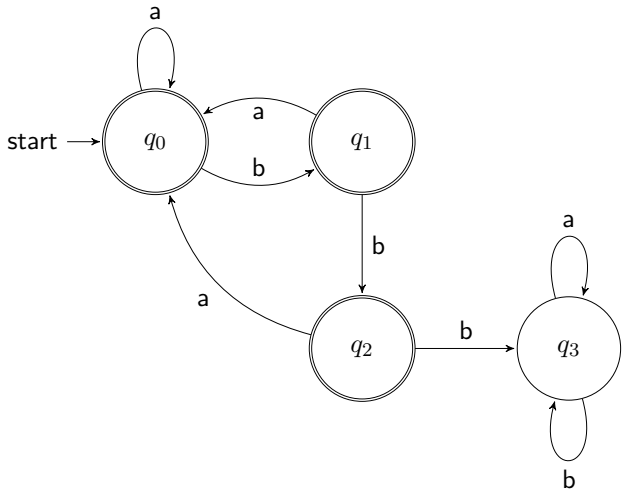
(2) Representamos a los símbolos del alfabeto de entrada por caracteres. Y representamos por el carácter \$ el final de la palabra de entrada.

(3) Representamos el cálculo del autómata mediante un bucle "while", en el que describimos mediante una instrucción switch-case las transiciones del autómata en las que cambia el estado.

(4) Al salir del bucle "while", comprobamos si el estado en el que estamos es un estado aceptador.

Ejemplo

Consideremos el autómata determinista M', que vimos en la última clase:



Ejemplo

Representamos a cada estado q_i por i. Podemos escribir entonces el siguiente método en Java para simular el autómata M':

```
public boolean simular (String entrada)
{ int q = 0, i = 0;
  char c = entrada.charAt(0);
  while (c != '$')
  { switch(q)
        { case 0:
        if (c == 'b') q = 1;
        break;
        case 1:
        if (c == 'a') q = 0; else if (c == 'b') q = 2;
        break;
        case 2:
        if (c == 'a') q = 0; else return false;
        break;}
  c = entrada.charAt(++i); } return true; }
```

Descripción básica

En la primera parte de la clase de hoy, mostraremos un método para simplificar los autómatas deterministas. Este método es importante, porque los autómatas deterministas representan programas, y los programas asociados a autómatas pequeños son más eficientes que los programas asociados a los autómatas grandes.

A continuación, definiremos los autómatas indeterministas, que son modelos computacionales equivalentes a los autómatas deterministas, pero más fáciles de deseñar.

Finalmente, introduciremos un algoritmo que nos permite transformar un autómata indeterminista en un autómata determinista equivalente.

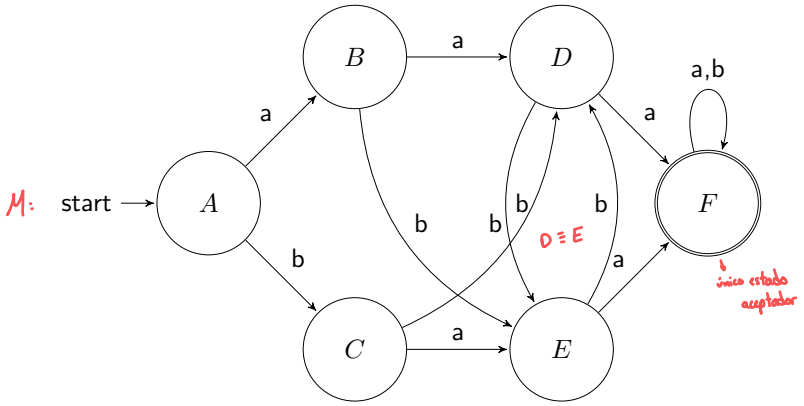
Simplificación de autómatas deterministas

Para simplificar un autómata determinista, la idea es eliminar un estado del autómata determinista que estemos considerando cuando haya otro estado que haga la misma función. Y, dicho de manera intuitiva, dos estados hacen la misma función, si para cualquier entrada que consideremos, se obtiene el mismo resultado tanto si se parte de un estado como del otro.

Formalmente, la definición es la siguiente. Sea $M = (K, \Sigma, \delta, q_0, F)$ un autómata determinista. Sean $p, q \in K$. Decimos que p, q son equivalentes (o indistingibles), si para todo $x \in \Sigma^*$, si $px \vdash_M^* p'$ y $qx \vdash_M^* q'$ entonces $p', q' \in F$ o $p', q' \notin F$.

Ejemplo

Consideremos el siguiente autómata determinista M:



Ejemplo

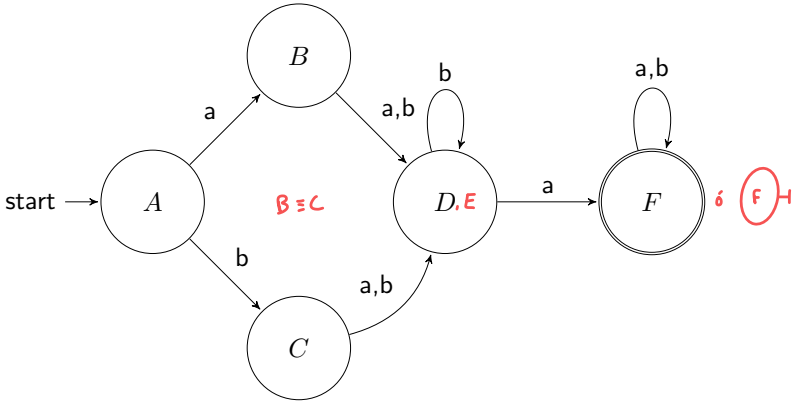
Tenemos que los estados D y E son equivalentes, ya que para toda palabra $x \in \{a, b\}^*$ se tiene que:

$$Dx \vdash_M^* F \iff x \in L(b^*a(a \cup b)^*),$$

$$Ex \vdash_M^* F \iff x \in L(b^*a(a \cup b)^*).$$

Por tanto, podemos juntar los estados D y E, obteniendo el siguiente autómata M' equivalente a M:

