Teoría de Autómatas y Lenguajes Formales

Leopoldo Altamirano, Miguel Arias,
Jesús González, Eduardo Morales,
Gustavo Rodríguez

Objetivo General

Proporcionar al estudiante los fundamentos de la teoría de autómatas así como los de lenguajes formales. También se incluye una introducción a las máquinas de Turing.

Temario

1. Introducción
2. Autómatas Finitos
3. Expresiones Regulares y Lenguajes
4. Propiedades de los Lenguajes Regulares
5. Gramáticas Libres de Contexto y Lenguajes
6. Autómata de Pila
7. Propiedades de los Lenguajes Libres de Contexto
8. Introducción a las Máquinas de Turing

Modalidad del Proceso de Enseñanza

El material del curso es impartido por el instructor con resolución de algunos problemas en clase. También se asignarn tareas prácticas a los estudiantes cuya solución será en algunos casos discutida en clase. Se tendrán sesiones de asesoría con los asistentes de curso.

Horario:

Clase:	Martes	9:30 a 12:00
	Jueves	9:30 a 12:00
Asosoria:	Martes	12:00 a 13:00
	Jueves	12:00 a 13:00

Evaluación:

Tareas	1/3
Examenes Parciales (2)	2/3
1^{er} Parcial	19-Junio-2008
2^{o} Parcial	15-Julio-2008

Calendarización.

Sem.	Fecha	Tema
1	27 y 29 mayo	Introducción
2	3 y 5 junio	Autómatas Finitos
3	10 y 12 junio	Expresiones Regulares y Lenguajes
4	17 y 19 junio	Props. Lenguajes Regulares (Primer examen)
5	24 y 26 junio	Gramáticas y Lenguajes Libres de Contexto
6	1 y 3 julio	Pushdown Automata
7	8 y 10 julio	Props. Lenguajes Libres de Contexto
8	15 y 17 julio	Intro. Máquina de Turing (Segundo examen)

Referencias

1. John E. Hopcroft, Rajeev Motwani and Jeffrey D. Ullman (2001). Automata Theory, Language and Computation. Addison-Wesley Publishing.

2. Michael Sipser (1997). Introduction to the Theory of Computation. PWS publishing company.

Capítulo 1

Introducción

1.1 Teoría de Autómatas

- Estudio de dispositivos o máquinas de cómputo abstractas
- Turing en los 30's estudió una máquina abstracta con las capacidades de las de hoy (en lo que podían calcular)
- La meta de Turing era describir la frontera entre lo que una máquina podía hacer y lo que no, su conclusión aplica no solo a las máquinas de Turing, sino a las máquinas actuales
- En los 40's y 50's se estudiaron los Autómatas Finitos
- Finales de los 50's, N. Chomsky inicia el estudio formal de las gramáticas
- En 1969 S. Cook extiende el trabajo de Turing para estudiar lo que se podía y no calcular (compute), y lo que se podía resolver eficientemente o no (NP-duros).

1.2 Introducción a Pruebas Formales

Para qué pruebas?

- Hacer hipótesis inductivas sobre partes de nuestros programas (iteración o recursión)
  - Hipótesis consistente con la iteración o recursión
- Entender como trabaja un programa correcto es esencialmente lo mismo que la prueba de teoremas por inducción
  - Pruebas deductivas (secuencia de pasos justificados)
  - Pruebas inductivas (pruebas recursivas de un estatuto parametrizado que se usa a sí mismo con valores más bajos del parámetro)

1.2.1 Pruebas Deductivas

Secuencia de enunciados cuya verdad nos lleva de un enunciado inicil (hipótesis) a una conclusión. Son del tipo "If H Then C".

Por ejemplo, probar que: *Si* $x \geq 4$ *Entonces* $2^x \geq x^2$. La idea general es probar que se cumple para 4 y probar que mientras x crece, el lado izquierdo duplica su valor con cada incremento, mientras que el lado derecho crece a una razón de $(\frac{x+1}{x})^2$.

También existen otras pruebas como pruebas por contradicción, por contraejemplos y de conjuntos, pero nos vamos a enfocar en las pruebas por inducción.

1.2.2 Pruebas por Inducción

- Para objetos definidos recursivamente
- La mayoría maneja enteros, pero en autómatas se trabaja con árboles y expresiones de varios tipos (expresiones regulares)

1.2.2.1 Inducciones estructurales

Probar el enunciado $S(X)$ respecto a una familia de objetos X (i.e., enteros, árboles) en dos partes:

1. *Base*: Probar directamente para uno o varios valores pequeños de X.
2. *Paso de Inducción*: Asumir $S(Y)$ para Y "más pequeña que" X; probar $S(X)$ a partir de lo que se asumió.

Un árbol binario con n hojas tiene $2n - 1$ nodos.

- Formalmente, $S(T)$: Si T es un árbol binario con n hojas, entonces T tiene $2n - 1$ nodos.
- Inducción respecto al tamaño = número de nodos de T.

Base: Si T tiene 1 hoja, se trata de un árbol de un nodo. $1 = 2 \times 1 - 1$, está bien.

Inducción: Asumir $S(U)$ para árboles con menos nodos que T. En particular, asumir para los sub-árboles de T:

- T debe consistir de una raíz más dos subárboles U y V.
- Si U y V tienen u y v hojas, respectivamente, y T tiene t hojas, entonces $u + v = t$.
- Por la hipótesis de inducción, U y V tienen $2u - 1$ y $2v - 1$ nodos, respectivamente.
- Entonces T tiene $1 + (2u - 1) + (2v - 1)$ nodos.

  $= 2(u + v) - 1$.
  $= 2t - 1$, probando con el paso de inducción.

Frecuentemente el enunciado que tenemos que probar es de la forma "X si y sólo si Y". Esto requiere hacer dos cosas:

1. Probar la parte-si: Asumir Y y probar X.

2. Probar la parte-si-y-sólo-si: Asumir X y probar Y.

1.2.3 Equivalencia de Conjuntos

Muchos hechos importantes en la teoría del lenguaje son de la forma de dos conjuntos de cadenas, descritas de dos maneras diferentes y que realmente son el mismo conjunto. Para probar que los conjuntos S y T son los mismos, probar:

- $x \in S$ si y sólo si $x \in T$. Esto es:
  - Asumir que $x \in S$; probar que $x \in T$.
  - Asumir que $x \in T$; probar que $x \in S$.

Ejemplo: Probar que $1 + 3 + 5 + \ldots + (2n - 1) = n^2$.

Base: $P(1) \rightarrow 1 = 1^2$

Inducción: Suponemos $P(k) : 1 + 3 + 5 + \ldots + (2k - 1) = k^2$. Para probar que $P(k + 1)$ es verdadera:

$P(k + 1) \leftarrow 1 + 3 + 5 + \ldots + [2(k + 1) - 1] = (k + 1)^2$

y esto se puede escribir como:

$1 + 3 + 5 + \ldots + (2k - 1) + [2(k + 1) - 1] = (k + 1)^2$

y la parte izquierda es igual a:

$k^2 + [2(k + 1) - 1] = k^2 + 2k + 1 = (k + 1)^2$

por lo tanto,

$1 + 3 + 5 + \ldots + [2(k + 1) - 1] = (k + 1)^2$

esto verifica $P(k + 1)$ y prueba que la ecuación es verdadera para cualquier entero positivo n.

Ejemplo: Probar que $1 + 2 + 2^2 + \ldots + 2^n = 2^{n+1} - 1$ para toda $n \geq 1$.

Base: $P(1) \leftarrow 1 + 2 = 2^{1+1} - 1, 3 = 2^2 - 1$, lo cual es verdadero.

Inducción: Asumimos que $P(k) : 1+2+2^2+\ldots+2^k = 2^{k+1}-1$, es verdadero e intentamos probar que $P(k+1)$ también lo es:

$P(k+1) : 1 + 2 + 2^2 + ... + 2^{k+1} = 2^{(k+1)+1} - 1$

Rescribiendo la ecuación anterior tenemos:

$$\begin{aligned} 1 + 2 + 2^2 + ... + 2^k + 2^{k+1} &= 2^{k+1} - 1 + 2^{k+1} \\ &= 2(2^{k+1}) - 1 \\ &= 2^{k+1+1} - 1 \end{aligned}$$

Por lo tanto:

$$1 + 2 + 2^2 + \ldots + 2^{k+1} = 2^{(k+1)+1} - 1$$

Ejercicio: Probar que $x^0 + x^1 + x^2 + \ldots + x^n = (x^{n+1} - 1)/(x - 1)$

Base: $P(1) : x^0 + x^1 = x + 1$, y $(x^2 - 1)/(x - 1) = ((x + 1)(x - 1))/(x - 1) = (x + 1)$, por lo tanto, la base si se cumple.

Inducción: Asumimos que $P(k) : x^0 + x^1 + x^2 + \ldots + x^k = (x^{k+1} - 1)/(x - 1)$ se cumple.

Probamos que también $P(k + 1)$ se cumple:

$P(k+1) : x^0 + x^1 + x^2 + \ldots + x^k + x^{k+1} = \frac{x^{k+1+1}-1}{(x-1)}$

Demostramos que el lado izquierdo es igual al lado derecho:

$\frac{x^{k+1}-1}{(x-1)} + x^{k+1} = \frac{(x^{k+1}-1)+(x-1)(x^{k+1})}{(x-1)} = \frac{x^{k+2}-1}{x-1}$

Ejercicio: Probar que $n^2 > 3n$ para $n \geq 4$.

Base: $P(4) : 4^2 > 3(4)$ ó $16 > 12$, lo cual es cierto

Inducción: Asumimos $P(k) : k^2 > 3k$ para $k > 4$ y tratamos de probar que $P(k + 1) : (k + 1)^2 > 3(k + 1)$

$$\begin{aligned}(k+1)^2 &= k^2+2k+1\\ &> 3k+2k+1 \text{ (por la Hipótesis de Inducción)}\\ &> 3k+2(4)+1 \text{ (porque } k>4)\\ &> 3k+3\\ &> 3(k+1).\end{aligned}$$

Ejercicios a resolver en clase:

1. $2+6+10+\ldots+(4n-2)=2n^2$
2. $1+5+9+\ldots+(4n-3)=n(2n-1)$
3. $1+3+6+\ldots+n(n+1)/2=\frac{n(n+1)(n+2)}{6}$
4. $5+10+15+\ldots+5n=\frac{5n(n+1)}{2}$
5. $1^2+2^2+\ldots+^n 2=\frac{n(n+1)(2n+1)}{6}$
6. $1^3+2^3+\ldots+n^3=\frac{n^2(n+1)^2}{4}$
7. $n^2>n+1$ para $n>2$

Ejercicios de tarea para la siguiente clase:

1. $2+4+6+...+2n=n(n+1)$
2. $1^2+3^2+...+(2n-1)^2=\frac{n(2n-1)(2n+1)}{3}$
3. Probar que $2^n<n!$, para $n\geq 4$

1.3 Teoría de Autómatas

1.3.1 Motivación

Los autómatas finitos se utilizan como modelos para:

- Software para diseñar circuitos digitales
- Analizador léxico de un compilador
- Buscar palabras clave en un archivo ó en el web
- Software para verificar sistemas de estados finitos, como protocolos de comunicaciones

1.4 Conceptos de Teoría de Autómatas

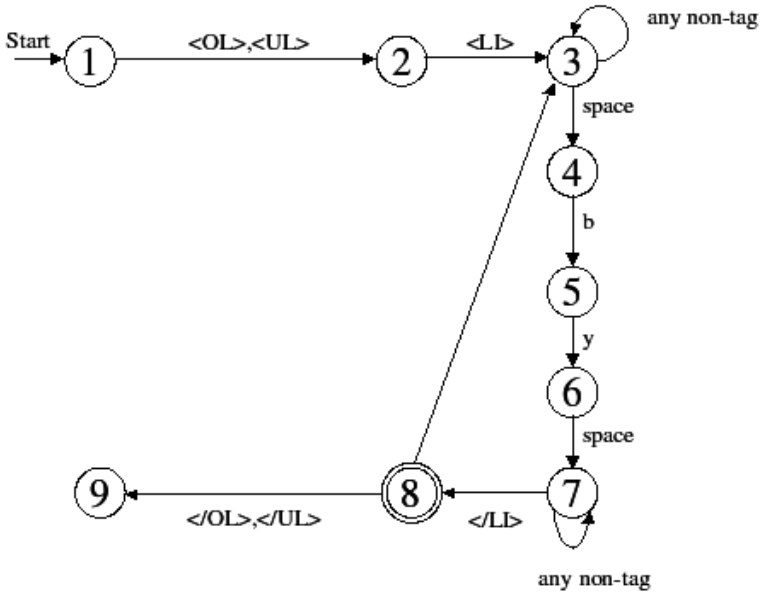
- Alfabeto = conjunto finito de símbolos (Σ)
  - $\Sigma = \{0, 1\}$, alfabeto binario.
  - $\Sigma = \{a, b, c, d, \ldots, z\}$, alfabeto del abecedario en minúsculas.
  - Conjunto de los caracteres ASCII.
- Cadena = secuencia finita de símbolos elegidos de un alfabeto.
  - 01101
  - *abracadabra*
  - La cadena vacía se denota como: ϵ,
  - Todas las cadenas de un alfabeto Σ de longitud k se denotan como Σ^k. E.g., $\Sigma^0 = \{\epsilon\}$, si $\Sigma = \{0, 1\}$, entonces, $\Sigma^1 = \{0, 1\}, \Sigma^2 = \{00, 01, 10, 11\}$, etc.
  - El conjunto de todas las cadenas de un alfabeto Σ se denota como Σ^*. $\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \ldots$. Sin la cadena vacía: Σ^+. Por lo que: $\Sigma^* = \Sigma^0 \cup \Sigma^+$.
- Lenguaje = conjunto de cadenas elegidas de algún alfabeto.
  - Nota: El lenguaje puede ser infinito, pero existe algún conjunto finito de símbolos de los cuales se componen todas sus cadenas.
  - El conjunto de todas las cadenas binarias que consisten de algún número de 0's seguidos por un número igual de 1's; esto es: $\{\epsilon, 01, 0011, 000111, \ldots\}$.

– C (el conjunto de programas compilables en C).

– Español.

Una manera importante de describir ciertos lenguajes simples, pero altamente útiles es la llamada "lenguajes regulares".

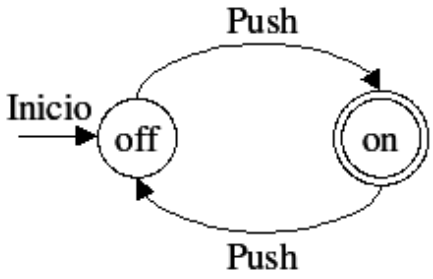
- Un grafo con un número finito de nodos, llamados *estados*.
- Los arcos se etiquetan con uno o más símbolos de algún alfabeto.
- Un estado es designado como el *estado de comienzo* ó *estado inicial*.
- Algunos estados son *estados finales* o *estados de aceptación*.
- El lenguaje de los **AF** (Autómata Finito) es el conjunto de cadenas que etiquetan rutas que van desde el estado inicial a algún estado de aceptación.
- Abajo, el **AF** explora documentos HTML, busca una lista de lo que podrían ser los pares de título-autor, quizás en una lista de lectura para algún curso de literatura.



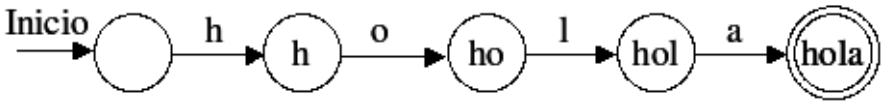
- Acepta cuando encuentra el final de un elemento de la lista.
- En una aplicación, las cadenas que casan con el título (antes de 'by') y autor (después) serían almacenadas en una tabla de pares de título-autor que son acumulados.

1.4.1 Ejemplos de Autómatas Finitos

Autómata Finito para modelar un switch de encendido/apagado:



Autómata Finito para reconocer la cadena "hola":



1.4.2 Representaciones Estructurales

- Gramáticas: Modelos útiles para diseñar SW que procesa datos con una estructura recursiva
  - Un parser de un compilador
  - Expresiones aritméticas, condicionales, etc.
  - Una regla: $E \Rightarrow E + E$
- Expresiones Regulares: Denotan la estructura de los datos, especialmente cadenas de texto
  - Autómatas pueden describir lo mismo que puede describir un autómata finito

- El estilo difiere del de las gramáticas
- Estilo de expresiones regulares de UNIX

  $[A-Z][a-z]*[\ ][A-Z][A-Z]$

  Representa palabras que inician con mayúscula seguidas de un espacio y dos letras mayúsculas (e.g., Ithaca NY)

  No incluye ciudades con nombres compuestos

  $([A-Z][a-z]*[\ ])*[A-Z][A-Z]$

1.5 Autómatas y Complejidad

Los autómatas son esenciales para el estudio de los límites de la computación

- Qué puede hacer una computadora?
  - Decidibilidad (decidability)
  - Problema decidible
- Qué puede hacer una computadora eficientemente?
  - Intractabilidad (intractability)
  - Polinomial vs Exponencial

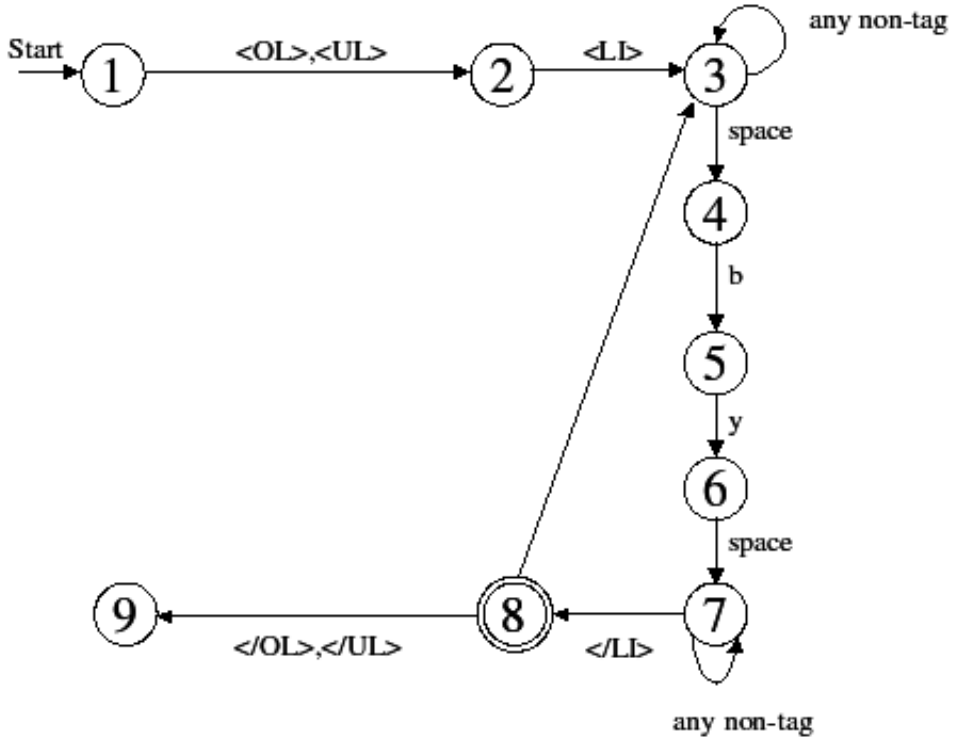
Ejercicios en clase:

1. Modelar un AF para cadenas de 0's y 1's que terminen siempre en 010.
2. Modelar un AF para las cadenas que terminan con tres 0's consecutivos.
3. Modelar un AF para todas las cadenas en donde cada bloque de 5 caracteres consecutivos contengan al menos dos 0's.

Ejercicios de tarea para la siguiente clase:

1. Modelar un AF donde el décimo símbolo de derecha a izquierda sea un 1.

2. Describir con palabras los conjuntos aceptados por los autómatas finitos de los diagramas de transiciones de las siguientes figuras:



Capítulo 2

Introducción a los Autómatas

Autómata: Conjunto de estados + Control → Cambio de estados en respuesta a una entrada.

