TEORIA DE LENGUAJES

TEORICO 2004

Apuntes recopilados por MARCELO SILVA

LENGUAJES REGULARES

Σ denota el alfabeto

Σ^* (Clausura sobre Σ) es el conjunto de strings (tiras de caracteres) que se pueden formar con el alfabeto

L es el lenguaje, que es el conjunto de símbolos de Σ sobre los que se aplican determinadas reglas

$L \subseteq \Sigma^*$

R_L son las relaciones definidas sobre el lenguaje

Def.: Sean $(x,y) \in \Sigma^*$, $xR_Ly \Leftrightarrow \forall z \in \Sigma^*$ se cumple que $(xz \in L \wedge yz \in L)$ o bien $(xz \notin L \wedge yz \notin L)$, es decir que las concatenaciones xz y yz o bien ambas pertenecen al lenguaje o ambas no pertenecen al lenguaje.

Ej.

$\Sigma = \{0,1\}$ L = tiras de largo par

a) $0R_L11$ no pertenece a L ya que tenemos 0z y 11z, donde si z es par 0z no, pero 11z si; y si z es impar, 0z es par pero 11z no, por lo cual no se cumple la definición anterior.
b) $1R_L11100$ si pertenece a L.

Def. Lenguaje Regular

Un lenguaje L se dice regular si existe una expresión regular que lo define.

Expresión Regular

i) 0 es una expresión regular que denota a 0
   a es una expresión regular que denota a $\{\varepsilon\}$
   a es una expresión regular que denota a $\{a\} \forall a \in \Sigma$
ii) Sea r1 una expresión regular que describe a L1 y r2 la que describe a L2,
   (r1|r2) es una ER (expresión regular) que describe a $L1 \cup L2$
   (r1.r2) es una ER que describe a L1.L2
   (r1)* es una ER que describe a L1*
iii) Estas son todas las ER sobre Σ (clausura de Kleene)

Precedencia de los operadores :

mayor *
↓ .
menor |

ej.: dado Σ = (a,b), L = Tiras que no tienen 2 b seguidas

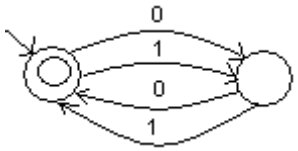
ER = $(b|\varepsilon)(a|ab)^*$

Autómatas Finitos

Dado un lenguaje, quiero saber si una tira pertenece a ese lenguaje. Los AF son herramientas para determinar esto, contestan la pregunta ¿ $x \in L$?

Un AF es una máquina de estados.

Ej: Si tenemos Σ = (0,1), y L = Tiras de largo par, con la ER = ((0|1)(0|1))*, el autómata sería:



Si al final de analizar la tira x de entrada estoy en el estado marcado con el circulo doble, entonces $x \in L$, sino $x \notin L$.

Autómatas Finitos Deterministas

Son aquellos AF donde dado un estado y un símbolo del alfabeto solo puedo ir a un único estado, está determinado. Es finito porque la cantidad de estados posibles es finita, y dado un estado me alcanza para saber si una tira es válida o no.

Def.: AFD : M:($Q, \Sigma, \delta, q_0, F$) donde

Q es la cantidad de nodos de la máquina de estados, es el conjunto de estados posibles

Σ es el alfabeto del lenguaje

δ es la función de transición de un estado a otro

q_0 es el estado inicial de la máquina de estados

F es el subconjunto de Q que representa los estados válidos, y gráficamente los representamos con doble circulo.

$\delta : Q \times \Sigma \rightarrow Q$

lo que se quiere hacer es $\delta (q_0, x)$, es decir que partiendo del estado inicial q_0 se analice la tira x. El tema acá es que $x \in \Sigma^*$, no a Σ como se define en $\delta = Q \times \Sigma \rightarrow Q$,

$\Rightarrow$ definimos $\hat{\delta} = Q \times \Sigma^* \rightarrow Q$, y la función a evaluar sería $\hat{\delta} (q_0, x)$

recordemos que

i) $\varepsilon \in \Sigma^*$

ii) si $a \in \Sigma^*$, $x \in \Sigma^* \Rightarrow xa \in \Sigma^*$

iii) Estas son todas las ER

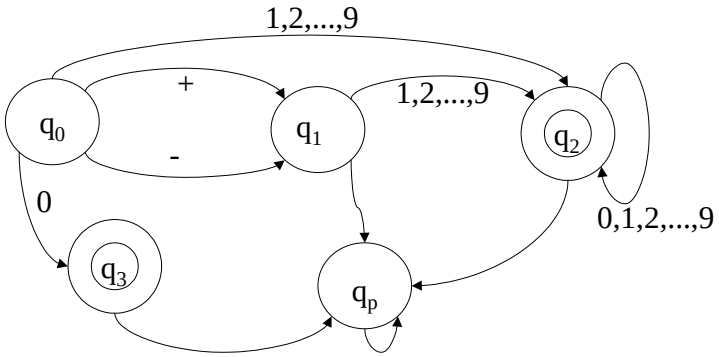
$\hat{\delta} (q, \varepsilon) = q$

$\hat{\delta}(q, xa) = \delta(\hat{\delta}(q, x), a) \ \forall\ q \in Q, a \in \Sigma, x \in \Sigma^*$

ej: Si tomamos la ER que modela los números enteros

(+ | - | ε) (1 | 2 | ... | 9) (0 | 1 | ... | 9)* | 0

el autómata sería



El arco superior que va de q_0 a q_1 sería la representación del ε inicial que NO se representa en el grafo

El estado q_p es el estado "pozo" cuando ocurre un error, que dependiendo de la implementación del autómata como se tratará.

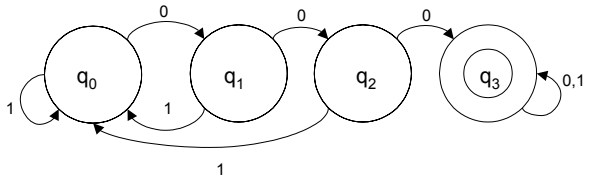
Lenguaje Regular (2da. Definición):

Un lenguaje se dice regular si existe un autómata finito determinista que lo reconoce. Es decir que un lenguaje L es regular si es aceptado por un autómata finito determinista definido como M:(Q, Σ, δ, q_0, F).
L = $\mathcal{L}$(M)
NOTA de notación : $\mathcal{L}$ (es la letra L mayúscula y cursiva) y representa al lenguaje L aceptado por el AFD M.

Las expresiones regulares (ER) sirven para definir un lenguaje
Los AFD sirven para reconocer si una tira pertenece o no a un lenguaje.

En los AFD, $\hat{\delta}(q_0, a) = \delta(q_0, a)$, ya que $\hat{\delta}(q_0, a) = \delta(\underbrace{\hat{\delta}(q_0, \varepsilon)}_{= q_0}, a) = \delta(q_0, a)$

Ej.: Σ = {0,1}, L = Tiras con al menos 3 ceros consecutivos
ER = (0|1)* 000 (0|1)*, ⇒ L es un lenguaje regular, y el AFD sería



donde cada estado significa algo distinto, por ejemplo q_0 es el estado en que faltan 3 ceros consecutivos, q_1 es el estado en que ya recibí un cero y faltan 2, q_2 es el estado en que ya recibí dos ceros y falta uno y finalmente q_3 es el estado solución.

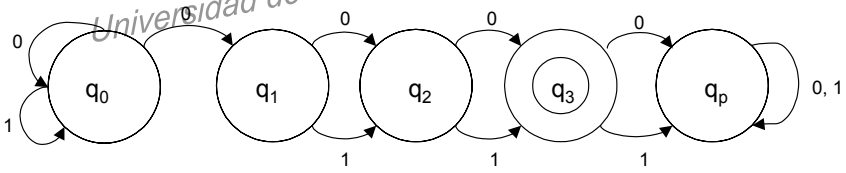
Ej.: Ej.: Σ = {0,1}, L = Tiras en que el tercer dígito desde la izquierda es 0
ER = (0|1)* 0 (0|1) (0|1) , $\Rightarrow$ L es un lenguaje regular. Para construir el AFD, como me interesan los 3 últimos dígitos tendríamos 2^3 = 8 estados posibles, algunos de los cuales serían finales y otros no. Este AFD es un poco más difícil de graficar.

Si el lenguaje estableciera que en lugar de ser 0 el 3er dígito desde la izquierda que sea el 5to entonces tendremos 2^5 = 32 estados posibles, algunos de los cuales serían finales y otros no. Este AFD es bastante más difícil de graficar.

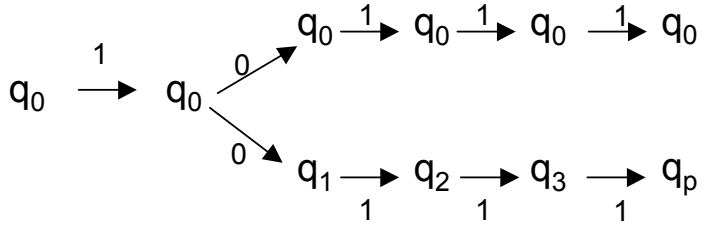
Si seguimos en este razonamiento llegaríamos a lenguajes cuyos AFD que serían imposibles de representar. Este problema se presenta porque el autómata es finito y determinista, es decir que dado un estado y un símbolo solo se puede ir a un estado.

Una alternativa para esto sería que no fuera determinista y que se pudiera "ir" a más de un estado. Sería como si se "partiera" el autómata en ejecuciones en paralelo.

Ej.:

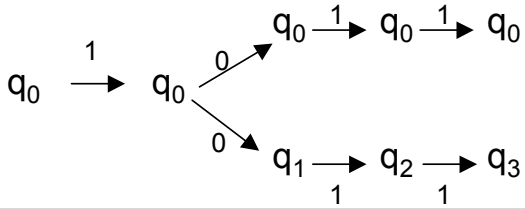


si evaluamos la tira 10111 y la representamos gráficamente sería



$\Rightarrow$ no quedaría en un estado válido (q_3) en ninguno de los caminos, por lo tanto la tira no es válida

si evaluamos la tira 1011



$\Rightarrow$ quedaría en un estado válido (q_3) en el camino inferior (uno de los caminos queda en un estado válido), por lo tanto la tira es válida

Lo que hemos modelado es un autómata finito no determinista

Autómata Finito NO Determinista

Def..: AFND : M:(Q, Σ, δ, q_0, F) donde

Q es la cantidad de nodos de la máquina de estados, es el conjunto de estados posibles
Σ es el alfabeto del lenguaje
δ es la función de transición de un estado a otro
q_0 es el estado inicial de la máquina de estados
F es el subconjunto de Q que representa los estados válidos, y gráficamente los representamos con doble circulo.

$\delta : Q \times \Sigma \rightarrow 2^Q$

lo que se quiere hacer es δ (q_0, x), es decir que partiendo del estado inicial q_0 se analice la tira x. El tema acá es que $x \in \Sigma^*$, no a Σ como se define en $\delta : Q \times \Sigma \rightarrow 2^Q$,

$\Rightarrow$ definimos $\hat{\delta} : Q \times \Sigma^* \rightarrow 2^Q$, y la función a evaluar sería $\hat{\delta}$ (q_0, x)

$\hat{\delta}$ (q, ε) = {q} que es el conjunto formado por q

$\hat{\delta}$ (q, wa) = {p/ $\exists$ r $\in$ $\hat{\delta}$ (q, w) $\wedge$ p $\in$ δ (r, a)}, siendo p $\subseteq$ Q y r $\subseteq$ Q (p y r son estados)

Intentemos buscar una notación más sencilla para expresar esto.

Si ponemos $\hat{\delta}$ (q, wa) = δ ($\hat{\delta}$ (q, w), a) no sería válido ya que δ no se aplica a conjunto de estados sino a 1 solo. Lo que podemos hacer para expresar esto es hacer una nueva extensión del δ.

$\tilde{\delta} : 2^Q \times \Sigma \rightarrow 2^Q$, $\tilde{\delta}$ (Q, a) = $\bigcup_{q \in Q_1}$ δ (q, a)

y hacemos

$\hat{\delta}$ (q, wa) = $\tilde{\delta}$ ($\hat{\delta}$ (q, w), a)

Def.: Lenguaje L aceptado por un AFND

$$L = \mathcal{L}(M) = \{ x / x \in \Sigma^*, \ \hat{\delta}(q_0, x) \cap F \neq \text{vacío} \}$$

O sea que el conjunto de estados finales luego de evaluar la tira x, si lo hacemos intersección con F (conjunto de estados solución) no es vacío. O lo que es lo mismo, que en el conjunto de estados finales de la evaluación de la tira x hay por lo menor un estado que pertenece al conjunto F de estados finales.

Lenguaje Regular (3ra. Definición):

Un lenguaje se dice regular si existe un autómata finito NO determinista que lo reconoce.

$\Rightarrow$ tenemos 3 definiciones para un Lenguaje regular; ER, AFD, AFND

y lo que tenemos que probar es que las 3 definiciones reconocen (o refieren) al mismo lenguaje. O lo que es lo mismo, si un lenguaje regular tiene un AFD que lo reconoce, tiene un AFND que lo reconoce y viceversa. AFD $\leftrightarrow$ AFND

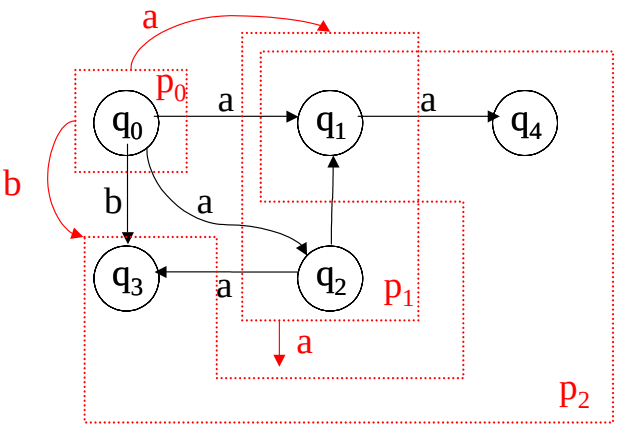
Los AFD se pueden considerar como un caso particular de los AFND, cuyo resultado es un conjunto de estados de largo 1 (o sea que tienen un solo estado).

AFD $\delta(q_0, a) = q_1$
AFND $\delta(q_0, a) = \{q_1\}$ (es un conjunto con un solo elemento)

Lo que tenemos que probar ahora es que todo AFND tiene un AFD equivalente.

Algoritmo AFND $\rightarrow$ AFD.

Supongamos que tenemos:
Algoritmo de entrada : AFND M:(Q, Σ, δ, q_0, F) / L = $\mathcal{L}$(M)
Algoritmo de salida : AFD M':(Q', Σ, δ', q_0', F') / L = $\mathcal{L}$(M')



En la figura representamos en negro el AFND. Lo que estamos representando en rojo y punteado son los subconjuntos de estados {p}, donde se agrupan todos los estados posibles a los que se puede ir desde uno dado, por ejemplo el {p1} es el subconjunto de estados a los que se puede ir desde q_0 con a.
$p \in Q'$ y $Q' \subseteq 2^Q$
$q_0' = \{ q_0 \}$ porque el primer estado (o estado de arranque) del AFD que estamos construyendo es particular y se corresponde con el primer estado del AFND. Esto no quiere decir que el q_0 no pueda formar parte de otros estados de Q'.

NOTACIÓN : Para este caso, el conjunto $\{q_0\}$ lo vamos a escribir como $[q_0]$.
$q_0' = \{ q_0 \} :: [q_0]$

Cuando en el AFND manejamos conjunto de estados $\{ q_i ,..., q_j \}$ en el AFD que estamos construyendo lo notaremos como $[q_i ,..., q_j]$, que es el "superestado" del nuevo AFD

En el AFD que estamos construyendo empezamos en el estado que contiene únicamente a q_0.

$F' = \{A / A \subseteq 2^Q , A \cap F \neq$ vacío$\}$, o sea que los estados finales del AFD que estamos construyendo son los subconjuntos de Q en los que al menos un estado es final o solución. Recordemos el ejemplo visto anteriormente para el lenguaje L = Tiras en que el tercer dígito desde la izquierda es 0, donde en el AFND si uno de los caminos termina en un estado solución, entonces la tira evaluada es aceptada como válida. Es lo mismo que se está definiendo en este caso sobre el AFD que estamos construyendo. Alcanza con que uno de los estados del "superestado" sea final en el AFND para que el "superestado" del AFD sea final.

Otra forma de expresar esto, con un poco de mayor rigor sería
$F' = \{ p \in Q' /$ el conjunto correspondiente a p (en M) contiene algún estado final (de M) $\}$

$$\delta'([q_i, ..., q_t], a) = [p_r, ... , p_l] \Leftrightarrow \tilde{\delta}(\{q_i, ... , q_t\}, a) = \{p_r, ..., p_l\}$$

(más adelante haremos referencia a esta como ©)

$$\delta'(q_0', x) \in F' \Leftrightarrow \tilde{\delta}(q_0, x) \cap F \neq \text{vacío}$$

o sea que el estado final del AFD que construimos pertenece a los estados solución si y solo si el estado pertenece a los estados solución del AFND.

Esto se demuestra por inducción completa sobre el largo de x (i.c. |x|)

$$\delta'(q_0', x) = [p_l, ... , p_t] \Leftrightarrow \quad \delta(q_0, x) = \{p_l, ... , p_t\}$$

Paso base : |x| = 0

$\wedge$

$\delta'(q_0', \varepsilon) = q_0' \Leftrightarrow \quad \delta(q_0, \varepsilon) = \{q_0\} :: [q_0]$ y esto es = q_0'

Paso inductivo :

Hi) Es válido para $|x| \leq h$

Ti) Es válido para $|x| = h + 1$

$\wedge \qquad \wedge \qquad \wedge$

$\delta'(q_0', x) = \quad \delta'(q_0', wa) = \delta'(\ \delta'(q_0', w), a) = [r_j, \ldots, r_s]$

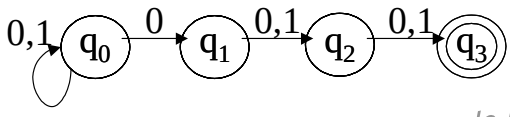
$\wedge \qquad \wedge$

por Hi) se tiene que $\delta'(q_0', w) = [p_l, \ldots, p_t] \Leftrightarrow \delta(q_0, w) = \{p_l, \ldots, p_t\}$

$\wedge \qquad \sim$

$\Rightarrow \quad \delta'([p_l, \ldots, p_t], a) = [r_j, \ldots, r_s] \quad \Leftrightarrow \delta(\{p_l, \ldots, p_t\}, a) = \{r_j, \ldots, r_s\}$, por la definición de δ' *(según se detalla en ©)*

Ej.: AFND



δ' al AFD que construiríamos del AFND

δ'	**0**	**1**
$[q_0]$	$[q_0, q_1]$	$[q_0]$
$[q_0, q_1]$	$[q_0, q_1, q_2]$	$[q_0, q_2]$
$[q_0, q_1, q_2]$	$[q_0, q_1, q_2, q_3]$	$[q_0, q_2, q_3]$
$[q_0, q_2]$	$[q_0, q_1, q_3]$	$[q_0, q_3]$
...	...	...

$\sim$

$\delta'([q_0, q_1], 0) = [q_0, q_1, q_2]$ porque $\delta(\{q_0, q_1\}, 0) = \{q_0, q_1, q_2\}$

$\sim$

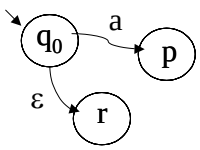
$\delta'([q_0, q_1], 1) = [q_0, q_2]$ porque $\delta(\{q_0, q_1\}, 1) = \{q_0, q_2\}$

y así sucesivamente.

Tercer modelo de autómata finito :

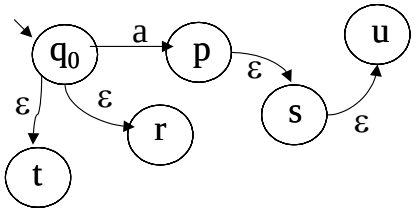
AFND - ε : Autómata Finito No Determinista con Transición Epsilon.

Es un AF que va a permitir una transición de estados sin consumir un carácter de la tira de entrada. Es decir que se permiten transiciones ε. ε NO ES UN SÍMBOLO DEL ALFABETO, representa la tira vacía. En este caso representa la transición de un estado a otro sin consumir un símbolo.



En este ejemplo, se podría arrancar en el estado inicial q_0 pero también se puede ir a r (o lo que es lo mismo se podría arrancar también desde r).

Se puede interpretar también que entre cada carácter de la tira hay un ε.