Ejemplo



Ejemplo

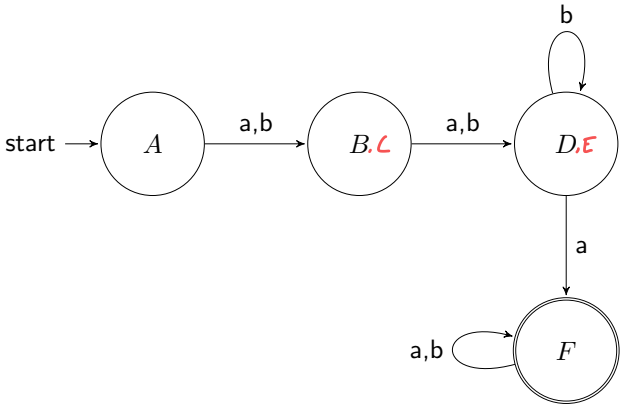
Ahora, observamos que los estados B y C son equivalentes, ya que para toda palabra $x \in \{a, b\}^*$ se tiene que:

$$Bx \vdash^*_{M'} F \iff x \in L((a \cup b)b^*a(a \cup b)^*),$$

$$Cx \vdash^*_{M'} F \iff x \in L((a \cup b)b^*a(a \cup b)^*).$$

Por tanto, obtenemos el siguiente autómata M'' equivalente a M:

Ejemplo



Se observa entonces que este autómata M'' ya no se puede simplificar.

Autómatas indeterministas

Los autómatas indeterministas son modelos computacionales equivalentes a los autómatas deterministas, pero más fáciles de construir, porque tienen generalmente menos estados y menos transiciones. La construcción de autómatas deterministas para diseñar programas se va haciendo más difícil a medida que crece el número de estados involucrados. Entonces, el modelo de autómata indeterminista nos ayuda a resolver este problema. Además, siempre se puede recurrir a autómatas indeterministas, ya que veremos que existe un algoritmo que nos permite transformar un autómata indeterminista en un autómata determinista equivalente. Entonces, cuando resulta complicado construir un autómata determinista para el diseño de un programa, podemos construir en primer lugar un autómata indeterminista, y a partir de él, mediante el algoritmo que veremos, construir el autómata determinista equivalente. El programa asociado a dicho autómata determinista será entonces el programa que buscamos.

Definición de autómata indeterminista

Un autómata indeterminista es una estructura $M = (K, \Sigma, \Delta, q_0, F)$ donde K, Σ, q_0, F son como en la definición de autómata determinista y Δ es un subconjunto de $K \times (\Sigma \cup \{\lambda\}) \times K$. ~ En el caso determinista: Δ és una funció de $K \times \Sigma$ en K.

La diferencia entre los autómatas deterministas y los indeterministas radica en que en el caso de los autómatas deterministas se ha de realizar exactamente una transición desde un estado para cada símbolo de la entrada, mientras que en el caso de los autómatas indeterministas se permite que desde un estado se realicen cero, una o más transiciones para cada símbolo de la entrada, y se permite además que se tengan transiciones con la palabra vacía, es decir, se permite pasar de un estado a otro sin leer ningún símbolo de la entrada.

Por tanto, en un autómata determinista el cómputo es único para cualquier entrada que sumistremos al autómata, mientras que un autómata indeterminista habrá en general varios cómputos para una entrada que le suministremos.

Noción de cómputo en un autómata indeterminista

Supongamos que $M = (K, \Sigma, \Delta, q_0, F)$ es un autómata indeterminista. A los elementos de Δ los llamaremos transiciones. Al igual que en el caso determinista, los cómputos de M se realizan aplicando transiciones de Δ.

Definimos una configuración de M como una palabra $px \in K\Sigma^*$. Si en un paso de cómputo del autómata nos encontramos en la configuración px, eso significa que el autómata se encuentra en el estado p y la x es la parte de la palabra de entrada aún no leída.

Si px, qy son configuraciones de M, decimos que px produce qy en un paso de cómputo , lo que representamos por $px \vdash_M qy$, si en M podemos pasar de px a qy aplicando una transición de Δ.

Si px, qy son configuraciones de M, decimos que px produce qy, lo que representamos por $px \vdash_M^* qy$, si en M podemos pasar de px a qy aplicando un número finito de transiciones de Δ.

Noción de lenguaje asociado a un autómata indeterminista

Supongamos que $M = (K, \Sigma, \Delta, q_0, F)$ es un autómata indeterminista. Decimos que una palabra $x \in \Sigma^*$ es reconocida (o aceptada) por M, si existe $q \in F$ tal que $q_0 x \vdash_M^* q$. ~ estado final

~ estado aceptador

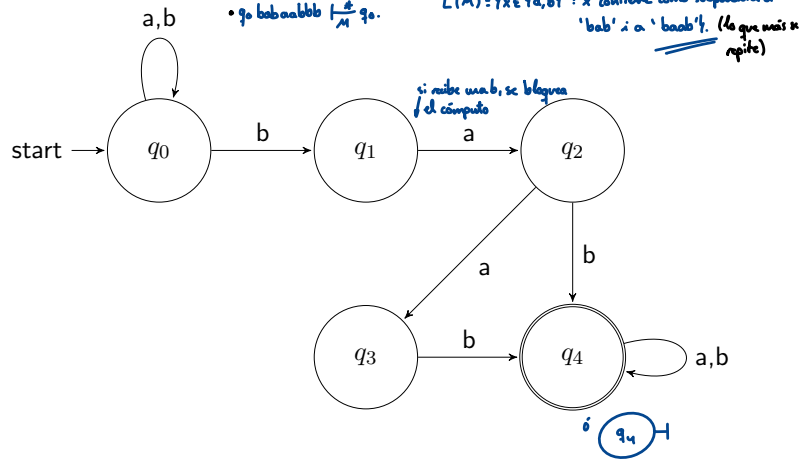
Por tanto, para que una palabra $x \in \Sigma^*$ sea reconocida por M tiene que existir un cómputo en el autómata que lea toda la palabra x y termine en un estado aceptador.

Definimos entonces el lenguaje reconocido (o aceptado) por M por

$$L(M) = \{x \in \Sigma^* : x \text{ es reconocida por } M\}.$$

Ejemplo 1

M:



Ejemplo 1

Consideremos la entrada $x = abbbabaabba$ para el autómata M. Vemos que hay varios cómputos de M para x. Por ejemplo, los dos siguientes cómputos reconocen la palabra x:

(1) $q_0abbabaabba \vdash_M q_0bbabaabba \vdash_M q_0babaabba \vdash_M q_1abaabba \vdash_M q_2baabba \vdash_M q_4aabba \vdash_M^* q_4\lambda$.