Tipo de Control:

- Determinístico: Para cada entrada, hay sólo un estado al que el autómata puede ir desde el estado en que se encuentre.
- No determinístico: Un autómata finito es no-determinístico cuando se permite que el **AF** tenga 0 o más estados siguientes para cada par estado-entrada.

Si añadimos la propiedad de no-determinismo, no añadimos poder al autómata. Osea que no podemos definir ningún lenguaje que no se pueda definir con el autómata determinístico.

Con la propiedad de no-determinismo se agrega eficiencia al describir una aplicación:

- Permite programar soluciones en un lenguaje de más alto nivel
- Hay un algoritmo para compilar un N-DFA en un DFA y poder ser ejecutado en una computadora convencional

Extensión del N-DFA para hacer saltos de un estado a otro espontáneamente, con la cadena vacía (ϵ) como entrada: ϵN-DFA. Estos autómatas también aceptan lenguajes regulares.

Ejemplo: Compra con dinero-electrónico. El cliente utiliza un archivo (envío por internet) para pagar al comerciante. El comerciante al recibir el archivo pide al banco que le transfieran el dinero.

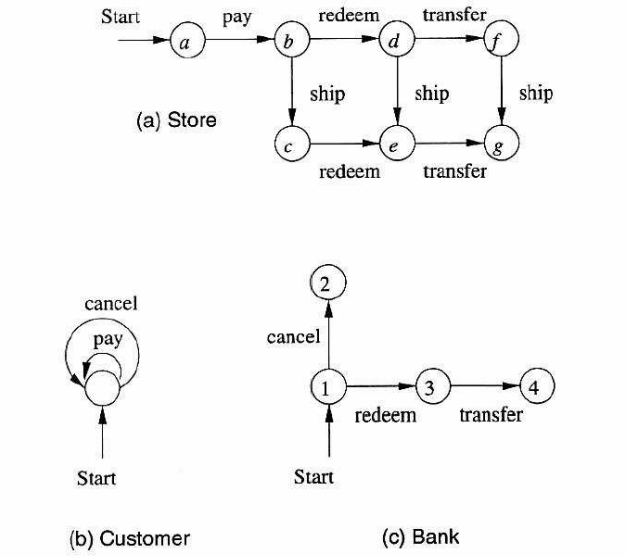


Figure 2.1: Finite automata representing a customer, a store, and a bank

Tarea: Leer la sección 2.1.1 – 2.1.5 del libro

2.1 Definición formal de un Autómata Finito Determinístico

Un AF se representa como la 5-tupla: $A = (Q, \Sigma, \delta, q_0, F)$. Donde:

1. Q es un conjunto finito de *estados*.
2. Σ es un alfabeto de *símbolos de entrada*.
3. q_0 es el estado *inicial/de comienzo*.

4. F representa cero o más *estados finales/de aceptación.*

5. δ es una *función de transición.* Esta función:

   - Toma un estado y un símbolo de entrada como argumentos.
   - Regresa un estado.
   - Una "regla" de δ se escribe como $\delta(q, a) = p$, donde q y p son estados, y a es un símbolo de entrada.
   - Intuitivamente: Si el **AF** está en un estado q, y recibe una entrada a, entonces el **AF** va al estado p (nota: $q = p$ OK).

Ejemplo: Un Autómata A que acepta $L = \{x01y | x, y \in \{0, 1\}^*\}$

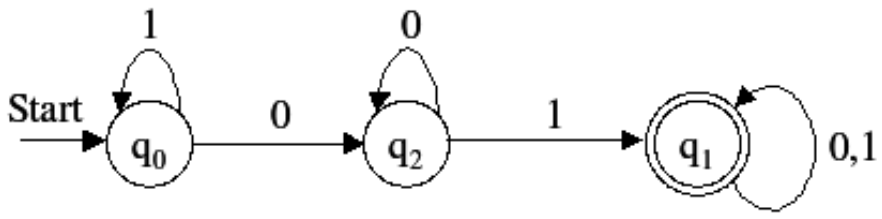
- El DFA acepta cadenas que tienen 01 en alguna parte de la cadena
- El lenguaje del DFA es el conjunto de cadenas que acepta $\{w|w$ tiene la forma $x01y$ para algunas cadenas x y y que consisten sólo de 0's y 1's.

El Autómata $A = (q_0, q_1, q_2, 0, 1, \delta, q_0, q_1)$

Autómata representado con una tabla de transiciones:

	0	1
$\rightarrow q_0$	q_2	q_0
$*q_1$	q_1	q_1
q_2	q_2	q_1

Autómata representado con un diagrama de transiciones:



Convenciones

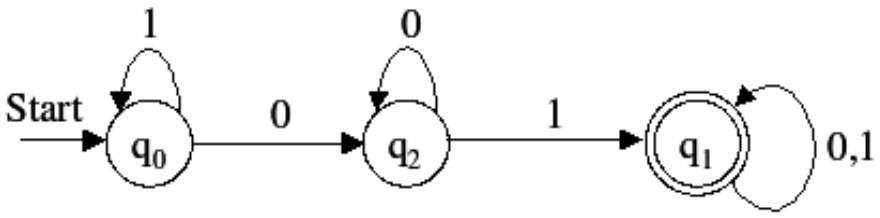
Se utilizan algunas convenciones para evitar mencionar el tipo de cada nombre, para esto utilizamos las siguientes reglas:

- Los símbolos de entrada son a, b, etc., o dígitos.
- Las cadenas de símbolos de entrada son $u, v, \ldots, z$.
- Los estados son q, p, etc.

Diagrama de Transiciones

Un **AF** se puede representar por medio de un grafo; los nodos representan estados; un arco de p a q se etiqueta con el conjunto de símbolos de entrada a tal que $\delta(q, a) = p$.

- No hay arco si a no existe.
- El estado de inicio se indica con la palabra "start" con una flecha.



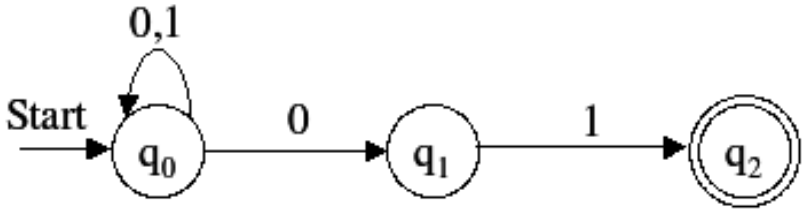
- Los estados de aceptación tienen doble círculo.

Funciones de transición extendidas ($\hat{\delta}$)

Intuitivamente, un **FA** acepta una cadena $w = a_1 a_2 \ldots a_n$ si hay una ruta en el diagrama de transiciones que:

1. Empieza en el estado de inicio,
2. Termina en un estado de aceptación, y
3. Tiene una secuencia de etiquetas $a_1, a_2, \ldots, a_n$.

Ejemplo: El siguiente **AF** acepta la cadena 01101:



Formalmente, extendemos la función de transición δ a $\hat{\delta}(q, w)$, donde w puede ser cualquier cadena de símbolos de entrada:

- Base: $\hat{\delta}(q, \epsilon) = q$ (i.e., nos quedamos en el mismo lugar si no recibimos una entrada).
- Inducción: $\hat{\delta}(q, w) = \delta(\hat{\delta}(q, x), a)$, donde x es una cadena, y a es un solo símbolo (i.e., ver a dónde va el AF con x, luego buscar la transición para el último símbolo a partir de ese estado).
- Hecho importante con una prueba inductiva directa: $\hat{\delta}$ realmente representa rutas. Esto es, si $w = a_1 a_2 \ldots a_n$, y $\delta(p_i, a_i) = p_{i+1}, \forall i = 0, 1, \ldots, n-1$, entonces $\hat{\delta}(p_0, w) = p_n$.

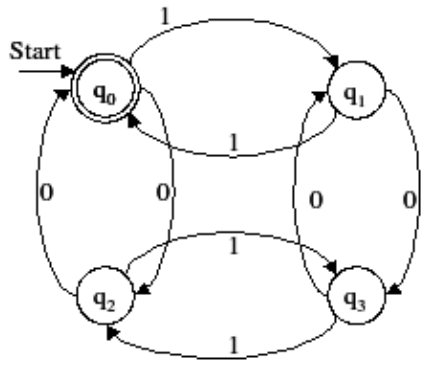
Aceptación de Cadenas: Un **AF** $A = (Q, \Sigma, \delta, q_0, F)$ acepta la cadena w si $\hat{\delta}(p_0, w)$ está en F.

Lenguaje de un AF: Un **AF** acepta el lenguaje $L(A) = w|\hat{\delta}(p0, w) \in F$.

Algunas confusiones frecuentes Una gran fuente de confusión cuando se trabaja con autómatas (o matemáticas en general) son los "errores de tipo":

- Ejemplo: No confundir A, un FA, i.e., un programa, con L(A), el cual es del tipo "conjunto de cadenas".
- Ejemplo: El estado de inicio q_0 es del tipo "estado" pero los estados de aceptación F son del tipo "conjunto de estados".
- Ejemplo engañoso: Es a un símbolo o una cadena de longitud 1? Respuesta: Depende del contexto, i.e., se usa en $\delta(q, a)$, donde es un símbolo, o en $\hat{\delta}(q, a)$, donde es una cadena?

Ejemplo: DFA que acepta todas y sólo las cadenas que tienen un número par de 0's y también un número par de 1's



- $\hat{\delta}(q_0, \epsilon) = q_0$.
- $\hat{\delta}(q_0, 1) = \delta\big(\hat{\delta}(q_0, \epsilon), 1\big) = \delta(q_0, 1) = q_1$.
- $\hat{\delta}(q_0, 11) = \delta\big(\hat{\delta}(q_0, 1), 1\big) = \delta(q_1, 1) = q_0$.
- $\hat{\delta}(q_0, 110) = \delta\big(\hat{\delta}(q_0, 11), 0\big) = \delta(q_0, 0) = q_2$.
- $\hat{\delta}(q_0, 1101) = \delta\big(\hat{\delta}(q_0, 110), 1\big) = \delta(q_2, 1) = q_3$.
- $\hat{\delta}(q_0, 11010) = \delta\big(\hat{\delta}(q_0, 1101), 0\big) = \delta(q_3, 0) = q_1$.
- $\hat{\delta}(q_0, 110101) = \delta\big(\hat{\delta}(q_0, 11010), 1\big) = \delta(q_1, 1) = q_0$.

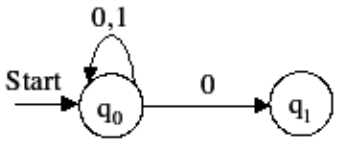
Representación tabular del autómata anterior:

	0	1
$\cdot \rightarrow q_0$	q_2	q_1
q_1	q_3	q_0
q_2	q_0	q_3
q_3	q_1	q_2

Ejemplo: Problema 2.2.1.a
Ejemplo: Problema 2.2.4.a

2.2 Autómata Finito No-Determinístico

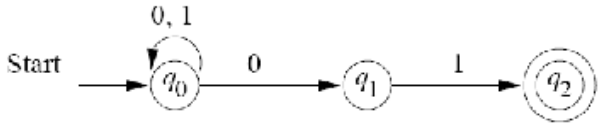
Un autómata finito es no-determinístico cuando se permite que el AF tenga 0 o más estados siguientes para cada par estado-entrada:



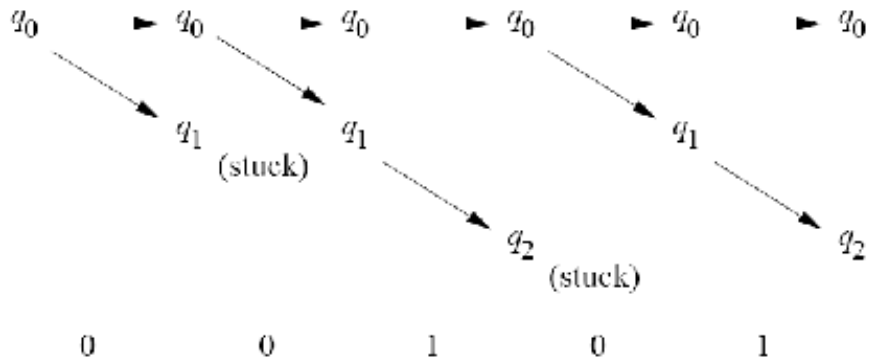
En el ejemplo anterior, se puede apreciar que de q_0 se puede ir a q_0 ó a q_1 con la entrada "0", y esto hace al AF ser no-determinista.

Un NFA puede estar en varios estados a la vez o se puede ver que "adivina" a qué estado ir.

Por ejemplo, el siguiente autómata acepta todas las cadenas que terminan en 01:



Lo que pasa al procesar como entrada a 00101 es:



Un NFA es una herramienta importante para diseñar procesadores de

cadenas, e.g., grep, analizadores léxicos, etc. Es fácil diseñar NFAs que encuentren secuencias de palabras en texto.

NFA: Formalmente, un NFA es una quíntupla $A = (Q, \Sigma, \delta, q_0, F)$, donde todo es un DFA, pero $\delta(q, a)$ nos regresa un conjunto de estados en lugar de un solo estado. De hecho puede ser vacío, tener un solo estado o tener más estados.

Un NFA acepta, al igual que un DFA, lenguajes regulares

Por ejemplo, para el NFA que acepta cadenas que acaban en 01 su función de transición δ es:

	0	1
$\rightarrow q_0$	$\{q_0, q_1\}$	$\{q_0\}$
q_1	$\emptyset$	$\{q_2\}$
q_2	$\emptyset$	$\emptyset$

Como puede observarse, todo se especifica en conjuntos.

2.2.1 Extensión a $\hat{\delta}$

Similarmente a un DFA, podemos definir la función de transición extendida $\hat{\delta}$ como sigue:

- Base: $\hat{\delta}(q, \epsilon) = q$
- Inducción: Supongamos w es de la forma $w = xa$, donde a es el símbolo terminal y x es el resto de w. Supongamos también que: $\hat{\delta}(q, x) = \{p_1, p_2, \ldots, p_k\}$.
  $\bigcup_{i=1}^{k} \delta(p_i, a) = \{r_1, r_2, \ldots, r_m\}$. Entonces $\cup\delta(q, w) = \{r_1, r_2, \ldots, r_m\}$.
  En otras palabras calculamos $\hat{\delta}(q, w)$ primero calculando $\hat{\delta}(q, x)$ y después siguiendo cualquier transición de algunos de esos estados etiquetada con a.

Por ejemplo, podemos calcular $\hat{\delta}(q_0, 00101)$ para el autómata anterior:
$\hat{\delta}(q_0, \epsilon) = \{q_0\}$

$\hat{\delta}(q_0, 0) = \delta(q_0, 0) = \{q_0, q_1\}$
$\hat{\delta}(q_0, 00) = \delta(q_0, 0) \cup \delta(q_1, 0) \cup \emptyset = \{q_0, q_1\}$
$\hat{\delta}(q_0, 001) = \delta(q_0, 1) \cup \delta(q_1, 1) = \{q_0\} \cup \{q_2\} = \{q_0, q_2\}$
$\hat{\delta}(q_0, 0010) = \delta(q_0, 0) \cup \delta(q_2, 0) = \{q_0, q_1\} \cup \emptyset = \{q_0, q_1\}$
$\hat{\delta}(q_0, 00101) = \delta(q_0, 1) \cup \delta(q_1, 1) = \{q_0\} \cup \{q_2\} = \{q_0, q_2\}$

Que tiene un estado final.

2.2.2 Lenguajes de un NFA

El lenguaje aceptado por un NFA, A, es: $L(A) = \{w|\hat{\delta}(q_0, w) \cap F \neq \emptyset\}$.

2.2.3 Equivalencia entre un DFA y un NFA

Un NFA es normalmente más fácil de definir, aunque al mismo tiempo, para cualquier NFA N existe un DFA D tal que $L(D) = L(N)$ y viceversa.

Para esto se usa la construcción de subconjunto que muestra un ejemplo de cómo un autómata se puede construir a partir de otro.

2.2.4 Construcción de Subconjunto

Para cada NFA existe un DFA equivalente (acepta el mismo lenguaje). Pero el DFA puede tener un número exponencial de estados.

Sea $N = (Q_N, \Sigma, \delta_N, q_0, F_N)$ un NFA. El DFA equivalente construido a partir del subconjunto de construcción es $D = (Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$, donde:

- $|Q_D| = 2^{|Q_N|}$; i.e., Q_D es el conjunto de todos los subconjuntos de Q_N.
- F_D es el conjunto de conjuntos S en Q_D tal que $S \cap F_N \neq \emptyset$.
- Para cualquier $S \subseteq Q_N$ y $a \in \Sigma$, $\delta_D(S, a) = \bigcup_{p \in S} \delta_N(p, a)$, osea, la unión de todos los estados a partir de p con entrada a. $\delta_D(\{q_1, q_2, \ldots, q_k\}, a) = \delta_N(p_1, a) \cup \delta_N(p_2, a) \cup \ldots \cup \delta_N(p_k, a)$.

La función de transición δ_D del NFA anterior es:

	0	1
$\emptyset$	$\emptyset$	$\emptyset$
$\rightarrow \{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$
$\{q_1\}$	$\emptyset$	$\{q_2\}$
$\star\{q_2\}$	$\emptyset$	$\emptyset$
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$
$\star\{q_0, q_2\}$	$\{q_0, q_1\}$	$\{q_0\}$
$\star\{q_1, q_2\}$	$\emptyset$	$\{q_2\}$
$\star\{q_0, q_1, q_2\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$

Al existir 3 estados, tenemos 8 subconjuntos. Esto mismo lo podemos poner como:

	0	1
A	A	A
$\rightarrow B$	E	B
C	A	D
$\star D$	A	A
E	E	F
$\star F$	E	B
$\star C$	A	D
$\star H$	E	F

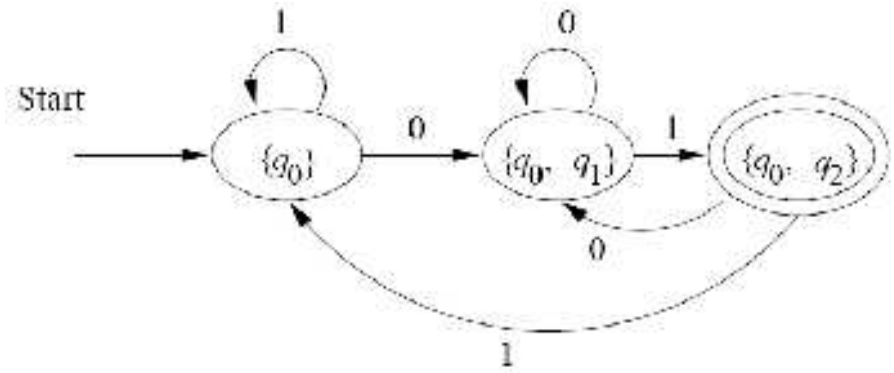
Lo cual es un DFA (simplemente cambiando la notación). También es importante notar que no todos los estados pueden ser alcanzados. En particular, sólo los estados B, E y F son accesibles, por lo que los demás los podemos eliminar.

Una forma de no construir todos los subconjuntos para después encontrar que sólo unos cuantos son accesibles, es construir la tabla sólo para los estados accesibles (*lazy evaluation*).

Para el ejemplo anterior: $\delta_D(q_0, 0) = q_0, q_1$
$\delta_D(q_0, 1) = q_1$
$\delta_D(q_0, q_1, 0) = q_0, q_1$

$\delta_D(q_0, q_1, 1) = q_0, q_2 = \delta_N(q_0, 1) \cup \delta_N(q_1, 1)$
$\delta_D(q_0, q_2, 0) = q_0, q_1$
$\delta_D(q_0, q_2, 1) = q_0$

Lo que nos queda:



Teorema clave: inducción de $|w|$ (la prueba está en el libro): $\hat{\delta}_D(\{q_0\}, w) = \hat{\delta}_N(q_0, w)$. Lo que queremos probar es que si $D = (Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$ es construido a partir del NFA $N = (Q_N, \Sigma, \delta_N, q_0, F_N)$ usando construcción de subconjuntos, entonces $L(D) = L(N)$.