Empezando en q_0, cuando leo la "a" se va al estado p, pero también al s y al u

Lo importante a considerar es que el ε tiene dos "significados", representando a la tira vacía y también representando a la transición sin consumir carácter

Def.: ε - Clausura (q) donde q es un estado

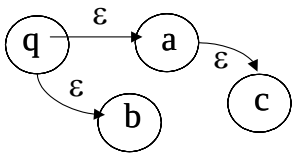
Llamamos epsilon clausura de q al conjunto de estados P a los que se llega desde q exclusivamente por caminos con arcos etiquetados ε. Cada estado pertenece implícitamente a su epsilon clausura.

ε - Clausura (P) donde P es un conjunto de estados

Llamamos epsilon clausura de P a la unión de las epsilon clausura de cada uno de los estados del conjunto P.

$$\varepsilon\text{ - Clausura (P)} = \bigcup_{q \in P} \varepsilon\text{ - Clausura (q)}$$

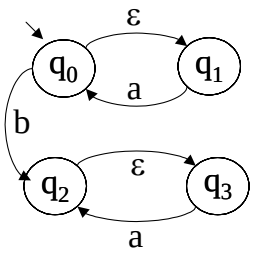
Ej. ε - Clausura ({q, p}) = {q, a, b, c}



Def.: AFND - ε : M:(Q, Σ, δ, q_0, F) donde

Q es la cantidad de nodos de la máquina de estados, es el conjunto de estados posibles
Σ es el alfabeto del lenguaje
δ es la función de transición de un estado a otro
q_0 es el estado inicial de la máquina de estados
F es el subconjunto de Q que representa los estados válidos, y gráficamente los representamos con doble circulo.

$\delta : Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow 2^Q$



Cuadro representación de las transiciones:

δ	a	b	ε
q_0	----	$\{q_2\}$	$\{q_1\}$
q_1			
...			

$\delta : Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow 2^Q$

$\wedge$

$\delta : Q \times \Sigma^* \rightarrow 2^Q$, ya que el Σ^* ya incluye a ε.

$\wedge$

$\delta (q, \varepsilon) = \varepsilon$ - clausura (q)

$\wedge \quad \sim \quad \wedge$

$\delta (q, wa) = \varepsilon$ - clausura (δ (δ (q, w), a)

a medida que vamos construyendo se va considerando la epsilon clausura, por lo tanto

$\wedge$

ε - clausura (δ (q, w)) está implícita. Consideremos por ejemplo que w = ε, $\Rightarrow$

$\wedge \quad \sim \quad \wedge$

ε - clausura (δ (q, εa)) = ε - clausura (δ (δ (q, ε), a)) =

$\sim$

ε - clausura (δ (ε - clausura (q), a))

Lenguaje Regular (4ta. Definición):

Un lenguaje se dice regular si existe un autómata finito NO determinista con transición ε que lo reconoce.

Def.: Lenguaje L aceptado por un AFND - ε

$\wedge$

$L = \mathcal{L}(M) = \{ x / x \in \Sigma^*, \ \delta (q_0, x) \cap F \neq \text{vacío}\}$

Veamos ahora que los lenguajes reconocidos por un AFND coinciden con los reconocidos por un AFND - ε. O sea que ambos AF reconocen al mismo lenguaje.

AFND $\leftrightarrow$ AFND - ε

El AFND se puede considerar como un caso particular del AFND - ε donde no hay transiciones ε. Por lo tanto queda probado que AFND $\rightarrow$ AFND - ε
Vamos a probar ahora que desde un AFND - ε se puede construir un AFND, o lo que es lo mismo, AFND - $\varepsilon \rightarrow$ AFND.

Algoritmo AFND - $\varepsilon \rightarrow$ AFND.

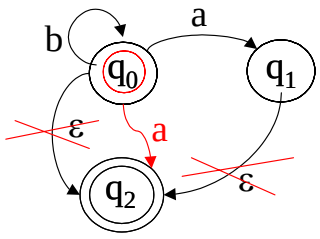
Entrada: AFND - ε M : $(Q , \Sigma, \delta , q_0 , F) / L = \mathcal{L}(M)$
Salida : AFND M': $(Q', \Sigma, \delta', q_0', F') / L = \mathcal{L}(M')$

$Q' = Q$
$q_0' = q_0$

$$F' = \begin{cases} F \\ F \cup \{ q_0\} \text{ si } \varepsilon \text{ - clausura } (q_0) \cap F \neq \text{vacío} \end{cases}$$

Esto quiere decir que si la ε - clausura (q_0) tiene un estado final, lo cual implica que el AFND - ε reconoce la tira vacía como válida ya que desde el estado inicial mediante una transición ε se llega a un estado final, se agrega q_0 como estado final a F', de forma que el AFND que estamos construyendo reconozca a la tira vacía como válida.

Al eliminar los caminos ε, puede ser que queden estados a los que no se llega nunca. Si a un estado solo se puede llegar mediante arcos ε, al eliminarlos quedan inaccesibles. Se podrían eliminar estos estados del conjunto Q' en un proceso posterior de depuración, pero por simplicidad del algoritmo se dejan (Q' = Q).



Este sería el AFND - ε para representar (b*a) | b*
Para convertirlo en un AFND, eliminamos los caminos ε, lo cual marcamos con rojo en el dibujo.

Se marca q_0 como estado final, ya que desde q_0 se puede ir a q_2 que si es un estado final mediante un camino ε, por lo cual reconoce la tira vacía como válida. Para que el AFND que construimos también reconozca esa tira como válida, entonces debemos marcar a q_0 como estado final.

Con esto no estamos introduciendo soluciones, ya que cualquier tira, si terminaba en el estado q_0, mediante una transición ε podía ir a un estado final, por lo tanto este AFND - ε la tomaba como válida. Al poner q_0 como final se obtiene el mismo resultado.

$\hat{\delta}'(q', a) = \varepsilon$ - clausura ($\tilde{\delta}'$ (ε - clausura (q'), a))

$\hat{\delta}'(q_0', x) = \hat{\delta}(q_0, x)$

Esto se demostrará por I. C., pero esto no se cumple para la tira x = ε (tira vacía), se cumple a partir del $|x| \geq 1$ (largo de x mayor o igual a 1). Esto pasa porque

$\hat{\delta}'(q_0', \varepsilon) = \{q_0'\}$ mientras que

$\hat{\delta}(q_0, \varepsilon) = \varepsilon$ - clausura (q_0), que no necesariamente es igual a $\{q_0'\}$

I.C. Paso base (a partir de |x| = 1)

$\hat{\delta}'(q_0', a) = \hat{\delta}(q_0, a)$ esto se cumple por la propia definición del δ', ya que

- $\hat{\delta}(q_0, a) = \varepsilon$ - clausura ($\tilde{\delta}$ (ε - clausura (q_0), a)) y
- $\hat{\delta}'(q_0', a) = \varepsilon$ - clausura ($\tilde{\delta}'$ (ε - clausura (q_0'), a))

Paso inductivo

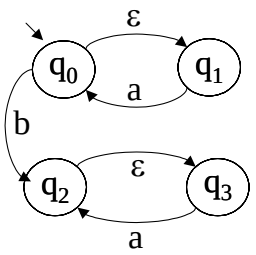
Hi) $\hat{\delta}'(q_0', x) = \hat{\delta}(q_0, x)$, con $|x| \leq h$ (lo llamamos P' y P respectivamente)

Ti) $\hat{\delta}'(q_0', x) = \hat{\delta}(q_0, x)$, con $|x| = h + 1$

Dem. Al x de la ti) lo podemos ver como wa, donde $|w| \leq h$, $|a| = 1$

$\hat{\delta}'(q_0', wa) = \tilde{\delta}'$ ($\hat{\delta}'$(q_0', w), a) = $\tilde{\delta}'$(P',a) = $\bigcup_{q \in P'} \delta'(q, a) = \bigcup_{q \in P} \delta(q, a) = \tilde{\delta}(P,a)$ =

= $\tilde{\delta}$ ($\hat{\delta}$ (q_0, w), a) = $\hat{\delta}(q_0, wa) = \hat{\delta}(q_0, x)$ **L.Q.Q.D.**



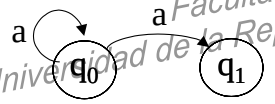
Cuadro representación de las transiciones:

δ'	a	b
q_0	{ q_0, q_1}	{ q_2, q_3}
q_1		
...		

Se sugiere calcular todas las ε - clausura de cada estado primero, dejarlas anotadas y luego aplicar la fórmula (el algoritmo). Conviene verificar primero todos los estados finales, y si la ε - clausura (q_0) tiene algún estado final, en cuyo caso se debe poner q_0 como final.
En el ejemplo se haría :
ε - clausura (q_0) = {q_0, q_1}
ε - clausura (q_1) = {q_1}
ε - clausura (q_2) = {q_2, q_3}
ε - clausura (q_3) = {q_3}

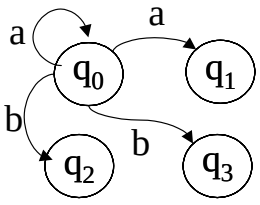
luego se haría

$\tilde{\delta}$' (q_0, a) = ε - clausura ($\tilde{\delta}$ (ε - clausura (q_0), a) = ε - clausura ($\tilde{\delta}$ ({q_0, q_1 }, a) =
= ε - clausura (δ (q_0, a) ∪ δ (q_1, a)) = ε - clausura ({q_0 }) = {q_0, q_1} ya que
δ (q_0, a) es el conjunto vacío o el pozo, y δ (q_1, a) = {q_0 }



$\tilde{\delta}$' (q_0, b) = ε - clausura ($\tilde{\delta}$ ({q_0, q_1 }, b) = ε - clausura (δ (q_0, b) ∪ δ (q_1, b)) =
= ε - clausura ({q_2 }) = {q_2, q_3} ya que δ (q_0, b) es el conjunto {q_2 }, y δ (q_1, b) = vacío

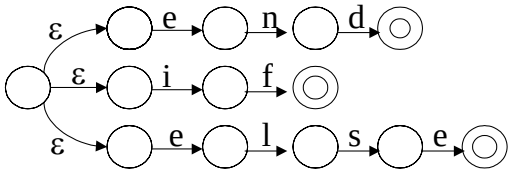
y así sucesivamente se iría construyendo.



Si tenemos un AFND - ε, es conveniente llevarlo a un AFD, ya que en términos de computación es más eficiente. Para hacer esto se aplica este último algoritmo y se obtiene un AFND, y a partir de este se le aplica el anterior y se obtiene un AFD.

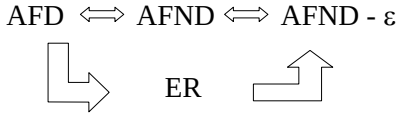
Un analizador lexicográfico de un compilador parte del AFND - ε y lo lleva al AFD aplicando estos algoritmos.

Ejemplo : {end}, {if}, {else}



Se construye el autómata para cada palabra por separado, se une mediante un estado inicial y las transiciones ε y después, con los algoritmos anteriores, se obtiene el AFD.

Vamos a probar ahora que a partir de una ER se puede obtener un AFND - ε, para cerrar y probar que



Algoritmo ER → AFND- ε.

Dado un ER r / L = ℒ(r), ∃ M : AFND - ε / L = ℒ(M)

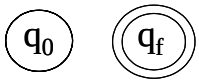
Este algoritmo construye, a partir de una ER, un AFND - ε con un único estado final del cual NO salen transiciones, o sea que M : (Q , Σ, δ , q_0 , {q_f }) donde δ (q_f, x) = {vacío}

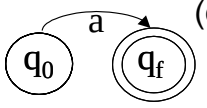
(en el ejemplo se sustituyen los estados finales por uno solo final al que se llega mediante transiciones ε)

ε → {ε }
∅ → {∅ }
a → {a }
$r_1|r_2$
$r_1.r_2$
r_1*

Demostración : La hacemos por IC sobre la cantidad de operadores.
PASO BASE:

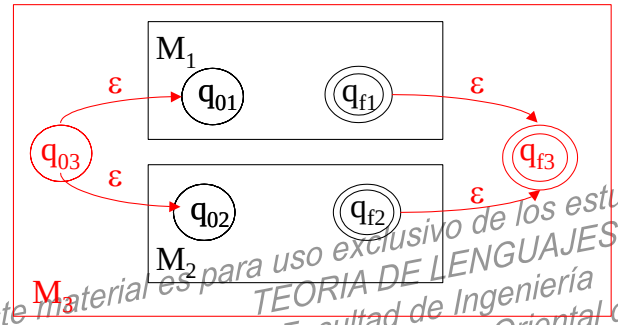
ε el AFND - ε que lo reconoce sería  (es decir que la tira vacía es la única solución válida)

∅ el AFND - ε que lo reconoce sería  (es decir que no tiene soluciones válidas)

a el AFND - ε que lo reconoce sería  (es decir que la única solución válida es la tira a)

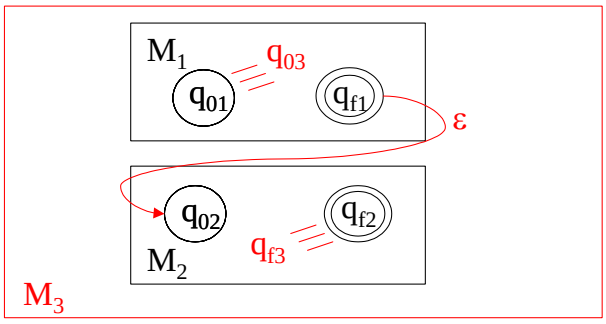
PASO INDUCTIVO :
Hi)
r_1 (con cantidad de operadores < k) $\Rightarrow \exists$ AFND - ε M_1: (Q_1 , Σ, δ_1 , q_{01} , {q_{f1} }) / $\mathcal{L}(M_1) = \mathcal{L}(r_1)$, donde δ_1 (q_{f1}, a) = {vacío}, $\forall$ a $\in \Sigma$ (o sea que desde el estado final no salen transiciones)
r_2 (con cantidad de operadores < k) $\Rightarrow \exists$ AFND - ε M_2: (Q_2 , Σ, δ_2 , q_{02} , {q_{f2} }) / $\mathcal{L}(M_2) = \mathcal{L}(r_2)$, donde δ_2 (q_{f2}, a) = {vacío}, $\forall$ a $\in \Sigma$ (o sea que desde el estado final no salen transiciones)
$Q_1 \cap Q_2$ = {vacío } (o sea que no tienen estados en común)

Ti) Veremos si existe un AFND - ε para $r_1 \mid r_2$
M_3: (Q_3 , Σ, δ_3 , q_{03} , {q_{f3} }) / $\mathcal{L}(M_3) = \mathcal{L}(r_1 \mid r_2)$, donde δ_3 (q_{f3}, a) = {vacío}, $\forall$ a $\in \Sigma$



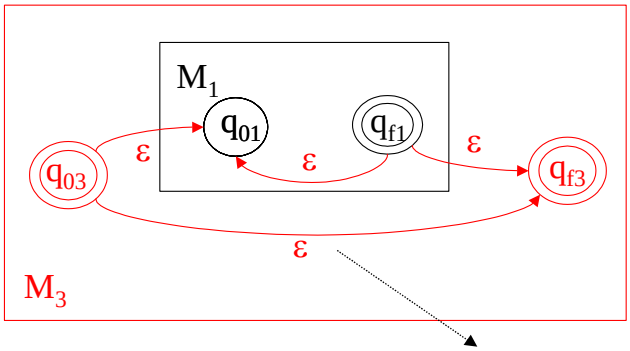
Creamos un estado inicial del que se puede ir a q_{01} y a q_{02} mediante transiciones ε.
$Q_3 = Q_1 \cup Q_2 \cup \{ q_{03} , q_{f3} \}$
$\delta_3 (q_i , a)$ = $\delta_1 (q_i , a)$ si $q_i \in Q_1$
= $\delta_2 (q_i , a)$ si $q_i \in Q_2$
$\delta_3 (q_{03}, \varepsilon)$ = {q_{01} , q_{02} }
$\delta_3 (q_{f1}, \varepsilon)$ = $\delta_3 (q_{f2}, \varepsilon)$ = {q_{f3} }

Ti) Veremos si existe un AFND - ε para $r_1 . r_2$
M_3: (Q_3 , Σ, δ_3 , q_{03} , {q_{f3} }) / $\mathcal{L}(M_3) = \mathcal{L}(r_1 . r_2)$, donde δ_3 (q_{f3}, a) = {vacío}, $\forall$ a $\in \Sigma$



$Q_3 = Q_1 \cup Q_2$ y además $q_{03} = q_{01}$ y $q_{f3} = q_{f2}$
$\delta_3 (q_i , a) = \delta_1 (q_i , a)$ si $q_i \in Q_1$
$= \delta_2 (q_i , a)$ si $q_i \in Q_2$
$\delta_3 (q_{f1}, \varepsilon) = \{q_{02} \}$

Ti) Veremos si existe un AFND - ε para r_1*
M_3: $(Q_3 , \Sigma, \delta_3 , q_{03} , \{q_{f3} \})$ / $\mathcal{L}(M_3) = \mathcal{L}(r_1^*)$, donde $\delta_3 (q_{f3}, a) = \{\text{vacío}\}$, $\forall$ a $\in \Sigma$



Se agrega esta transición para asegurarnos que r_1* reconozca la tira vacía, que no sabemos si r_1 la reconoce

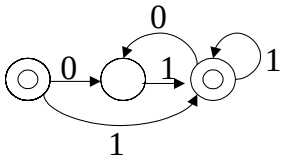
$Q_3 = Q_1 \cup \{ q_{03} , q_{f3} \}$
$\delta_3 (q_i , a) = \delta_1 (q_i , a)$ si $q_i \in Q_1$
$\delta_3 (q_{03}, \varepsilon) = \{ q_{01} , q_{f3} \} = \delta_3 (q_{f3}, \varepsilon)$

Con esto se demuestra que dada una ER se puede obtener un AFD.

En la práctica, manualmente no se utiliza este mecanismo ya que genera AFD que son muy engorrosos de manejar. Veamos un ejemplo de esto:

ER = (01|1)*

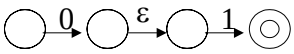
Haciéndolo a “mano” :



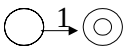
Haciéndolo mediante el algoritmo :

1) AFND - ε para 0 y para 1

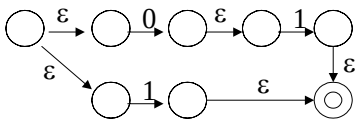
2) Los uno mediante el ε por la concatenación (operación “.” entre 0 y 1)



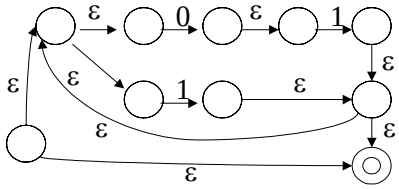
3) AFND - ε para 1



4) Uno los AFND - ε de 2) y 3) mediante el procedimiento para el operador "|"
5) Aplicamos el procedimiento para el operador "*" al AFND - ε de 4)



Como se ve, el AFND - ε que se obtiene mediante la aplicación del algoritmo es bastante más engorroso que el que se obtiene haciéndolo en forma manual.

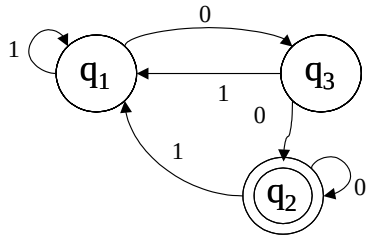


Algoritmo AFD → ER.

Para completar el círculo de transformaciones entre ER y AF nos queda ver como hacemos para obtener un ER a partir de un AFD.

R^k_{ij} conjunto de tiras que van de q_i al q_j sin pasar por un estado mayor que k, con i, j entre 1 y n y k entre 0 y n.
$Q = \{ q_1, q_2, ..., q_n \}$ $|Q| = n$ (autómata numerado con "n" estados)

Ej. R^0_{3i} esta es la transición => {1}
El conjunto de tiras supra 0 son el delta (R)



$\{ \bigcup_{q_i \in F} R^n_{1j} \}$ es el conjunto de estados finales. Puedo asociar estas tiras a unas ER.