(2) $q_0abbabaabba \vdash_M q_0bbabaabba \vdash_M q_0babaabba \vdash_M q_0abaabba \vdash_M q_0baabba \vdash_M q_1aabba \vdash_M q_2abba \vdash_M q_3bba \vdash_M q_4ba \vdash_M q_4a \vdash_M q_4\lambda$.

Y hay también cómputos que no reconocen la palabra x. Por ejemplo, el cómputo que va leyendo todos los símbolos de x permaneciendo siempre en el estado q_0.

Normalmente, en un autómata indeterminista habrá computos que reconozcan una entrada y cómputos que no la reconozcan. Pero para que una palabra esté en el lenguaje de un autómata indeterminista, basta con que haya un cómputo que reconozca la palabra.

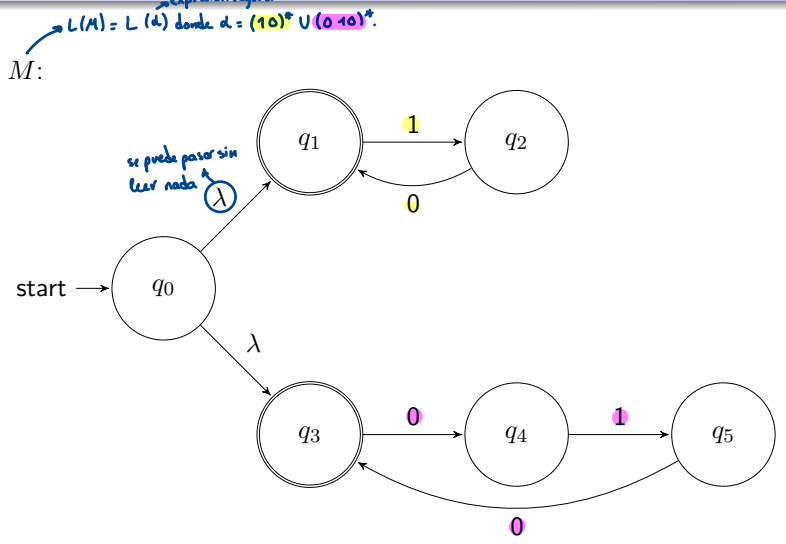
Ejemplo 1

En este autómata M, vemos que para que una palabra x sea reconocida, tiene quen haber un cómputo para esa palabra que parta del estado inicial q_0 y termine en el estado aceptador q_4. Y eso sucederá cuando o bien la palabra x contenga bab o bien contenga baab. Por tanto,

$L(M) = \{x \in \{a, b\}^* :$ en x aparece la palabra bab o la palabra baab$\}$.

Ejemplo 2

M:

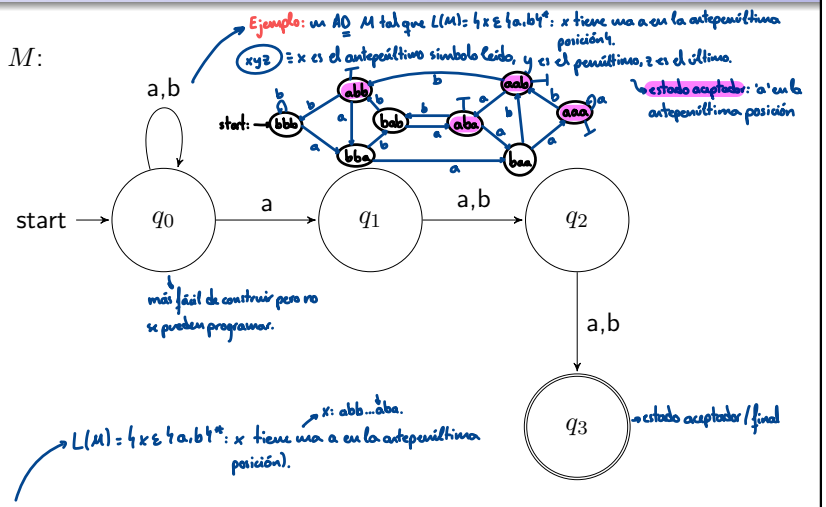


Ejemplo 2

Los estados aceptadores de M' son los estados q_1 y q_3. Si un cómputo del autómata termina en el estado q_1, se habrá reconocido una palabra de la forma $(10)^n$ donde $n \geq 0$. Y si un cómputo del autómata termina en el estado q_3, se habrá reconocido una palabra de la forma $(010)^n$ donde $n \geq 0$. Por tanto,

$L(M) = L(\alpha)$ donde α es la expresión regular $(10)^* \cup (010)^*$.

Ejemplo 3



Se tiene que $L(M)$ es el lenguaje de las palabras $x \in \{a,b\}^*$ tales que a es el antepenúltimo símbolo de x.

Equivalencia entre autómatas deterministas e indeterministas

Dos autómatas M y M' son equivalentes, si reconocen el mismo lenguaje, es decir, si $L(M) = L(M')$.

La equivalencia entre los autómatas deterministas y los autómatas indeterministas viene dada por el siguiente teorema.

Teorema

Para todo lenguaje L, existe un autómata determinista M tal que $L = L(M)$ si y sólo si existe un autómata indeterminista M tal que $L(M) = L$.

Por tanto, para todo autómata indeterminista M existe un autómata determinista M' que es equivalente a M.

Equivalencia entre autómatas deterministas e indeterministas

El siguiente concepto será utilizado en el algoritmo para transformar un autómata indeterminista en un autómata determinista equivalente.