Queremos probar por inducción en w que $\hat{\delta}_D(\{q_0\}, w) = \hat{\delta}_N(q_0, w)$. Las dos funciones de transición regresan conjuntos de estados de Q_N, pero la determinística lo interpreta como uno solo de sus estados Q_D.

Base: $w = \epsilon$, en este caso $\hat{\delta}_D(\{q_0\}, \epsilon) = \hat{\delta}_N(q_0, \epsilon)$.

Inducción: Tomamos w de longitud $n + 1$ y asumimos que se cumple el enunciado para n, o sea que $\hat{\delta}_D(\{q_0\}, x) = \hat{\delta}_N(q_0, x)$. Sean estos dos conjuntos de estados $= \{p_1, p_2, \ldots, p_k\}$. Dividimos a w en xa. La definición de $\hat{\delta}$ para el NFA nos dice que: $\hat{\delta}_N(q_0, w) = \bigcup_{i=1}^{k} \delta_N(p_i, a)$.

Por la construcción de subconjuntos: $\delta_D(\{p_1, p_2, \ldots, p_k\}, a) = \bigcup_{i=1}^{k} \delta_N(p_i, a)$

Usando esto, y que $\hat{\delta}_D(\{q_0\}, x) = \{p_1, p_2, \ldots, p_k\}$ tenemos que:

$\hat{\delta}_D(\{q_0\}, w) = \delta_D(\hat{\delta}_D(\{q_0\}, x), a) = \delta_D(\{p_1, p_2, \ldots, p_k\}, a) = \bigcup_{i=1}^{k} \delta_N(p_i, a)$

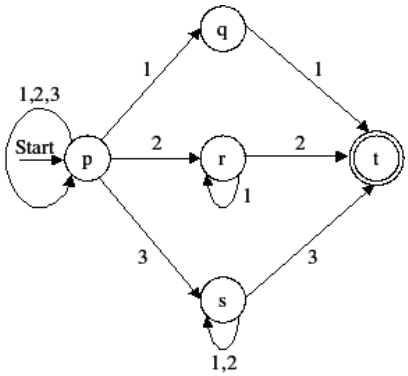
Tanto D como N aceptan w cuando contiene un estado en FN.

Consecuencia: $L(D) = L(N)$.

Ejemplo: Problema 2.3.1

Ejemplo: En este ejemplo un tanto imaginario, se diseñará un **NFA** para aceptar cadenas sobre el alfabeto $\{1, 2, 3\}$ de tal manera que el último símbolo aparezca previamente, sin ninguna intervención de un símbolo más alto entre esa previa aparición del símbolo, e.g., $\ldots 11, \ldots 21112, \ldots 312123$.

- Truco: Utilizar el estado de inicio con el significado "Creo que todavía no se ha visto el símbolo que corresponde al símbolo final".
- Otros tres estados representando una elección de que el símbolo con que acaba la cadena se ha visto y se recuerda de que símbolo se trata.



Ejemplo: Subconjunto de Construcción del NFA Previo.

Un truco práctico importante utilizado por analizadores léxicos y otros procesadores de texto es ignorar los (frecuentemente muchos) estados que no son accesibles desde el estado de inicio (i.e., no hay ruta que lleve a ellos). Para el ejemplo anterior de NFA, de los 32 subconjuntos posibles, solo 15 son accesibles. Calculando las transiciones "por demanda" obtenemos el siguiente δ_D:

	1	2	3
$\longrightarrow$ p	pq	pr	Ps
pq	pqt	pr	Ps
pqt	pqt	pr	Ps
pr	pqr	prt	Ps
prt	pqr	prt	Ps
ps	pqs	prs	Pst
pst	pqs	prs	Pst
prs	pqrs	prst	Pst
prst	pqrs	prst	Pst
pqs	pqst	prs	Pst
pqst	pqst	prs	Pst
pqr	pqrt	prt	Ps
pqrt	pqrt	prt	Ps
pqrs	pqrst	prst	Pst
pqrst	pqrst	prst	Pst

Ejemplo: Problema 2.4.1.a

2.3 Autómatas Finitos y Lenguajes Formales

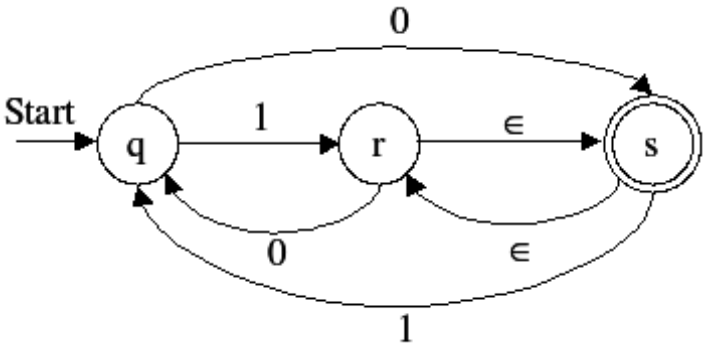
2.3.1 Autómata Finito con Transiciones-ϵ

Sea ϵ una etiqueta en arcos.

No hay ningún cambio extra: la aceptación de w todavía se da como la existencia de la ruta desde un estado de inicio a un estado de aceptación con etiqueta w.

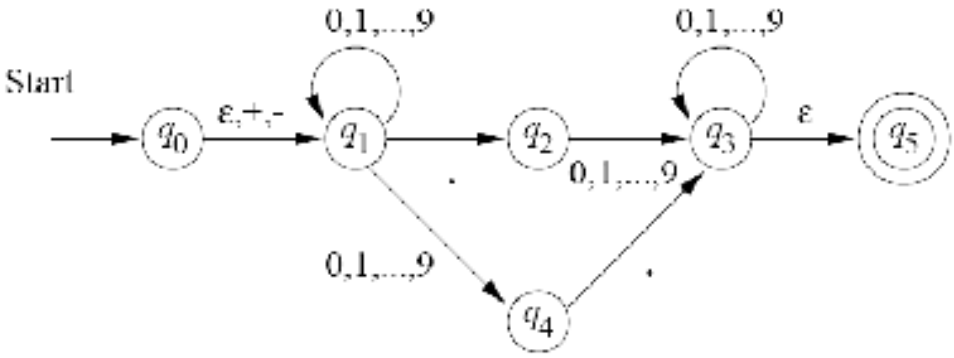
Pero ϵ puede aparecer en los arcos, y significa que hay una cadena vacía (i.e., no tiene una contribución visible para w).

Ejemplo:



001 es aceptado siguiendo la ruta q, s, r, q, r, s, con la etiqueta $0\epsilon 01\epsilon = 001$.

Podemos diseñar un autómata que acepte cadenas de números que tengan un signo al inicio opcional, seguida posiblemente de una cadena de decimales, seguida de un punto decimal y posiblemente de otra cadena de decimales.



Más formalmente: Un ϵ-NFA es una quíntupla $(Q, \Sigma, \delta, q_0, F)$, donde δ es una función de $Q \times \Sigma \cup \{\epsilon\}$ al conjunto potencia de Q.

La tabla de transición del ϵ-NFA del ejemplo anterior es:

	ϵ	$+,-$	.	$0,\ldots,9$
$\rightarrow q_0$	$\{q_1\}$	$\{q_1\}$	$\emptyset$	$\emptyset$
q_1	$\emptyset$	$\emptyset$	$\{q_2\}$	$\{q_1, q_4\}$
q_2	$\emptyset$	$\emptyset$	$\emptyset$	$\{q_3\}$
q_3	$\{q_5\}$	$\emptyset$	$\emptyset$	$\{q_3\}$
q_4	$\emptyset$	$\emptyset$	$\{q_3\}$	$\emptyset$
$\star q_5$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

2.4 Eliminación de las Transiciones-ϵ

Las transiciones-ϵ son una conveniencia, pero no incrementan la potencia de los FA's. Para eliminar las transiciones-ϵ:

1. Calcular la cerradura transitiva sólo para los arcos ϵ

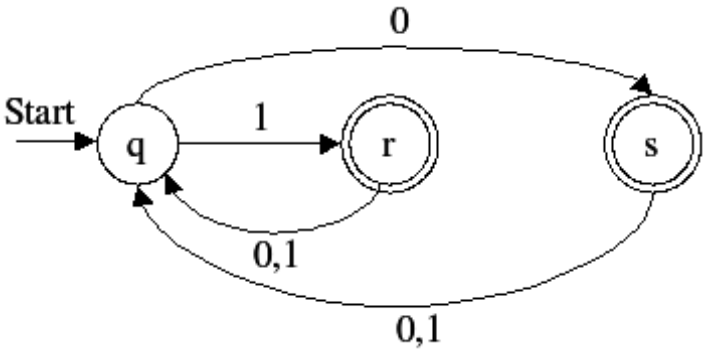
   Ejemplo:

   $q \longrightarrow \{q\}; r \longrightarrow \{r, s\}; s \longrightarrow \{r, s\}$.

2. Si un estado p puede alcanzar al estado q por medio de arcos ϵ, y existe una transición de q a r en la entrada a (no ϵ), entonces añádase una transición de p a r con la entrada a.

3. Convertir el estado p en un estado de aceptación siempre y cuando p pueda alcanzar algún estado de aceptación q por medio de arcos ϵ.

4. Eliminar todas las transiciones-ϵ.



Ejemplo:

De la misma forma como lo hicimos anteriormente, podemos definir las transiciones extendidas para ϵ-NFA.

Base: $\hat{\delta}(q, \epsilon) = ECLOSE(q)$

Inducción: $\hat{\delta}(q, xa) = \bigcup_{p \in \delta(\hat{\delta}(q,x),a)} ECLOSE(p)$

Por ejemplo, $\hat{\delta}(q_0, 5.6)$ es: $\hat{\delta}(q_0, \epsilon) = \{q_0, q_1\} = ECLOSE(q_0)$
$\delta(q_0, 5) \cup \delta(q_1, 5) = \{q_1, q_4\}$
$ECLOSE(q_1) \cup ECLOSE(q_4) = \{q_2, q_3\}$ $\delta)q_1, .) \cup \delta(q_4, .) = \{q_1, q_4\}$
$ECLOSE(q_2) \cup ECLOSE(q_3) = \{q_2, q_3, q_5\} = \hat{\delta}(q_0, 5.)$
$\delta)q_2, 6) \cup \delta(q_3, 6) \cup \delta(q_5, 6) = \{q_3\}$
$\delta(q_3) = \{q_3, q_5\} = \hat{\delta}(q_0, 5.6)$

Como antes, el lenguaje aceptado por un ϵ-NFA, E, es: $L(E) = \{w | \hat{\delta}(q_0, w) \cap F \neq \emptyset\}$, osea todas las cadenas w que van de un estado inicial q_0 a al menos un estado final.

Se puede demostrar que se puede construir un DFA a partir de un ϵ-NFA siguiendo un esquema parecido al de construcción de subconjuntos visto para NFA.

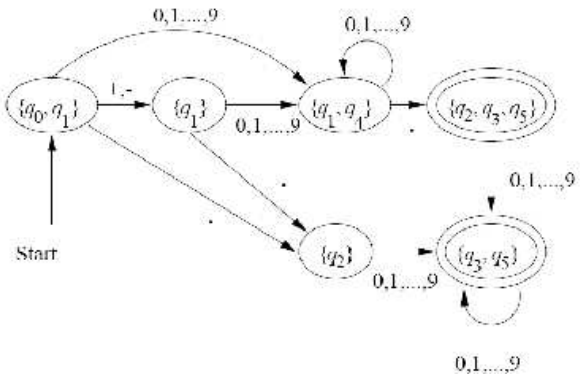
$Q_D = \{S | S \subseteq Q_E \wedge S = ECLOSE(S)\}$
$q_D = ECLOSE(q_0)$

$F_D = \{S | S \subseteq Q_D \wedge S \cap F_E \neq \emptyset\}$
$\delta_D(S, a) = \bigcup ECLOSE(p) | p \in \delta(t, a), t \in S\}$

lo que se calcula para todos los $a \in \Sigma$ y conjuntos $S \in Q_D$.

Ejemplo: Problema 2.5.6

Por ejemplo, el DFA correspondiente al ϵ-NFA de números decimales (omitiendo todas las transiciones a estados "muertos") es:



Se puede demostrar que un lenguaje L es aceptado por algún ϵ-NFA E si y solo si L es aceptado por un DFA. Para esto, hacia un sentido es fácil (cambiando un DFA a un ϵ-NFA) y para el otro sentido se hace lo mismo que hemos hecho antes, probando con el caso base y el caso inductivo, donde partimos $w = xa$, asumimos que es verdad para x y probamos para w, solo que ahora tomando la cerradura ϵ o ECLOSE para los estados.

Capítulo 3

Expresiones Regulares

Es un equivalente algebraico para un autómata.

- Utilizado en muchos lugares como un lenguaje para describir patrones en texto que son sencillos pero muy útiles.
- Pueden definir exactamente los mismos lenguajes que los autómatas pueden describir: Lenguajes regulares
- Ofrecen algo que los autómatas no: Manera declarativa de expresar las cadenas que queremos aceptar
- Ejemplos de sus usos
  - Comandos de búsqueda, e.g., grep de UNIX
  - Sistemas de formateo de texto: Usan notación de tipo expresión regular para describir patrones
  - Convierte la expresión regular a un DFA o un NFA y simula el autómata en el archivo de búsqueda
  - Generadores de analizadores-léxicos. Como Lex o Flex.
  - Los analizadores léxicos son parte de un compilador. Dividen el programa fuente en unidades lógicas (tokens). okens como while, números, signos ($+, -, <$, etc.)
  - Produce un DFA que reconoce el token

3.1 Operadores y Operandos

Si E es una expresión regular, entonces $L(E)$ denota el lenguaje que define E. Las expresiones se construyen de la manera siguiente:

- Un operando puede ser:
  1. Una variable, que pertenece o se representa por medio de un lenguaje.
  2. Un símbolo, que se representa a sí mismo como un conjunto de cadenas, i.e., a representa al lenguaje $\{a\}$ (formalmente, $L(a) = \{a\}$).
  3. ϵ, representado por $\{\epsilon\}$ (un lenguaje).
  4. $\emptyset$, representando a $\emptyset$ (el lenguaje vacío).
- Los operadores son:
  1. $+$, que representa la unión. $L(E + F) = L(E) \cup L(F)$.
  2. Yuxtaposición (i.e., símbolo de no operador, como en xy que significa $x \times y$) para representar la concatenación. $L(EF) = L(E)L(F)$, donde la concatenación de los lenguajes L y M es $\{xy | x \in L \text{ y } y \in M\}$, la concatenación también se representa por un punto "$\cdot$" ó como "dot".
  3. $*$ para representar la cerradura. $L(E^*) = (L(E))^*$, donde $L^* = \{\epsilon\} \cup L \cup LL \cup LLL \cup \ldots$.
  4. Si E es una expresión regular, entonces E^* es una expresión regular, que denota la cerradura de $L(E)$. Esto es, $L(E^*) = (L(E))^*$.
  5. Si E es una expresión regular, entonces (E), E entre paréntesis, también es una expresión regular que denota el mismo lenguaje que E. Formalmente: $L((E)) = L(E)$.
- Precedencia de los operadores de Expresiones-Regulares
  1. El asterisco de la cerradura tiene la mayor precedencia. Aplica sólo a la secuencia de símbolos a su izquierda que es una expresión regular bien formada

2. Concatenación sigue en precedencia a la cerradura, el operador "dot".
3. Después de agrupar los asteriscos a sus operandos, se agrupan los operadores de concatenación a sus operandos
4. Se agrupan todas las expresiones yuxtapuestas (adyacentes sin operador interviniendo)
5. Concatenación es asociativa y se sugiere agrupar desde la izquierda (i.e. 012 se agrupa (01)2).
6. La unión (operador +) tiene la siguiente precedencia, también es asociativa.
7. Los paréntesis pueden ser utilizados para alterar el agrupamiento

Ejemplos

- $L(001) = 001$.
- $L(0 + 10^*) = \{0, 1, 10, 100, 1000, \ldots\}$.
- $L((0(0 + 1))^*)$ = el conjunto de cadenas de 0's y 1's, de longitud par, de tal manera que cada posición impar tenga un 0.

Ejemplo 3.2: Expresión regular para el conjunto de cadenas que alterna 0's y 1's:

- $(01)^* + (10)^* + 0(10)^* + 1(01)^*$
- $(\epsilon + 1)(01)^*(\epsilon + 0)$

3.2 Equivalencia de Lenguajes de FA y Lenguajes RE

- Se mostrará que un NFA con transiciones-ϵ puede aceptar el lenguaje de una RE.

- Después, se mostrará que un RE puede describir el lenguaje de un DFA (la misma construcción funciona para un NFA).
- Los lenguajes aceptados por DFA, NFA, ϵ-NFA, RE son llamados lenguajes regulares.

3.2.1 De DFA's a Expresiones Regulares

Teorema 3.4: Si $L = L(A)$ para algún DFA A, entonces existe una expresión regular R tal que $L = L(R)$.

Prueba: Suponiendo que A tiene estados $\{1, 2, \ldots, n\}$, n finito. Tratemos de construir una colección de RE que describan progresivamente conjuntos de rutas del diagrama de transiciones de A

$R_{ij}^{(k)}$ es el nombre de la RE con lenguaje el conjunto de cadenas w

w es la etiqueta de la ruta del estado i al estado j de A. Esta ruta no tiene estado intermedio mayor a k. Los estados inicial y terminal no son intermedios, i y/o j pueden ser igual o menores que k

Para construir $R_{ij}^{(k)}$ se utiliza una definición inductiva de $k = 0$ hasta $k = n$

BASE: $k = 0$, implica que no hay estados intermedios. Sólo dos clases de rutas cumplen con esta condición:

1. Un arco del nodo (estado) i al nodo j
2. Una ruta de longitud 0 con un solo nodo i

Si $i \neq j$, solo el caso 1 es posible.

Examinar el DFA A y encontrar los símbolos de entrada a tal que hay una transición del estado i al estado j con el símbolo a

- Si no hay símbolo a, entonces $R_{ij}^{(0)} = \emptyset$.

- Si hay sólo un símbolo a, entonces $R_{ij}^{(0)} = a$.
- Si hay varios símbolos $a_1, a_2, \ldots, a_k$, entonces $R_{ij}^{(0)} = a_1 + a_2 + \ldots + a_k$.

Si $i = j$, sólo se permiten rutas de longitud 0 y loops del estado i a él mismo.

- La ruta de longitud 0 se representa con ϵ
- Si no hay símbolo a, entonces $R_{ij}^{(0)} = \epsilon$.
- Si hay sólo un símbolo a, entonces $R_{ij}^{(0)} = \epsilon + a$.
- Si hay varios símbolos $a_1, a_2, \ldots, a_k$, entonces $R_{ij}^{(0)} = \epsilon + a_1 + a_2 + \ldots + a_k$.

INDUCCIÓN: Suponemos que hay una ruta del estado i al estado j que no pasa por ningún estado mayor que k.

Se consideran 2 casos.

1. La ruta no pasa por el estado k: Etiqueta de la ruta esta en el lenguaje $R_{ij}^{(k-1)}$.
2. La ruta pasa por el estado k al menos una vez:

   Se divide la ruta en varias partes, una división cada que se pasa por el estado k

   Primero del estado i al estado k, después, varios pasos del estado k a sí mismo, finalmente, del estado k al estado j.

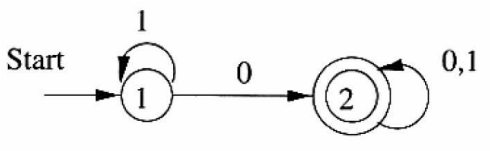
   Las etiquetas de estas rutas se representan con la RE $R_{ik}^{(k-1)}(R_{kk}^{(k-1)})^* R_{kj}^{(k-1)}$

Si combinamos las expresiones de las rutas de los dos tipos: $R_{ij}^{(k)} = R_{ij}^{(k-1)} + R_{ik}^{(k-1)}(R_{kk}^{(k-1)})^* R_{kj}^{(k-1)}$ para todas las etiquetas de las rutas del estado i al j que no pasan por estados mayores que k.

Eventualmente tendremos $R_{ij}^{(n)}$.