Def.. R^k_{ij} $k = 0$

$R^0_{ij} = \{ a / \delta(q_i, a) = q_i \}$ $i \neq j$

$R^0_{ii} = \{ \varepsilon \} \cup \{ a / \delta(q_i, a) = q_i \}$ $i = j$

$k > 0$

$R^k_{ij} = R^{k-1}_{ij} \cup R^{k-1}_{ik} . (R^{k-1}_{kk})^* R^{k-1}_{kj}$

R^{k-1}_{ij} son los caminos que no pasan por k y van de i a j

$R^2_{14} = R^1_{14} \cup R^1_{12} . (R^1_{22})^* R^1_{24}$

Sea la ER r^0_{ij} de R^0_{ij}

$i \neq j \longrightarrow \quad a_1 \mid a_2 \mid \ldots. \mid a_t \quad / \exists\, \delta\,(q_i\,,\, a_n) = q_j$, donde $n \in 1 \,..\, t$

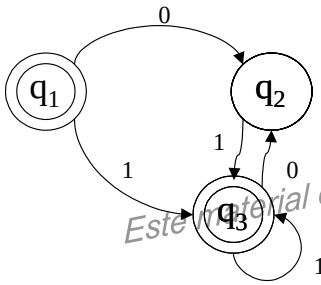
$i \neq j \longrightarrow \quad a_1 \mid a_2 \mid \ldots. \mid a_t \mid \varepsilon$

H) $\exists\ r^{k-1}_{ij}$ para c/ R^{k-1}_{ij}
T) $\exists\ r^{k}_{ij}$ para c/ R^{k}_{ij}

Se cumple porque $r^k_{ij} = r^{k-1}_{ij} \mid r^{k-1}_{ik}\,(r^{k-1}_{kk})^* \, r^{k-1}_{kj}$

Para todos estos conjuntos de tiras puedo encontrar una ER.

Por lo tanto, la ER que define el lenguaje L es la concatenación de todas las ER de las tiras formadas por los estados finales.



$r^3_{11} \mid r^3_{13}$ $\qquad r^3_{11} = r^2_{11} \mid r^2_{13}\,.\,(r^2_{33})^*\,.\,r^2_{31}$

Partiendo de un AFD puedo encontrar una ER que defina el lenguaje reconocido por este autómata.

Siempre va del estado inicial al último estado.
k es el estado mayor.

Máquinas secuenciales

Los AF son un tipo de máquina secuencial. Veremos otro tipo de estas que son los autómatas con salida.

Autómatas con salida

Son “traductores”, que lo que hacen es traducir una entrada en la salida. Es un modelo en el que no se construye un lenguaje sino que directamente se traduce la tira de entrada. Un ejemplo de estos son los transductores, que se utilizan para procesar el lenguaje natural, en los que por ejemplo dado un verbo genera (o traduce esta entrada) en todas las conjugaciones del mismo.

En estos autómatas no hay estados finales, procesan una entrada y devuelven una salida.

M : (Q , Σ , Δ , δ , λ , q_0), donde

Q = conjunto de estados
Σ = alfabeto
Δ = alfabeto de salida
δ = transición
λ = función de salida. Es la que indica que es lo que se imprime, cual es la salida
q_0= estado inicial

Máquina de MEALY

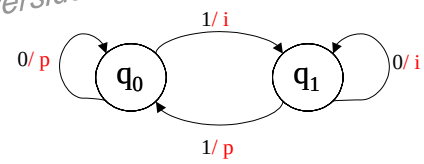
δ : Q x Σ → Q dada una pareja estado símbolo me dice a que estado voy
λ : Q x Σ → Δ dada una pareja estado símbolo me dice que símbolo se imprime

Ej. Construir un autómata que cuando tenga una cantidad par de 1 imprima “p”, y cuando sea impar imprima “i”

Σ = { 0 , 1}, Δ = { p , i }

δ	0	1
q_0	q_0	q_1
q_1	q_1	q_0

λ	0	1
q_0	p	i
q_1	i	p



Al poner el “0/p” en el arco que va de q_0 a q_0 se representa el hecho que estoy en el estado q_0 , recibo un 0, imprimo una “p” y quedo nuevamente en el estado q_0.
Al igual que en loa AF, tendremos extensiones para las funciones.

$\hat{\delta}$ = Q x Σ* → Q, y la función a evaluar sería $\hat{\delta}(q_0, x)$

$\hat{\delta}(q, \varepsilon) = q \quad \forall q \in Q$

$\hat{\delta}(q, aw) = \hat{\delta}(\delta(q, a), w) \quad \forall q \in Q, a \in \Sigma, w \in \Sigma^*$

$\hat{\lambda}$ = Q x Σ* → Δ* , y la función a evaluar sería $\hat{\lambda}(q_0, x)$

$\hat{\lambda}(q, \varepsilon) = \varepsilon \quad \forall q \in Q$ (si no leo nada, entonces no imprimo nada)

$\hat{\lambda}(q, aw) = \lambda(q, a) . \hat{\lambda}(\delta(q, a), w) \quad \forall q \in Q, a \in \Sigma, w \in \Sigma^*$

Ej.:

$\hat{\lambda}(q_0, 01) = \lambda(q_0, 0) . \hat{\lambda}(\delta(q_0, 0), 1) = \lambda(q_0, 0) . \lambda(q_0, 1) . \hat{\lambda}(\delta(q_0, 1), \varepsilon) =$
$= p . i . \varepsilon \Rightarrow$ se imprimiría "pi"

Máquina de MOORE

$\delta : Q \times \Sigma \to Q$ dada una pareja estado símbolo me dice a que estado voy
$\lambda : Q \to \Delta$ dado un estado me dice que símbolo se imprime

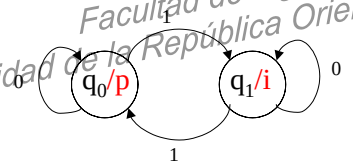
La diferencia con la maq. de Mealy es la función de salida. En esta la salida está asociada únicamente al estado en el que estoy.

Ej. Construir un autómata que cuando tenga una cantidad par de 1 imprima "p", y cuando sea impar imprima "i"

$\Sigma = \{ 0 , 1\}, \Delta = \{ p , i \}$

δ	0	1
q_0	q_0	q_1
q_1	q_1	q_0

λ	0	1
q_0	p	i
q_1	i	p



Estoy en q_0 y leo un 0, me quedo en q_0 y se imprime una "p".

$\hat{\delta} = Q \times \Sigma^* \to Q$

$\hat{\delta}(q, \varepsilon) = q \quad \forall q \in Q$

$\hat{\delta}(q, aw) = \hat{\delta}(\delta(q, a), w) \; \forall q \in Q, a \in \Sigma, w \in \Sigma^*$

$\hat{\lambda} = Q \to \Delta^*$

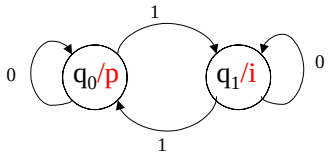
Como no podemos trabajar directamente con $\hat{\lambda}$ se define un $\lambda' = Q \times \Sigma^* \to \Delta^*$

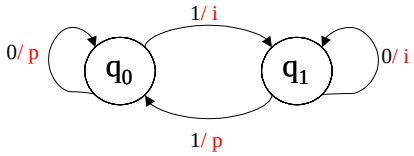
$\lambda'(q, \varepsilon) = \varepsilon \quad \forall q \in Q$ (si no leo nada, entonces no imprimo nada)
$\lambda'(q, aw) = \lambda(q, a) . \lambda'(\delta(q, a), w)$

Equivalencia MOORE → MEALY

M : (Q , Σ , Δ , δ , λ , q_0) → M' : (Q' , Σ , Δ , δ' , λ' , q_0')
δ (q, a) = p donde p y q ∈ Q, a ∈ Σ
λ (p) = b donde b ∈ Δ

La función δ' (de MEALY) coincide con la δ (de MOORE) y Q' (de MEALY) coincide con Q (de MOORE).
δ' (q, a) = δ (q, a)
λ' (q, a) = b o sea que al estado de salida de MOORE le asocio la salida del estado en MEALY





δ (q_0 , 0) = q_0
λ (q_0) = p
⇒ λ' (q_0 , 0) = p

δ (q_0 , 1) = q_1
λ (q_1) = i
⇒ λ' (q_0 , 1) = i

δ (q_1 , 0) = q_1
λ (q_1) = i
⇒ λ' (q_1 , 0) = i

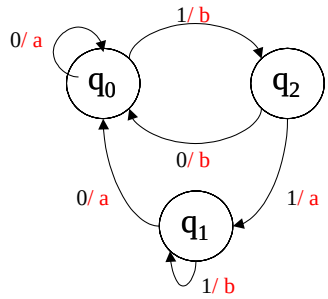
δ (q_1 , 1) = q_0
λ (q_0) = p
⇒ λ' (q_1 , 1) = p

Equivalencia MEALY → MOORE

M : (Q , Σ , Δ , δ , λ , q_0) → M' : (Q' , Σ , Δ , δ' , λ' , q_0')
δ (q, a) = p donde p y q ∈ Q, a ∈ Σ
λ (q, a) = b donde b ∈ Δ

p^b ∈ Q' → los estados de Q' se definen como q^s ∀ s ∈ Δ. Esto quiere decir que se hace para <u>todos</u> los símbolos de Δ.
λ' (p^b) = b
δ' (q^s, a) = p^b

Ej. $\Sigma = \{ 0 , 1\}$, $\Delta = \{ a , b \}$

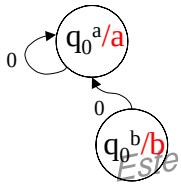


A partir de esta máquina de MEALY construyamos la correspondiente de MOORE.

Tomemos primero $\delta (q_0 , 0) = q_0$
$\lambda (q_0 , 0) = a$
$\lambda' (q_0^a) = a$

Se hace entonces $\delta' (q_0^a , 0) = q_0^a$
$\delta' (q_0^b , 0) = q_0^a$

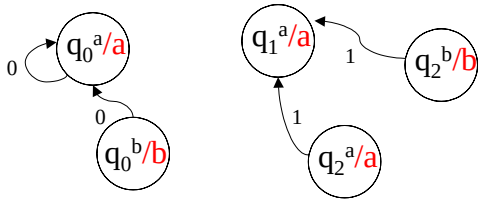
y se empezaría a construir la máquina de la forma :



Si tomamos a $\delta (q_2 , 1) = q_1$
$\lambda (q_2 , 1) = a$
$\lambda' (q_1^a) = a$

Se hace $\delta' (q_2^a , 1) = q_1^a$
$\delta' (q_2^b , 1) = q_1^a$

y la máquina estaría quedando como



El estado inicial en la máquina de MOORE resultante puede ser cualquiera de los dos q_0

Esta forma de construcción puede generar estados a los que nunca se puede llegar, por lo cual luego de obtenida se puede hacer una depuración de estos en varios ciclos, de forma de eliminar los estados “basura” que se generan en el proceso.

Tomemos por ejemplo el lenguaje L de las tiras que tengan únicamente 1 o 2 dígitos uno.
$\Sigma = \{ 0 , 1 \}$
$xR_l y$ si $\forall z \in \Sigma^*$, $xz \in L \Leftrightarrow yz \in L$ y $xz \notin L \Leftrightarrow yz \notin L$

ej.: $001R_l101$ NO ES CORRECTO, ya que si agregamos un 1 a ambas tiras, 0011 pertenece a L pero 1011 no pertenece a L, lo cual no cumple con la definición de R_l.

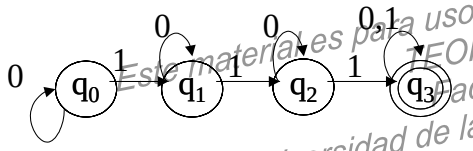
M: un AFD M : $(Q , \Sigma , \delta , q_0 , F)$

Def.: $x R_m y$ decimos que x , y están en la misma relación

$R_m \Leftrightarrow \hat{\delta}(q_0, x) = \hat{\delta}(q_0, y)$ $x, y \in \Sigma^*$

o sea que partiendo del estado inicial, ambas tiras terminan en el mismo estado. R_m es una relación de equivalencia.

Para un AFD, cada uno de los estados está definiendo una clase de equivalencia.



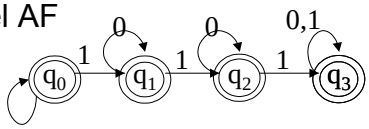
$100 R_m 010000$, ambas tiras (100) y (10000) están en la misma clase de equivalencia.

$\hat{\delta}(q_0, 100) = q_1$, $\hat{\delta}(q_0, 10000) = q_1$

La cantidad de clases de equivalencia está definida por la cantidad de estados del AF.

Cada clase de equivalencia tiene una ER asociada, y por lo tanto cada estado tiene una ER asociada. En el ejemplo anterior,
para q_0 la ER =0*
para q_1 la ER =0*10*
para q_2 la ER =0*10*10*
para q_3 la ER =0*10*10*10*1(0|1)*

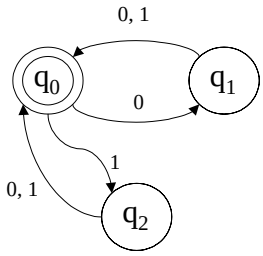
como notación, se pueden elegir tiras representativas de cada estado, ej:
para q_0 tomamos [ε]
para q_1 tomamos [1]
para q_2 tomamos [11]
para q_3 tomamos [111]

Si tenemos el AF 

reconoce el mismo lenguaje que 

pero el primero tiene 4 clases de equivalencias mientras que el segundo tiene solo una. En este caso no se cumple que $R_M = R_L$. Esto se cumple únicamente si el AFD es el mínimo.
Siempre se cumpla que la cantidad de clases de un lenguaje va a ser menor o igual a la cantidad de clases del autómata que lo reconoce.

Cantidad de clases de $R_L \leq$ cantidad de clases de R_M



Este AFD tiene 3 clases de equivalencias $\Rightarrow R_M = 3$. Reconoce el lenguaje de las tiras de largo par, y por eso el lenguaje tiene 2 clases, las tiras de largo par y los de largo impar, $\Rightarrow R_L = 2$
0 R_L 1 es cierta, ya que si se le agregan 0 o 1 a ambas tiras, o bien ambas tiras pertenecen a L o ambas no pertenecen a L. Sin embargo
0 R_M 1 no es cierta, ya que

$\hat{\delta}(q_0, 0) = q_1$ y $\hat{\delta}(q_0, 1) = q_2$ y por lo tanto no se cumple con la definición de R_M

TEOREMA DE MYHILL – NERODE

L es regular $\Leftrightarrow$ cantidad de clases de R_L es finita.
(demostraremos primero la $\Rightarrow$) L regular $\Rightarrow$ hay un AFD que lo reconoce, $\Rightarrow$ hay un número finito de estados $\Rightarrow$ tiene una cantidad finita de clases de R_M , y como $R_M \geq R_L \Rightarrow$ R_L tiene una cantidad finita de clases.

(demostraremos ahora la $\Leftarrow$) La cantidad de clases de R_L es finita $\Rightarrow$ L es un lenguaje regular.
La idea es construir un AFD a partir de las clases de equivalencias. Esto es construir
M : (Q , Σ, δ , q_0 , F) / L = $\mathcal{L}$(M)
Q = {clases de R_L }
q_0 = la clase que representa a la tira vacía. En el ejemplo anterior sería la [ε]
F = { [x] / x $\in$ L } son aquellas clases asociadas a las tiras que pertenecen al lenguaje, de las que [x] es la representación.

δ ([x], a) = [xa] con a $\in \Sigma$. Hay que tener en cuenta que [x] y [xa] son etiquetas que representan a clases. Se podría poner δ ([x], a) = [ya]

$\wedge$

δ ([ε], x) = [x]

o sea que el AFD M acepta a la tira x $\Leftrightarrow$ $\wedge$ δ ([ε], x) $\in$ F, $\Leftrightarrow$ [x] $\in$ F, $\Rightarrow$ x $\in$ L.

COROLARIO DEL TEOREMA

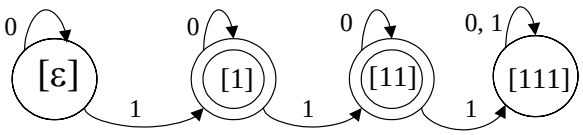
El AFD que se obtiene es el mínimo, ya que parte de las clases de R_L y por lo tanto la cantidad de clases de este AFD (R_M) será igual a las de R_L (porque lo construimos de esta forma. Por lo tanto no existe otro AFD que pueda tener menor cantidad de clases, ya que Cantidad de clases de R_L $\leq$ Cantidad de clases de R_M

Ej. Tomando nuevamente el lenguaje de las tiras que contengan 1 o 2 unos. Las clases de este L serán:

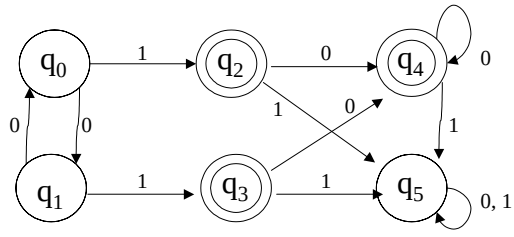
[ε]
[1]
[11]
[111]

donde [1] y [11] son finales ya que reconocen a las tiras del lenguaje. Los obtenemos en forma intuitiva a partir de la definición del L

δ ([ε], 0) = [ε]
δ ([ε], 1) = [1]
δ ([1], 0) = [1]
δ ([1], 1) = [11]
δ ([11] , 0) = [11]
δ ([11] , 1) = [111]
δ ([111] , 0) = [111]
δ ([111] , 1) = [111]



Una vez que tengo las clases de R_L, el AFD es inmediato.
El problema que se plantea con este mecanismo es justamente encontrar las clases de R_L.
Si por ejemplo el L lo dan como un AF en lugar de hacerlo en lenguaje natural y nos piden hallar el mínimo o nos preguntan si es mínimo



Lo primero que hacemos es hallar las clases de R_M, lo que es lo mismo que hallar el conjunto de tiras que llegan a cada estado del AF. Tenemos que construir una ER para cada uno de estos.

C_0 - δ (q_0, x) = q_0
C_1 - δ (q_0, x) = q_1
C_2 - δ (q_0, x) = q_2
C_3 - δ (q_0, x) = q_3
C_4 - δ (q_0, x) = q_4
C_5 - δ (q_0, x) = q_5

Para construir la ER de cada clase se hace analizando “artesanalmente” el AFD

C_0 – (00)*
C_1 – 0(00)*
C_2 – (00)*1
C_3 – 0(00)*1 = (00)*01
C_4 – (C_2 | C_3)00* = ((00)*1 | 0(00)*1)00* = 0*100*
C_5 – (C_2 | C_3 | C_4)1(0|1)*

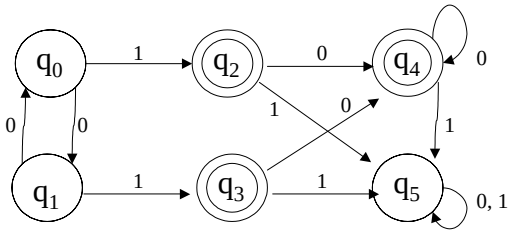
Donde C_2 , C_3 y C_4 son los estados finales, $\Rightarrow$ si los concateno obtengo una ER que define el lenguaje (reconoce las tiras válidas).
$\Rightarrow$ C_2 | C_3 | C_4 = (00)*1 | (00)*01 | 0*100* = 0*1 | 0*100* = 0*1 (ε | 00*) = 0*10*
$\Rightarrow$ el lenguaje es el que reconoce las tiras con un solo 1,
$\Rightarrow$ las clases de R_L son [ε], [1], [11] donde la clase final es [1]. Una vez que tenemos esto la construcción del AF mínimo es inmediata.

ALGORITMO DE MINIMIZACIÓN

Entrada: AFD M : (Q , Σ, δ , q_0 , F) / L = $\mathcal{L}$(M)
Salida : AFDmin M’: (Q’, Σ, δ’, q_0’, F’) / L = $\mathcal{L}$(M’)

Básicamente hay que agrupar estados de M que conformen un único estado.

Se trata de ir quitando estados que no se puedan distinguir. Por ejemplo, en el AF



estando en q_0 o en q_1, si viene un 1 llego a un estado final, por lo cual la tira es aceptada, pero no se puede distinguir si vino de q_0 o de q_1. Por lo tanto el algoritmo de minimización se trata de realizar una partición (Π) tal que vaya agrupando estados.

1) Π_0 : Q - F
   q_2 ,q_3 y q_4 como son estados finales, en algún sentido son indistinguibles, y por otro lado tengo al resto de los estados. O sea que la partición inicial va a estar formada por 2 conjuntos, {Q – F} y {F}
   Π_0 $\underbrace{Q - F}_{C_1}$ $\underbrace{F}_{C_2}$
   El algoritmo sigue analizando en cada uno de esos 2 conjuntos el comportamiento de los estados según las tiras que vienen.

2) Para cada C_i de la partición Πcorriente empiezo a considerar de a pares los estados de ese subconjunto, y según el comportamiento de las transiciones para todo par de estados y para todo símbolo del alfabeto.