Si $M = (K, \Sigma, \Delta, q_0, F)$ es un autómata indeterminista y $p \in K$ definimos el λ-cierre de p por

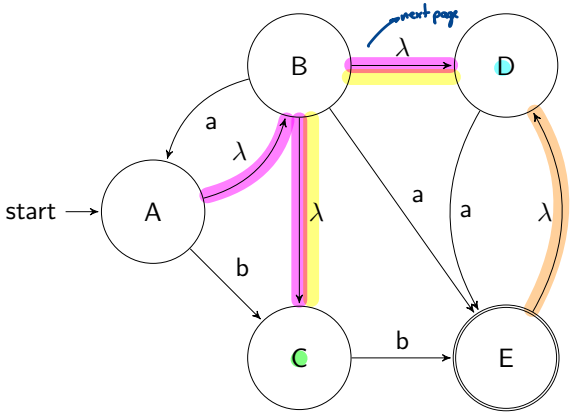
$$\Lambda(p) = \{q \in K : p\lambda \vdash_M^* q\}.$$

Por tanto, $\Lambda(p)$ es el conjunto de aquellos estados a los que podemos llegar desde el estado p sin leer ningún símbolo de la entrada.

Obsérvese que todo estado pertenece a su λ-cierre, es decir, para todo estado p se tiene que $p \in \Lambda(p)$.

Ejemplo

Consideremos el siguiente autómata indeterminista:



Ejemplo

Para simplificar la notación, representamos a un conjunto de estados $\{X_1, \dots X_n\}$ por la secuencia $X_1 \dots X_n$.

Tenemos entonces:

$\Lambda(A) = \{A, B, C, D\} = ABCD$,

$\Lambda(B) = BCD$,

$\Lambda(C) = C$,

$\Lambda(D) = D$,

$\Lambda(E) = DE$.

Algoritmo para transformar un autómata indeterminista en un autómata determinista equivalente

Sea $M = (K, \Sigma, \Delta, q_0, F)$ un autómata indeterminista. Definimos el autómata determinista $M' = (K', \Sigma, \delta', q_0', F')$ de la siguiente manera:

Si X es un conjunto, $\mathcal{P}(X) = \{y : y \subseteq X\}$.

(1) $K' = P(K)$ = conjunto de subconjuntos de K,

(2) $q_0' = \Lambda(q_0)$,

(3) $F' = \{X \in K' : X \cap F \neq \emptyset\}$,

(4) Si $X \in K'$ y $a \in \Sigma$, definimos

$$\delta'(X, a) = \bigcup \{\Lambda(q) : \text{ existe un estado } p \in X \text{ tal que } (p, a, q) \in \Delta\}.$$

Por tanto, para calcular $\delta'(X, a)$ se ha de hacer lo siguiente:

Algoritmo para transformar un autómata indeterminista en un autómata determinista equivalente

(1) Obtener todos los estados $q \in K$ para los cuales existe un estado $p \in X$ de manera que $(p, a, q) \in \Delta$.

terna

(2) Computar $\Lambda(q)$ para todo estado q obtenido en la etapa (1).

(3) Tomar la unión de los conjuntos $\Lambda(q)$ computados en (2).

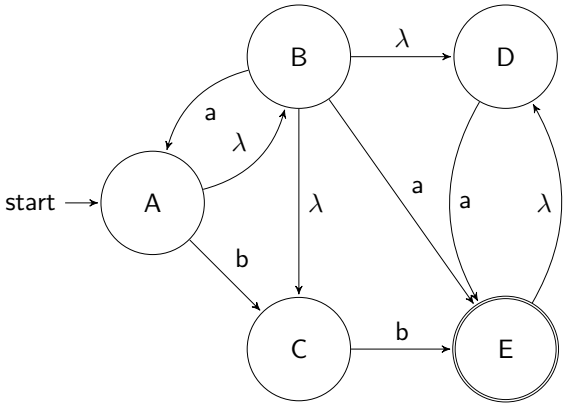
Se puede demostrar entonces que $L(M) = L(M')$, es decir, M y M' son equivalentes.

Algoritmo para transformar un autómata indeterminista en un autómata determinista equivalente

Por tanto, para obtener un autómata determinista equivalente a M, consideramos en primer lugar el estado inicial $q_0' = \Lambda(q_0)$, y computamos $\delta'(q_0', a)$ para cada símbolo a de Σ. A continuación, computamos de nuevo la función δ' para los estados nuevos que hayan salido. Y así continuamos, hasta que no salgan estados nuevos. Marcaremos entonces como estados aceptadores del autómata determinista M' aquellos estados que contengan algún estado aceptador del autómata indeterminista M.

Ejemplo

Consideremos el siguiente autómata indeterminista visto anteriormente:



Ejemplo

Recordemos que $\Lambda(A) = ABCD$, $\Lambda(B) = BCD$, $\Lambda(C) = C$, $\Lambda(D) = D$ y $\Lambda(E) = DE$.

El estado inicial del auómata determinista M es $\Lambda(A) == ABCD$. Tenemos entonces:

$\delta'(ABCD, a) = \Lambda(A) \cup \Lambda(E) = ABCDE$,

$\delta'(ABCD, b) = \Lambda(C) \cup \Lambda(E) = CDE$,

$\delta'(ABCDE, a) = \Lambda(A) \cup \Lambda(E) = ABCDE$,

$\delta'(ABCDE, b) = \Lambda(C) \cup \Lambda(E) = CDE$,

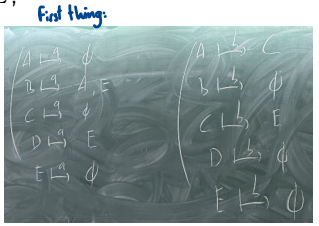
$\delta'(CDE, a) = \Lambda(E) = DE$,

$\delta'(CDE, b) = \Lambda(E) = DE$,

$\delta'(DE, a) = \Lambda(E) = DE$,

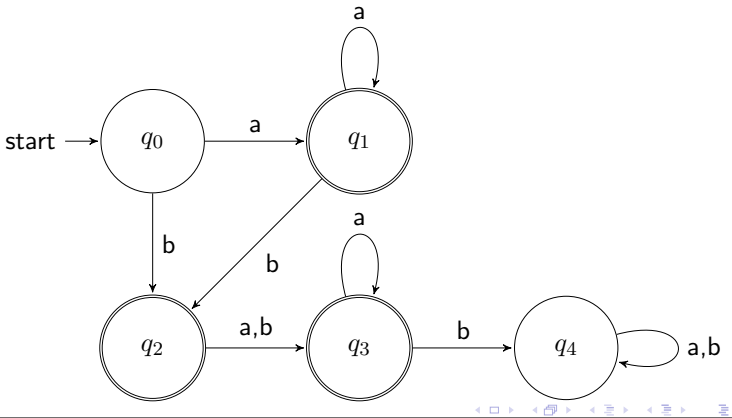
$\delta'(DE, b) = \emptyset$,

$\delta'(\emptyset, a) = \delta'(\emptyset, b) = \emptyset$.