Asumimos que 1 es el estado inicial. El estado de aceptación puede ser un conjunto de estados La expresión regular para el lenguaje del autómata es la suma (unión) de todas las expresiones $R_{1j}^{(n)}$ tal que j es un estado de aceptación.

Ejemplo: Un DFA que acepta todas las cadenas que tienen al menos un 0



Inicialmente sustituimos para la base: $R_i j^{(0)} = \epsilon\ R_{ij}^{(0)} = \epsilon + a\ R_{ij}^{(0)} = \epsilon + a_1 + a_2 + \ldots + a_k$

$R_{11}^{(0)}$	$\epsilon + 1$
$R_{12}^{(0)}$	0
$R_{21}^{(0)}$	$\emptyset$
$R_{22}^{(0)}$	$(\epsilon + 0 + 1)$

Ahora para el paso de inducción: $R_{ij}^{(1)} = R_{ij}^{(0)} + R_{i1}^{(0)}(R_{11}^{(0)})^* R_{1j}^{(0)}$

	Por sustitución directa	Simplificado
$R_{11}^{(1)}$	$\epsilon + 1 + (\epsilon + 1)(\epsilon + 1)^*(\epsilon + 1)$	1^*
$R_{12}^{(1)}$	$0 + (\epsilon + 1)(\epsilon + 1)^*0$	1^*0
$R_{21}^{(1)}$	$\emptyset + \emptyset(\epsilon + 1)^*(\epsilon + 1)$	$\emptyset$
$R_{22}^{(1)}$	$\epsilon + 0 + 1 + \emptyset(\epsilon + 1)^*0$	$\epsilon + 0 + 1$

$R_{ij}^{(2)} = R_{ij}^{(1)} + R_{i2}^{(1)}(R_{22}^{(1)})^* R_{2j}^{(1)}$

	Por sustitución directa	Simplificado
$R_{11}^{(2)}$	$1^* + 1^*0(\epsilon + 0 + 1)^*\emptyset$	1^*
$R_{12}^{(2)}$	$1^*0 + 1^*0(\epsilon + 0 + 1)^*(\epsilon + 0 + 1)$	$1^*0(0 + 1)^*$
$R_{21}^{(2)}$	$\emptyset + (\epsilon + 0 + 1)(\epsilon + 0 + 1)^*\emptyset$	$\emptyset$
$R_{22}^{(2)}$	$\epsilon + 0 + 1 + (\epsilon + 0 + 1)(\epsilon + 0 + 1)^*(\epsilon + 0 + 1)$	$(0 + 1)^*$

Construcción de la RE final:

Unión de todas las expresiones donde el primer estado es el estado inicial y el segundo el estado de aceptación

- Estado inicial: 1
- Estado final: 2
- Sólo necesitamos $R_{12}^{(2)}$
- $1^*0(0+1)^*$

Este método funciona también para NFA y ϵ-NFA pero su construcción es muy costoso, hasta en el orden de 4^n símbolos.

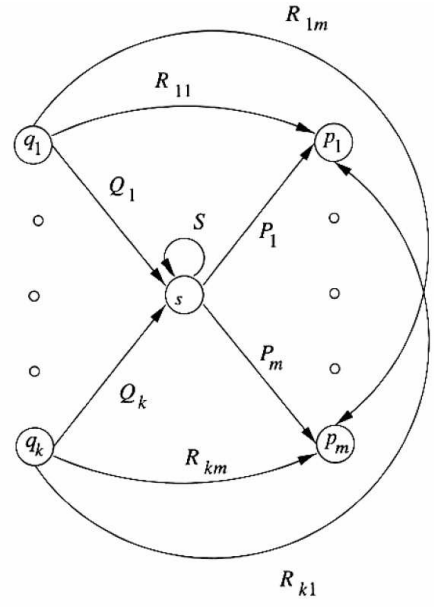
3.2.2 Conversión de un DFA a una RE por Eliminación de Estados

Evita duplicar trabajo en algunos puntos del teorema 3.4.

Ahora utilizaremos autómatas que podrán tener RE como etiquetas.

El lenguaje del autómata es la unión de todas las rutas que van del estado inicial a un estado de aceptación.

- Concatenando los lenguajes de las RE que van a través de la ruta.
- En la siguiente figura se muestra un autómata al cual se va a eliminar el estado "s".



- Se eliminan todos los arcos que incluyen a "s"
- Se introducen, para cada predecesor q_i de s y cada sucesor p_j de s, una RE que representa todas las rutas que inician en q_i, van a s, quizás hacen un loop en s cero o más veces, y finalmente van a p_j.

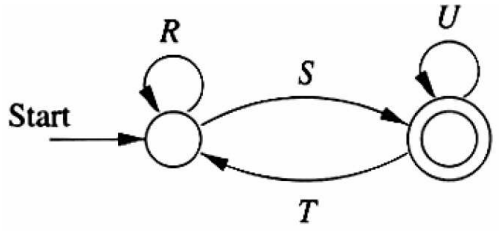
  La expresión para estas rutas es $Q_i S^* P_j$.

  Esta expresión se suma (con el operador unión) al arco que va de q_i a p_j.

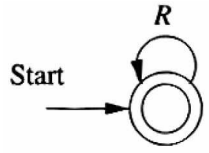
  Si este arco no existe, se añade primero uno con la RE $\emptyset$
- El autómata resultante después de la eliminación de "s" es el siguiente

3.2.2.1 Estrategia para construir el autómata

1. Para cada estado de aceptación q, aplicar el proceso de reducción para producir un autómata equivalente con RE como etiquetas en los arcos. Eliminar todos los estados excepto q y el estado inicial q_0.
2. Si $q \neq q_0$, se genera un autómata con 2 estados como el siguiente, una forma de describir la RE de este autómata es $(R + SU^*T)^*SU^*$

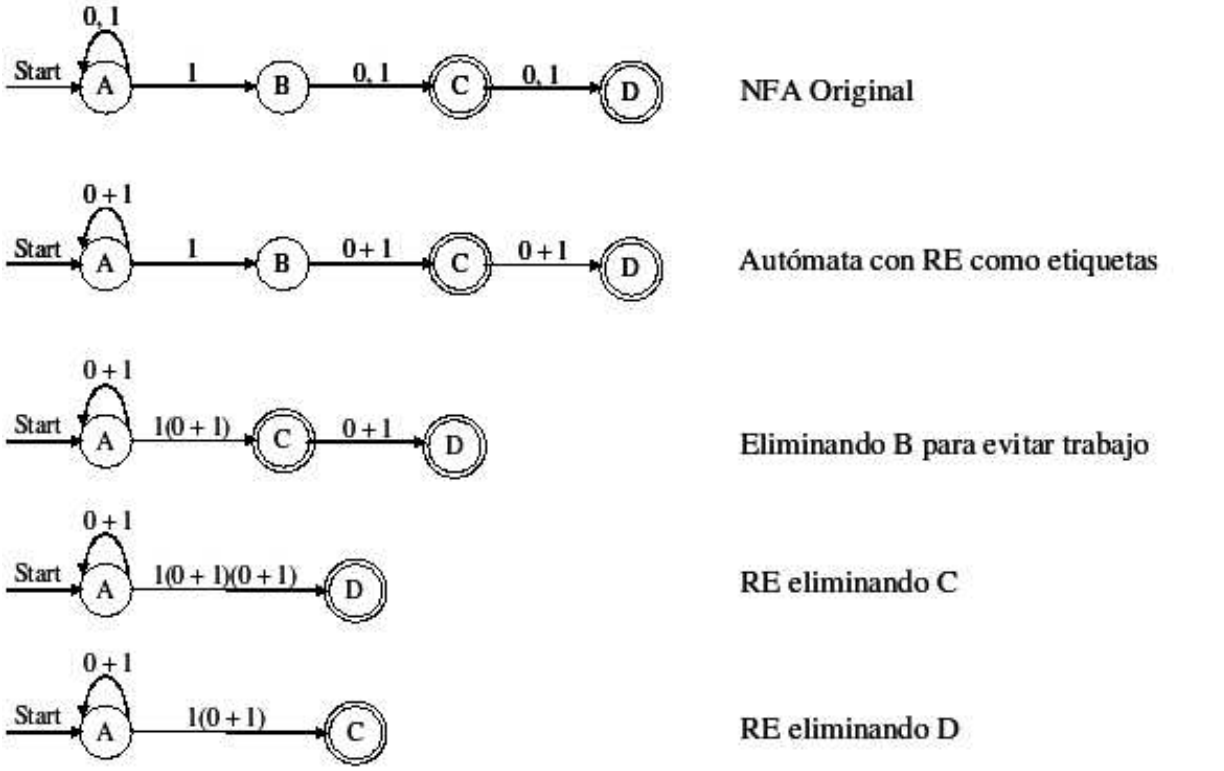


3. Si el estado inicial también es un estado de aceptación, también se debe hacer una eliminación de estados del autómata original que elimine todos los estados menos el inicial y dejamos un autómata como el siguiente:



4. La RE final es la suma (unión) de todas las expresiones derivadas del autómata reducido para cada estado de aceptación por las reglas 2 y 3.

Ejemplo: para el siguiente NFA que acepta cadenas de 0's y 1's de manera que tienen un 1 dos o tres posiciones antes del final.



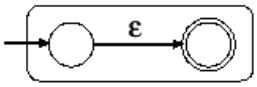
3.2.3 Convirtiendo una RE a un Autómata

Teorema 3.7: Todo lenguaje definido por una RE también esta definido por un autómata finito.

Prueba: Suponemos $L = L(R)$ para la expresión regular R. Mostramos que $L = L(E)$ para algún ϵ-NFA E con:

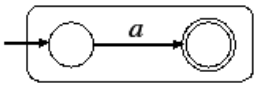
1. Exactamente un estado de aceptación
2. Sin arcos que lleguen al estado inicial
3. Sin arcos que salgan del estado de aceptación

Base: cumpliendo las condiciones 1, 2, y 3.



El lenguaje es ∈

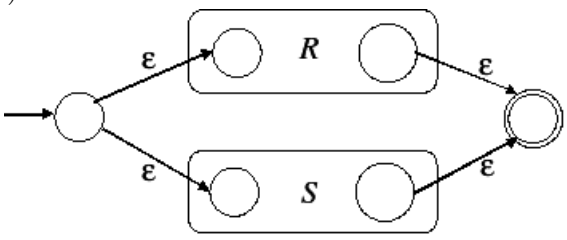
El lenguaje es ϕ



El lenguaje es la RE *a*, y
sólo contiene la cadena *a*

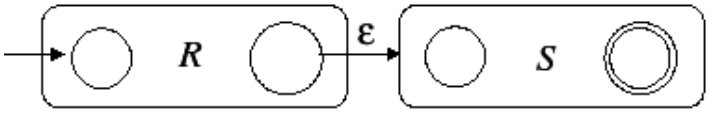
Inducción:

a)



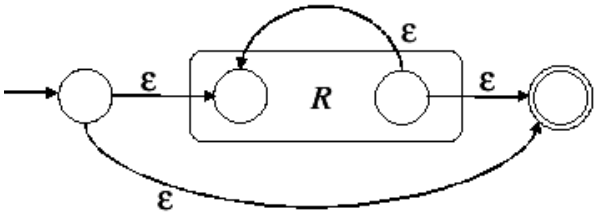
Lenguaje: $L(R) \cup L(S)$

b)



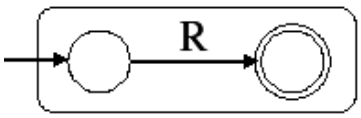
Lenguaje: $L(R)L(S)$

c)



El lenguaje es la R^*

d)



El lenguaje es R ó (R)

Ejemplo 3.8: Convertir la RE $(0+1)^*1(0+1)$ a un ϵ-NFA.

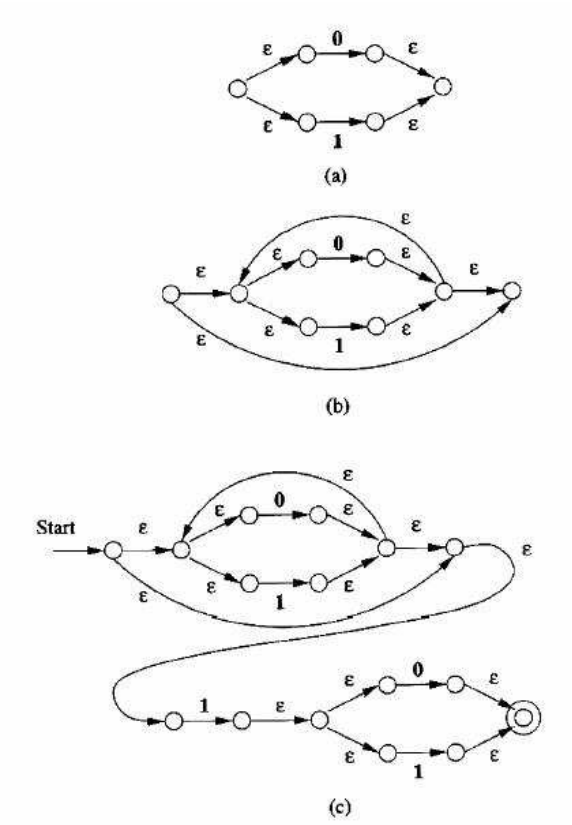


Figure 3.18: Automata constructed for Example 3.8

3.3 Leyes Algebraicas de las Expresiones Regulares

3.3.1 Asociatividad y Conmutatividad

- Ley conmutativa para la unión: $L + M = M + L$
- Ley asociativa para la unión: $(L + M) + N = L + (M + N)$
- Ley asociativa para la concatenación: $(LM)N = L(MN)$

NOTA: La concatenación no es conmutativa, es decir $LM \neq ML$

3.3.2 Identidades y Aniquiladores

- Una identidad para un operador es un valor tal que cuando el operador se aplica a la identidad y a algún otro valor, el resultado es el otro valor.

  0 es la identidad para la adición: $0 + x = x + 0 = x$.

  1 es la identidad para la multiplicación: $1 \times x = x \times 1 = x$
- Un aniquilador para un operador es un valor tal que cuando el operador se aplica al aniquilador y algún otro valor, el resultado es el aniquilador.

  0 es el aniquilador para la multiplicación: $0 \times x = x \times 0 = 0$

  no hay aniquilador para la suma

3.3.3 Leyes

- $\emptyset$ es la identidad para la unión: $\emptyset + L = L + \emptyset = L$
- ϵ es la identidad para la concatenación: $\epsilon L = L\epsilon = L$
- $\emptyset$ es el aniquilador para la concatenación: $\emptyset L = L\emptyset = \emptyset$

NOTA: Estas leyes las utilizamos para simplificaciones

3.3.3.1 Leyes Distributivas

- Como la concatenación no es conmutativa, tenemos dos formas de la ley distributiva para la concatenación:

  Ley Distributiva Izquierda para la concatenación sobre unión: $L(M + N) = LM + LN$

  Ley Distributiva Derecha para la concatenación sobre unión: $(M + N)L = ML + NL$

3.3.3.2 Ley de Idempotencia

- Se dice que un operador es idempotente si el resultado de aplicarlo a dos argumentos con el mismo valor es el mismo valor

  En general la suma no es idempotente: $x+x \neq x$ (aunque para algunos valores sí aplica como $0 + 0 = 0$)

  En general la multiplicación tampoco es idempotente: $x \times x \neq x$

- La unión e intersección son ejemplos comunes de operadores idempotentes

  Ley idempotente para la unión: $L + L = L$

3.3.3.3 Leyes que involucran la cerradura

- $(L^*)^* = L^*$ (Idempotencia para la cerradura)
- $\emptyset^* = \epsilon$
- $\epsilon^* = \epsilon$
- $L+ = LL^* = L^*L, L+$ se define como $L + LL + LLL + \ldots$
  $L^* = \epsilon + L + LL + LLL + \ldots$
  $LL^* = L\epsilon + LL + LLL + LLLL + \ldots$
- $L^* = L + +\epsilon$
- $L? = \epsilon + L$

3.3.3.4 Descubriendo leyes para RE

- Se puede proponer una variedad infinita de leyes para RE
- Hay una metodología general para hacer estas pruebas más fácilmente?

  Se reduce a probar la igualdad de dos lenguajes específicos

  Técnica ligada a los operadores para RE, no se puede extender a otros operadores (como intersección)
- Para probar que $(L + M)^* = (L^*M^*)^*$

  Probamos que las cadenas que están en $(L + M)^*$ también están en $(L^*M^*)^*$

  Probamos que las cadenas que están en $(L^*M^*)^*$ también están en $(L + M)^*$
- Cualquier RE con variables se puede ver como una RE concreta sin variables

  Ver cada variable como si fuera un símbolo diferente

  La expresión $(L + M)^*$ se puede ver como $(a + b)^*$

  Utilizamos esta forma como una guía para concluir sobre los lenguajes

Teorema 3.13: Sea E una RE con variables $L_1, L_2, \ldots, L_m$. Se forma una RE concreta C reemplazando cara ocurrencia de L_i por un símbolo a_i, para $i = 1, 2, \ldots, m$. Luego para cada lenguaje $L_1, L_2, \ldots, L_m$, cada cadena w en $L(E)$ puede escribirse como $w_1w_2 \ldots w_k$, donde cada w_i esta en uno de los lenguajes, como L_{ji}, y la cadena $a_{j1}a_{j2} \ldots a_{jk}$ esta en el lenguaje $L(C)$.

Menos formal, podemos construir $L(E)$ iniciando con cada cadena en $L(C)$, como $a_{j1}a_{j2} \ldots a_{jk}$, y sustituyendo por cada a_{ji}'s cualquier cadena del lenguaje correspondiente L_{ji}.

Base (3 casos):

- Si $E = \epsilon$, la expresión congelada (concreta) también es ϵ
- Si $E = \emptyset$, la expresión congelada también es $\emptyset$

- Si $E = a$, la expresión congelada es también a. Ahora $w \epsilon L(E)$ sí y solo sí existe $u \epsilon L(a)$, tal que $w = u$ y u está en el lenguaje de la expresión congelada, i.e. $u \epsilon a$.

Inducción (3 casos):

- Caso 1 (Unión): $E = F + G, L(E) = L(F) + L(G)$. Utilizamos las expresiones concretas C y D formadas de F y G respectivamente y las sustituimos en todas sus apariciones.

  Obtenemos: $L(C + D) = L(C) + L(D)$

  Suponemos que w es una cadena en $L(E)$ cuando las variables de E se reemplazan por lenguajes específicos. Entonces w esta en $L(F)$ o en $L(G)$.

- Caso 2 (Concatenación):

- Case 3 (Cerradura):

La prueba de una Ley Algebraica para una RE

Probar si $E = F$ es verdadero. E y F son dos RE con el mismo conjunto de variables.

PASOS:

1. Convertir E y F a RE concretas C y D, respectivamente, reemplazando cada variable por un símbolo concreto.
2. Probar si $L(C) = L(D)$. Si es cierto, entonces $E = F$ es una ley verdadera y si no, la "ley" es falsa.

NOTA: No se verán las pruebas para decidir si dos RE denotan el mismo lenguaje hasta la sección 4.4.

Capítulo 4

Propiedades de los Lenguajes Regulares

Existen diferentes herramientas que se pueden utilizar sobre los lenguajes regulares:

- El lema de Pumping: cualquier lenguaje regular satisface el *pumping lemma*, el cual se puede usar para probar que un lenguaje no es regular.
- Propiedades de cerradura: se pueden construir autómatas a partir de componentes usando operaciones, v.g., dado un lenguaje L y M construir un autómata para $L \cap M$.
- Propiedades de decisión: análisis computacional de autómatas, v.g., probar si dos autómatas son equivalentes.
- Técnicas de minimización: útiles para construir máquinas más pequeñas.

La clase de lenguajes conocidos como lenguajes regulares tienen al menos 4 descripciones: $DFA, NFA, \epsilon - NFA$ y RE.

No todos los lenguajes son regulares, por ejemplo, $L = \{0^n 1^n | n \geq 1\}$. Si suponemos que el lenguaje es regular y es aceptado por un DFA A de k estados, si lee al menos k 0s se tiene que cumplir que dos estados se repitan,

esto es que para $i < j, p_i = p_j$. Llamemos a este estado q. Si $\delta(q, 1^i) \in F$ entonces el máquina acepta erróneamente $0^j 1^i$, y si $\notin F$ entonces la máquina rechaza erróneamente $0^i 1^i$.