   Para todo C_i de la partición Πcorriente
   Para todo par p, q $\in C_i$
   Para todo a $\in \Sigma$
   } calculo δ (p, a) = p' y δ (q, a) = q'

   Si $\underbrace{p' \text{ y } q' \in C_i}$ distintos $\Rightarrow$ $\underbrace{p \text{ y } q \in C_i'}$ distintos
   Πcorriente Πcorriente + 1

Ej: con el autómata anterior
Π_0 C_1 = {q_0 q_1 q_5 } C_1 = {q_2 q_3 q_4 }
Empecemos con el conjunto C_1 = { q_0 q_1 q_5}
Tomemos q_0 q_5
δ (q_0, 0) = q_1 } ambos $\in C_1$ δ (q_0, 1) = q_2 $\in C_2$ } ambos $\notin$ al mismo C_j
δ (q_5, 0) = q_5 δ (q_5, 1) = q_5 $\in C_1$
O sea que el p y el q (q_0 , q_5) son distinguibles, y por lo tanto voy empezando a formar las clases de Π_1 $\Rightarrow$ $q_0 \in C_1'$ y $q_5 \in C_2'$
Tomemos q_0 q_1
δ (q_0, 0) = q_1 } ambos $\in C_1$ δ (q_0, 1) = q_2 $\in C_2$ } ambos $\in C_2 \Rightarrow q_0, q_1 \in C_1'$
δ (q_1, 0) = q_0 δ (q_1, 1) = q_3 $\in C_2$
$\Rightarrow$ C_1' = {q_0, q_1} C_2' = {q_5}
Tomemos q_1 q_5
No es necesario hacerlo ya que vimos que q_0 es equivalente a q_1 y q_0 ya es distinguible con q_5. No tiene sentido hacer q_1q_5 ya que va a dar que son distinguibles.

Ahora analicemos al conjunto C_2 = { q_2 q_3 q_4}
Tomemos q_2 q_3
δ (q_2, 0) = q_4 δ (q_3, 0) = q_4 δ (q_2, 1) = q_5 δ (q_3, 1) = q_5
Tomemos q_3 q_4
δ (q_3, 0) = q_4 δ (q_4, 0) = q_4 δ (q_3, 1) = q_5 δ (q_4, 1) = q_5
Tomemos q_2 q_4
δ (q_2, 0) = q_4 δ (q_4, 0) = q_4 δ (q_2, 1) = q_5 δ (q_4, 1) = q_5
$\Rightarrow$ q_2 q_3 q_4 son indistinguibles

Hagamos ahora el paso 3 del algoritmo
Si $\Pi_t \neq \Pi_{tm}$ entonces
hacer $\Pi_t \leftarrow \Pi_{tm}$
repetir paso 2
Si no, Π_t es la Π_{final}
FinSi

NOTA : Si la función de transición δ no es completa, lo que tenemos que hacer para poder aplicar el algoritmo es totalizarla, por ejemplo mediante agregar un estado pozo.

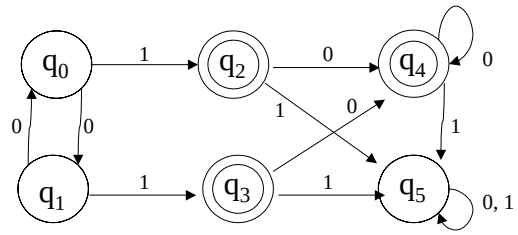
Siguiendo con el procedimiento, se tiene que armar M'
Q' : C_k / $C_k \in \Pi_{final}$
q_0' : C_k / $q_0 \in C_k$ (en el ejemplo, $q_0' = q_0\ q_1$)
F' : C_k / $C_k \cap F \neq \phi$
$\delta'(C_k, a) = C_s$ si $\delta(q, a) = p$, $a \in \Sigma$, $q \in C_k$ y $p \in C_s$

Terminando el paso 2 del algoritmo,
Π_1 $C_1 = \{q_0\ q_1\}$ $C_2 = \{q_5\}$ $C_3 = \{q_2\ q_3\ q_4\}$

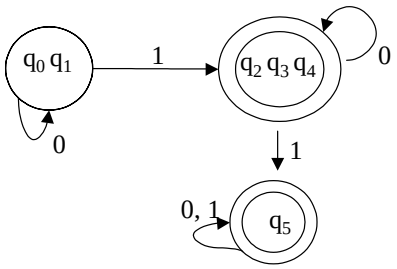
Como $\Pi_0 \neq \Pi_1$ se tiene que aplicar nuevamente el paso 2 con Π_1 y obtener un Π_2

Π_2 $C_1 = \{q_0\ q_1\}$ $C_2 = \{q_5\}$ $C_3 = \{q_2\ q_3\ q_4\}$

Como $\Pi_1 = \Pi_2$, por lo tanto se termina el algoritmo
O sea que partiendo de



obtenemos



Algoritmo de minimización de máquinas secuenciales

(en particular para máquinas secuenciales de Meally)

Meally $M : (Q , \Sigma , \Delta , \delta , \lambda , q_0)$

$\delta : Q \times \Sigma \rightarrow Q$ dada una pareja estado símbolo me dice a que estado voy

$\lambda : Q \times \Sigma \rightarrow \Delta$ dada una pareja estado símbolo me dice que símbolo se imprime

Def.: Función respuesta $f: Q \times \Sigma^* \rightarrow \Delta^*$ (λ : Meally λ':Moore)

(es para independizarnos de la máquina)

$$f(q, x) = \begin{cases} \overset{\wedge}{\lambda}(q, x) \\ \lambda'(q, x) \end{cases}$$

O sea que unificamos en una única función "respuesta" la función lambda (λ)

Def. Equivalencia de estados : Dada una máquina secuencial MS, 2 estados p y q $\in$ Q, se dice que pEq sii $\forall$ x $\in \Sigma^*$, f(p, x) = f(q, x)

O sea que dos estados son equivalentes si emiten la misma salida (respuesta) para la misma entrada (para cualquier tira de Σ^*).

También se puede hablar de estados equivalentes en función del largo de la tira

Def. Equivalencia de largo n : pE_nq sii $\forall$ x $\in \Sigma^*$ / |x| = n, f(p, x) = f(q, x)

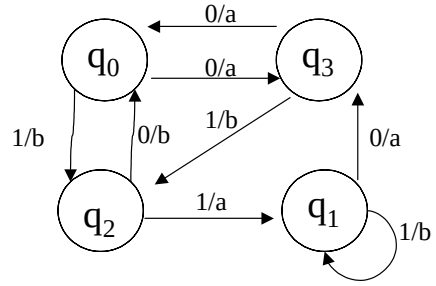
Podría pasar que 2 estados fueran equivalentes en largo 1 y no lo fueran en largo2, por lo tanto no serían equivalentes.

Def.: Sea $M_1 : (Q_1 , \Sigma , \Delta , \delta_1 , \lambda_1 , q_0)$

$M_2 : (Q_2 , \Sigma , \Delta , \delta_2 , \lambda_2 , q_0')$

Las dos máquinas M_1 y M_2 son equivalentes si $\forall$ p $\in Q_1 \wedge$ q $\in Q_2$, pEq (o sea que δ_1 (p, x) = δ_2 (q, x) $\forall$ x $\in \Sigma^*$) y recíprocamente

En estas máquinas con salida la minimización tendrá por objetivo obtener otra



MS pero con menor cantidad de estados.

Los estados serán equivalentes si para la misma tira generan la misma salida.

Π_0 $\forall$ p y q $\in$ subconjuntos de estados Q_1 / λ_1 (p, a) = λ_1 (q, a), $\forall$ a $\in \Sigma$

λ_1 (q_0, 0) = a λ_1 (q_0, 1) = b

λ_1 (q_1, 0) = a λ_1 (q_1, 1) = b

λ_1 (q_2, 0) = b λ_1 (q_2, 1) = a

$\lambda_1 (q_3, 0) = a$ $\lambda_1 (q_3, 1) = b$

$\delta_1 (q_0, 0) = q_3$ $\delta_1 (q_0, 1) = q_2$
$\delta_1 (q_1, 0) = q_3$ $\delta_1 (q_1, 1) = q_1$
$\delta_1 (q_2, 0) = q_0$ $\delta_1 (q_2, 1) = q_1$
$\delta_1 (q_3, 0) = q_0$ $\delta_1 (q_3, 1) = q_2$

hay que repetir para cada par p, q $\in C_i$ $\delta_1 (p, a) = p'$ $\delta_1 (q, a) = q'$ (es igual que para AFD)

Π_0	$\{ q_2 \}$		$\{ q_0, q_1, q_3 \}$
Π_1	$\{ q_2 \}$	$\{ q_0, q_3 \}$	$\{ q_1 \}$
Π_2	$\{ q_2 \}$	$\{ q_0, q_3 \}$	$\{ q_1 \}$

Si $\Pi_i <> \Pi_{i+1}$ hay que seguir. Como en este caso se llega a $\Pi_1 = \Pi_2$ entonces se finaliza.

Cada clase de la Π_{final} es un estado de la máquina secuencial mínima.
$\delta_2 (C_i , a) = C'_i$ si $\delta_1 (p, a) = p'$, $p \in C_i \wedge p' \in C'_i$

q_0 va a ser aquella clase que contenga a q_0
$\lambda_2 (C_i , a) = b$ si $\lambda_1 (q, a) = b$, con $q \in C_i$

PROPIEDADES DE LOS LENGUAJES REGULARES

PUMPING LEMMA

Sea L un lenguaje regular, entonces existe "n" natural que depende del lenguaje L en donde para toda tira perteneciente a L cuyo largo sea mayor o igual a "n", existe al menos una (puede existir más de una) descomposición de esta tira en 3 sub tiras u v w que cumplen que:

1) $| u v | \leq n$
2) $|v| \geq 1$
3) $\forall i \geq 0 \quad u v^i w \in L$

H) L regular

T) $\exists n / \forall z \in L, |z| \geq n, \exists$ al menos una descomposición z = u.v.w tal que :

i) $| u v | \leq n$
ii) $|v| \geq 1$
iii) $\forall i \geq 0 \quad u v^i w \in L$

Demostración:

L es regular, por lo tanto existe un AFD M : $(Q , \Sigma, \delta , q_0 , F) / L = \mathcal{L}(M)$

La cantidad de estados del autómata es n $\quad |Q| = n$

Consideramos arbitrariamente una tira z cualquiera de L cuya cantidad de símbolos es mayor o igual a "n"

$z = a_1, a_2, ..., a_m$ con $a_i \in \Sigma$

$|z| = m$ con $m \geq n$

$\delta (q_0, \varepsilon) = q_0$

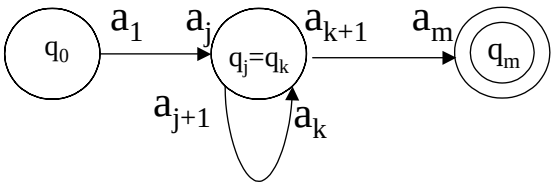
$\delta (q_0, a_1, a_2, ..., a_t) = q_t$

$\delta (q_0, a_1, a_2, ..., a_m) = q_m$

Para reconocer la tira $a_1, a_2, ..., a_m$ paso por m + 1 estados, que pueden repetirse.

$|\{ q_1, q_2, ..., q_m \}| = m + 1 > n$ son los estados por lo que pasa cuando consume la tira, pueden haber estados repetidos.

Asumimos que j y k es el estado que se repite, j < k ($q_j = q_k$ es el estado que se repite)



A partir de esto definimos las sub-tiras u v w en que vamos a partir a z

$u = a_1, a_2, ..., a_j$

$v = a_{j+1}, a_{j+2}, ..., a_k$

$w = a_{k+1}, a_{k+2}, ..., a_m$

1) $| u\ v | \leq n$ $| a_1, a_2, ..., a_j , a_{j+1}, ..., a_k | = k \leq |Q| = n$ desde a_1 hasta a_k pasé a lo sumo por $|Q|$ estados

2) $| v | = | a_{j+1}, ..., a_k | \geq 1$, ya que $j < k$

3) $\forall\ i \geq 0$ $u\ v^i\ w \in L$

$\hat{\delta}(q_0, u) = \hat{\delta}(q_0, a_1, a_2, ..., a_j) = q_j$

$\hat{\delta}(q_j, v) = \hat{\delta}(q_j, a_{j+1}, a_{j+2}, ..., a_k) = q_k = q_j$ o sea que $\hat{\delta}(q_j, v^i) = q_j$

$\hat{\delta}(q_k, w) = \hat{\delta}(q_j, w) = \hat{\delta}(q_j, a_{k+1}, a_{k+2}, ..., a_m) = q_m$

$\hat{\delta}(q_0, uv^iw) = \hat{\delta}(q_j, v^iw) = \hat{\delta}(q_j, w) = q_m \qquad \forall\ i \geq 0$

$q_m \in F$ ya que q_m es el estado que marcamos como estado final
Es de notar que en ningún momento importó cuanto es "i"
Si tengo uv^0w es la tira uw , no hago el ciclo j – k.

Ejemplo : $L = \{x / x$ es de la forma $0^k 1^k$ con $k \geq 0\ \}$
Este lenguaje no es regular, para probarlo se usa el contra recíproco del Pumping Lemma:

Contra recíproco del Pumping Lemma:

H) Dado un lenguaje L, $\forall\ n \in N\ \exists\ z \in L$, $|z| \geq n$, para toda descomposición $z = u\ v\ w$ alguna de las siguientes tres condiciones no se cumple

1) $| u\ v | \leq n$
2) $|v| \geq 1$
3) $\forall\ i \geq 0$ $u\ v^i\ w \in L$

T) entonces L no es regular.

En la práctica se usa que las condiciones 1 y 2 se cumplen y falla la 3.

Pasos a seguir para aplicar el contra recíproco del Pumping Lemma:

1) Elijo un n
2) Elijo una descomposición z que pertenezca al lenguaje, en función del n, de tal forma que $|z| \geq n$
   Por ejemplo, $L = \{0^k 1^k$ con $k \geq 1\ \}$
   $z = 0^n 1^n$ $\quad |z| = 2n \geq n$
3) Ahora debería estudiar toda posible descomposición de z y esto complica el análisis, y la solución en este caso sería utilizar descomposiciones de z que verifican las condiciones 1) y 2), y en todo el resto de las posibles descomposiciones de z no se va a cumplir alguna de esas 2 propiedades 1) o 2).

Una posibilidad para esto sería tomar
$u = 0, v = 0$ y $w = 0^{n-2} 1^n$
donde $|uv| = 2 < n$, $|v| = 1 \geq 1 \Rightarrow$ se verifican las condiciones 1) y 2).

Otra posibilidad es
$u = \varepsilon, v = 0$ y $w = 0^{n-1}1^n$
donde $|uv| = 1 < n$, $|v| = 1 \geq 1 \Rightarrow$ se verifican las condiciones 1) y 2).

Planteándolo en forma genérica
$u = 0^p, v = 0^q$ y $w = 0^{n-p-q} 1^n$ con $q \geq 1$ y $p+q \leq n$ se verifican las condiciones 1) y 2).
Así que a estos casos tengo que probarles que no se cumple la tercera condición, ya que se cumplen las 2 primeras.

$z_i = u v^i w = 0^p (0^q)^i 0^{n-p-q} 1^n = 0^p 0^{iq} 0^{n-p-q} 1^n = 0^{(i-1)q+n} 1^n$
vemos que para i = 0 tenderemos
$z_0 = 0^{n-q} 1^n$ y esta tira no pertenece a L ya que $q \geq 1$ y la cantidad de ceros sería menor que la cantidad de unos.

Veamos ahora otras descomposiciones que cumplan con las condiciones 1 y 2, y veremos que no hay otra que las cumpla. Esto tiene que ver con elegir inteligentemente el z, por ejemplo el visto $z = 0^n 1^n$. Entonces, cualquier otra descomposición no verifica alguna de las primeras 2 condiciones.
Por ejemplo,
$u = 0^{n-p}, v = 0^p 1^p$ $w = 1^{n-p}$. Para i = 0, verifica la tercer condición.
Pero para i = 3, $0^{n-p} (0^p 1^p)^3 1^{n-p}$ no la verifica.
NOTA: Es un error frecuente en las pruebas el tomar $(0^p 1^p)^3$ como $0^{3p} 1^{3p}$ cuando en realidad es $0^p1^p 0^p1^p 0^p1^p$

Como fin de la demostración hay que poner una cláusula que diga:
Estas son todas las descomposiciones que cumplen con las condiciones 1 y 2 y que cualquier otra descomposición no cumple con alguna de estas, y por lo tanto L no es regular.

Es fundamental incluir esta cláusula de clausura, ya que sino se considera que está mal.

Una de las habilidades a considerar en un ejercicio de este tipo es la elección del z.

Tomemos por ejemplo

$Z = 0^{[n/2]} 1^{[n/2]}$, donde $[n/2]\,[n/2] \geq n$

$$\overbrace{\underbrace{00000}_{u}\underbrace{....00}_{v}\underbrace{111}_{w}.....1}^{[n/2]\qquad[n/2]}$$

1) $u = 0^p$ $pq \le n$
$v = 0^q$ $q \ge 1$
$w = 0^{[n/2] - p - q}\ 1^{[n/2]}$
$z_i = uv^iw = 0^p\ (0^q)^i\ 0^{[n/2] - p - q}\ 1^{[n/2]} = 0^p\ 0^{iq}\ 0^{[n/2] - p - q}\ 1^{[n/2]}$
$= 0^{p + iq + n/2 - p - q}\ 1^{[n/2]} = 0^{\,n/2 + q\,(i - 1)}\ 1^{[n/2]}$
Para $i = 2 \Rightarrow z_2 = 0^{\,n/2 + q}\ 1^{[n/2]}$, con $q \ge 1 \Rightarrow$ la cantidad de 0 es mayor que la de 1 $\Rightarrow z_2$ no pertenece a L

Todavía no puedo afirmar que L no es regular, porque primero hay que probar que esto se cumple para toda descomposición.

00000....00111.....1
u v w

1) $u = 0^{[n/2] - q}$ $n/2 + r \le n$
$v = 0^q\ 1^r$ $q + r \ge 1$
$w = 1^{[n/2] - r}$
$z_i = uv^iw = 0^{[n/2] - q}\ (0^q\ 1^r)^i\ 1^{[n/2] - r}$
OJO: $(0^q\ 1^r)^i$ *NO ES* $= 0^{qi}\ 1^{ri}$, es $0^q1^r\ 0^q1^r \ldots 0^q1^r$ i veces
Para $i = 2 \Rightarrow z_2 = 0^{\,n/2 - q}\ 0^q1^r\ 0^q1^r\ 1^{n/2 - r}$

El caso de r = 0 está contemplado en el anterior, $\Rightarrow$ L NO ES REGULAR

Se puede demostrar más fácil si se elige mejor la tira:

$Z = 0^{2N}\ 1^{2N}$ $|uv| \le N$
o
$Z = 0^N\ 1^N$ $|v| \ge 1$

Hay un solo elemento de uvw para demostrar

N N
00000....00111.....1
u v w

Ejemplo : $L = \{(ab)^k\ c^k\ , k \ge 1\}$

abababababab..... ababccccccc.....ccc
u v w
abababababab..... ababccccccc.....ccc
u v w

o sea que hay varias descomposiciones para plantear
$z = (ab)^n\ c^n$
$u = (ab)^{n - j}$ $v = (ab)^j$ $w = c^n$ El problema es que esta descomposición no cumple con $| u\ v | \le n$ ya que en $z = (ab)^n\ c^n$ $|z| = 3n$

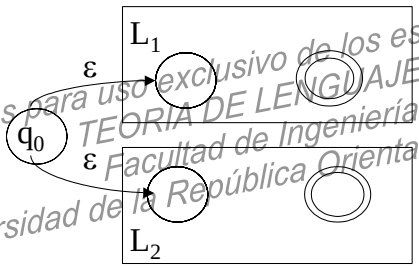
Lo correcto es tomar
$u = (ab)^p$ $v = (ab)^q$ $w = (ab)^{n-p-q} c^n$ con $q \geq 1$ y $2(p+q) \leq n$ Con estas restricciones se cumple 1 y 2. O sea que recién tengo una familia de descomposiciones posibles que cumplen las condiciones 1 y 2, y tengo que seguir
$u = (ab)^p$ $v = (ab)^q a$ $w = b\,(ab)^{n-p-q-1} c^n$

$u = (ab)^p a$ $v = b\,(ab)^q$ $w = (ab)^{n-p-q-1} c^n$

$u = (ab)^p a$ $v = b\,(ab)^q a$ $w = b\,(ab)^{n-p-q-2} c^n$

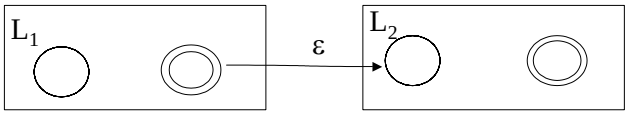
Propiedad UNION

Tenemos L1, L2 lenguajes regulares.
La unión de dos lenguajes regulares es un lenguaje regular. L1 U L2 es regular.
Como L1 y L2 son regulares, tienen un ER que los define, ⇒ L1 U L2 tiene una ER que lo define que es r1|r2.
Si tengo AF en lugar de ER, igualmente puedo construir un AF para L1UL2



Propiedad CONCATENACIÓN

La concatenación de dos LR es un LR
L1.L2 se hace r1.r2



Si tengo AF en lugar de ER se hace

Y se definen los estados iniciales de L1.L2 como los iniciales de L1, y los finales son los de L2.