Ejemplo

Por tanto, los estados de M' son $q_0 = ABCD, q_1 = ABCDE, q_2 = CDE, q_3 = DE$ y $q_4 = \emptyset$. Y como E es el único estado aceptador de M, los estados aceptadores de M' son q_1, q_2 y q_3. El grafo del autómata determinista obtenido es entonces el siguiente:



Introducción

En primer lugar, veremos un algoritmo que nos permite encontrar una expresión regular que describe el lenguaje de una autómata indeterminista sin transiciones con la palabra vacía.

Después de eso, estudiaremos una de las aplicaciones básicas de la teoría de autómatas, como es el diseño de programas eficientes de búsqueda.

A continuación, mostraremos cómo se puede diseñar el analizador léxico de un compilador.

Por último, empezaremos a estudiar el tipo de autómatas que se utilizan en el diseño del analizador sintáctico de un compilador.

Autómatas y expresiones regulares

La equivalencia viene dada por el siguiente teorema.

Teorema

Para todo lenguaje L, existe un autómata indeterminista M tal que $L(M) = L$ si y sólo si existe una expresión regular α tal que $L = L(\alpha)$.

también para un determinista

Este teorema expresa entonces la equivalencia entre el concepto de expresión regular y el concepto de autómata indeterminista.

El Lema de Arden

El algoritmo que veremos está basado en el llamado Lema de Arden. Para poder enunciar dicho lema, necesitamos definir el siguiente concepto.

Una ecuación lineal sobre un alfabeto Σ es una expresión

$$X = AX \cup B$$

donde A y B son lenguajes sobre el alfabeto Σ y X es una incógnita.

↳ $A, B \subseteq \Sigma^*$

Lema de Arden. (a) A^*B es solución de la ecuación $X = AX \cup B$. $\rightsquigarrow A^*B = AA^*B \cup B$

(b) Si $\lambda \notin A$, entonces A^*B es la única solución de la ecuación $X = AX \cup B$.

En el algoritmo que vamos a ver ahora se utiliza la parte (a) del Lema de Arden.

Observación

Obsérvese que el concepto de autómata determinista es un caso particular del concepto de autómata indeterminista, ya que si $M = (K, \Sigma, \delta, q_0, F)$ es un autómata determinista, se tiene que δ es una función de $K \times \Sigma$ en K, y por tanto δ es un subconjunto de $K \times \Sigma \times K$, y por consiguiente δ es un subconjunto de $K \times (\Sigma \cup \{\lambda\} \times K$.

A continuación, vamos a ver un algoritmo que nos permite encontrar una expresión regular que describe el lenguaje de una autómata indeterminista que no tenga transiciones con la palabra vacía. Por tanto, este algoritmo sirve para encontrar una expresión regular que describe el lenguaje de una autómata determinista. Si el autómata indeterminista tiene transiciones con la palabra vacía, deberemos transformar previamente dicho autómata en un autómata determinista equivalente, utilizando el algoritmo que vimos en la última clase.

Algoritmo para encontrar una expresión regular asociada a un autómata indeterminista

Sin tener en cuenta λ.

La entrada del algoritmo es un autómata indeterminista $M = (K, \Sigma, \Delta, q_0, F)$ sin transiciones con la palabra (λ) vacía. Y la salida es una expresión regular α tal que $L(\alpha) = L(M)$.

Sean $q_0, \ldots, q_n$ los estados de K, donde q_0 es el estado inicial. Para todo $i \leq n$ definimos

↳ $K = \{q_0, q_1, \ldots, q_n\}$.

$$X_i = L(q_i) = \{x \in \Sigma^* : \exists q \in F(q_i x \vdash_M^* q)\}.$$

Por tanto, $L(M) = X_0$.

$X_0 = L(q_0) = L(M)$

hay un cómputo de $q_i x$ a q.

Algoritmo para encontrar una expresión regular asociada a un autómata

Entonces, para todo estado $q_i \in F$ ponemos la ecuación

$$X_i = \bigcup \{aX_j : j \leq n, (q_i, a, q_j) \in \Delta\} \cup \lambda$$

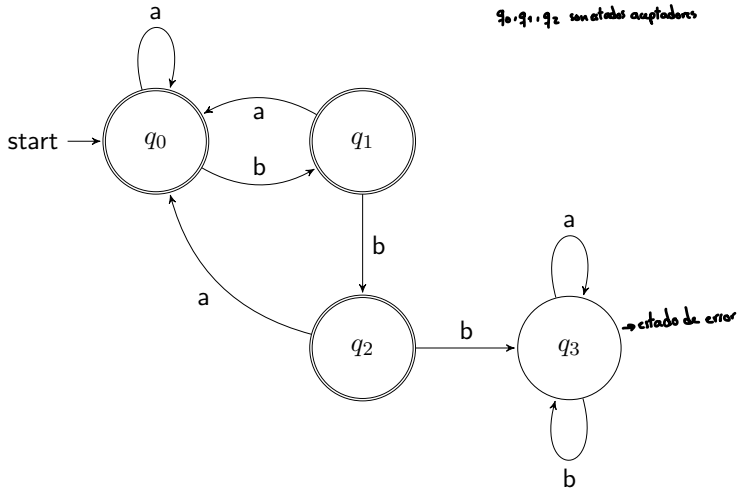
y para todo estado $q_i \notin F$ ponemos la ecuación

$$X_i = \bigcup \{aX_j : j \leq n, (q_i, a, q_j) \in \Delta\}$$

El algoritmo consiste entonces en resolver el sistema de ecuaciones, utilizando la parte (a) del Lema de Arden.

Ejemplo

Consideremos el autómata determinista M', que vimos anteriormente:



Ejemplo

Tenemos las siguientes ecuaciones asociadas al autómata M:

(1) $X_0 = aX_0 \cup bX_1 \cup \lambda$.

(2) $X_1 = aX_0 \cup bX_2 \cup \lambda$.