El problema es que el DFA tiene que tener historia de cuántos 0's lleva para poder aceptar el mismo número de 1's y que este número es variable.

4.1 Lema de Pumping

Si L es un lenguaje regular, entonces existe una constante n tal que cada cadena $w \in L$, de longitud n o más, puede ser escrita como $w = xyz$, donde:

1. $y \neq \epsilon$
2. $|xy| \leq n$
3. Para toda $i \geq 0, wy^i z$ también está en L. Notese que $y^i = y$ repetida i veces; $y^0 = \epsilon$.

Lo que dice es que si tenemos una cadena con una longitud mayor al número de estados del autómata, entonces una cadena no vacía y puede ser repetida (*pumped*) un número arbitrario de veces.

4.1.1 Prueba del Lema de Pumping

Como se da por hecho que L es regular, debe haber un DFA A tal que $L = L(A)$. Si A tiene n estados; escogemos esta n para el lema de pumping.

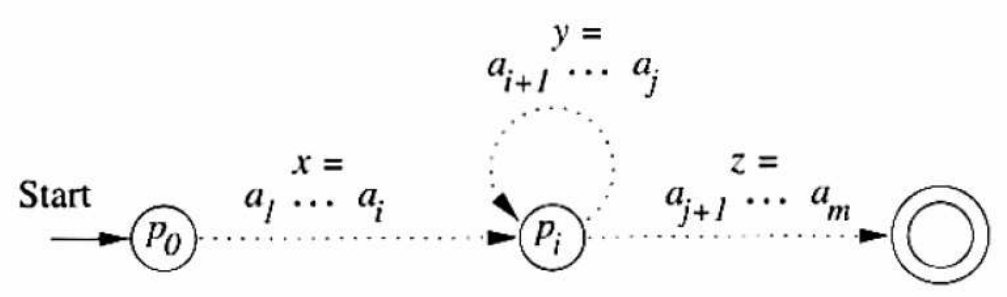
Sea w una cadena de longitud $\geq n$ en L, como en $w = a_1 a_2 \ldots a_m$, donde $m \geq n$.

Sea q_i el estado en que A esta después de leer los primeros i símbolos de w.

q_0 = estado de inicio, $q_1 = \delta(q_0, a_1), q2 = \delta'(q_0, a_1 a_2)$, etc.

Como sólo hay n estados diferentes, dos de $q_0, q_1, \ldots, q_n$ deben ser los mismos; digamos $q_i = q_j$, donde $0 \leq i < j \leq n$.

Sea $x = a_1 \ldots a_i; y = a_{i+1} \ldots a_j; z = a_{j+1} \ldots a_m$. Entonces, si repetimos el ciclo desde q_i a q_i con la etiqueta $a_{i+1} \ldots a_j$ cero veces por una ocasión, o más, se puede probar que xy^iz es aceptado por A.



4.1.2 Uso del Lema de Pumping

El Lema de Pumping se utiliza para mostrar que un lenguaje L no es regular.

- Se inicia asumiendo que L es regular.
- Luego, debe haber alguna n que sirva como la constante de PL. Puede que no sepamos el valor de n, pero podemos seguir con el resto del "juego" con n como un parámetro.
- Escogemos una w que sabemos que está en L. Típicamente, w depende de n.
- Aplicando el PL, sabemos que w puede descomponerse en xyz, satisfaciendo las propiedades del PL. De nuevo, puede que no sepamos como descomponer w, así que utilizaremos x, y, z como parámetros.
- Derivamos una contradicción escogiendo i (la cual puede depender de n, x, y, y/ó z) tal que xy^iz no está en L.

Ejemplo: considere el lenguaje de cadenas con el mismo número de 0's y 1's. Por el pumping lemma, $w = xyz, |xy| \leq n, y \neq \epsilon$ y $xy^kz \in L$.

$$w = \underbrace{000\ldots}_{x}\underbrace{\ldots0}_{y}\underbrace{0111\ldots11}_{z}$$

En particular, $xz \in L$, pero xz tiene menos 0's que 1's.

Ejemplo 2: Supongamos que $L = 1^p : p$ es primo es regular. Sea n el parámetro del pumping lemma y seleccionemos un primo $p \geq n + 2$.

$$w = \overbrace{\underbrace{111\ldots}_{x}\underbrace{\ldots1}_{y}\underbrace{111\ldots11}_{z}}^{p}$$

$|y| = m$

Ahora $xy^{p-m}z$ está en L. $|xp^{p-m}z| = |xz|+(p-m)|y| = p-m+(p-m)m = (1+m)(p-m)$

Que no es un número primo, a menos que uno de los factores sea 1. Pero: $y \neq \epsilon \Rightarrow 1+m > 1$ y $m = |y| \leq |xy| \leq n, p \geq n+2 \Rightarrow p-m \geq n+2-n = 2$

Problema 1: Considere el problema 0^n10^n y demuestre que no es regular.

Problema 2: Considere que el conjunto de cadenas de 0's cuya longitud es un cuadrado perfecto; formalmente $L = \{0^i | i \text{ es un cuadrado}\}$.

Suponemos que L es regular. Entonces hay una n constante que satisface las condiciones del PL.

Considere $w = 0^{n^2}$, que seguramente estará en L.

Entonces $w = xyz$, donde $|xy| \leq n$ y $y \neq \epsilon$

Por PL $xyyz$ está en L. Pero la longitud de $xyyz$ es más grande que n^2 y no más grande que $n^2 + n$.

Sin embargo, el próximo cuadrado perfecto después de n^2 es $(n+1)^2 = n^2 + 2n + 1$.

Así, $xyyz$ no es de longitud cuadrada y no está en L.

Como hemos derivado una contradicción, la única asunción que no se ha probado –que L es regular– debe ser una falla, y tenemos una "prueba por contradicción" que L no es regular.

4.1.3 Propiedades de Cerradura

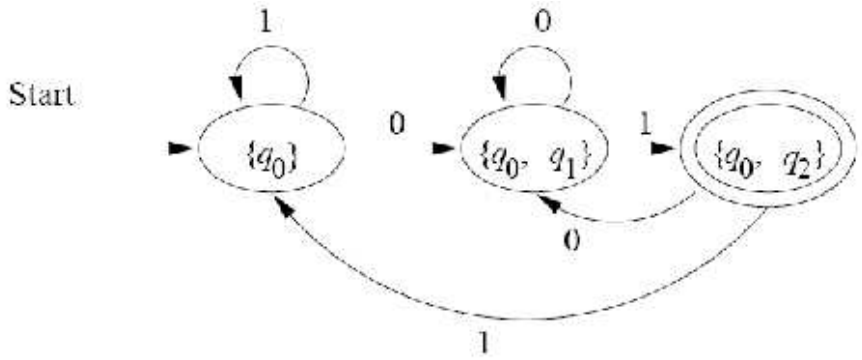
Algunas operaciones sobre lenguajes regulares garantizan producir lenguajes regulares:

- Unión: $L \cup M$ lenguajes con cadenas en L, M o en ambos.
- Intersección: $L \cap M$ lenguajes con cadenas en ambos.
- Complemento: $\overline{L}$ cadenas que no están en L.
- Diferencia: $L \backslash M$ o $L - M$.
- Inversión: $L^R = \{w^R : w \in L\}$
- Cerradura: L^*
- Concatenación: $L.M$
- Homomorfismo (substitución): $h(L) = \{h(w) \in L\}h$ es un homomorfismo.
- Homomorfismo inverso (substitución inversa): $h^{-1}(L) = \{w \in \Sigma : h(w) \in L, h : \Sigma \rightarrow\}$ es un homomorfismo.

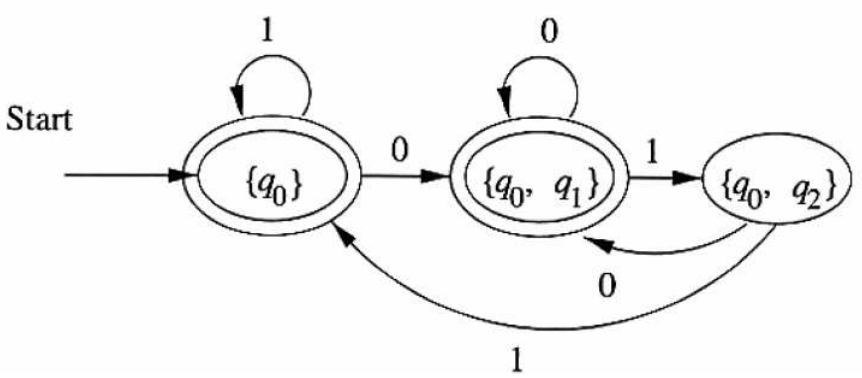
Unión: la unión de lenguajes regulares es regular. Sea $L = L(E)$ y $M = L(F)$. Entonces $L(E + F) = L \cup M$, por la definición de "+" en RE.

Complemento: Si L es un lenguaje regular sobre Σ, entonces también lo es $\overline{L} = \Sigma^* \ L$. Todos los estados son de aceptación excepto los F.

Ejemplo: Sea L definido por el siguiente DFA (el lenguaje de cadenas que terminan en 01):



Entonces $\overline{L}$ es descrito por (el lenguaje de cadenas que no terminan en 01):

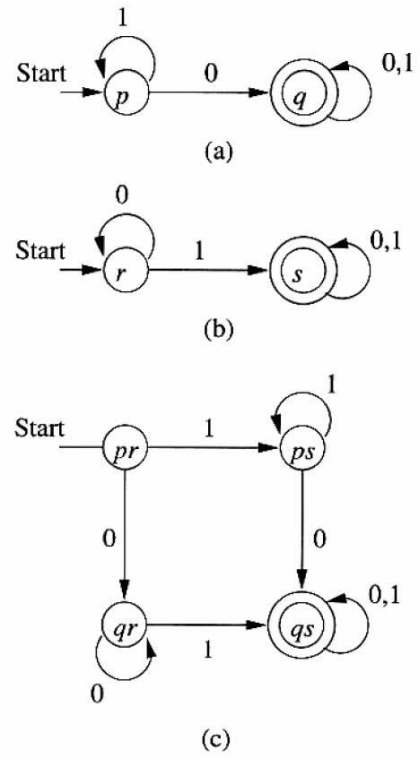


Las cadenas de un número diferente de 0's y 1's es difícil probarlo con el *pumping lemma*. Sin embargo, ya probamos que $L = 0^n1^n$ no es regular. $M = 0^n1^m, n \neq m$ es $\overline{L}$. Como L no es regular su complemento tampoco.

Intersección: Si L y M son regulares, entonces también $L \cap M$. Usando las leyes de Morgan: $L \cap M = \overline{\overline{L} \cup \overline{M}}$.

Para esto también se puede construir un autómata que simula A_L y A_M en paralelo y llega a un estado de aceptación si A_L y A_M también lo hacen.

Ejemplo: (a) todos las cadenas con un 0, (b) todas las cadenas con un 1, (c) todas las cadenas con un 0 y un 1.



Diferencia: $L \backslash M$ lenguaje en L pero no en M, $L \backslash M = L \cap \overline{M}$.

Inversión: w^R es la cadena invertida w. Ejemplo: $0010^R = 0100$.

Inversión de un lenguaje L es el lenguaje con todas las cadenas de L invertidas. Por ejemplo: $L = \{001, 10, 111\}$, $L^R = \{100, 01, 111\}$.

Se puede probar que $L(E^R) = (L(E))^R$.

En particular, $E^R = E$ para $\emptyset$ conjunto vacío y a.

Si $E = F + G$, entonces, $E^R = F^R + G^R$

Si $E = F \cdot G$, entonces $E^R = G^R \cdot F^R$

Por ejemplo, $L(E_1) = \{01, 111\}$ y $L(E_2) = \{00, 10\}$.

$L(E_1)L(E_2) = \{0100, 0110, 11100, 11110\}$

$L^R(E_1) = \{10, 111\}$ y $L^R(E_2) = \{00, 01\}$

$L^R(E_1)L^R(E_2) = \{0010, 00111, 0110, 01111\} = (L(E_1)L(E_2))^R$

Si $L = (0+1)0^*$, $L^R = (0^*)^R(0+1)^R = 0^*(0+1)$

Si $E = F^*$, entonces $E^R = (F^R)^*$

Homomorfismo

Un *homomorfismo* sobre Σ es una función $h : \Sigma^* \to \Theta^*$, donde Σ y Θ son alfabetos.

Sea $w = a_1a_2 \ldots a_n \in \Sigma$. Entonces $h(w) = h(a_1)h(a_2) \ldots h(a_n)$ y $h(L) = \{h(w) \in L\}$.

Ejemplo: sea $h : \{0,1\}^* \to \{a,b\}^*$ definido como $h(0) = ab$, y $h(1) = \epsilon$. Entonces $h(0011) = abab$ y $h(L(10^*1)) = L((ab)^*)$.

$h(L)$ es regular si L es regular.

Sea $L = L(E)$ para una RE E, queremos probar que $L(h(E)) = h(L)$.

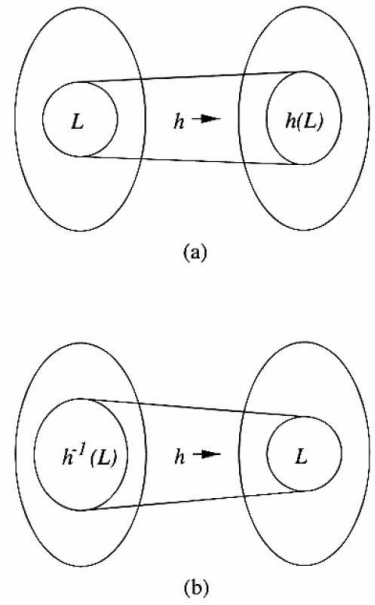
Base: E es ϵ ó $\emptyset$, $h(E) = E$ y $L(h(E)) = L(E) = h(L(E))$.

E es un solo símbolo, a. Entonces $L(E) = a$, y $L(h(E)) = L(h(a)) = \{h(a)\} = h(L(E))$.

Inducción: Hay tres casos, dependiendo de si $R = R_1 + R_2, R = R_1R_2$, ó $R = R_1^*$.

- $L = E + F, L(h(E + F)) = L(h(E) + h(F)) = L(h(E)) \cup L(h(F)) = h(L(E)) \cup h(L(F)) = h(L(E) \cup L(F)) = h(L(E + F))$
- $L = E \cdot F, L(h(E \cdot F)) = L(h(E)) \cdot L(h(F)) = h(L(E)) \cdot h(L(F)) = h(L(E) \cdot L(F))$
- $L = E^*, L(h(E^*)) = L(h(E)^*) = h(L(E))^* = h(L(E^*))$

Homomorfismo Inverso: Sea $h : \Sigma^* \to \Theta^*$ un homomorfismo. Sea $L \subseteq \Theta^*$, entonces $h^{-1}(L) = \{w | w \in \Sigma^* : h(w) \in L\}$.

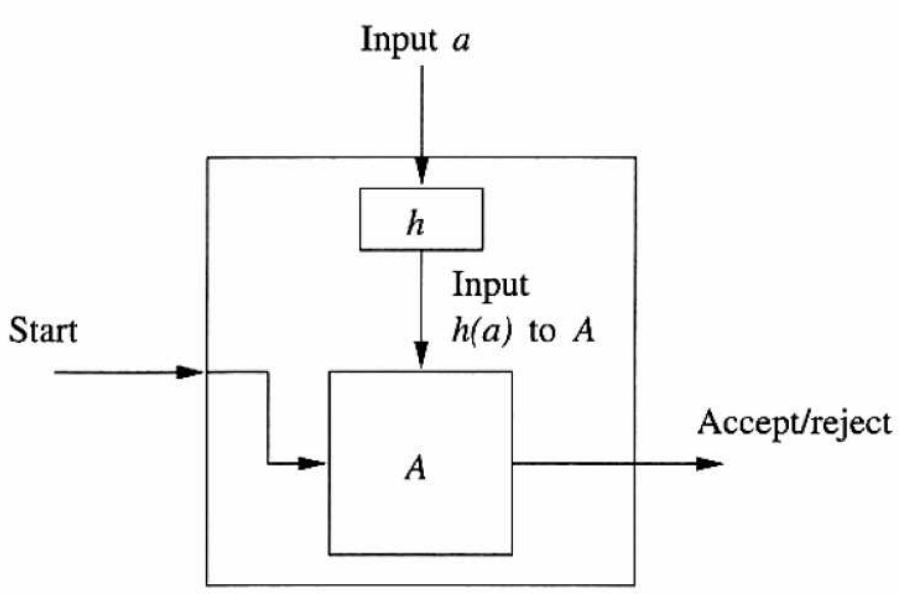


Sea $h : \Sigma^* \to \Theta^*$ un homomorfismo. Sea $L \subseteq \Theta^*$ un lenguaje regular, entonces $h^{-1}(L)$ es regular.

El DFA del homomorfismo inverso usa los estados del DFA original, pero cambia los símbolos de entrada de acuerdo a la función h antes de decidir el siguiente estado. En la prueba lo importante es que los estados de aceptación de los dos autómatas son los mismos y que $h(a)$, para cualquier símbolo a dentro del alfabeto del DFA original puede ser ϵ, un símbolo o muchos símbolos.

$\gamma(q, a) = \hat{\delta}(q, h(a))$.

La prueba se hace por inducción sobre $|w|$. $\hat{\gamma}(q_0, w) = \hat{\delta}(q_0, h(w))$.



4.1.4 Propiedades de decisión

Algunas de las preguntas que nos podemos hacer acerca de lenguajes son si el lenguaje es vacío, si una cadena particular pertenece al lenguaje o si dos descripciones definen el mismo lenguaje.

También podemos cuestionarnos acerca del costo computacional requerido para resolver estas preguntas o por ejemplo el requerido para hacer la conversión entre una representación a otra.

Transformar un ϵ-NFA a un DFA: Si el ϵ-NFA tiene n arcos. Para calcular $ECLOSE(p)$ se requieren seguir a lo más n^2 arcos. El DFA tiene a lo más 2^n estados. Para cada símbolo y cada subconjunto el calcular la función de transición para todos los estados, requiere a lo más n^3 pasos, lo cual nos da una complejidad total de $O(n^3 2^n)$. En general el número de estados del DFA es de orden lineal (digamos s), por lo que en la práctica la complejidad se reduce a $O(n^3 s)$.

Transformar un DFA a un NFA: Sólo se requiere poner corchetes a los estados, lo cual nos da O(n).

Transformar un FA a una RE: $O(n^3 4^n)$. Es todavía peor si el FA es NFA. Si lo convertimos primero a un DFA nos da: $O(n^3 4^{n^3 2^n})$.

Transformar de una RE a un FA: se puede construir un autómata en n pasos. Si eliminamos transiciones ϵ toma $O(n^3)$. Si se requiere un DFA puede tomar un número exponencial de pasos.

Decidir si un lenguaje es vacío: el probar si existe un camino entre un estado inicial a uno final o de aceptación, es simplemente un problema de ver si un nodo está conectado en un grafo, lo cual tiene una complejidad de $O(n^2)$.

Probar por pertenencia a un lenguaje regular: ver si una cadena es miembro del lenguaje. Si la cadena w es de longitud n para un DFA, esto es de complejidad $O(n)$. Si es un NFA de s estados, entonces la complejidad es: $O(ns^2)$. Si es un ϵ-NFA entonces la complejidad es $O(ns^3)$.