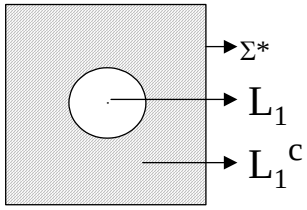
Propiedad CLAUSURA DE KLEENE

La clausura de Kleene L1* de un lenguaje regular L1 es un lenguaje regular. Si tengo r1, se hace r1*

Propiedad COMPLEMENTO

Si L1 es regular, el complemento es regular.
$L1 \subseteq \Sigma^*$

M : $(Q , \Sigma, \delta , q_0 , F)$ / $L1 = \mathcal{L}(M)$
M': $(Q', \Sigma, \delta', q_0', F')$ / $L1^c = \mathcal{L}(M')$

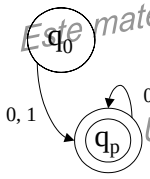


$Q' = Q$
$F' = Q - F$
$\delta' = \delta$
$q_0' = q_0$

En el AF se intercambian los estados finales y los que no lo son. El único problema en este caso es que tenemos que asegurarnos que la función δ sea total (es decir que estén representadas todas las transiciones).

Ej. 

Este AF reconoce la tira { ε }, pero como no es completo si solo hago el complemento del estado, $L1^c$ no reconoce las tiras Σ^* - { ε }. Para hacer el complemento tenemos que considerar (o agregar) un estado pozo.

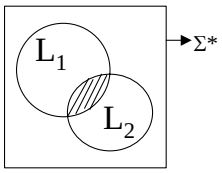


O lo que es igual, $F' = Q - F \cup q_p$ (estados pozo)

Propiedad INTERSECCIÓN

La intersección de dos lenguajes regulares es un lenguaje regular.

$L3 = L1 \cap L2$

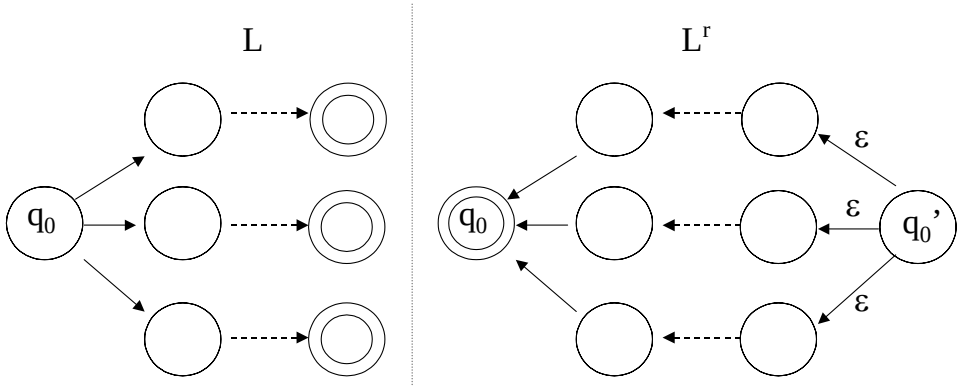


Ahora, como L1 y L2 son conjuntos, se puede hacer
$L3 = (L1^c \cup L2^c)^c$ (propiedad de Morgan de Teoría de Conjuntos)
Luego, como L1 es regular, el complemento es regular
Como L2 es regular, el complemento es regular
Como la unión de lenguajes regulares es regular, $L1^c \cup L2^c$ y como el complemento es regular, L3 es regular.

Propiedad REVERSO

El reverso de una tira es la tira leída al revés. Si el lenguaje L1 es regular, el lenguaje reverso (que reconoce las tiras leídas al revés) también es regular.

$x = a_1\ a_2\ ....\ a_m$
$x^r = a_m\ a_{m-1}\ ....\ a_1$
$L^r = \{ x^r / x \in L \}$

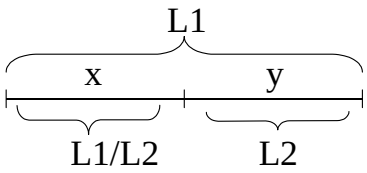


Lo que se hace es cambiar la orientación de las flechas de las transiciones, se pone al estado inicial de L como estado final de L^r y se agrega un nuevo estado inicial a L^r desde donde se hacen transiciones ε a los estados finales de L.

Propiedad COCIENTE

El cociente de L1 y L2 es el conjunto de los x tal que ∃ y ∈ L2 y xy ∈ L1.

$L3 = L1|L2 = \{ x / \exists\ y \in L2,\ xy \in L1 \}$



Si $\varepsilon \in L2, \Rightarrow L1 \subseteq L1 / L2$
Ej.: 1*|1* = 1*
1*01*|0 = 1* porque 0 concatenado con 1* es 1*0, que pertenece a L1
1*01*|00 = { }
Si L1 es un lenguaje regular, L2 es un lenguaje cualquiera (no necesita ser regular), entonces L3 = L1 | L2 es regular.

Propiedad SUSTITUCIÓN

Es una función que va de símbolos de un cierto alfabeto a lenguajes (lo denotamos como Δ^*)

$f : \Sigma \rightarrow \Delta^*$
f(0) = {a} al 0 se le asocia el lenguaje a.
f(1) = b* le asocia las tiras formadas por b.

$f(\varepsilon) = \varepsilon$
f(xa) = f(x) . f(a)

La sustitución de un lenguaje es la unión de cada una de las tiras del lenguaje.
$f(L) = \bigcup_{x \in L} f(x)$

Propiedades de la Sustitución

f(L1 U L2) = f(L1) U f(L2)
f(L1.L2) = f(L1).f(L2)
f(L1*) = (f(L1))*

ej.: f(0) = {a}
f(1) = b*
f(010) = ab*a ya que f(010) = f(0) . f(1) . f(0) = a b* a
f(0*1*) = (f(0))*.(f(1))* = a* b*

Propiedad HOMOMORFISMO

Es un caso particular de sustitución. La diferencia con esta es que la función no da como resultado un lenguaje sino que es una cadena de caracteres.

$h : \Sigma \rightarrow \Delta^*$ h(0) = aa h(1) = aaa

Los lenguajes regulares son cerrados bajo el homomorfismo.

Propiedad HOMOMORFISMO INVERSO

$h^{-1} : \Delta^* \rightarrow \Sigma^*$

Primero tenemos que tener definido un homomorfismo.
$h^{-1}(y) = \{x / h(x) = y\}$

$h^{-1}(L) = \{x / h(x) \in L\}$

h(0) = aa h(1) = aaa

$h^{-1}(a^6) = \{11,000\}$, ya que h(11) = aaaaaa y h(000) = aaaaaa

La propiedad dice que dado un lenguaje regular y un homomorfismo, el homomorfismo inverso sobre este es un lenguaje regular.

IMPORTANTE : Si L es regular, L' es regular. Por esto podemos afirmar que si L' no es regular, entonces L no es regular. Ahora, si L' es regular NO PODEMOS AFIRMAR QUE L sea regular.

Ejemplo de cómo podemos utilizar estas propiedades:
$L = \{ 0^i 1^j 2^k / i = j + k \text{ con } i, j, k \geq 0\}$

Este lenguaje reconoce las tiras que tienen igual cantidad de 0 que de 1 y 2.
Supongamos que L es un lenguaje regular.
$h:\{0, 1, 2\} \rightarrow \{a, b\}$
Si tomamos el homomorfismo definido por h(0) = a, h(1) = b, h(2) = b
$\Rightarrow h(L) = \{a^k b^k, k \geq 0 \}$ que como vimos anteriormente NO ES REGULAR,
$\Rightarrow$ por la propiedad de homomorfismo inverso podemos afirmar que L no es regular.

GRAMÁTICAS

LENGUAJES : Están compuesto por símbolos (alfabeto) y reglas.

Ejemplo :
```
Function frutaid (n:integer):integer;
    begin
        -----
        -----
    end
```

Reglas :
Fun → function IDENT (LPARAM):TIPO; cuerpo
LPARAM → IDENT : IDENT, LPARAM
LPARAM → ε

El conjunto de símbolos y reglas que permiten definir un lenguaje es la gramática. En el manual de cada lenguaje de programación se detalla la gramática del mismo (Ej. Pascal, Cobol, etc.)

La gramática surge para definir lo que se llama PARSER, o analizador sintáctico. Este analiza si el programa que desarrollamos es correcto, y para esto se basa en las reglas definidas en la gramática. Va construyendo un árbol y analizando este al final se podrá decir su es correcto o no.

Por ejemplo, XML es un metalenguaje que define como se estructurará un documento.

Lo primero que haremos es definir un tipo de gramáticas que se llaman libres de contexto o independientes del contexto

Gramática Libre del Contexto

G:(V, T, P, S) donde

V = Conjunto de símbolos que le vamos a llamar Símbolos Variables. Son aquellos que me ayudan a definir un lenguaje, a partir de los cuales se define este. En el ejemplo anterior, el símbolo IDEN se usa para definir que en el lenguaje final se debe poner un identificador, que debe cumplir con las reglas del símbolo IDENT. De alguna forma esto estaría definiendo en sí mismo un lenguaje, que es el que define a los identificadores.

T = Conjunto de símbolos que se denominan Terminales. Son los que corresponden al alfabeto, son los finales. Por ejemplo, serían las frases finales del lenguaje.

S = Símbolo Variable ($S \in V$) que identifica al símbolo inicial o principal. Es aquel a partir del cual se construye todo el lenguaje. En el ejemplo anterior sería Fun, ya que a partir de este sed define el resto.

P = Reglas del lenguaje (reglas de producción). Todas las reglas tienen la siguiente estructura :
$P : A \rightarrow \alpha$, donde $A \in V$ y $\alpha \in (V \cup T)^*$
Del lado izquierdo SOLO pueden haber UN símbolo variable. Del lado derecho pueden existir cualquier cantidad de combinaciones de símbolos variables y terminales, incluido el ε.

DERIVACIÓN :

En una gramática llamaremos derivar a pasar de una determinada configuración a otra, en base a las reglas de producción.

$\exists A \to \beta \in P, A \in V, \alpha, \gamma \in (V \cup T)^*$

$\alpha A \gamma \Rightarrow \alpha \beta \gamma$ donde $\Rightarrow$ es el símbolo que representa a la derivación.

Ej.: En el ejemplo anterior, empezamos aplicando la derivación en la 1° regla.

Fun $\Rightarrow$ $\underbrace{\text{function IDENT}}_{\alpha}$ $\underbrace{\text{(LPARAM)}}_{A}$$\underbrace{\text{:TIPO;cuerpo}}_{\gamma}$ $\Rightarrow$

$\Rightarrow$ $\underbrace{\text{function IDENT}}_{\alpha}$ $\underbrace{\text{(IDENT:TIPO,LPARAM)}}_{\beta}$$\underbrace{\text{:TIPO;cuerpo}}_{\gamma}$

Si seguimos así, podemos hacer Fun $\Rightarrow^*$ x, que significa que derivando "n" veces a partir de Fun obtengo un x, que es una frase de mi lenguaje y debe estar compuesto únicamente por símbolos terminales. Si no llego, entonces no pertenece al lenguaje.

Ej: Σ^* , $\Sigma = \{0, 1\}$

Def. recursiva de Σ^* 1) $\varepsilon \in \Sigma^*$

2) $ax \in \Sigma^*$, si $a \in \Sigma$ y $x \in \Sigma^*$

3) Estas son todas las tiras del lenguaje.

Las reglas de producción serían :

$S \to \varepsilon$

$S \to 0S$

$S \to 1S$

y para derivar aplicamos cualquier regla, según lo que se quiera obtener.

Si aplico $S \to \varepsilon$, obtengo la tira vacía : $S \Rightarrow \varepsilon$

Si aplico $S \to 0S$, y luego $S \to \varepsilon$, obtengo la tira 0 : $S \Rightarrow 0S \Rightarrow 0$

Si aplico $S \to 1S$, luego $S \to 0S$ y luego $S \to \varepsilon$, obtengo la tira 10 :

$S \Rightarrow 1S \Rightarrow 10S \Rightarrow 10$

Def.: Sea una GLC (gramática libre de contexto) G:(V, T, P, S), el conjunto de tiras generado es el conjunto de las tiras pertenecientes a los símbolos terminales que pueden ser derivados a partir del símbolo inicial.

$L = \mathcal{L}(G) = \{x, x \in T^* / S \Rightarrow^* x\}$

Ejemplo: $L = \{0^k 1^k, k>1\}$, las reglas de la gramática serían :

$S \to 01$

$S \to 0S1$

Lo que hay que probar es que $L \equiv \mathcal{L}(G)$, o sea que toda tira de L es generada por la gramática, y que toda tira de la gramática pertenece al lenguaje. O sea

1) $L \subseteq \mathcal{L}(G)$

2) $\mathcal{L}(G) \subseteq L$

Con el 2) me aseguro que por más que aplique cualquier regla de la gramática, la tira generada pertenece al lenguaje.

Para probar 1) se hace Inducción Completa sobre el largo de las tiras de mi lenguaje.|
Para probar 2) se hace Inducción Completa en la cantidad de pasos de las derivaciones (en la cantidad de derivaciones).

NOTACIÓN: como el símbolo $\Rightarrow$ se utiliza para denotar la derivación, se utilizará el símbolo " **»** " para indicar el "**por lo tanto**" o "**entonces**".

1) IC en el largo de x

PB) k=1, 01 $\in$ L, $\exists$ S $\rightarrow$ 01 » se cumple

Hi) |x| = 2k, x $\in$ L » x $\in \mathcal{L}$(G). Se toma |x| = 2k porque las tiras son de largo par.

Ti) |x| = 2k + 2, x $\in$ L » x $\in \mathcal{L}$(G).

x $\in$ L » $x = 0^{k+1}1^{k+1} = 00^k1^k1$. Haremos $y = 0^k1^k$ » x = 0y1 con y $\in$ L,

» |y| = 2k, » por Hi) y $\in \mathcal{L}$(G) , lo que significa que S $\Rightarrow^*$ y. Poniendo 0 y 1 a ambos lados tenemos 0S1 $\Rightarrow^*$ 0y1 » como x = 0y1, 0S1 $\Rightarrow^*$ x. Como $\exists$ una regla S $\rightarrow$ 0S1, S $\Rightarrow$ 0S1 $\Rightarrow^*$ x » S $\Rightarrow^*$ x » x $\in \mathcal{L}$(G) **LQQD**

2) IC en la cantidad de pasos en las derivaciones. Tengo tantas demostraciones como reglas que solo tengan símbolos terminales a la derecha.

PB) S $\rightarrow$ 01, 01 $\in$ L

Hi) $S \Rightarrow^{\leq h} x \in T^*$ » x $\in$ L ($\Rightarrow^{\leq h}$ significa que en h o menos derivaciones se obtiene x partiendo de S)

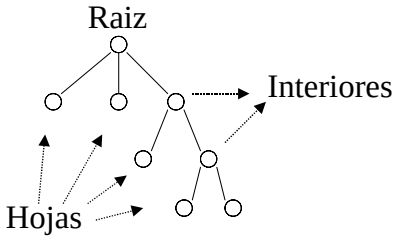
Ti) $S \Rightarrow^{=h+1} x \in T^*$ » x $\in$ L

$S \Rightarrow 0S1 \Rightarrow^h x = 0y1$ » $S \Rightarrow^h y$ además como x $\in T^*$ » y $\in T^*$, » (por Hi) y $\in$ L.

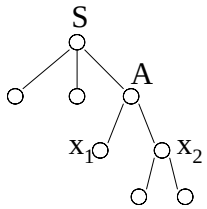
$y = 0^h1^h$ y x = 0y1, » $x = 00^h1^h1 = 0^{h+1}1^{h+1}$ » x $\in$ L **LQQD**

ARBOL DE DERIVACIÓN :

Es una estructura de árbol donde los nodos (o vértices) interiores son siempre variables, mientras que las hojas son siempre terminales o ε. La raíz del árbol es el símbolo inicial.



Que un nodo A tenga “hijos” (x_1, x_2, .., x_n) significa que $\exists$ la regla $A \rightarrow (x_1, x_2, .., x_n)$ con $A \in P$



Ejemplo: Supongamos que tenemos una gramática con las siguientes reglas (representaría el álgebra de Boole).

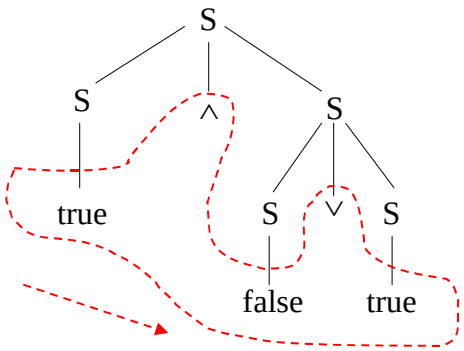
$S \rightarrow S \wedge S$
$S \rightarrow S \vee S$
$S \rightarrow (S)$
$S \rightarrow true$
$S \rightarrow false$

Esto se puede representar también juntándolo en uno o varios renglones utilizando el “pipe” (|) como operador “or”.

$S \rightarrow S \wedge S \mid S \vee S \mid (S) \mid true \mid false$

Supongamos que queremos hallar el árbol de derivación asociado a la tira

true $\wedge$ false $\vee$ true



Las hojas leídas de izquierda a derecha es la tira que se quiere representar. Para construir el árbol lo que se hace es aplicar las reglas de la gramática.

Teorema : (Marca la relación entre una derivación y el árbol de derivación).
Sea GLC : (V, T, P, S)
$\forall x \in T^*$, $S \Rightarrow^* x$ si y solo si $\exists$ un árbol de derivación para x.
Ambas demostraciones (para cada lado del si y solo si) se hacen por IC, para un lado se hace sobre las reglas de derivación, y para el otro sobre la altura del árbol. No se demostrará en el curso, se deja como ejercicio.

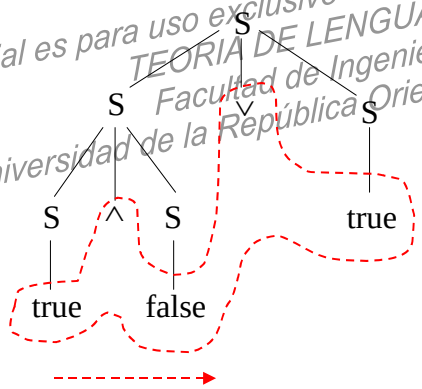
Def.: Derivación de más a la izquierda

Es aquella en la que cada vez que estamos derivando (aplicando las reglas) siempre se va a derivar primero la variable de más a la izquierda.
Para el ejemplo true $\wedge$ false $\vee$ true haríamos :
S $\Rightarrow_1$ S $\wedge$ S $\Rightarrow_4$ true $\wedge$ S $\Rightarrow_2$ true $\wedge$ S $\vee$ S $\Rightarrow_5$ true $\wedge$ false $\vee$ S $\Rightarrow_4$ true $\wedge$ false $\vee$ true
NOTACIÓN : cuando ponemos por ejemplo $\Rightarrow_1$ significa que estamos aplicando la regla 1 de las definidas.

Def.: Derivación de más a la derecha

Es aquella en la que cada vez que estamos derivando (aplicando las reglas) siempre se va a derivar primero la variable de más a la derecha.
Para el ejemplo true $\wedge$ false $\vee$ true haríamos :
S $\Rightarrow_2$ S $\vee$ S $\Rightarrow_4$ S $\vee$ true $\Rightarrow_1$ S $\wedge$ S $\vee$ true $\Rightarrow_5$ S $\wedge$ false $\vee$ true $\Rightarrow_4$ true $\wedge$ false $\vee$ true
NOTACIÓN : cuando ponemos por ejemplo $\Rightarrow_1$ significa que estamos aplicando la regla 1 de las definidas.

El árbol de derivación anterior se podría construir de otra forma :



Estamos ante un problema de ambigüedad, ya que según como se apliquen las reglas se obtiene un árbol diferente, o bien se pueden aplicar las reglas de varias formas diferentes.

Gramática Ambigua

Se dice que una gramática es ambigua si se puede construir por lo menos uno de los siguientes casos :

1) Se pueden construir 2 árboles de derivación diferentes.
2) Se pueden construir 2 derivaciones de más a la izquierda distintas.
3) Se pueden construir 2 derivaciones de más a la derecha distintas.