(3) $X_2 = aX_0 \cup bX_3 \cup \lambda = aX_0 \cup \lambda$.

(4) $X_3 = aX_3 \cup bX_3$.

Obsérvese que en la ecuación para X_2, tenemos que $aX_0 \cup bX_3 \cup \lambda = aX_0 \cup \lambda$, porque $X_3 = L(q_3) = \emptyset$, y por tanto $bX_3 = \emptyset$.

Ejemplo

Sustituyendo (3) en (2), obtenemos:

$(5) X_1 = aX_0 \cup baX_0 \cup b \cup \lambda = (a \cup ba)X_0 \cup b \cup \lambda.$

Ahora, sustituyendo (5) en (1), obtenemos:

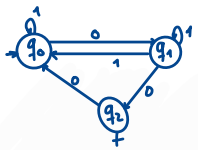
$(6) X_0 = aX_0 \cup b(a \cup ba)X_0 \cup bb \cup b \cup \lambda = (a \cup b(a \cup ba))X_0 \cup bb \cup b \cup \lambda.$

Aplicando entonces el Lema de Arden a (6), obtenemos:

$(7) X_0 = (a \cup b(a \cup ba))^* \cdot (bb \cup b \cup \lambda).$

Por tanto, $L(M) = X_0 = L(\alpha)$ donde $\alpha = (a \cup b(a \cup ba))^* \cdot (bb \cup b \cup \lambda)$.

Ejemplo



(0) $X_0 = 1X_0 \cup 0X_1$

(1) $X_1 = 1X_1 \cup 1X_0 \cup 0X_2$

(2) $X_2 = 0X_0 \cup \lambda$

Aplicando Arden a (0) obtenemos:

(3) $X_0 = 1^*0X_1$

Sustituyendo (3) en (2) obtenemos:

(4) $X_2 = 01^*0X_1 \cup \lambda$

Sustituyendo (4) en (1) obtenemos:

(5) $X_1 = 1X_1 \cup 1X_0 \cup 001^*0X_1 \cup 0$

Sustituyendo (3) en (5) obtenemos:

(6) $X_1 = 1X_1 \cup 11^*0X_1 \cup 001^*0X_1 \cup 0 = (1 \cup 11^*0 \cup 001^*0)X_1 \cup 0$

Aplicando Arden a (6), obtenemos:

(7) $X_1 = (1 \cup 11^*0 \cup 001^*0)^*0$

Sustituyendo (7) en (3):

(8) $X_0 = 1^*0(1 \cup 11^*0 \cup 001^*0)^*0$

$L(M)$

Búsqueda de un patrón en un texto

Estudiamos una aplicación básica de los autómatas, y es su uso para diseñar programas eficientes de búsqueda.

Consideremos el problema de buscar en un texto una secuencia de caracteres, a la cual denominamos "patrón". Tenemos entonces un patrón p y un texto t, que suponemos largo en relación al patrón, y queremos buscar el patrón p en el texto t. Un programa en JAVA que hace esta función sería el siguiente.

```
int buscar (String texto, String patron, int n, int m)
//n y m son las longitudes del texto y patrón respectivamente
{ int i =0,j; boolean trobat = false;
  while ((i <= n-m) && (trobat == false)
        { j=0;
        while(texto.charAt(i+j)== patron.charAt(j) && j < m -1)
                                                    j = j+1;
        if (texto.charAt(i+j) == patron.charAt(j) ) trobat = true;
        i = i + 1;}
return trobat;}
```

Búsqueda de un patrón en un texto

Es un programa ineficiente

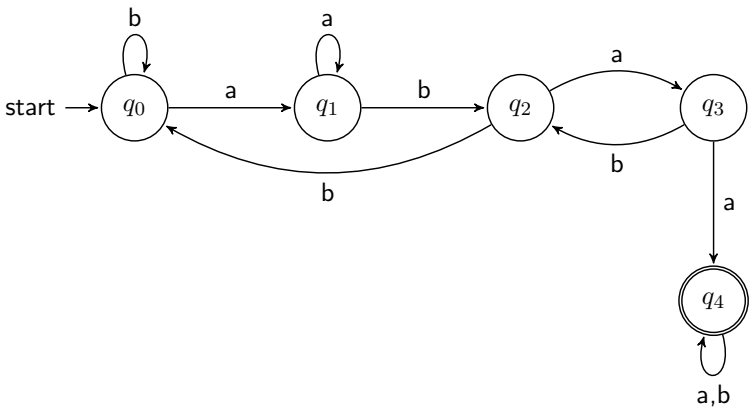
Observamos que el programa se sitúa en una posición del texto, y entonces compara los símbolos del patrón con los del texto a partir de la posición considerada. Si se produce una discrepancia, el programa se sitúa en la siguiente posición del texto y vuelve a comparar los caracteres del patrón con los del texto. Si se encuentra el patrón, el programa asigna a la variable trobat el valor true y sale del bucle "while" externo.

Búsqueda de un patrón en un texto

Sin embargo, este programa se puede mejorar, ya que vemos que no saca partido de la información que va recogiendo a medida que explora el texto. Para poner un ejemplo de este hecho, supongamos que trabajamos con el alfabeto $\Sigma = \{a, b\}$, que el patrón es la palabra $abaa$ y que el texto tiene la forma $ababaaabba$..... Entonces, el programa anterior se sitúa inicialmente en la primera posición del texto, y compara $abaa$ con $abab$, Como no son iguales, el programa pasa a la segunda posición del texto y compara $abaa$ con $baba$. Sin embargo, esto es ineficiente, ya que al comparar el patrón $abaa$ con $abab$, tenemos que las posiciones tercera y cuarta del texto coinciden con las dos primeras posiciones del patrón. Por tanto, un programa eficiente debe situarse en la quinta posición del texto y no en la segunda. Para conseguir hacer esto, hay que considerar un autómata determinista que reconozca el patrón $abaa$, y a continuación programar dicho autómata utilizando el algoritmo que ya conocemos. El autómata determinista que reconoce el patrón $abaa$ es el siguiente:

Búsqueda de un patrón en un texto

tiempo lineal



Búsqueda de un patrón en un texto

Se tiene entonces que si n es la longitud del texto, el tiempo de ejecución del programa asociado al anterior autómata determinista está en $O(n)$, es decir, tiene complejidad lineal, por lo que es un programa que mejora sustancialmente al primer programa que mostramos.