4.2 Equivalencia y minimización de autómatas

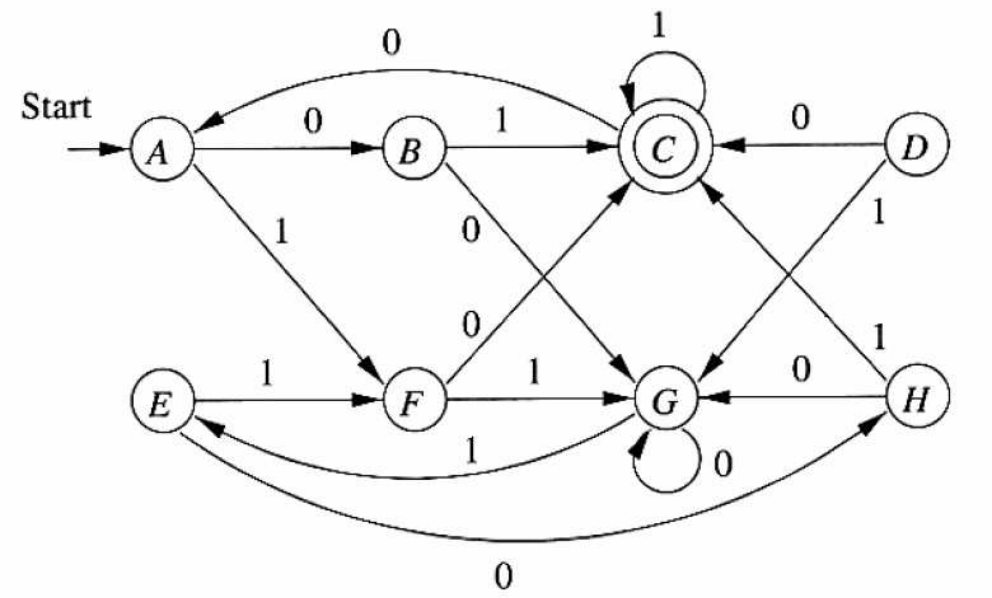
Lo que queremos saber es si dos autómatas diferentes definen el mismo lenguaje.

Primero definiremos lo que son estados equivalentes.

Dos estados p y q dentro de un autómata son *equivalentes*: $p \equiv q \Leftrightarrow \forall w \in \Sigma^* : \hat{\delta}(p, w) \in F \Leftrightarrow \hat{\delta}(q, w) \in F$.

Si no, entonces se dice que son *distinguibles*. Osea que p y q son distinguibles si: $\exists w : \hat{\delta}(p, w) \in F \wedge \hat{\delta}(q, w) \notin F$ o viceversa.

Ejemplo:

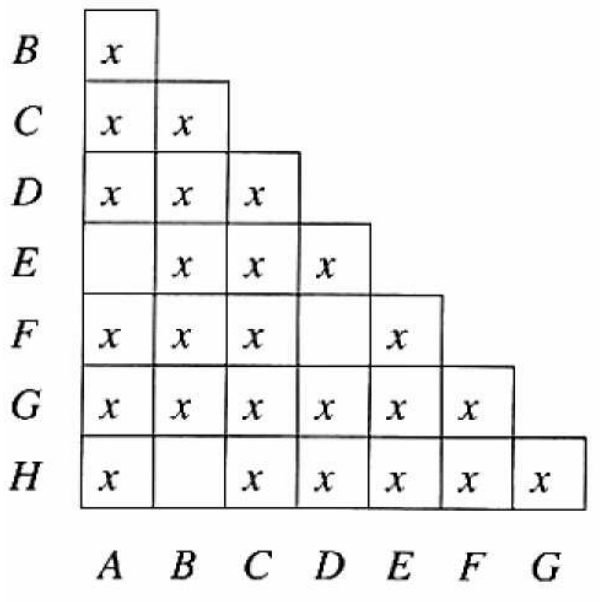


$$\hat{\delta}(C,\epsilon) \in F, \hat{\delta}(G,\epsilon) \notin F \Rightarrow C \neq G$$
$$\hat{\delta}(A,01) = C \in F, \hat{\delta}(G,01) = E \notin F \Rightarrow A \neq G$$
$$\hat{\delta}(A,\epsilon) = A \notin F, \hat{\delta}(E,\epsilon) = E \notin F$$
$$\hat{\delta}(A,1) = F = \hat{\delta}(E,1) \therefore \hat{\delta}(A,1x) = \hat{\delta}(E,1x) = \hat{\delta}(F,1x)$$
$$\hat{\delta}(A,00) = G = \hat{\delta}(E,00)$$
$$\hat{\delta}(A,01) = C = \hat{\delta}(E,01)$$
$$\therefore A \equiv E$$

También podemos encontrar los pares equivalentes usando el algoritmo de llenado de tabla (*table-filling algorithm*).

Base: Si $p \in F \wedge q \neq F \Rightarrow p \neq q$
Inducción: Si $\exists a \in \Sigma : \delta(p,a) \neq \delta(q,a) \Rightarrow p \neq q$

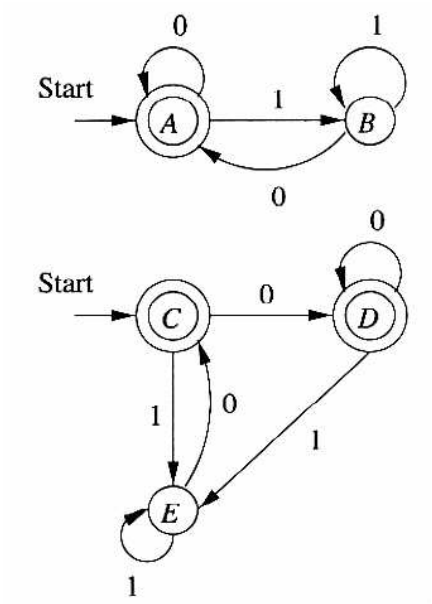
Por ejemplo, para el DFA anterior:



4.2.1 Prueba de equivalencia entre lenguajes regulares

- Sea L y M dos lenguajes regulares (dados en alguna forma).
- Convierte L y M a DFA's
- Junta los dos DFA's
- Prueba si los dos estados iniciales son equivalentes, en cuyo caso $L = M$. Si no son equivalentes, entonces $L \neq M$.

Ejemplo:



Los dos DFA's aceptan: $L(\epsilon + (0+1)^*0)$. Si los consideramos a los dos como un solo autómata, el algoritmo de llenado de tabla nos da:

B	x			
C		x		
D		x		
E	x		x	x
	A	B	C	D

Lo que nos deja los siguientes pares de estados equivalentes: $\{A, C\}, \{A, D\}, \{C, D\}$ y $\{B, E\}$. Como A y C son equivalentes, entonces los dos autómatas son equivalentes.

4.2.2 Minimización de un DFA

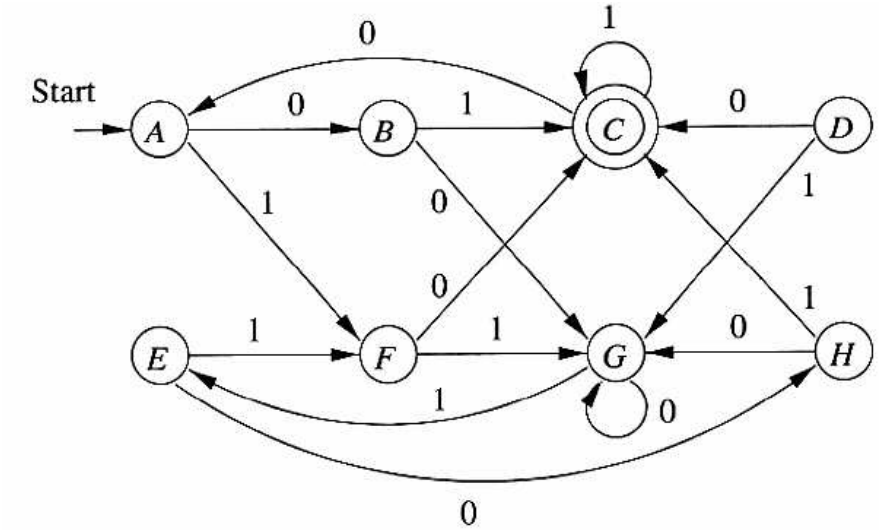
Una consecuencia de encontrar estados equivalentes, es que podemos reemplazar los estados equivalentes por un solo estado representado por su unión.

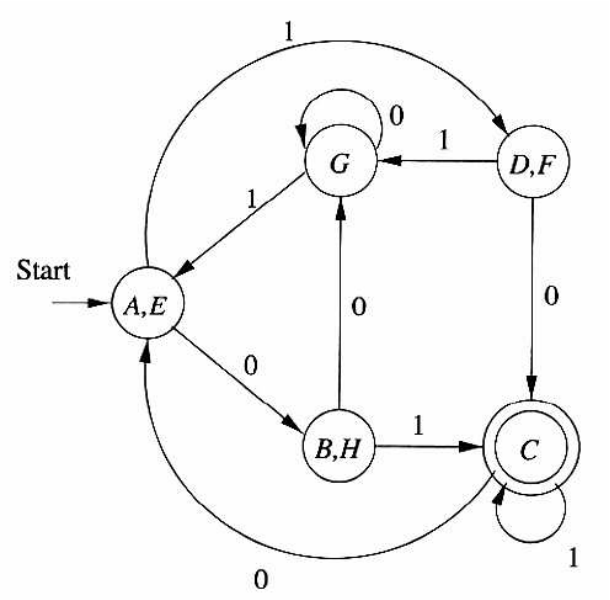
Por otro lado, la relación de equivalencia cumple con ser reflexiva, simétrica y transitiva.

La prueba de equivalencia de estados es transitiva, osea que si encontramos que p y q son equivalentes y q y r son equivalentes, entonces p y r son equivalentes.

Ejemplo: el siguiente autómata, lo podemos minimizar usando los estados

equivalentes.





Nota: no podemos aplicar el algoritmo de llenado de tabla a un NFA.

Ejemplo 4.4.1:

Capítulo 5

Gramaticas Libres de Contexto

5.1 Introducción

- Hemos visto que muchos lenguajes no son regulares. Por lo que necsitamos una clase mas grande de lenguages
- Las Gramáticas Libres de Contexto (*Context-Free Languages*) o CFL's jugaron un papel central en lenguaje natural desde los 50's y en los compiladores desde los 60's
- Las Gramáticas Libres de Contexto forman la base de la sintáxis BNF
- Son actualmente importantes para XML y sus DTD's (*document type definition*)

Vamos a ver los CFG's, los lenguajes que generan, los árboles de parseo, el *pushdown automata* y las propiedades de cerradura de los CFL's.

Ejemplo: Considere $L_{pal} = \{w \in \Sigma^* : w = w^R\}$

Por ejemplo, $oso \in L_{pal}, anitalavalatina \in L_{pal}$,

Sea $\Sigma = \{0, 1\}$ y supongamos que L_{pal} es regular.

Sea n dada por el *pumping lemma*. Entonces $0^n10^n \in L_{pal}$. Al leer 0^n el FA debe de entrar a un ciclo. Si quitamos el ciclo entonces llegamos a una contradicción.

Definamos L_{pal} de forma inductiva.

Base: $\epsilon, 0$ y 1 son palíndromes.

Inducción: Si w es un palíndrome, también $0w0$ y $1w1$.

Ninguna otra cosa es palíndrome.

Las CFG's son un mecanismo formal para la definiciónn como la de palíndrome.

$P \to \epsilon$
$P \to 0$
$P \to 1$
$P \to 0P0$
$P \to 1P1$

donde 0 y 1 son símbolos terminales.

P es una variable o símbolo no terminal o categoría sintáctica.

P es en esta gramática también el símbolo inicial.

1-5 son producciones o reglas. La variable definida (parcialmente) en la producción también se llama la *cabeza* de la producción y la cadena de cero, 1 o más símbolos terminales o variables a la derecha de la producción se llama el *cuerpo* de la producción.

5.2 Definición formal de CFG's

Una grámatica libre de contexto se define con $G = (V, T, P, S)$ donde:

- V es un conjunto de *variables*
- T es un conjunto de *terminales*

- P es un conjunto finito de *producciones* de la forma $A \to \alpha$, donde A es una variables y $\alpha \in (V \cup T)^*$
- S es una variable designada llamada el *símbolo inicial*

Ejemplo: $G_{pal} = (\{P\}, \{0,1\}, A, P)$, donde $A = \{P \to \epsilon, P \to 0, P \to 1, P \to 0P0, P \to 1P1\}$.

Muchas veces se agrupan las producciones con la misma cabeza, e.g., $A = \{P \to \epsilon|0|1|0P0|1P1\}$.

Ejemplo: Expresiones regulares sobre $\{0,1\}$ se pueden definir por la gramática: $G_{regex} = (\{E\}, \{0,1\}, A, E)$, donde $A = \{E \to 0, E \to 1, E \to E.E, E \to E+E, E \to E^*, E \to (E)\}$.

Ejemplo: expresiones (simples) en un lenguaje de programación típico. Los operadores son $+$ y $*$, y los argumentos son identificadores, osea *strings* que empiezan con a o b en $L((a+b)(a+b+0+1)^*)$.

Las expresiones se definen por la gramática: $G = (\{E, I\}, T, P, E)$ donde $T = \{+, *.(,), a, b, 0, 1\}$ y P es el siguiente conjunto de producciones:
$1) E \to I$
$2) E \to E + E$
$3) E \to E * E$
$4) E \to (E)$
$5) I \to a$
$6) I \to b$
$7) I \to Ia$
$8) I \to Ib$
$9) I \to I0$
$10) I \to I1$

5.3 Derivaciones usando gramáticas

- Inferencia recursiva, usando las producciones del cuerpo a la cabeza. Con esto reconocemos si una cadena está en el lenguaje definido por la gramática.

Ejemplo de una inferencia recursiva de la cadena: $a * (a + b00)$.

Cadenas	Cabeza	Del Leng. de	Cadenas usadas
(i) a	I	5	–
(ii) b	I	6	–
(iii) $b0$	I	9	(ii)
(iv) $b00$	I	9	(iii)
(v) a	E	1	(i)
(vi) $b00$	E	1	(iv)
(vii) $a + b00$	E	2	(v),(vi)
(viii) $(a + b00)$	E	4	(vii)
(ix) $a * (a + b00)$	E	3	(v),(viii)

- Derivaciones, usando las producciones de la cabeza al cuerpo. Con esto derivamos cadenas que pertenecen a la gramática.

  Para esto introducimos un nuevo símbolo: $\Rightarrow$

  Sea $G = (V, T, P, S)$ una CFG, $A \in V, \{\alpha, \beta\} \subset (V \cup T)^*$ y $A \to \gamma \in P$.

  Entonces, escribimos: $\alpha A \beta \Rightarrow_G \alpha\gamma\beta$ o si se sobre-entiende G: $\alpha A \beta \Rightarrow \alpha\gamma\beta$ y decimos que $\alpha A \beta$ *deriva* $\alpha\gamma\beta$.

Definimos $\overset{*}{\Rightarrow}$ como la cerradura reflexiva y transitiva de $\Rightarrow$. Lo que quiere decir es que usamos uno a mas pasos de derivación.

Ideas:

- *Base*: Sea $\alpha \in (V \cup T)^*$, entronces $\alpha \overset{*}{\Rightarrow} \alpha$ (osea que cada cadena se deriva a sí misma).
- *Inducción*: Si $\alpha \overset{*}{\Rightarrow} \beta$, y $\beta \Rightarrow \gamma$, entonces $\alpha \overset{*}{\Rightarrow} \gamma$

Ejemplo: La derivación de $a*(a+b00)$ a partir de E en la gramática anterior sería:
$E \Rightarrow E * E \Rightarrow I * E \Rightarrow a * E \Rightarrow a * (E) \Rightarrow a * (E + E) \Rightarrow a * (I + E) \Rightarrow a * (a + E) \Rightarrow a * (a + I) \Rightarrow a * (a + I0) \Rightarrow a * (a + I00) \Rightarrow a * (a + b00)$

Podemos abreviar y simplemente poner: $E \overset{*}{\Rightarrow} a * (a + b00)$

La derivación y la inferencia recursiva son equivalentes, osea que si podemos inferir que una cadena de símbolos terminales w está en el lenguaje de una variable A entonces $A \stackrel{*}{\Rightarrow} w$ y al revés.

Nota 1: en cada paso podemos tener varios reglas de las cuales escoger, e.g.: $I * E \Rightarrow a * E \Rightarrow a * (E)$ o $I * E \Rightarrow I * (E) \Rightarrow a * (E)$

Nota 2: no todas las opciones nos llevan a derivaciones exitosas de una cadena en particular, por ejemplo: $E \Rightarrow E + E$ no nos lleva a la derivación de $a * (a + b00)$.

5.4 Derivaciones más a la izquierda y más a la derecha

Para restringir el número de opciones para derivar una cadena.

- Derivación más a la izquierda (*leftmost derivation*): $\Rightarrow_{lm}$ siempre reemplaza la variable más a la izquierda por uno de los cuerpos de sus producciones.
- Derivación más a la derecha (*rightmost derivation*): $\Rightarrow_{rm}$ siempre reemplaza la variable más a la derecha por uno de los cuerpos de sus producciones.

Ejemplo: la derivación anterior la podemos hacer como derivación más a la izquierda:

$E \Rightarrow_{lm} E * E \Rightarrow_{lm} I * E \Rightarrow_{lm} a * E \Rightarrow_{lm} a * (E) \Rightarrow_{lm} a * (E + E) \Rightarrow_{lm} a*(I+E) \Rightarrow_{lm} a*(a+E) \Rightarrow_{lm} a*(a+I) \Rightarrow_{lm} a*(a+I0) \Rightarrow_{lm} a*(a+I00) \Rightarrow_{lm} a * (a + b00)$ o simplemente $E \stackrel{*}{\Rightarrow}_{lm} a * (a + b00)$

Por otro lado, también la podemos hacer más a la derecha:

$E \Rightarrow_{rm} E * E \Rightarrow_{rm} E * (E) \Rightarrow_{rm} E * (E + E) \Rightarrow_{rm} E * (E + I) \Rightarrow_{rm} E*(E+I0) \Rightarrow_{rm} E*(E+I00) \Rightarrow_{rm} E*(E+b00) \Rightarrow_{rm} E*(I+b00) \Rightarrow_{rm} E*(a+b00) \Rightarrow_{rm} I*(a+b00) \Rightarrow_{rm} a*(a+b00)$ o simplemente $E \stackrel{*}{\Rightarrow}_{rm} a*(a+b00)$.

Cualquier derivación tiene una derivación equivalente más a la izquierda y una más a la derecha.

Si w es una cadena de símbolos terminales y A es una variable, $A \stackrel{*}{\Rightarrow} w$ si y solo si $A \stackrel{*}{\Rightarrow}_{lm} w$ y si y solo si $A \stackrel{*}{\Rightarrow}_{rm} w$.

5.5 El Lenguaje de la Gramática

Si $G(V, T, P, S)$ es una CFG, entonces el lenguaje de G es: $L(G) = \{w \in T^* : S \stackrel{*}{\Rightarrow}_G w\}$, osea el conjunto de cadenas sobre T^* derivadas del símbolo inicial.

Si G es una CFG al $L(G)$ se llama *lenguaje libre de contexto.*

Por ejemplo, $L(G_{pal})$ es un lenguaje libre de contexto.

Teorema: $L(G_{pal}) = \{w \in \{0,1\}^* : w = w^R\}$

Prueba: ($\Rightarrow$) Suponemos $w = w^R$. Mostramos por inducción en $|w|$ que $w \in L(G_{pal})$.

Base: $|w| = 0$ or $|w| = 1$. Entonces w es $\epsilon, 0$ o 1. Como $P \to \epsilon, P \to 1$ y $P \to 0$ son producciones, concluimos que $P \stackrel{*}{\Rightarrow}_G w$ en todos los casos base.

Inducción: Suponemos $|w| \geq 2$. Como $w = w^R$, tenemos que $w = 0x0$ o $w = 1x1$ y que $x = x^R$.

Si $w = 0x0$ sabemos de la hipótesis inductiva que $P \stackrel{*}{\Rightarrow} x$, entonces $P \Rightarrow 0P0 \stackrel{*}{\Rightarrow} 0x0 = w$, entonces $w \in L(G_{pal})$.

El caso para $w = 1x1$ es similar.

($\Leftarrow$): Asumimos que $w \in L(G_{pal})$ y tenemos que mostrar que $w = w^R$.

Como $w \in L(G_{pal})$, tenemos que $P \stackrel{*}{\Rightarrow} w$. Lo que hacemos es inducción sobre la longitud de $\stackrel{*}{\Rightarrow}$.

Base: La derivación de $\stackrel{*}{\Rightarrow}$ se hace en un solo paso, por lo que w debe de ser $\epsilon, 0$ o 1, todos palíndromes.