La salida de un analizador sintáctico es un árbol de derivación. El problema de la ambigüedad esta en que el árbol resultante de aplicar correctamente las reglas puede ser inválido.
Veamos por ejemplo:

S → IF <cond> THEN <sent>
| IF <cond> THEN <sent> ELSE <sent>
| <…..>

Si tenemos la tira IF c1 THEN IF c2 THEN s1 ELSE s2 en este caso, el ELSE ¿se le aplicaría al primer IF o al segundo? Según como lo tome el analizador puede ser válida la salida pero incorrecta, ya que el ELSE se puede aplicar al IF incorrecto.

TEOREMA: Todo lenguaje regular es un Lenguaje Libre de Contexto (generado por una gramática libre de contexto).
Para demostrar esto lo que haremos es demostrar que para todo LR tengo una GLC que lo genera. Como un Lenguaje Regular está definido por una Expresión Regular, para la demostración haremos IC sobre la forma de construcción de la ER (IC en el número de operadores).

Paso BASE

ER	GLC : (V, T, P, S) con V={S}, T={a}
ϕ	P = {}
ε	P = { $S \to \varepsilon$ }
$a \in \Sigma$	P = { $S \to a$ }

Paso INDUCTIVO

Hi) $L_1 = \mathcal{L}(r_1) \Rightarrow \exists\ G_1 : (V_1, T_1, P_1, S_1) / L_1 = \mathcal{L}(G_1)$
$L_2 = \mathcal{L}(r_2) \Rightarrow \exists\ G_2 : (V_2, T_2, P_2, S_2) / L_2 = \mathcal{L}(G_2)$

Ti1) $r_1|r_2$, $L_3 = \mathcal{L}(r_1 \mid r_2) \Rightarrow \exists\ G_3 : (V_3, T_3, P_3, S_3) / L_3 = \mathcal{L}(G_3)$
Ti2) $r_1.r_2$, $L_4 = \mathcal{L}(r_1 . r_2) \Rightarrow \exists\ G_4 : (V_4, T_4, P_4, S_4) / L_4 = \mathcal{L}(G_4)$
Ti3) r_1^* , $L_5 = \mathcal{L}(r_1^*) \Rightarrow \exists\ G_5 : (V_5, T_5, P_5, S_5) / L_5 = \mathcal{L}(G_5)$
G_1, G_2, G_3, G_4, G_5 son GLC

Suponemos que $V_1 \cap V_2 = \varnothing$, ya que siempre podemos hacer un cambio de símbolos para que esto no ocurra. Es para que no se confundan los símbolos de uno con los del otro.

Demostración Ti1) $r_1|r_2$, $L_3 = \mathcal{L}(r_1 \mid r_2) \Rightarrow \exists\ G_3 : (V_3, T_3, P_3, S_3) / L_3 = \mathcal{L}(G_3)$
$V_3 = V_1 \cup V_2 \cup \{S_3\}$
$T_3 = T_1 \cup T_2$
$P_3 = P_1 \cup P_2 \cup \{S_3 \to S_1 , S_3 \to S_2\}$
Si se empieza a ejecutar la derivación por $S_3 \to S_1$ entonces a partir de ahí aplico las reglas de P_1 (que está incluido en P_3). Si empiezo a aplicar las reglas por $S_3 \to S_2$ entonces a partir de ahí aplico las reglas de P_2 (que también está incluido en P_3).

$S_3 \Rightarrow S_1 \Rightarrow^* x$, $x \in L_1$
$S_3 \Rightarrow S_2 \Rightarrow^* x$, $x \in L_2$
Como S_3 solo genera tiras de L_1 o de L_2, entonces L_3 genera tiras de $r_1 \mid r_2$, por lo tanto $L_3 = \mathcal{L}(G_3)$

Demostración Ti2) $r_1 . r_2$, $L_4 = \mathcal{L}(r_1 . r_2) \Rightarrow \exists\ G_4 : (V_4, T_4, P_4, S_4)$ / $L_4 = \mathcal{L}(G_4)$
$V_4 = V_1 \cup V_2 \cup \{S_4\}$
$T_4 = T_1 \cup T_2$
$P_4 = P_1 \cup P_2 \cup \{S_4 \rightarrow S_1\ S_2\}$
$S_4 \Rightarrow S_1\ S_2 \Rightarrow^* w_1\ S_2 \Rightarrow^* w_1\ w_2$, $w_1 \in L_1$ y $w_2 \in L_2$
Por lo tanto, $S_4 \Rightarrow^* w_1\ w_2$ o sea que L_4 genera la concatenación de tiras de L_1 y L_2 , por lo tanto $L_4 = \mathcal{L}(G_4)$.

Demostración Ti3) r_1^* , $L_5 = \mathcal{L}(r_1^*) \Rightarrow \exists\ G_5 : (V_5, T_5, P_5, S_5)$ / $L_5 = \mathcal{L}(G_5)$
$V_5 = V_1 \cup \{S_5\}$
$T_5 = T_1 \cup T_2$
$P_5 = P_1 \cup \{S_5 \rightarrow \varepsilon \mid S_1\ S_5\}$
Con la primera regla aseguramos que cumple con aceptar la tira $\{\varepsilon\}$ que requiere la clausura de Kleene, pero también aseguramos que la recursión definida en $S_5 \rightarrow S_1\ S_5$ termina en algún momento.

Con esto demostramos que todo lenguaje regular es libre de contexto. El recíproco no se cumple, es decir que todo lenguaje libre de contexto no es un lenguaje regular. Para probar esto usamos un contraejemplo: 0^N1^N, que como vimos se puede definir una GLC que lo genere, pero no es regular, como vimos en la propiedad de Pumping Lema.

Gramáticas regulares

Def. Lineales derechas

Se dice que una gramática libre de contexto G: (V, T, P, S) que es de la forma $A \rightarrow w \mid wB$, A y B $\in$ V y w $\in$ T*, es una gramática regular lineal derecha. Del lado izquierdo de las reglas son SOLO variables, mientras que del lado derecho puede tener a lo sumo una única variable y esta debe ser el último símbolo (el de más a la derecha).

Def. Lineales izquierdas

Se dice que una gramática libre de contexto G: (V, T, P, S) que es de la forma $A \rightarrow w \mid Bw$, A y B $\in$ V y w $\in$ T*, es una gramática regular lineal izquierda. Del lado izquierdo de las reglas son SOLO variables, mientras que del lado derecho puede tener a lo sumo una única variable y esta debe ser el primer símbolo (el de más a la izquierda).

Si una gramática libre de contexto lineal izquierda o derecha genera un lenguaje regular

Ejemplo : (0 | 1)*
Una gramática lineal derecha para este LR sería:
$A \rightarrow 0A \mid 1A \mid \varepsilon$

Una gramática lineal izquierda para este LR sería:
$A \rightarrow A0 \mid A1 \mid \varepsilon$

Cuando en los ejercicios se pide que se construya una gramática para un lenguaje regular, se está pidiendo que se haga una gramática regular, o sea una gramática lineal izquierda o lineal derecha.

Ejemplo : 001*0
Una gramática lineal izquierda sería
$A \rightarrow B0$
$B \rightarrow C \mid B1$
$C \rightarrow 00$

NOTA : Lo que tenemos que asegurar en estos casos es tener una regla que haga que pare la recursión.

Una gramática lineal derecha sería
$A \rightarrow 00B$
$B \rightarrow 0 \mid 1B$

Def.: Variable Positiva

Una variable se dice que es positiva, en una GLC, si existe alguna derivación que parta de ella y llegue a una tira válida, o sea que se cumpla que
$A \Rightarrow^* w$, con $w \in T^*$

$S \rightarrow 0S1 \mid 0D \mid BC$
$A \rightarrow 11A \mid 1$
$B \rightarrow B0 \mid 0$
$C \rightarrow 00C$
$D \rightarrow D0 \mid E \mid \varepsilon$
$E \rightarrow 0$

En este caso, C no es una variable positiva ya que la derivación no terminaría nunca, nunca se llega a una tira válida.

Def.: Variable Alcanzable

En una GLC una variable se dice que es alcanzable desde el símbolo inicial si existe alguna derivación que partiendo de S (símbolo inicial) se llega a esta, o sea que se cumpla que
M es alcanzable si $S \Rightarrow^* \alpha M \beta$, con $M \in V$ y $\alpha, \beta \in (V \cup T)^*$

En el ejemplo anterior, A no es alcanzable.

Def.: Variable Útil

En una GLC una variable se dice que es útil si interviene en alguna derivación que obtenga una tira del lenguaje
M es útil si $S \Rightarrow^* \alpha M \beta \Rightarrow^* w$, con $M \in V$, $\alpha, \beta \in (V \cup T)^*$, $w \in T^*$

Una variable útil es alcanzable y positiva. El recíproco no se cumple. En el ejemplo anterior B es alcanzable y positiva, pero no es útil ya que como está junto con C, si aplicamos la regla $S \rightarrow BC$ no llegaremos a una tira válida del lenguaje ya que en C la derivación no se puede terminar (C no es una variable positiva).

Def.: Variable Anulable

En una GLC una variable se dice que es anulable si desde ella se puede derivar la tira vacía.
M es anulable si $M \Rightarrow^* \varepsilon$, con $M \in V$

A menos que el lenguaje acepte las tiras vacías como válidas, buscaremos que no hayan reglas anulables. Esto significa no tener Producciones-ε. Si el lenguaje acepta la tira vacía, la única regla que tendremos será $S \rightarrow \varepsilon$.

Def.: Producción Unitaria

Son las reglas de la GLC que del lado derecho solo tienen una variable.
$A \rightarrow B$ con $A, B \in V$

Buscaremos no tener producciones unitarias en nuestra GLC.

Def.: Gramática Simplificada

Son aquellas que no contienen producciones unitarias, no contienen producciones-ε y todos sus símbolos son útiles.

Simplificación de producciones - ε

Lo que hacemos es construir una gramática nueva sin producciones - ε. Se debería probar (no se hará en el curso) es que el lenguaje que genera esta nueva gramática es el mismo que la original, sin las tiras vacías. Esto es probar que $L(G) - \{ \varepsilon \} = L(G')$. La nueva gramática G' no reconoce las tiras vacías, y si es necesario (ya que el lenguaje original si las reconoce), entonces se debe agregar la regla $S \rightarrow \varepsilon$.

Ejemplo : Si tenemos las siguientes reglas de G:

$A \rightarrow BCD$, $B \rightarrow \varepsilon$, $D \rightarrow \varepsilon$, ...

lo que haría el algoritmo es eliminar $B \rightarrow \varepsilon$ y $D \rightarrow \varepsilon$, y agregar las reglas

$A \rightarrow CD$, $A \rightarrow BC$ y $A \rightarrow C$, manteniendo la $A \rightarrow BCD$ (por si hay otras reglas que se deriven de B o D).

ALGORITMO :

1) Eliminar la producción - ε : $M_i \rightarrow \varepsilon$
2) Para toda regla que sea del tipo $A \rightarrow M_1 M_2 ... M_i ... M_n$ con M_i anulable agregar la regla $A \rightarrow M_1 M_2 ... M_{i-1} M_{i+1} ... M_n$
3) Si hay más producciones - ε (o sea anulables) ir al paso 1)
4) Si no hay mas $\Rightarrow$ FIN

Simplificación de producciones unitarias

Lo que hacemos es construir una gramática nueva sin producciones unitarias.
Se debería probar (no se hará en el curso) es que el lenguaje que genera esta nueva gramática es el mismo que la original.

Ejemplo : Si tenemos las siguientes reglas de G : $A \rightarrow B$, $B \rightarrow \alpha$, ...
Lo que hace el algoritmo es eliminar la producción unitaria $A \rightarrow B$ y para toda regla que tenga B del lado izquierdo (como $B \rightarrow \alpha$) hago $A \rightarrow \alpha$. Igualmente se mantiene la regla $B \rightarrow \alpha$.

ALGORITMO :

1) Eliminar la producción unitaria : $A \rightarrow B$
2) Para toda regla que sea del tipo $B \rightarrow \alpha$ agregar la regla $A \rightarrow \alpha$. Si se llega a una regla del tipo $A \rightarrow A$, estas se eliminan.
3) Si hay más producciones unitarias ir al paso 1)
4) Si no hay mas $\Rightarrow$ FIN

Eliminación de variables no positivas

Lo que hacemos es construir una gramática nueva que contenga únicamente variables positivas.
Se debería probar (no se hará en el curso) es que el lenguaje que genera esta nueva gramática es el mismo que la original.

G : (V, T, P, S) $\rightarrow$ G' : (POSITIVAS, T, P', S)

S (símbolo inicial) es el mismo en ambas gramáticas, ya que si S en G no es una variable positiva, el lenguaje original no genera nada, salvo el conjunto vacío.

Hay que construir el conjunto {POSITIVAS}, y lo empezamos con aquellas variables tales que existe una regla en P que la tengan en el lado izquierdo y donde el lado derecho son todos terminales. Luego iremos agregando aquellas variables para las que exista una regla en P que la contengan en el lado

izquierdo y donde el lado derecho son todos terminales o variables que ya pertenezcan a {POSITIVAS}.

ALGORITMO :

1) Construir {POSITIVAS} = { A / A $\in$ V, $\exists$ una regla A $\rightarrow$ w $\in$ P, w $\in$ T*}
2) Para toda variable A que no pertenezca {POSITIVAS}
   Si existe una regla A $\rightarrow \alpha \in$ P / $\alpha \in$ ({POSITIVAS} $\cup$ T)*
   Entonces {POSITIVAS} = {POSITIVAS} $\cup$ { A } (agregar A al conjunto {POSITIVAS}).
3) Si cambió en conjunto {POSITIVAS} ir al paso 2)
4) Si no cambió, entonces hacer P' = {reglas $\in$ P / todas las variables que intervienen $\in$ {POSITIVAS})
5) FIN

En el ejemplo anterior:
P = { S $\rightarrow$ 0S1 | 0D | BC , A $\rightarrow$ 11A | 1 , B $\rightarrow$ B0 | 0 , C $\rightarrow$ 00C , D $\rightarrow$ D0 | E | ε , E $\rightarrow$ 0}

Se haría :
Paso 1) $\{POSITIVAS\}_1$ = {A, B, D, E}
Paso 2) $\{POSITIVAS\}_2$ = {A, B, D, E, S}
Paso 4) P' = { S $\rightarrow$ 0S1 | 0D , A $\rightarrow$ 11A | 1 , B $\rightarrow$ B0 | 0 , D $\rightarrow$ D0 | E | ε , E $\rightarrow$ 0}
y en G' quedaría V' = {POSITIVAS} = {A, B, D, E, S}

<u>Eliminación de variables no alcanzables</u>

Lo que hacemos es construir una gramática nueva que contenga únicamente variables alcanzables.
Se debería probar (no se hará en el curso) es que el lenguaje que genera esta nueva gramática es el mismo que la original.

G : (V, T, P, S) $\rightarrow$ G' : (ALCANZABLES, T, P', S)

S (símbolo inicial) es el mismo en ambas gramáticas, ya que si S es por si mismo alcanzable, y por lo tanto pertenece a G'.

Hay que construir el conjunto {ALCANZABLES}, y lo empezamos asignando el símbolo inicial S. Luego iremos agregando aquellas variables para las que exista una regla en P que contengan del lado derecho variables que ya pertenezcan a {ALCANZABLES}.

ALGORITMO :

1) Construir {ALCANZABLES} = { S }
2) Para toda regla
   Si existe una regla $A \rightarrow \alpha B \beta \in P$ / $A \in$ {ALCANZABLES} , α y $\beta \in T^*$ y $B \notin$ {ALCANZABLES}
   Entonces {ALCANZABLES} = {ALCANZABLES} $\cup$ { B } (agregar B al conjunto {ALCANZABLES}).
3) Si cambió en conjunto {ALCANZABLES} ir al paso 2)
4) Si no cambió, entonces hacer P' = {reglas $\in$ P / todas las variables que intervienen $\in$ {ALCANZABLES})
5) FIN

En el ejemplo anterior:
P = { $S \rightarrow 0S1 \mid 0D \mid BC$, $A \rightarrow 11A \mid 1$, $B \rightarrow B0 \mid 0$, $C \rightarrow 00C$, $D \rightarrow D0 \mid E \mid \varepsilon$, $E \rightarrow 0$}

Se haría :
Paso 1) $\{ALCANZABLES\}_1$ = { S }
Paso 2) $\{ALCANZABLES\}_2$ = { S, D }
Paso 2) $\{ALCANZABLES\}_3$ = { S, D, B }
Paso 2) $\{ALCANZABLES\}_4$ = { S, D, B, C }
Paso 2) $\{ALCANZABLES\}_5$ = { S, D, B, C, E }
Paso 4) P' = { $S \rightarrow 0S1 \mid 0D \mid BC$, $B \rightarrow B0 \mid 0$, $C \rightarrow 00C$, $D \rightarrow D0 \mid E \mid \varepsilon$, $E \rightarrow 0$}
V' quedaría = {ALCANZABLES} = { B, C, D, E, S}

Orden de aplicación de los algoritmos

Es importante considerar el orden en que se aplican los algoritmos para lograr una gramática simplificada. Por ejemplo, no se puede aplicar el algoritmo para eliminar las producciones unitarias antes del utilizado para eliminar las producciones - ε, ya que este último puede a su vez generar producciones unitarias, lo que obligaría a aplicar el algoritmo para eliminar las producciones unitarias nuevamente. Algo similar ocurre con las variables no positivas y las no alcanzables, ya que el algoritmo para eliminar las variables no positivas puede generar variables no alcanzables, y si se aplican en orden inverso luego será necesario aplicar nuevamente la eliminación de las variables no alcanzables. Es de notar que el algoritmo para eliminar las producciones unitarias no genera producciones - ε, y el de eliminar variables no alcanzables no genera variables no positivas.

1) Se aplica el algoritmo para eliminar las producciones - ε.
2) Se aplica el algoritmo para eliminar las producciones unitarias.
3) Se aplica el algoritmo para eliminar las variables no positivas.
4) Se aplica el algoritmo para eliminar las variables no alcanzables.

G	$\rightarrow$	G'	$\rightarrow$	G''	$\rightarrow$	G'''	$\rightarrow$	$G_{simplificada}$
elimino		elimino		elimino		elimino		
prod - ε		prod unit.		var. no pos		var. no alcan.		

Aplicando los algoritmos anteriores en orden al ejemplo anterior, quedaría

$S \rightarrow 0S1 \mid 0D \mid 0$

$D \rightarrow D0 \mid 0$

Este ejemplo generaría el lenguaje $0^n\ 1^m$ con $n > m \geq 0$. Para demostrar esto se debería demostrar que L = L(G), para ambos lados.

Normalización

Una gramática se dice que está normalizada cuando está en alguna de las siguientes formas normales:

Forma Normal de Chomsky

Una gramática está en Forma Normal de Chomsky (FNC), cuando todas sus reglas son de la forma :

$A \rightarrow BC$, $B \rightarrow a$ con $a \in T$, A, B y $C \in V$

O sea que las reglas tienen del lado derecho o bien símbolos terminales o bien 2 variables.

Teorema de Normalización de Chomsky

Cualquier lenguaje libre de contexto que no contenga producciones ε puede ser generado por una gramática que esté en la forma normal de Chomsky (FNC).

La idea para normalizar una gramática como la del ejemplo anterior y llevarlas a una FNC sería :

1) Para cada regla que no esté en FNC sustituir cada terminal por una variable y una regla desde la variable al terminal. Sustituir i por T_i y agregar la regla $T_i \rightarrow i$.

   $S \rightarrow 0S1 \mid 0D \mid 0$

   $D \rightarrow D0 \mid 0$

   Lo hacemos

   $S \rightarrow T_0ST_1 \mid T_0D \mid 0$

   $D \rightarrow D\ T_0 \mid 0$

   $T_0 \rightarrow 0$

   $T_1 \rightarrow 1$

2) Para las reglas que tengan del lado derecho más de 2 variables hacer un nuevo cambio de variable agrupando de a 2 y generando nuevas reglas.

   Aplicando a las anteriores se haría

   $S \rightarrow AT_1 \mid T_0D \mid 0$

   $D \rightarrow D\ T_0 \mid 0$

   $T_0 \rightarrow 0$

   $T_1 \rightarrow 1$

   $A \rightarrow T_0\ S$

Este algoritmo parte de una gramática simplificada.