Es posible incluso, dados un texto y un patrón, escribir un programa que en primer lugar construye el autómata determinista que reconoce el patrón y, a continuación, simula el programa asociado al autómata que ha construido.

Fases en el diseño de un compilador

El diseño de compiladores es la principal aplicación de la teoría de autómatas. Un compilador es un programa interno del ordenador, que recibe como entrada un programa escrito en un lenguaje de programación de alto nivel (como Java o C), y entonces determina si el programa está correctamente escrito, y si es así traduce dicho programa a código máquina. Por tanto, los compiladores crean archivos ejecutables cuyo código entiende el ordenador. Los compiladores son esenciales en programación, ya que sin ellos no sería posible programar en lenguajes de alto nivel.

En el diseño de un compilador, se distinguen las tres siguientes fases:

(1) Análisis léxico.

(2) Análisis sintáctico.

(3) Análisis semántico.

Categorías sintácticas de un lenguaje de programación:

1) Identificadores
2) Cada tipo básico de datos
3) Cada símbolo relacional
4) Cada operador aritmético
5) Cada símbolo de puntuación
6) Cada palabra reservada.

Fases en el diseño de un compilador

El programa-fuente, es decir, el programa escrito en lenguaje de alto nivel que se va a traducir a código máquina, se almacena en un fichero de caracteres, al que se le llama fichero de entrada. El analizador léxico agrupa entonces los símbolos que va leyendo del fichero de entrada en categorías sintácticas, es decir, identifica las categorías sintácticas de los símbolos que va leyendo, y a continuación pasa esa información al analizador sintáctico. Las categorías sintácticas (o tokens) de un lenguaje de programación son la clase de los identificadores, cada tipo básico de datos (tipo integer, tipo float, tipo char, etc), cada símbolo relacional, cada operador aritmético, cada símbolo de puntuación y cada palabra reservada.

Recordemos que los identificadores son los nombres que damos a los tipos, literales, variables, clases, métodos, paquetes y sentencias de un programa.

En el lenguaje C, un identificador es una palabra formada por letras, dígitos y el carácter de subrayado, de manera que el primer carácter de la palabra no es un dígito.

Fases en el diseño de un compilador

En la segunda fase del diseño del compilador, en la fase de análisis sintáctico, se determina si el programa está escrito correctamente según las reglas del lenguaje de programación. El analizador sintáctico de un compilador trabaja siempre con las categorías sintácticas de los símbolos del programa que le suministra el analizador léxico. Y esto es así, porque sería mucho más complicado diseñar un analizador sintáctico que trabajase con los símbolos del programa en lugar de con las categorías. La utilización de las categorías sintácticas simplifica, por tanto, el diseño del analizador sintáctico.

Si no se detectan errores en la fase del análisis sintáctico, se pasa entonces a la fase del análisis semántico. En dicha fase, se controlan los aspectos semánticos como la correspondencia de tipos en las asignaciones o en los parámetros de llamadas a funciones, y si no hay tales errores semánticos se procede a efectuar la traducción.

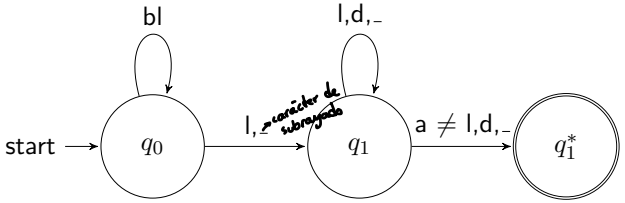
Diseño de analizadores léxicos

Para diseñar el analizador léxico de un compilador, se ha de definir un autómata indeterminista que reconozca cada una de las categorías sintácticas del lenguaje de programación que estemos considerando.

Para simplificar la construcción, denotamos por l a cualquier letra y por d a cualquier dígito. Y denotamos por bl al carácter blanco.

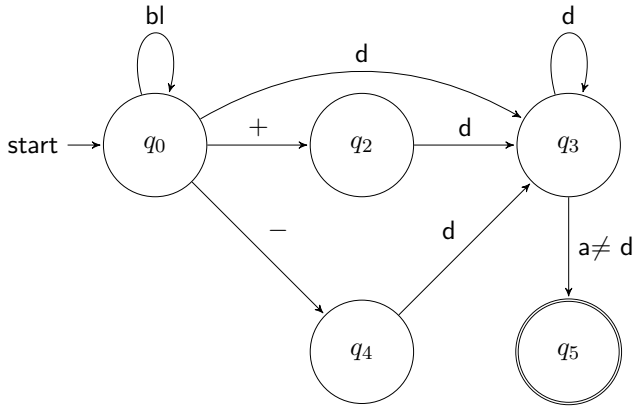
Diseño de analizadores léxicos

Tenemos el siguiente autómata para reconocer los identificadores de C:



Diseño de analizadores léxicos

Para el tipo "integer", tenemos el siguiente autómata:

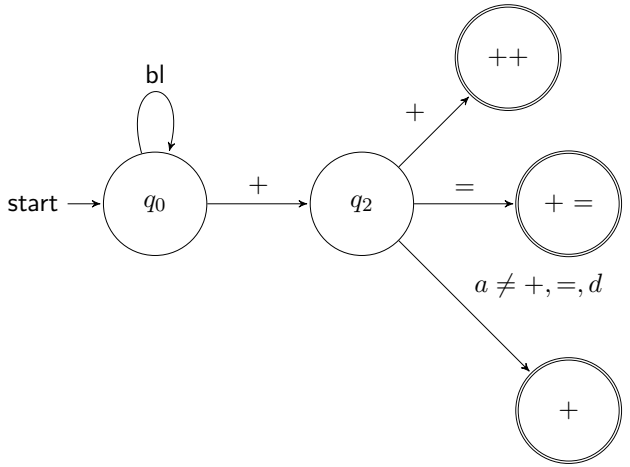


Diseño de analizadores léxicos

Ahora, a partir de los estados q_2 y q_4, podemos reconocer las categorías sintácticas $+$, $-$, $++$, $--$, $+=$ y $-=$.