Inducción: Sea $n \geq 1$ y suponemos que la derivación toma $n+1$ pasos y que el enunciado es verdadero para n pasos. Osea, si $P \overset{*}{\Rightarrow} x$ en n pasos, x es palíndrome. Por lo que debemos de tener para $n+1$ pasos:

$w = 0x0 \overset{*}{\Leftarrow} 0P0 \Leftarrow P$ o $w = 1x1 \overset{*}{\Leftarrow} 1P1 \Leftarrow P$ donde la segunda derivación toma n pasos.

Por la hipótesis inductiva, x es un palíndrome, por lo que se completa la prueba.

5.6 Sentential Forms

Sea $G = (V, T, P, S)$ una CFG y $\alpha \in (V \cup T)^*$. Si $S \overset{*}{\Rightarrow} \alpha$ decimos que α está en forma de sentencia /*sentential form*).

Si $S \Rightarrow_{lm} \alpha$ decimos que α es una forma de sentencia izquierda (*left-sentencial form*), y si $S \Rightarrow_{rm} \alpha$ decimos que α es una forma de sentencia derecha (*right-sentencial form*).

$L(G)$ son las formas de sentencia que estan en T^*.

Ejemplo: Si tomamos la gramática del lenguaje sencillo que definimos anteriormente, $E * (I + E)$ es una forma de sentencia ya que: $E \Rightarrow E * E \Rightarrow E * (E) \Rightarrow E * (E + E) \Rightarrow E * (I + E)$.

Esta derivación no es ni más a la izquierda ni más a la derecha.

Por otro lado: $a * E$ es una forma de sentencia izquierda, ya que: $E \Rightarrow_{lm} E * E \Rightarrow_{lm} I * E \Rightarrow_{lm} a * E$ y $E * (E + E)$ es una forma de sentencia derecha, ya que: $E \Rightarrow_{rm} E * E \Rightarrow_{rm} E * (E) \Rightarrow_{rm} E * (E + E)$

5.7 Árboles de Parseo

Si $w \in L(G)$ para alguna CFG, entonces w tiene un árbol de parseo (*parse tree*), el cual nos da la estructura (sintáctica) de w. w puede ser un programa, un *query* en SQL, un documento en XML, etc.

Los árboles de parseo son una representación atlernativa de las derivaciones e inferencias recursivas.

Pueden existir varios árboles de parseo para la misma cadena.

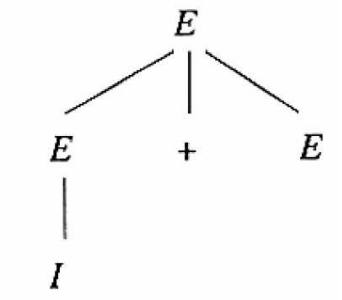
Idealmente nos gustaría que existiera solo uno, i.e., que el lenguaje fuera no ambigüo. Desafortunadamente no siempre se puede quitar la ambigüedad.

5.7.1 Construyendo Árboles de Parseo

Sea $G = (V, T, P, S)$ una CFG. Un árbol es un árbol de parseo de G si:

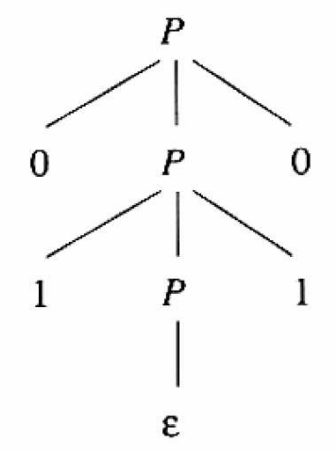
1. Cada nodo interior está etiquetado con una variable en V
2. Cada hoja está etiquetada con un símbolo en $V \cup T \cup \{\epsilon\}$. Cada hoja etiquetada con ϵ es el único hijo de su padre.
3. Si un nodo interior tiene etiqueta A y sus hijos (de izquierda a derecha) tienen etiquetas: $X_1, X_2, \ldots, X_k$, entonces: $A \to X_1, X_2, \ldots, X_k \in P$

Ejemplo: En la gramática:
$E \to I$
$E \to E + E$
$E \to E * E$
$E \to (E)$
$\vdots$
el siguiente es un árbol de parseo:



Este árbol muestra la derivación: $E \stackrel{*}{\Rightarrow} I + E$.

Ejemplo: En la gramática:
$P \to \epsilon$
$P \to 0$
$P \to 1$
$P \to 0p0$
$P \to 0p0$
el siguiente es un árbol de parseo:



Este árbol muestra la derivación: $P \stackrel{*}{\Rightarrow} 0110$.

5.7.2 El producto de un árbol de parseo

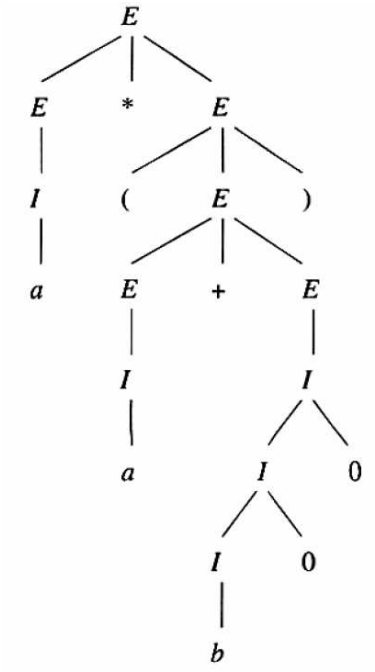
El producto (*yield*) de un árbol de parseo es la cadena de hojas de izquierda a derecha.

Son en especial relevantes los árboles de parseo que:

1. El producto es una cadena terminal
2. La raíz esté etiquetada con el símbolo inicial

El conjunto de productos de estos árboles de parseo nos definen el lenguaje de la gramática.

Ejemplo: considere el siguiente árbol de parseo

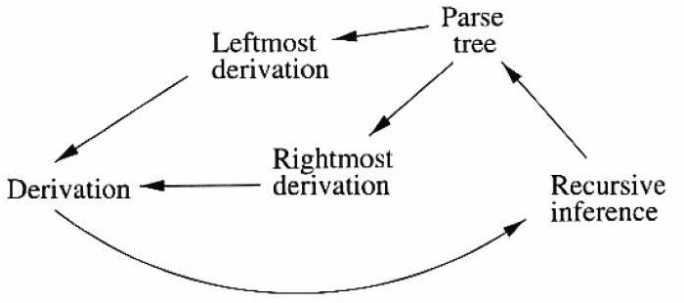


cuyo producto es: $a*(a+b00)$ (podemos comparar este árbol con la derivación que hicimos antes).

Sea $G = (V, T, P, S)$ una CFG y $A \in V$. Vamos a demostrar que lo siguiente es equivalente:

- Podemos determinar por inferencia recursiva que w esté en el lenguaje de A
- $A \stackrel{*}{\Rightarrow} w$
- $A \Rightarrow^*_{lm} w$ y $A \Rightarrow^*_{rm} w$
- Existe un árbol de parseo de G con raíz A que produce w

El siguiente es el plan a seguir para probar las equivalencias:

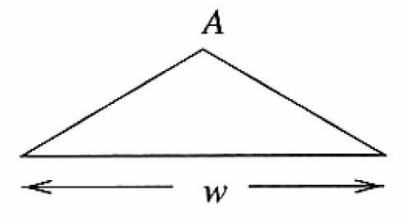


5.7.3 De inferencias a árboles

Teorema: Sea $G = (V, T, P, S)$ una CFG y supongan que podemos mostrar que w está en el lenguaje de una variable A. Entonces existe un árbol de parseo para G que produce w.

Prueba: la hacemos por inducción en la longitud de la inferencia

Base: Un paso. Debemos de usar la producción $A \to w$. El árbol de parseo es entonces:

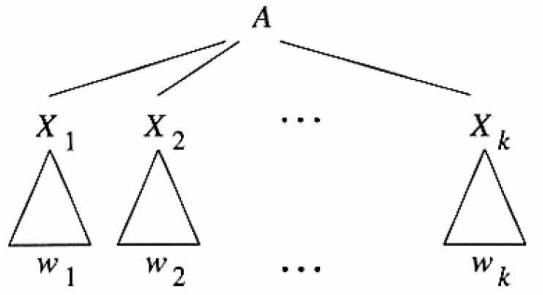


Inducción: w es inferido en $n + 1$ pasos. Suponemos que el último paso se baso en la producción: $A \to X_1, X_2, \ldots, X_k$, donde $X_i \in V \cup T$.

Descomponemos w en: $w_1 w_2 \ldots w_k$, donde $w_i = X_i$ cuando $X_i \in T$, y cuando $X_i \in V$, entonces w_i fué previamente inferida en X_i en a los más n pasos.

Por la hipótesis de inferencia existen i árboles de parseo con raíz X_i que producen w_i. Entonces el siguiente en una árbol de parseo de G con raíz en

A que produce w:



5.7.4 De árboles a derivaciones

Mostraremos como construir una derivación más a la izquierda de un árbol de parseo.

Ejemplo: De una gramática podemos tener la siguiente derivación: $E \Rightarrow I \Rightarrow Ib \Rightarrow ab$.

Entonces, para cualquier α y β existe uan derivación $\alpha E\beta \Rightarrow \alpha I\beta \Rightarrow \alpha Ib\beta \Rightarrow \alpha ab\beta$.

Por ejemplo, supongamos que tenemos la derivación:

$E \Rightarrow E + E \Rightarrow E + (E)$

Entonces podemos escoger $\alpha =$ "$E+($" y $\beta =$ "$)$" y seguir con la derivación como:

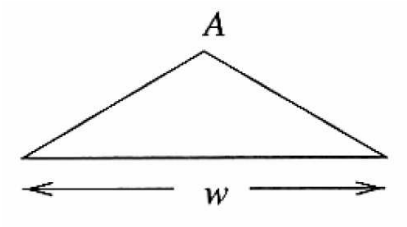
$E + (E) \Rightarrow E + (I) \Rightarrow E + (Ib) \Rightarrow E + (ab)$

Por esto es que las CFG se llaman libres de contexto (substitución de cadenas por variables, independientes del contexto).

Teorema: Sea $G = (V, T, P, S)$ una CFG y supongamos que existe un árbol de parseo cuya raíz tiene etiqueta A y que produce w. Entonces $A \Rightarrow^*_{lm} w$ en G.

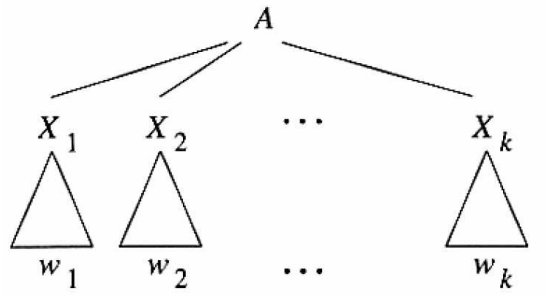
Prueba: la hacemos por inducción en la altura del árbol.

Base: La altura es 1, y el árbol debe de verse asi:



Por lo tanto $A \to w \in P$ y $A \Rightarrow_{lm} w$.

Inducción: Altura es $n + 1$. El árbol debe de verse así:



Entonces $w = w_1 w_2 \dots w_k$ donde:

1. Si $X_i \in T$, entonces $w_i = X_i$
2. Si $X_i \in V$, entonces debe de ser la raíz de un subárbol que nos da w_i, $X_i \Rightarrow^*_{lm} w_i$ en G por la hipótesis inductiva

Ahora mostramos $A \Rightarrow^*_{lm} w_1 w_2 \dots w_i X_{i+1} X_{i+2} \dots X_k$, obtener una derivación más a la izquierda. Probamos sobre i:

Base: Sea $i = 0$. Sabemos que: $A \Rightarrow_{lm} X_1 X_2 \dots X_k$

Inducción: Hacemos la hipótesis inductiva: $A \Rightarrow^*_{lm} w_1 w_2 \dots w_{i-1} X_i X_{i+1} \dots X_k$

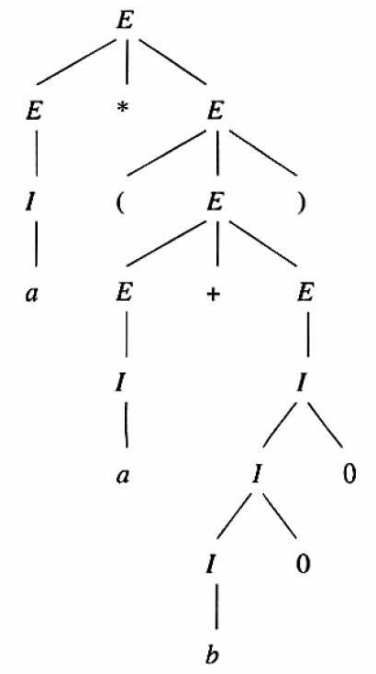
Caso 1: Si $X_i \in T$, no hacemos nada ya que $X_i = w_i$ que nos da: $A \Rightarrow^*_{lm} w_1 w_2 \dots w_i X_{i+1} \dots X_k$

Case 2: $X_i \in V$. Por la hipótesis inductiva existe una derivación $X_i \Rightarrow_{lm}$

$\alpha_1 \Rightarrow_{lm} \alpha_2 \ldots \Rightarrow_{lm} w_i$. Por la propiedad libre de contexto de las revivaciones podemos proceder como:

$$\begin{aligned} A \Rightarrow^*_{lm} \quad & w_1 w_2 \ldots w_{i-1} X_i X_{i+1} \ldots X_k \Rightarrow^*_{lm} \\ & w_1 w_2 \ldots w_{i-1} \alpha_1 X_{i+1} \ldots X_k \Rightarrow^*_{lm} \\ & w_1 w_2 \ldots w_{i-1} \alpha_2 X_{i+1} \ldots X_k \Rightarrow^*_{lm} \\ & \ldots \\ & w_1 w_2 \ldots w_{i-1} w_i X_{i+1} \ldots X_k \Rightarrow^*_{lm} \end{aligned}$$

Ejemplo: Construyamos la derivación más a la izquierda del árbol:



Sopongamos que construimos inductivamente la derivación más a la izquierda: $E \Rightarrow_{lm} I \Rightarrow_{lm} a$ que corresponde con la rama izquierda del árbol, y la derivación más a la izquierda:

$$E \Rightarrow_{lm} (E) \Rightarrow_{lm} (E + E) \Rightarrow_{lm} (I + E) \Rightarrow_{lm} (a + E) \Rightarrow_{lm} (a * I) \Rightarrow_{lm}$$

$(a + I0) \Rightarrow_{lm} (a + I00) \Rightarrow_{lm} (a + b00)$ correspondiendo a la rama derecha del árbol.

Para las derivaciones correspondientes del árbol completo empezamos con $E \Rightarrow_{lm} E * E$ y expandimos la primera E con la primera derivación y la segunda E con la segunda derivación:

$E \Rightarrow_{lm} E * E \Rightarrow_{lm} I * E \Rightarrow_{lm} a * E \Rightarrow_{lm} a * (E) \Rightarrow_{lm} a * (E + E) \Rightarrow_{lm} a*(I+E) \Rightarrow_{lm} a*(a+E) \Rightarrow_{lm} a*(a*I) \Rightarrow_{lm} a*(a+I0) \Rightarrow_{lm} a*(a+I00) \Rightarrow_{lm} a * (a + b00)$

De forma similar podemos convertir un árbol en una derivación más a la derecha. Osea, si existe un árbol de parseo con raíz etiquetada con la variable A que produce $w \in T^*$, entonces existe: $A \Rightarrow^*_{rm} w$.

5.7.5 De derivaciones a inferencias recursivas

Supongamos que $A \Rightarrow X_1X_2 \ldots X_k \stackrel{*}{\Rightarrow} w$, entonces: $w = w_1w_2 \ldots w_k$ donde $X_i \stackrel{*}{\Rightarrow} w_i$.

El factor w_i se puede extraer de $A \stackrel{*}{\Rightarrow} w$ viendo únicamente a la expansión de X_i.

Por ejemplo: $E \Rightarrow a * b + a$ y

$$E \Rightarrow \underbrace{E}_{X_1} \underbrace{*}_{X_2} \underbrace{E}_{X_3} \underbrace{+}_{X_4} \underbrace{E}_{X_5}$$

Tenemos que: $E \Rightarrow E * E \Rightarrow E * E + E \Rightarrow I * E + E \Rightarrow I * I + E \Rightarrow I * I + I \Rightarrow a * I + I \Rightarrow a * b + I \Rightarrow a * b + a$

Viendo solo a la expansión de $X_3 = E$ podemos extraer: $E \Rightarrow I \Rightarrow b$

Teorema: Sea $G = (V, T, P, S)$ una CFG. Suponga $A \Rightarrow^*_G w$ y que w es una cadena de símbolos terminales. Entonces podemos inferir que w está en el lenguaje de la variable A.

Prueba: la hacemos por inducción en la longitud de la derivación $A \Rightarrow^*_G w$

Base: Un paso. Si $A \Rightarrow_G w$ entonces debe de existir una producción $A \rightarrow w$ en P. Por lo que podemos inferir que w está en el lenguaje de A.

Inducción: Suponemos $A \stackrel{*}{\Rightarrow}_G w$ en $n+1$ pasos. Escribimos la derivación como:

$A \Rightarrow_G X_1 X_2 \ldots X_k \stackrel{*}{\Rightarrow}_G w$

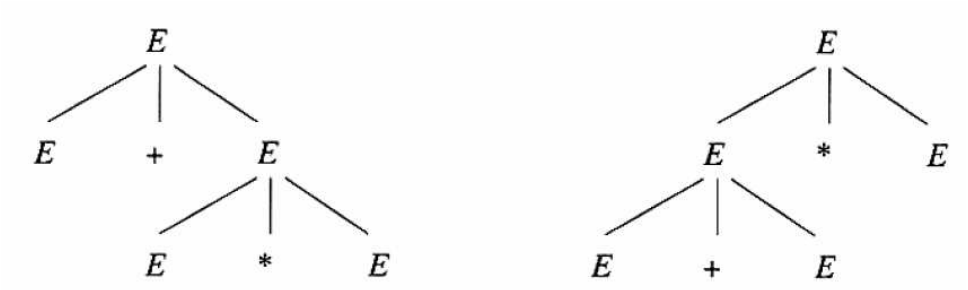
Como vimos, podemos partir w como $w_1 w_2 \ldots w_k$ donde $X_i \stackrel{*}{\Rightarrow}_G w_i$. Además $X_i \stackrel{*}{\Rightarrow}_G w_i$ puede usar a lo más n pasos.

Ahora tenemos una produccuón $A \rightarrow X_1 X_2 \ldots X_k$ y sabemos por la hipótesis inductiva que podemos inferir que w_i está en el lenguaje de X_i.

Por lo que podemos inferir que $w_1 w_2 \ldots w_k$ está en el lenguaje de A.

5.8 Ambigüedad en Gramáticas y Lenguajes

En la gramática:
$E \rightarrow I$
$E \rightarrow E + E$
$E \rightarrow E * E$
$E \rightarrow (E)$
...
la sentencia $E + E * E$ tiene dos derivaciones: $E \Rightarrow E + E \Rightarrow E + E * E$ y $E \Rightarrow E * E \Rightarrow E + E * E$ lo cual nos da dos árboles de parseo:



Si tuvieramos números, e.g., 2,4 y 6, en lugar de las E's nos daría 26

por un lado y 36 por el otro. La existencia de varias derivaciones no es gran problema, sino la existencia de varios árboles de parseo.