Forma Normal de Greibach

Una gramática está en Forma Normal de Greibach (FNG), cuando todas sus reglas son de la forma :

$A \rightarrow a\alpha$ con $a \in T$, $\alpha \in V^*$

Teorema de Normalización de Chomsky

Cualquier lenguaje libre de contexto que no contenga producciones ε puede ser generado por una gramática que esté en la forma normal de Greibach (FNG). El algoritmo para llevar una gramática a FNG no lo veremos en el curso. Se parte de una gramática que esté en la forma normal de Chomsky.

Veremos ahora un modelo de autómata para responder la pregunta si $x \in L$, siendo L un lenguaje libre de contexto.

Autómatas Push Down (APD)

La idea de estos autómatas es manejar un stack para reconocer si es una tira válida del lenguaje. Existen APD deterministas y NO deterministas. Al contrario de lo que pasaba con los autómatas finitos, no hay una correspondencia entre estos. El conjunto de lenguajes aceptados por un APD determinista está incluido en el NO determinista, pero no son equivalentes.

APD : $(Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$

Q = Conjunto de estados del autómata.

Σ = Conjunto de símbolos del lenguaje (alfabeto)

q_0 = Estado inicial

F = Conjunto de estados finales

Γ = Alfabeto con el cual se manipula al stack. Se puede manejar con un conjunto de símbolos diferentes a Σ , aunque podría ser igual.

Z_0 = Es la referencia al inicio del stack. Se puede ver como el puntero en memoria a la base del stack. Si este está como  cuando quitamos el Z_0 no se puede hacer más “push” al stack.

δ = función de transición que deberá considerar los estados, el alfabeto de entrada y el stack.

$\delta : Q \times \Sigma \cup \{\varepsilon\} \times \Gamma \rightarrow 2^{Q \times \Gamma^*}$

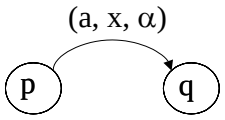
Si en el resultado, el Γ^* es de la forma $\Gamma\ \Gamma$ estamos haciendo un “push” al stack. Si es de la forma Γ estamos haciendo un “pop” al stack, mientras que si es ε es porque hemos llegado al final del stack.
Para poder manipular el stack sin consumir símbolos de la tira, utilizaríamos transiciones ε de la misma forma que en los autómatas finitos.

$\delta(q, a, z) = \{(P_1, \alpha_1), \ldots, (P_k, \alpha_k)\}$, $q \in Q$, $a \in \Sigma \cup \{\varepsilon\}$, $z \in \Gamma$, $P_{1..k} \in Q$, $\alpha_{1..k} \in \Gamma$

Ejemplo : Si se tiene el lenguaje 001#100, se haría
$\delta(q_0 , 0, z_0) = \{(q_0, 0z_0)\}$
O sea que partiendo del estado q_0,consumo el primer carácter (0) de la tira y como resultado queda en el mismo estado, agregando un símbolo al stack.

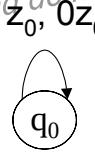
$\delta(q_0 , 1, z_0) = \{(q_0, 1z_0)\}$
$\delta(q_0 , 0, 0) = \{(q_0, 00)\}$
$\delta(q_0 , 0, 1) = \{(q_0, 01)\}$
$\delta(q_0 , 1, 0) = \{(q_0, 10)\}$
$\delta(q_0 , 1, 1) = \{(q_0, 11)\}$
...

Lo podemos ver también en forma gráfica :



Un arco de un estado q a uno p lo vamos a rotular con una terna (a, x, α), donde p y q $\in$ Q, a $\in \Sigma$, x $\in \Gamma$, $\alpha \in \Gamma^*$.

En el ejemplo anterior, a la transición $\delta(q_0 , 0, z_0) = \{(q_0, 0z_0)\}$ la representaríamos como :



Descripción Instantánea
Es como si se sacara una fotografía del APD en un momento dado. Es una terna (q, w, α), con q $\in$ Q es el estado corriente, w $\in \Sigma^*$ es lo que queda por consumir de la tira de entrada y $\alpha \in \Gamma^*$ es el stack propiamente dicho. El símbolo de más a la derecha de α (si no está vacío) es Z_0.

Si hacemos la transición δ(q, a, z) = (p, Xz), si sacamos la "foto" antes, tengo (q, ax, zα). Para representar la secuencia de fotos utilizamos el símbolo "$\vdash$", y hacemos
(q, ax, zα) $\vdash$ (p', x, Xzα) donde tenemos la foto inicial, la transición y la foto luego de esto.
Si tuviera la transición δ(q, a, z) = (p', ε) sería (q, ax, zα) $\vdash$ (p', x, α), esto representa el pop.
$\vdash^*$ es la descripción instantánea luego de aplicar varias (n) transiciones, de igual forma que antes usábamos $\Rightarrow^*$ para denotar la aplicación de varias transiciones.

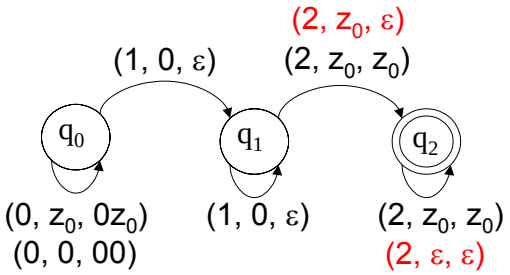
Lenguaje aceptado por estado final

$L_{Aceptado\ por\ Estado\ Final} = L_{AEF} = \{ w \in \Sigma^* / (q_0, w, z_0) \vdash^* (p, \varepsilon, \alpha) \text{ con } p \in F, \ \alpha \in \Gamma^* \}$

Lenguaje aceptado por stack vacío

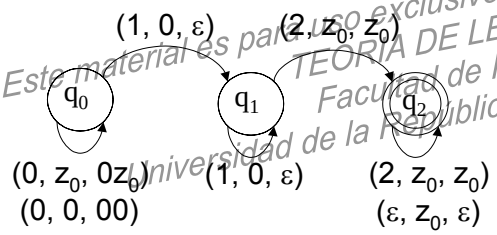
$L_{Aceptado\ por\ Stack\ Vacio} = L_{ASV} = \{ w \in \Sigma^* / (q_0, w, z_0) \vdash^* (p, \varepsilon, \varepsilon) \}$

Ejemplo: L_{AEF} $0^k1^k2^p$ k y p ≥ 1



Se puede borrar el stack Cuando empiezo a recibir 2, se marcan en rojo.

L_{ASV} $0^k1^k2^p$ k y p ≥ 1



Autómata Push Down Determinista

Tienen 2 condiciones :

I) Dado un estado q, símbolo a y tope del stack z, si existe y está definida una transición solo puede existir una.
$|\delta(q, a, z)| \leq 1 \quad a \in \Sigma \cup \{ \varepsilon \}, z \in \Gamma, q \in Q$

II) Si existe una transición ε, entonces no puede existir otra transición del mismo estado y tope del stack que consuma un símbolo de la tira de entada.
$\text{Si } \exists\ \delta(q, \varepsilon, X) \Rightarrow \text{no } \exists\ \delta(q, b, X)\ \forall\ b \in \Sigma$

El ejemplo anterior de L_{ASV} es NO DETERMINISTA ya que para el estado q_2 no se cumple la condición II), puesto que tenemos
$\delta(q_2, \varepsilon, z_0) = \{(q_2, \varepsilon)\}$ y también $\delta(q_2, 2, z_0) = \{(q_2, z_0)\}$

En los APD Deterministas, $\delta : Q \times (\Sigma \cup \{ \varepsilon \}) \times \Gamma \rightarrow q \times \Gamma^*$

Equivalencia entre L_{AEF} y L_{ASV}

Todo lenguaje aceptado por un APD por estado final es aceptado por un APD por satck vacío y recíprocamente.

($\Rightarrow$) M : $(Q, \Sigma, \Gamma, \delta, q_0, z_0, F) / \mathcal{L} = L_{AEF}$ (M)

M' : $(Q', \Sigma, \Gamma', \delta', q_0', z_0', \varnothing) / \mathcal{L} = L_{ASV}$ (M')

La idea es armar un algoritmo que construya M' partiendo de M. Desde los estados finales voy a un nuevo estado cuando se acabe la tira, donde se quitan todas las tiras del stack mediante transiciones ε. Tenemos que tener en consideración si en M había una transición que sacaba el Z_0, ya que podría quedar el stack vacío. Para solucionar esto es que hacemos que M' tenga un z_0' y lo primero que se hace es pasar desde el estado inicial de M' (q_0') al q_0 con una transición ε, agregando el z_0 al stack de M' (quedaría Z_0Z_0'). A partir de ahí todas las transiciones de M' serían iguales a las de M.

$\delta'(q_0', \varepsilon, z_0') = \{(q_0, Z_0Z_0')\}$

$\delta'(q, a, X) = \delta(q, a, X)$ con $x \in \Gamma$, $q \in Q$, $a \in \Sigma$

$\delta'(p, \varepsilon, X) = \{(q_s', \varepsilon)\} \forall\ p \in F, q_s' \in Q', X \in \Gamma'$

$\delta'(q_s', \varepsilon, z) = \{(q_s', \varepsilon)\} \forall\ q_s' \in Q', z \in \Gamma'$

$Q' = Q \cup \{(q_s', q_0')\}$

$\Gamma' = \Gamma \cup \{(Z_0')\}$

($\Leftarrow$) M : $(Q, \Sigma, \Gamma, \delta, q_0, z_0, \varnothing) / \mathcal{L} = L_{ASV}$ (M)

M' : $(Q', \Sigma, \Gamma', \delta', q_0', z_0', \{q_s\}) / \mathcal{L} = L_{AEF}$ (M')

La idea es armar un algoritmo que construya M' partiendo de M. Desde cada estado de M, si se vació el stack (se llega a Z_0) y no hay más símbolos en la tira, voy a un nuevo estado final (q_s) mediante transiciones ε. Hacemos que M' tenga un z_0' y lo primero que se hace es pasar desde el estado inicial de M' (q_0') al q_0 con una transición ε, agregando el z_0 al stack de M' (quedaría Z_0Z_0'). A partir de ahí todas las transiciones de M' serían iguales a las de M.

$Q' = Q \cup \{(q_s', q_0')\}$

$\Gamma' = \Gamma \cup \{(Z_0')\}$

$\delta'(q_0', \varepsilon, z_0') = \{(q_0, Z_0Z_0')\}$

$\delta'(q, a, X) = \delta(q, a, X)$ con $x \in \Gamma$, $q \in Q$, $a \in \Sigma$

$\delta'(p, \varepsilon, z_0') = \{(q_s', \varepsilon)\} \forall\ p \in Q$

$$\underbrace{(q_0', x, z_0') \vdash \underbrace{(q_0, x, Z_0Z_0') \vdash^* (q, \varepsilon, Z_0')}_{M} \vdash (q_s, \varepsilon, \varepsilon)}_{M'}$$

Equivalencia entre GLC y APD

Todo lenguaje que no contenga ε y es generado por una GLC es aceptado por un APD que lo reconoce por stack vacío y recíprocamente.

($\Rightarrow$) A partir de las reglas de producción de la gramática, se construirá el APD. Para esto es requerido que la gramática esté en forma normal de Greibach (FNG)

G : (V, T, P, S) $\rightarrow$ M : (Q, Σ, Γ, δ, q_0, z_0, $\varnothing$) $\mathcal{L}$ = L (G) = L (M)

Se construirá simulando derivaciones de más a la izquierda.
Q = { q_0} el APD solo tendrá un estado.
Σ = T el conjunto de terminales constituyen el alfabeto.
Γ = V
z_0 = S

Como la gramática está en FNG, todas las reglas de producción son del tipo :
$A \rightarrow a\alpha$ $a \in T$, $\alpha \in V^*$
$(q, \alpha) \in \delta(q, a, A)$
$S \Rightarrow^* x$ si y solo si $(q_0, x, S) \vdash (q_0, \varepsilon, \varepsilon)$ $x \in T^*$

Lo que tenemos que demostrar es que si a partir de S llego a una tira de terminales, entonces el APD la reconoce.

Ejemplo : $S \rightarrow aSB \mid aB$
$B \rightarrow b$

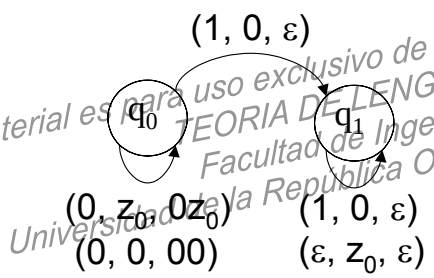
Según el teorema anterior haríamos:
$\delta(q, a, S) = \{(q, SB), (q, B)\}$
$\delta(q, b, B) = \{(q, \varepsilon)\}$

Veamos la tira x = aaabbb
Para la gramática sería :
$S \Rightarrow_1 aSB \Rightarrow_1 aaSBB \Rightarrow_2 aaaBBB \Rightarrow_3 aaabBB \Rightarrow_3 aaabbB \Rightarrow_3 aaabbb$
Para el APD sería :
Inicia con el stack = SZ_0
Leo “a”, aplico la 1° regla agrego B al stack = SBZ_0
Leo “a”, aplico la 1° regla agrego B al stack = $SBBZ_0$
Leo “a”, aplico la 2° regla quito S del stack = BBZ_0
Leo “b”, aplico la 3° regla quito B del stack = BZ_0
Leo “b”, aplico la 3° regla quito B del stack = Z_0
Leo “b”, aplico la 3° regla quito Z_0 del stack = $\varnothing$
Por lo tanto se llega al final de la tira y está el stack vacío, por lo que el APD la reconoce como válida.

($\Leftarrow$) M : $(Q, \Sigma, \Gamma, \delta, q_0, z_0, \varnothing) \rightarrow$ G : (V, T, P, S) $\mathcal{L} = L(G) = L(M)$
Donde:
$T = \Sigma$
$V = \{S\} \cup \{[q, A, p]$, con p y q $\in Q$, $A \in \Gamma\}$ es decir que las variables serán la S y combinaciones entre los estados y los símbolos con los que se manipula el stack.
$P = \{S \rightarrow [q_0, z_0, q] \quad \forall\ q \in Q$
$[q, A, q_{n+1}] \rightarrow a\ [q_1, B_1, q_2]\ [q_2, B_2, q_3] \ldots [q_n, B_n, q_{n+1}]$ y tendremos esta regla si $\exists\ (q_1, B_1 B_2 \ldots B_n) \in \delta(q, a, A)\ \forall\ q \in Q$. (con estas se simula el "push" del APD).
$[q, A, q1] \rightarrow A$ si $\exists\ (q1, \varepsilon) \in \delta(q, a, A)$ (con estas simulamos el "pop" del APD).
}

Lo que tenemos que demostrar en este caso es que
$(q_0, x, z_0) \vdash^* (q, \varepsilon, \varepsilon)$ si y solo si $S \Rightarrow^* x$

Ejemplo: Dado el siguiente APD que reconoce las tiras $0^K\ 1^K$



Aplicando el algoritmo del teorema anterior para obtener la gramática haríamos:
$S \rightarrow [q_0, z_0, q_0] \mid [q_0, z_0, q_1]$
Partiendo de la transición $\delta(q_0, 0, z_0) = \{(q_0, 0z_0)\}$ haríamos
$[q_0, z_0, q_0] \rightarrow 0\ [q_0, 0, q_0]\ [q_0, z_0, q_0] \mid 0\ [q_0, 0, q_1]\ [q_1, z_0, q_0]$
$[q_0, z_0, q_1] \rightarrow 0 \ldots$
...
y así sucesivamente con todas las combinaciones posibles entre los estados. Haciéndolo de esta forma se transforma en algo inmanejable, ya que al hacer la combinación de los estados para obtener las reglas, se aumenta exponencialmente la cantidad de reglas. Muchas de estas no serán útiles o alcanzables, ya que el algoritmo NO GENERA UNA GRAMÁTICA SIMPLIFICADA.
En la práctica no se utiliza este mecanismo para obtener la gramática.

Propiedades de los Lenguajes Libres de Contexto (LLC)

$L = \{ a^k b^k c^k,\ k \geq 1\}$

Este lenguaje no es regular, y como se vio anteriormente, esto se probaría mediante el contra recíproco del Pumping Lemma. Hay que tener en consideración en este caso que si el k estuviera acotado entonces el lenguaje SI SERÍA REGULAR.

Recordamos el LEMA : Un árbol binario de altura h tiene cantidad de hojas $\leq 2^h$

Pumping Lemma

Si L es un lenguaje libre de contexto que no contiene ε entonces existe un n natural donde para toda tira perteneciente a L de largo mayor o igual a n, existe al menos una descomposición de esa tira en 5 sub-tiras uvwxy que cumplen:

$|uv| \geq 1$

$|vwx| \leq n$

$\forall\ i \geq 0,\ u\ v^i\ w\ x^i\ y \in L$

H) L es un LLC que no contiene ε

T) $\exists\ n \in N\ /\ \forall\ z \in L$ y $|z| \geq n$, entonces $\exists$ al menos una descomposición $z = uvwxy$ tal que :

i) $|uv| \geq 1$

ii) $|vwx| \leq n$

iii) $\forall\ i \geq 0,\ u\ v^i\ w\ x^i\ y \in L$

Dem: Sea una gramática G : (V, T, P, S) en forma normal de Chomsky / $\mathcal{L} = L(G)$. |V| = k (cantidad de variavles de V es igual a k).

Tomamos el siguiente número natural : $n = 2^{k+1}$ (expresado en función de la cantidad de variables). Consideremos una tira z de largo mayor o igual a 2^{k+1}

$| z | \geq 2^{k+1}$

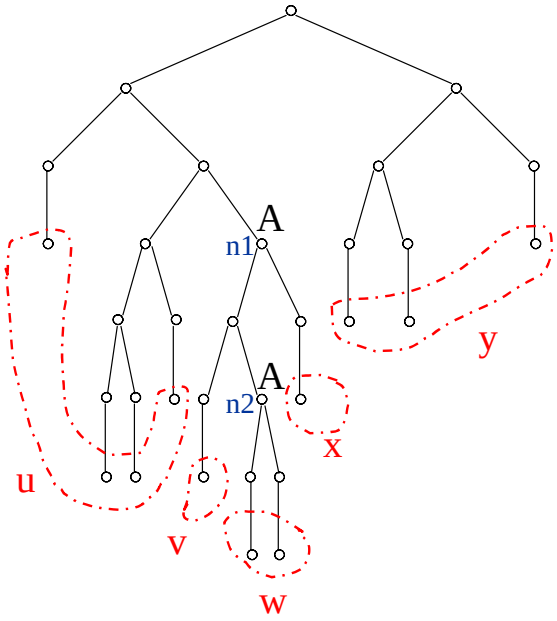
Recordemos que las reglas de las gramáticas en FNC son del estilo

$A \rightarrow BC \quad A \rightarrow a$

$\Rightarrow$ el árbol de derivación de estas gramáticas es binario. Por el lema anterior, la altura de este árbol es mayo o igual a k + 1. Como tenemos k variables, que la altura sea $\geq$ k+1 quiere decir que alguna de las variables se repite.

Si se repite más de una vez la misma variable, elegimos las 2 que estén más cerca del resultado (más cerca de las hojas.

Lo mismo si se repiten varias variables distintas, tomamos el par que esté más cerca de las hojas.



Notese que la concatenación de uvwxy es el z, no queda ningún nodo afuera
La derivación sería : $S \Rightarrow^* uAy$
$A \Rightarrow^* vAx$
$A \Rightarrow^* w$

Veamos si cumple con las condiciones del teorema :

i) $|vx| \geq 1$
Como la gramática está en FNC, no puede tener una derivación $A \Rightarrow^* A$, y si $v = x = \varepsilon$ (o sea que fueran ambos vacíos) la segunda derivación que estamos aplicando sería justamente $A \Rightarrow^* A$, por lo tanto no pueden ser ambos = ε, $\Rightarrow |vx| \geq 1$.

ii) $|vwx| \leq n$
Tomemos el nodo "n1" que estamos considerando. Desde ahí para abajo, a lo sumo la altura será ≤ k+1, ya que como elegimos de todas las variables que se repitieran la que estuvieran más cerca de las hojas, no se repetirá otra variable, y si estuvieran todas las k, la altura sería a lo sumo k+1.
$\Rightarrow$ altura (n1) ≤ k + 1, $\Rightarrow$ cant.hojas(n1) ≤ 2k+1 = n, $\Rightarrow |n1| \leq n$, y como n1 = vwx, $\Rightarrow |vwx| \leq n$

iii) $\forall\, i \geq 0,\ u\, v^i\, w\, x^i\, y \in L$
$S \Rightarrow^* uAy \Rightarrow^* uvAxy \Rightarrow^* uvvAxxy \Rightarrow^* uv^iAx^iy \Rightarrow^* uv^iwx^iy$,
$\Rightarrow \forall\, i > 0,\ u\, v^i\, w\, x^i\, y \in L$ ya que hay una secuencia de derivación que lo construye. Si i = 0 , $S \Rightarrow^* uAy \Rightarrow^* uwy$, que también ∈ L,
$\Rightarrow$ se cumple.