En concreto, para reconocer las categorías sintácticas $+$, $+=$ y $++$, tenemos el siguiente autómata:

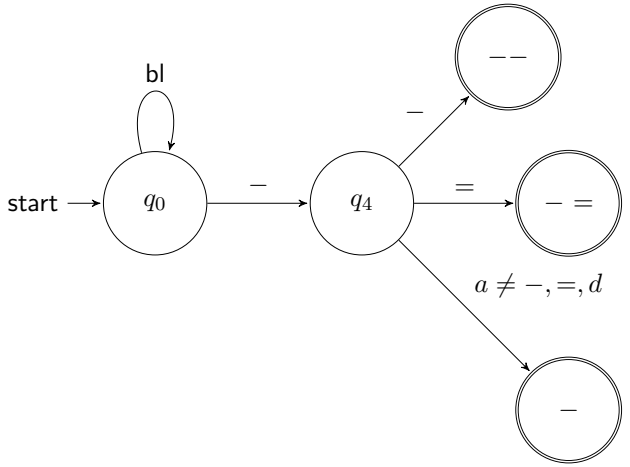
Diseño de analizadores léxicos



Diseño de analizadores léxicos

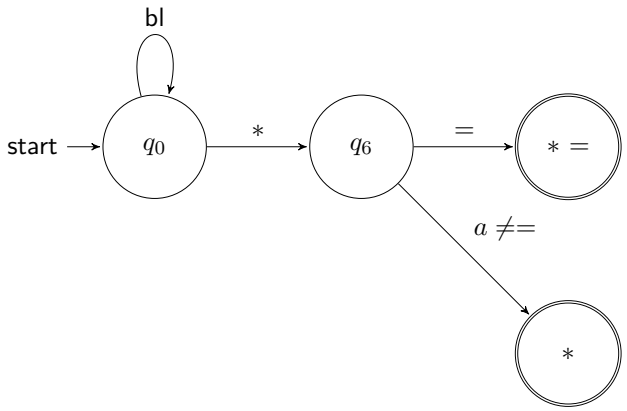
Y para reconocer las categorías sintácticas $-$, $-=$ y $--$, tenemos el siguiente autómata:

Diseño de analizadores léxicos



Diseño de analizadores léxicos

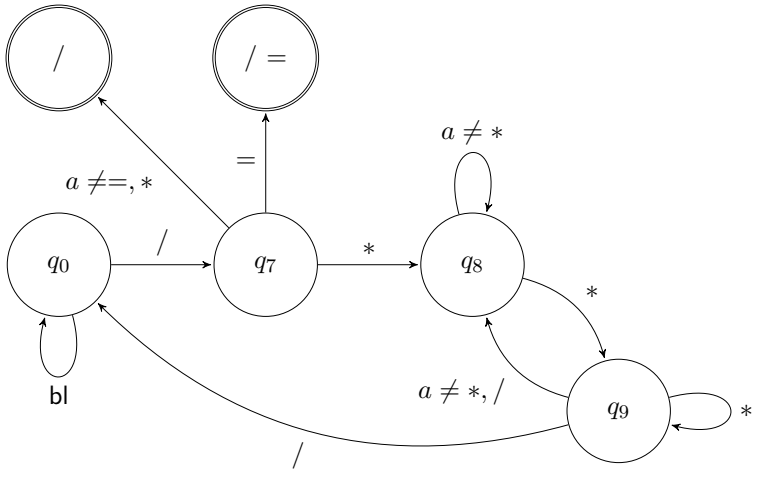
Ahora, para reconocer las categorías sintácticas $*$ y $* =$, tenemos el siguiente autómata:



Diseño de analizadores léxicos

Si entra el símbolo $/$, es posible que entre el símbolo para la división o la categoría sintáctica $/ =$ o es posible que tengamos el comienzo de un comentario. El analizador léxico debe saltar los comentarios, ya que éstos no tienen interés para el análisis sintáctico. En el siguiente autómata, eliminamos entonces los comentarios que comienzan por los símbolos $/*$ y terminan con los símbolos $*/$. Análogamente, se eliminarían los comentarios de una sola línea.

Diseño de analizadores léxicos



Diseño de analizadores léxicos

De esta forma, vamos reconociendo en el autómata las categorías sintácticas del lenguaje de programación. A continuación, transformamos este autómata indeterminista en un autómata determinista M equivalente, añadiendo un estado de error, es decir, añadiendo un estado no aceptador, del cual nunca saldremos y al cual accederemos desde cualquier otro estado cuando entre un símbolo que no interesa. Por tanto, los estados del autómata determinista M serán los mismos estados del autómata indeterminista más el estado de error.

Diseño de analizadores léxicos

En dicho autómata M, se observa que hay categorías sintácticas que son reconocidas al leer el último símbolo de la categoría, como es el caso de las categorías $++$ y $--$. Sin embargo, hay otras categorías sintácticas que son reconocidas al leer el símbolo siguiente de la categoría, como es el caso de los identificadores y del tipo entero, y de los operadores $+$ y $-$. Se dice entonces que estos estados aceptadores van adelantados un carácter, ya que reconocen la categoría cuando leen el símbolo siguiente.

Al escribir posteriormente el programa asociado al autómata, cuando lleguemos a un estado aceptador que vaya adelantado un carácter, no deberemos leer el siguiente símbolo de la entrada, porque dicho símbolo ya está leído. Para programar entonces el analizador léxico, debemos programar el autómata determinista M con las siguientes modificaciones:

Diseño de analizadores léxicos

(1) Cuando se llegue a un estado aceptador, se ha de imprimir la categoría sintáctica reconocida y se ha de volver al estado inicial.

(2) Cuando se llegue a un estado aceptador que vaya adelantado un carácter, no se ha de leer el siguiente carácter de la entrada.

(3) Cuando se llegue al estado q_1^*, se ha de explorar la tabla de las palabras reservadas para determinar si la palabra leída está en la tabla. Si no lo está, se da como salida "identificador". Y si lo está, se da como salida la palabra reservada.

Bibliografía para el tema 3

- Teoría de autómatas y lenguajes formales (D. Kelley)
  Capítulos 1 y 2.
- Compiladores: teoría e implementación (J. Ruiz Catalán)
  Capítulos 1 y 2.