Con esto demostramos el teorema, ya que encontramos una descomposición que cumple con las propiedades.
Al igual que en el caso anterior, lo que se aplica en la práctica es el contra recíproco

Contra recíproco del Pumping Lemma

H) Dado un lenguaje L, $\forall$ n $\in$ N $\exists$ z $\in$ L, |z| $\geq$ n, para toda descomposición z = u v w x y alguna de las siguientes tres condiciones no se cumple

i) | v x | $\geq$ 1
ii) | v w x | $\leq$ n
iii) $\forall$ i $\geq$ 0, u v^i w x^i y $\in$ L

T) entonces L no es libre de contexto.
En la práctica se usa que las condiciones 1 y 2 se cumplen y falle la 3.
O sea que $\exists$ i $\geq$ 0 / u v^i w x^i y $\notin$ L

Por ejemplo, L = {a^k b^k c^k con k $\geq$ 1 }
Pasos a seguir para aplicar el contra recíproco del Pumping Lemma:

A. Elijo un n
B. Elijo una descomposición z que pertenezca al lenguaje, en función del n, de tal forma que |z| $\geq$ n $\Rightarrow$ z = a^k b^k c^k |z| = 3k $\geq$ n
C. Ahora debería estudiar toda posible descomposición de z y esto complica el análisis. La solución en este caso sería utilizar descomposiciones de z que verifican las condiciones 1) y 2), y en todo el resto de las posibles descomposiciones de z no se va a cumplir alguna de esas 2 propiedades.

1) Tomemos la siguiente descomposición, con v y x entre las “a”
aaa....aabbb....bbccc....cc
| | | |
v x
u = a^s
v = a^p
w = a^r
x = a^q
y = $a^{n-s-p-r-q}$ b^n c^n
Para que se cumpla la condición 1) es necesario que p + q $\geq$ 1. Esta descomposición cumple con las condiciones 1 y 2. Estudiemos la condición 3).
z_i = u v^i w x^i y = a^s $(a^p)^i$ a^r $(a^q)^i$ $a^{n-s-p-r-q}$ b^n c^n = $a^{s-s+r-r+n-p-q+ip+iq}$ b^n c^n =
z_i = $a^{n+(p+q)(i-1)}$ b^n c^n $\Rightarrow$ si tomamos i = 2
z_i = a^{n+p+q} b^n c^n que $\notin$ L

2) Tomemos la siguiente descomposición, con v y x entre las “b”
aaa....aabbb....bbccc....cc
| | | |
v x
u = a^n b^s
v = b^p
w = b^r
x = b^q
y = $b^{n-s-p-r-q}$ c^n
Para que se cumpla la condición 1) es necesario que p + q $\geq$ 1.
Se resuelve de la misma manera que en el caso 1)

3) Tomemos la siguiente descomposición, con v y x entre las “c”

aaa....aabbb....bbccc....cc
| | | |
v x

$u = a^n b^n c^s$
$v = c^p$
$w = c^r$
$x = c^q$
$y = c^{n-s-p-r-q}$

Para que se cumpla la condición 1) es necesario que $p + q \geq 1$.
Se resuelve de la misma manera que en el caso 1)

4) Tomemos la siguiente descomposición, con v entre las “a” y x entre las “b”

aaa....aabbb....bbccc....cc
| | | |
v x

$u = a^{n-p-r}$
$v = a^p$
$w = a^r b^s$
$x = b^q$
$y = b^{n-s-q} c^n$

Para que se cumpla la condición 1) es necesario que $p + r - q < n$.
Para cumplir con la condición 2) es necesario que $p + r + s + q \leq n$
$z_i = u\ v^i\ w\ x^i\ y = a^{n-p-r} (a^p)^i a^r b^s (b^q)^i b^{n-s-q} c^n = a^{n-p-r+ip} b^{n-s-q+iq} c^n =$
$z_i = a^{n+p(i-1)} b^{n+q(i-1)} c^n \Rightarrow$ si tomamos i = 2
$z_i = a^{n+p} b^{n+q} c^n$ que $\notin L$

NOTA : La siguiente descomposición no se puede hacer pues no cumple con la segunda condición :

aaa....aabbb....bbccc....cc
| | | |
v x

5) Tomemos la siguiente descomposición, con v entre las “b” y x entre las “c”

aaa....aabbb....bbccc....cc
| | | |
v x

$u = a^n b^p$
$v = b^q$
$w = b^{n-p-q} c^r$
$x = c^s$
$y = c^{n-r-s}$

Para que se cumpla la condición 1) es necesario que $q + s \geq 1$.
Para cumplir con la condición 2) es necesario que $r + s \leq p$

6) Tomemos la siguiente descomposición, con v entre las “ab” y x entre las “b”
aaa....aabbb....bbccc....cc
| | | |
v x
$u = a^p$
$v = a^{n-p}\ b^q$
$w = b^r$
$x = b^s$
$y = b^{n-q-r-s}\ c^n$
Para que se cumpla la condición 1) es necesario que $n-p+q+s \geq 1$.
Para cumplir con la condición 2) es necesario que $q + r + s \leq p$

Asi sucesivamente se seguiría resolviendo las distintas descomposiciones posibles.

Lema de OGDEN

Es una propiedad más fuerte que el Pumping Lemma.
Si L es un Lenguaje Libre de Contexto, existe un “n” natural tal que para todo “z” que pertenezca a L se cumple que $dist(z) \geq n$, y existe una descomposición z = uvwxy tal que cumple con las condiciones mencionadas más abajo.

“dist” es una función que permite “contar” las posiciones distinguidas en la tira Z. Lo que hacemos es “distinguir” (se puede ver como elegir o marcar) n símbolos de Z. $dist : T \rightarrow N$

H) L es LLC
T) $\exists\ n \in N / \forall\ z \in L$ y $dist(z) \geq n$, $\exists\ z = uvwxy$ tal que:
i) $dist(vx) \geq 1$
ii) $dist(vwx) \leq n$
iii) $\forall\ i \geq 0,\ \ z_i = u\ v^i\ w\ x^i\ y \in L$

La demostración no se hará en el curso. Es similar a la del Pumping Lema. Podemos ver al Pumping Lema como un caso particular del lema de Ogden donde se distingue a toda la tira.

Contra recíproco del lema de OGDEN

H) Sea L un lenguaje, $\forall\ n \in N\ \exists\ z \in L$ con $dist(z) \geq n / \forall\ z = uvwxy$ alguna de las siguientes condiciones no se cumple:
i) $dist(vx) \geq 1$
ii) $dist(vwx) \leq n$
iii) $\forall\ i \geq 0,\ \ z_i = u\ v^i\ w\ x^i\ y \in L$
T) L no es un LLC (lenguaje libre de contexto)

Generalmente se buscan descomposiciones que cumplan con las condiciones i) y ii) y fallen en la iii), o sea $\exists\ i \geq 0 / z_i = u\ v^i\ w\ x^i\ y \notin L$

Ej: $z = a^p b^n c^n d^n$ con $p>0$

Sean b^n las posiciones distinguidas

Para poder aplicar el contra recíproco vamos a plantear todas las descomposiciones que cumplen las condiciones i) y ii) y fallan en la iii),
$\Rightarrow$ dist(vx) $\geq$ 1 y dist(vwx) $\leq$ n

aaaaa ….. aaabbbbbb ….. bbbbccccc …… cccddddd …… ddddd

Caso 1) | v | | x |
Caso 2) | v | | x |
Caso 3) | v | | x |
Caso 4) | v | | x |
Caso 5) | v | | x |

.... y así sucesivamente todas las descomposiciones
Tenemos que plantear cada caso:

Caso 1) $u = a^t b^r$
$v = b^p$
$w = b^l$
$x = b^q$
$y = b^{n-r-p-l-q} c^n d^n$ $p + q \geq 1$ (para cumplir la condición i))

$z_i = a^t b^r (b^p)^i b^l (b^q)^i b^{n-r-p-l-q} c^n d^n$
$z_i = a^t b^{r+ip+l+iq+n-r-p-l-q} c^n d^n$
$z_i = a^t b^{n+(i-1)(p+q)} c^n d^n$
tomando i=0 $z_0 = a^t b^{n-p-q} c^n d^n \notin L$, porque $cant_b(z_0) < cant_c(z_0)$

Los demás casos se plantearía :
a) $v \in a^p$, $x \in b^n$
b) $v \in a^p$, $x \in b^n c^n$
c) $v \in a^p$, $x \in b^n c^n d^n$
d) $v \in a^p$, $x \in a^p b^n$
e) $v \in a^p$, $x \in a^p b^n c^n$
f) $v \in a^p$, $x \in a^p b^n c^n d^n$
g) $v \in a^p b^n$, $x \in b^n$
$x \in b^n c^n$
$x \in b^n c^n d^n$
h) $v \in a^p b^n c^n$, $x \in c^n$
$x \in c^n d^n$
$x \in d^n$

h) $v \in a^p b^n c^n d^n$, $x \in d^n$
….
Cada uno de estos casos habría que demostrarlos, de la misma forma en que se hizo el primer caso. Al igual que con el Pumping Lema, se podría agrupar aquellos que se demuestran en forma similar.
Se pueden presentar algunos casos en forma más genérica, que resume varios casos :
h') $v \in a^p b^n c^r$, $x \in c^p d^q$ con r, p y q cumpliendo determinadas condiciones.

El cuidado que hay que tener con esta aplicación es elegir apropiadamente los símbolos distinguidos. Si en este ejemplo hubiéramos elegido como distinguidos a $b^n c^n d^n$, la cantidad de casos que debemos plantear es mucho menor.

Propiedades de Clausura de los LLC

Los LLC son armados bajo las siguientes operaciones:

1) Unión
2) Concatenación
3) Clausura de Kleen
4) Reverso
5) Sustitución

1) Unión : L_1 es LLC, L_2 es LLC $\Rightarrow$ $L_1 \cup L_2$ es LLC
Demo:
L_1 es LLC $\Rightarrow \exists\ G_1\ (V_1, T_1, P_1, S_1)$
L_2 es LLC $\Rightarrow \exists\ G_2\ (V_2, T_2, P_2, S_2)$
$V_1 \cap V_2 = \varnothing$
$G_3\ (V_3, T_3, P_3, S_3)$
$V_3 = V_1 \cup V_2 \cup \{ S_3 \}$
$T_3 = T_1 \cup T_2$
$P_3 = P_1 \cup P_2 \cup \{ S_3 \rightarrow S_1 | S_2 \}$

2) Concatenación : L_1 es LLC, L_2 es LLC $\Rightarrow$ $L_1 . L_2$ es LLC
Demo:
L_1 es LLC $\Rightarrow \exists\ G_1\ (V_1, T_1, P_1, S_1)$
L_2 es LLC $\Rightarrow \exists\ G_2\ (V_2, T_2, P_2, S_2)$
$V_1 \cap V_2 = \varnothing$
$G_3\ (V_3, T_3, P_3, S_3)$
$V_3 = V_1 \cup V_2 \cup \{ S_3 \}$
$T_3 = T_1 \cup T_2$
$P_3 = P_1 \cup P_2 \cup \{ S_3 \rightarrow S_1 . S_2 \}$

3) Clausura de Kleen : L_1 es LLC $\Rightarrow$ L_1^* es LLC
Demo:
L_1 es LLC $\Rightarrow \exists\ G_1\ (V_1, T_1, P_1, S_1)$
$G_3\ (V_3, T_3, P_3, S_3)$
$V_3 = V_1 \cup \{ S_3 \}$
$T_3 = T_1$
$P_3 = P_1 \cup \{ S_3 \rightarrow S_3\ S_1 | \varepsilon \}$

4) Reverso : L es LLC $\Rightarrow L^r$ es LLC
Demo: Básicamente hay que aplicar el reverso al lado derecho de todas las reglas.
Ej. $L = a^n b^n$ hay una regla del tipo aSb | ab
$L^r = b^n a^n$ la regla sería bSa | ba

L_1 es LLC $\Rightarrow \exists\ G_1 (V_1, T_1, P_1, S_1)$
$L^r \rightarrow G_3 (V_3, T_3, P_3, S_3)$
$V_3 = V_1$
$T_3 = T_1$
$P_3 = \{ \forall\ A \rightarrow \alpha \in P, \exists\ A \rightarrow \alpha^r \}, \quad \alpha \in (V_1 \cup T_1)^*$
$S_3 = S_1$

5) Sustitución : L es LLC , $f : \Sigma \rightarrow \Delta^* \Rightarrow f(L)$ es LLC
Ejemplo :
$L = \{ a^n b^n \}$
$f(a) = 0^n 1^m$ S_0
$f(b) = \# w^s \# w^r \quad w \in (0,1)^*$ S_1

Ej.: $f(a\ b) = 0^n 1^m \# w^s \# w^r$

$S \rightarrow aSb \mid ab$
$S \rightarrow S_0\ S\ S_1 \mid S_0\ S_1$ Esta es la gramática de la sustitución.
$S_0 \rightarrow 0\ S_0\ 1 \mid 0\ S_0 \mid 01 \mid 0$
$S_1 \rightarrow \# T$
$T \rightarrow 0\ T_0 \mid 1\ T_1 \mid \#$

6) Intersección : La intersección de dos LLC no necesariamente es un LLC, no se cumple para todo par de lenguajes.

$L_1 = \{a^q b^p c^p\}$ es un LLC
$L_2 = \{a^j b^j c^t\}$ es un LLC

$L_1 \cap L_2 = \{ a^k b^k c^k \}$ NO es un LLC

$\{0^n 1^m\} \cap \{ \# w^s \# w^r \}$ es LLC porque la intersección es vacía, que es regular.

Complemento : $L_1 \cap L_2 = (L_1^c \cup L_2^c)^c$
El complemento no puede ser LLC porque sino la intersección lo sería, y vimos que no lo es.

$\Rightarrow$ El complemento de un LLC no necesariamente es LLC

Proposición :
L es LLC, R es regular $\Rightarrow L \cap R$ es LLC

Jerarquía de Chomsky

Lenguajes recursivamente numerables : Son aquellos que pueden ser definidos por las gramáticas irrestrictas.

Gramáticas Irrestrictas

G_{IRES} (U, T, P, S) $\alpha \rightarrow \beta$ $\alpha, \beta \in (V \cup T)^*, \alpha \neq \varepsilon$

La diferencia con las gramáticas anteriores (lo que las caracteriza) son las reglas de producción. Del lado izquierdo o derecho puedo tener cualquier combinación de símbolos o variables. Del lado izquierdo no puedo tener ε.

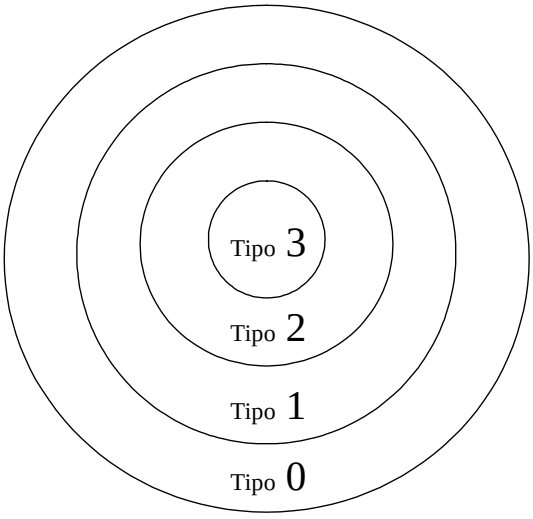
Lenguajes sensibles al contexto

Son lenguajes constructores, es decir que siempre se genera igual o mayor cantidad de símbolos, no se reduce

$\alpha \rightarrow \beta \quad |\alpha| \leq |\beta|$

Cuando en un ejercicio se pide generar la gramática, siempre se debe hacer la que corresponde al lenguaje (regular, libre de contexto o no libres de contexto, recursivamente numerables o sensibles al contexto)

Los lenguajes son de los siguientes tipos:
Tipo 0 : Recursivamente numerables
Tipo 1 : Sensibles al contexto
Tipo 2 : Libres de contexto
Tipo 3 : Regulares



Teorema de jerarquía:
Los lenguajes tipo i+1 están contenidos en los tipo i, i = 0 .. 2

Ej. Gramática irrestricta $a^k a^k a^k$ con $k \geq 1$
$S \rightarrow aSBc \mid abc$
$cB \rightarrow Bc$
$bB \rightarrow bb$
$S \Rightarrow aSBc \Rightarrow aaSBcBc \Rightarrow aaabcBcBc \Rightarrow aaabBccBc \Rightarrow aaabbccBc \ldots$

Ej.: $a^n \# b^m$, con $m = p*n$ (o sea que la cantidad de b es múltiplo de la cantidad de a)

1) $S \rightarrow PS \mid A$
2) $A \rightarrow aA \mid \#$
3) $Pa \rightarrow aBp$
4) $P\# \rightarrow \#$
5) $Ba \rightarrow aB$
6) $B\# \rightarrow \#b$

PPPP ….. PPPaaaa ….. aaa#
↓ aplicando la regla 3
…… aBPaa …
↓
…… aBaBPa …
↓
…..... aBaBa …… aBP#
↓ aplicando la regla 5
…..... aaBBa …… aBP#
↓
…..... aaBaB …… aBP#

y aplicando sucesivamente las reglas se llegaría a tener para cada “p” la misma cantidad de “B” que de “a”. Lo que se hace es introducir un símbolo “B” por cada “p” y mediante las reglas lleva junto al “#”, para luego convertirlo en “#b”.

Máquina de Turing

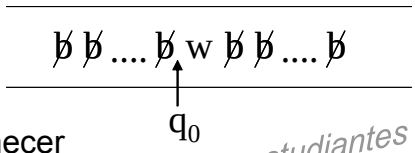
Está asociada a las gramáticas irrestrictas. En este tipo de máquinas se maneja memoria infinita.

M : (Q, Σ, Γ, δ, q_0, ∄ , F)

Maneja una "cinta infinita", tanto en su comienzo como en el final.

Q = Conjunto de caracteres
Σ = Alfabeto de entrada
Γ = Alfabeto de la cinta
δ = Función de transición
q_0 = Es un estado de Q distinguido como inicial
∄ = Simboliza el carácter blanco
F = Conjunto de estados finales

w ∈ Σ* es la tira a chequear
El resto de la cinta está con blancos. ∄ ∉ Σ. El símbolo que denota el blanco no puede pertenecer al alfabeto de entrada.



∄ ∈ Γ
Σ ⊂ Γ - {∄}
δ : Q x Γ → Q x Γ x {I, D}

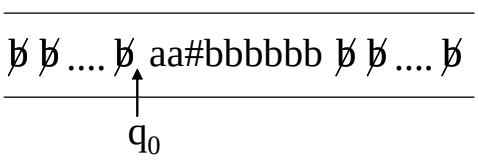
La función de transición permite que desde un estado, cuando se accede a un símbolo de la entrada se pueda mover a la izquierda o a la derecha, leer el símbolo siguiente o el anterior, e inclusive se puede reemplazar el símbolo por otro.

La tira va a ser reconocida cuando la máquina quede en un estado final. No importa como quede la cinta.

Esto es una Máquina de Turing Determinista, con desplazamiento a la izquierda y a la derecha (1 solo lugar) y con un único estado final.

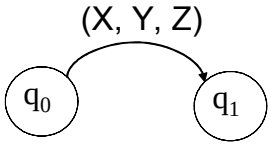
Hay otras máquinas de Turing no deterministas, con varias cintas, con desplazamiento a la derecha y a la izquierda y que se quede en el mismo lugar, con N estados finales, etc. El tema es que son todas equivalentes.

Ejemplo : Máquina de Turing para a^n # b^m, con m = p*n (o sea que la cantidad de b es múltiplo de la cantidad de a)



La idea sería marcar para cada “a” una “b” correspondiente. Si marqué todas las “a” y no me quedan “b” sin marcar, entonces desmarco las “a” y vuelvo a empezar (deja las “b” marcadas). Si llego a que quedan “a” sin marcar pero no quedan “b”, entonces no es válida. Si marqué todas las “a” y llego a $\not b$ (marqué todas las “b”) entonces es válida.

$\delta(q_0, a) = (q_1, X, D)$ lo que quiere decir es que se reemplaza la “a” por X, y se mueve a la derecha
$\delta(q_1, a) = (q_1, a, D)$
$\delta(q_1, \#) = (q_2, \#, D)$
…



donde X es el símbolo leído, Y es el símbolo que queda en la cinta y Z es el desplazamiento {I, D}

Para el ejemplo anterior sería :