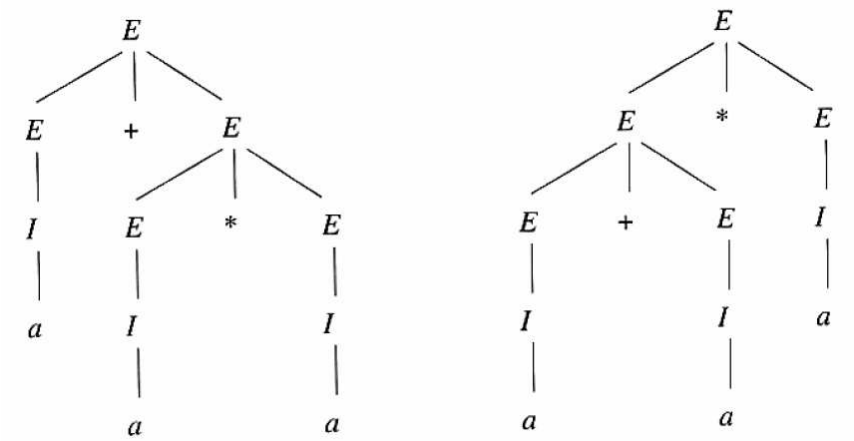
Por ejemplo, en la misma gramática:
...
$I \to a$
$I \to b$
$I \to Ia$
$I \to Ib$
$I \to I0$
$I \to I1$
la cadena $a + b$ tiene varias derivaciones:
$E \Rightarrow E + E \Rightarrow I + E \Rightarrow a + E \Rightarrow a + I \Rightarrow a + b$
$E \Rightarrow E + E \Rightarrow E + I \Rightarrow I + I \Rightarrow I + b \Rightarrow a + b$

sin embargo, sus árboles de parseo son los mismos y la estructura de $a + b$ no es ambígüa.

Definición: Sea $G = (V, T, P, S)$ una CFG. Decimos que G es *ambígüa* si existe una cadena en T^* que tenga más de un árbol de parseo.

Si todas las cadenas en $L(G)$ tienen a lo más un árbol de parseo, se dice que G es *no ambígüa*.

Ejemplo: La cadena $a + a * a$ tiene dos árboles de parseo:



5.8.1 Removiendo la ambigüedad de las gramáticas

Las buenas noticias: A veces se puede remover la ambigüedad "a mano"

Las malas noticias: no hay un algoritmo para hacerlo

Peores noticias: Algunos CFLs solo tienen CFGs ambígüas.

En la gramática: $E \rightarrow I|E+E|E*E|(E)$ y $I \rightarrow a|b|Ia|Ib|I0|I1$ existen dos problemas:

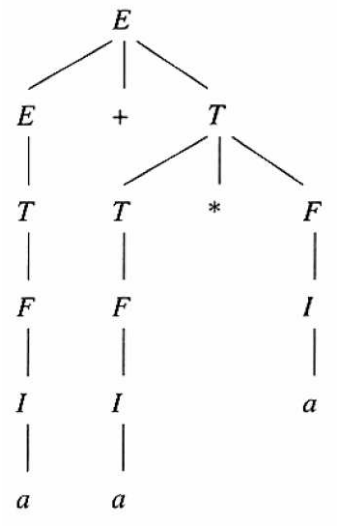
1. No hay precedencia entre $*$ y $+$
2. No existe un agrupamiento en las secuencias de operadores, e.g., $E + E + E$ significa: $E + (E + E)$ o $(E + E) + E$.

Solución: podemos introducir más variables para forzar un agrupamiento uniforme:

- Un *factor* (F) es una expresión que no puede separarse por un $*$ o $+$
- Un *término* (T) es una expresión que no puede separarse por un $+$
- El resto son expresiones que pueden separarse por $*$ o $+$

La gramática queda:
$I \rightarrow a|b|Ia|Ib|I0|I1$
$F \rightarrow I|(E)$
$T \rightarrow F|T * F$
$E \rightarrow T|E + T$

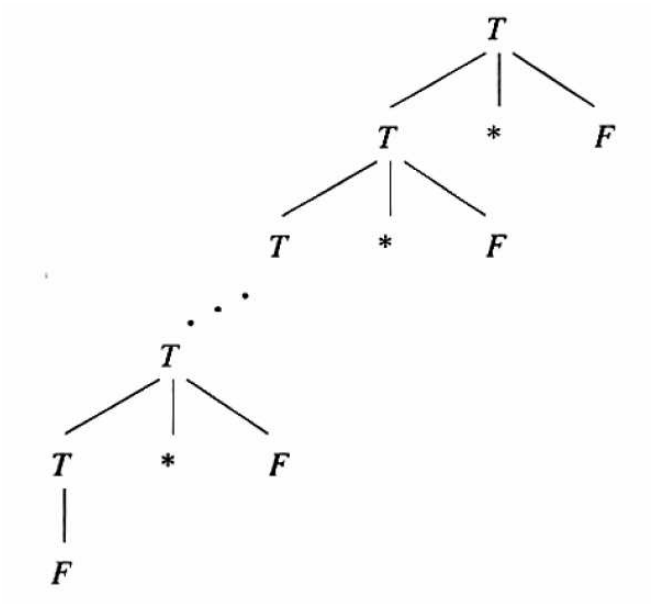
Con esto, el único árbol de parseo de $a + a * a$ es:



Las razones por las cuales la gramática nueva es no ambígüa son:

- Un factor es o un identificador o (E) para una expresión E
- El único árbol de parseo de una secuencia $f_1 * f_2 * \ldots * f_{n-1} * f_n$ de

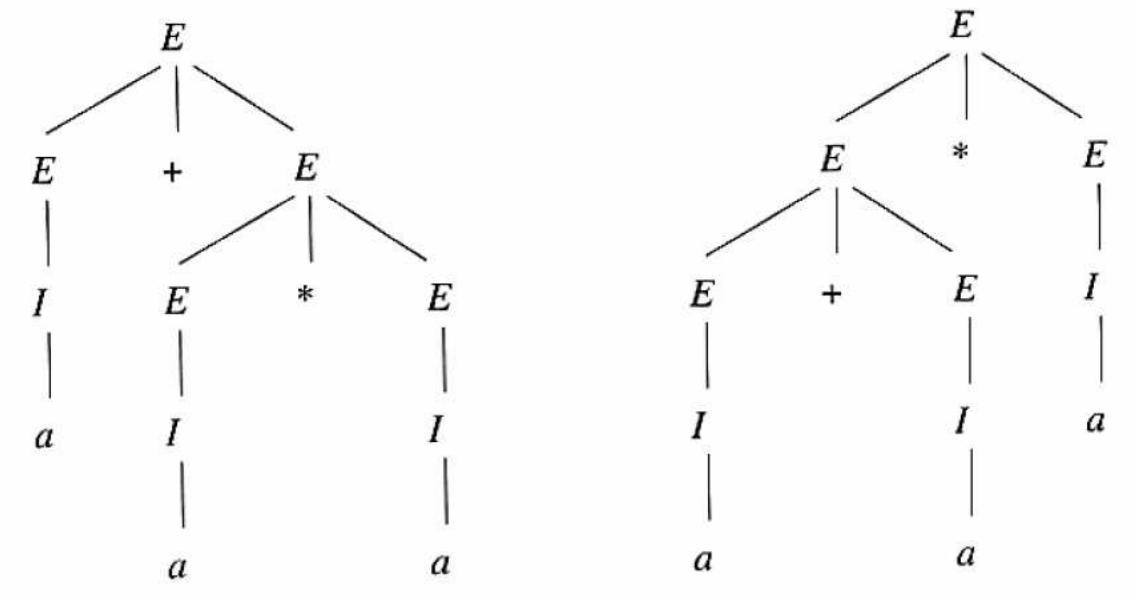
factores es el que da $f_1 * f_2 * \ldots * f_{n-1}$ como término y f_n como factor.



- Una expresión es una sequencia de: $t_1 + t_2 + \ldots + t_{n-1} + t_n$ de términos t_i y solo se puede parsear con $t_1 + t_2 + \ldots + t_{n-1}$ como una expresión y con t_n como término.

5.8.2 Derivaciones más a la izquierda y ambigüedad

En gramáticas no ambígüas, las derivaciones más a la izquierda y más a la derecha son únicas. Los dos árboles de derivación de $a + a * a$ son:



que nos da dos derivaciones:
$E \Rightarrow_{lm} E + E \Rightarrow_{lm} I + E \Rightarrow_{lm} a + E * E \Rightarrow_{lm} a + I * E \Rightarrow_{lm} a + a * E \Rightarrow_{lm} a + a * I \Rightarrow_{lm} a + a * a$
y
$E \Rightarrow_{lm} E + E * E \Rightarrow_{lm} I + E * E \Rightarrow_{lm} a + E * E \Rightarrow_{lm} a + I * E \Rightarrow_{lm} a + a * E \Rightarrow_{lm} a + a * I \Rightarrow_{lm} a + a * a$

En general:

- Se tienen un árbol de parseo pero varias derivaciones
- Muchas derivaciones más a la izquierda implican muchos árboles de parseo
- Muchas derivaciones más a la derecha implican muchos árboles de parseo

Teorema: For cualquier CFG G, una cadena de terminales w tiene dos árboles de parseo diferentes si y solo si w tiene dos derivaciones más a la izquierda distintas desde el símbolo inicial.

Esquema de la prueba: (Solo si) Si dos árboles de parseo difieren, tienen un nodo con diferentes producciones. Las derivaciones más a la izquierda correspondientes usaran sus derivaciones en estas dos producciones diferentes y por lo tanto serán distintas.

(Si) Analizando como se construye un árbol de parseo a partir de una derivación más a la izquierda, debe de ser claro que dos derivaciones distintas producen dos árboles de parseo distintos.

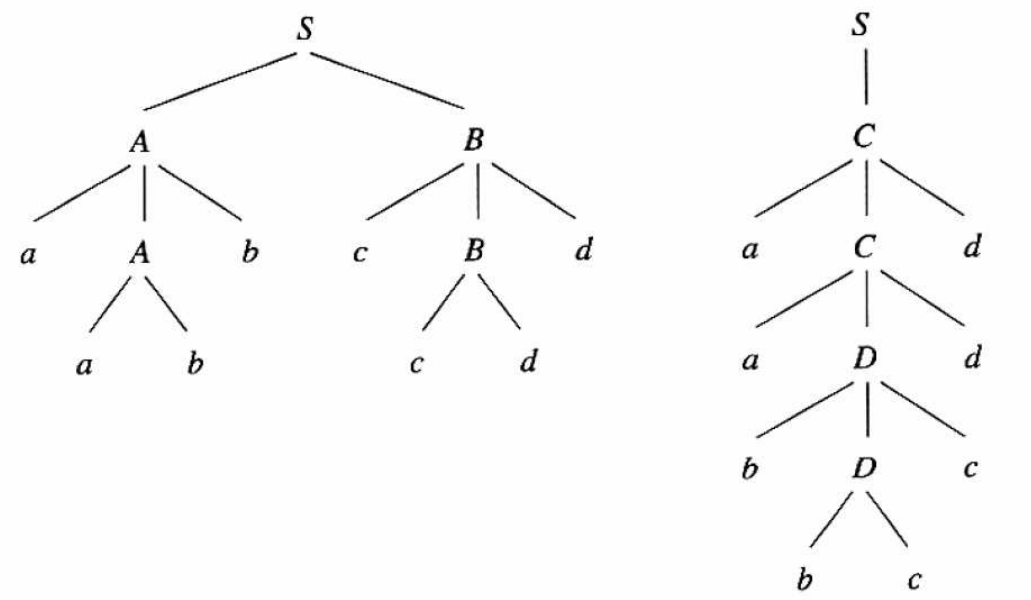
5.8.3 Ambigüedad Inherente

Un CFL L es *inherentemente ambígüo* si todas las gramáticas para L son ambígüas.

Ejemplo: Considere $L = \{a^n b^n c^m d^m : n \geq 1, m \geq 1\} \cup \{a^n b^m c^m d^n : n \geq 1, m \geq 1\}$

Una gramática para L es:
$S \rightarrow AB|C$
$A \rightarrow aAb|ab$
$B \rightarrow cBd|cd$
$C \rightarrow aCd|aDd$
$D \rightarrow bDc|bc$

Veamos los árboles para: *aabbccdd*:



Vemos que existen dos derivaciones más a la izquierda:

$S \Rightarrow_{lm} AB \Rightarrow_{lm} aAbB \Rightarrow_{lm} aabbB \Rightarrow_{lm} aabbcBd \Rightarrow_{lm} aabbccdd$
y
$S \Rightarrow_{lm} C \Rightarrow_{lm} aCd \Rightarrow_{lm} aabDdd \Rightarrow_{lm} aabbccdd$

Se puede mostrar que todas las gramáticas para L se comportan como la anterior. L es inherentemente ambígüo.

Capítulo 6

Autómatas de Pila (*Push-down automata*)

6.1 Pushdown Automata

Las gramáticas libres de contexto tienen un tipo de autómata que las define llamado *pushdown automata*.

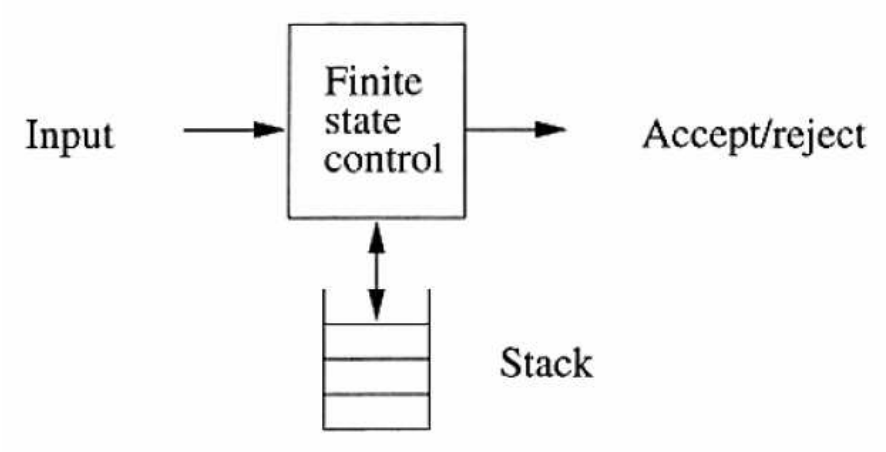
Un *pushdown automata* (PDA) es básicamente un ϵ-NFA con un *stack*, en donde se puede almacenar una cadena y por lo tanto se puede recordar información.

Sin embargo, sólo puede acceder a esta información en forma LIFO por lo que existen lenguajes reconocidos por una computadora pero no por un PDA, por ejemplo: $\{0^n1^n2^n | n \geq 1\}$.

En una transición el PDA:

1. Consume un símbolo de entrada o hace una transición vacía (ϵ)
2. Se va a un nuevo estado (o se queda en el mismo)
3. Reemplaza el primer elemento del *stack* por alguna cadena (puede ser el mismo símbolo de arriba del *stack*, lo que corresponde con no hacer

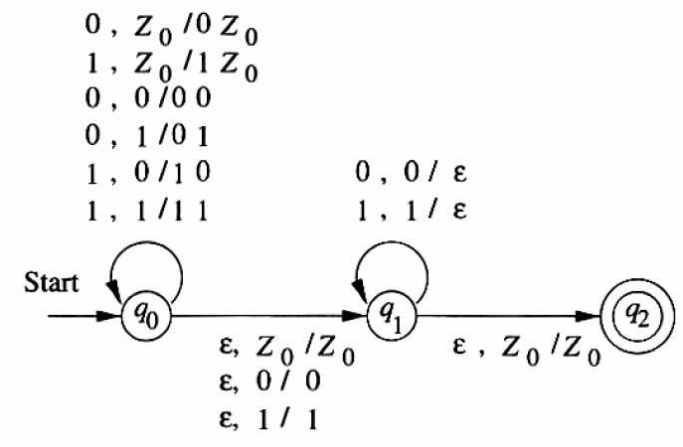
nada, hace *pop*, lo que corresponde con ϵ, o hace *push* de una cadena al *stack*)



Ejemplo: sea $L_w = \{ww^R : w \in \{0,1\}^*\}$ con una gramática: $P \to 0P0, P \to 1P1, P \to \epsilon$. Un PDA para L_{ww^r} tiene 3 estados y opera de la siguiente manera:

1. Adivina que esta leyendo w. Se queda en el estado 0 y *push* el símbolo de entrada al *stack*.
2. Adivina que está en medio de la cadena (ww^R) y se mueve espontáneamente al estado 1.
3. Está leyendo w^R y compara el valor con el valor de hasta arriba del *stack*. Si son iguales hace un *pop* y se queda en el estado 1.
4. Si se vacía el *stack*, se va al estado 2 y acepta.

El PDA para L_{ww^r} es el siguiente diagrama de transición:



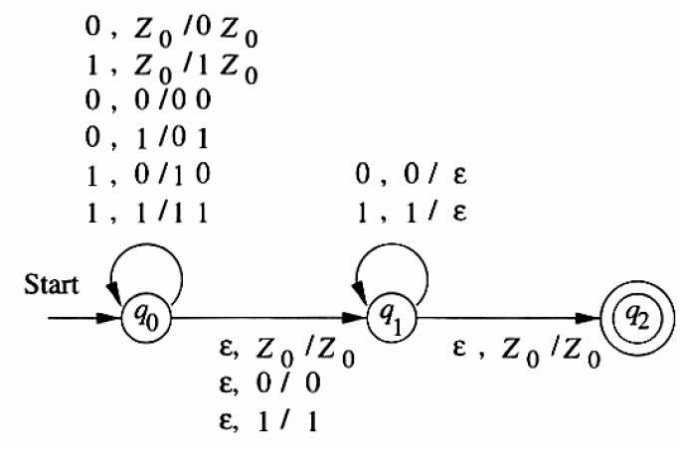
Los nodos, nodo inicial y final, son como los hemos visto antes. La diferencia principal es que en las transiciones (arcos) la etiqueta $a, X/\alpha$ significa que $\delta(q, a, X)$ tiene el par (p, α). O sea nos dice la entrada y cómo estaba y cómo queda la parte superior del *stack*.

Lo único que no nos dice es cuál es el símbolo inicial del *stack*. Por convención se usa Z_0.

Formalmente, un PDA es una séptupla: $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$, donde:

- Q: es un conjunto finito de estados
- Σ: es un alfabeto finito de entradas
- Γ: es un alfabeto finito del *stack*
- $\delta : Q \times \Sigma \cup \{\epsilon\} \times \Gamma \rightarrow 2^{Q \times \Gamma^*}$ es la función de transición
- q_0: es el estado inicial
- $Z_0 \in \Gamma$: es el símbolo inicial del *stack*
- $F \subseteq Q$: es el conjunto de estados finales o de aceptación

Ejemplo: El PDA



Es la siguiente tupla: $P = (\{q_0, q_1, q_2\}, \{0, 1\}, \{0, 1, Z_0\}, \delta, q_0, Z_0, \{q_2\})$.

Donde δ está definida por la siguiente tabla:

	$0, Z_0$	$1, Z_0$	$0, 0$	$0, 1$	$1, 0$	$1, 1$	ϵ, Z_0	$\epsilon, 0$	$\epsilon, 1$
$\rightarrow q_0$	$q_0, 0Z_0$	$q_0, 1Z_0$	$q_0, 00$	$q_0, 01$	$q_0, 11$	q_1, Z_0	$q_1, 0$	$q_1, 1$	
q_1			q_1, ϵ			q_1, ϵ	q_2, Z_0		
$*q_2$									

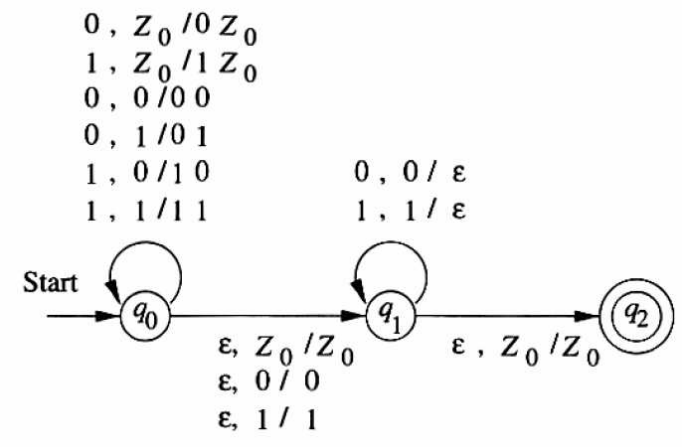
Descripciones instantáneas o IDs. Una ID es una tripleta (q, w, γ) donde q es un estado, w es lo que falta de la entrada y γ es el contenido del *stack*. Esto es lo que para los FA es el $\hat{\delta}$.

Usamos $\vdash$ para representar un movimiento en un PDA.

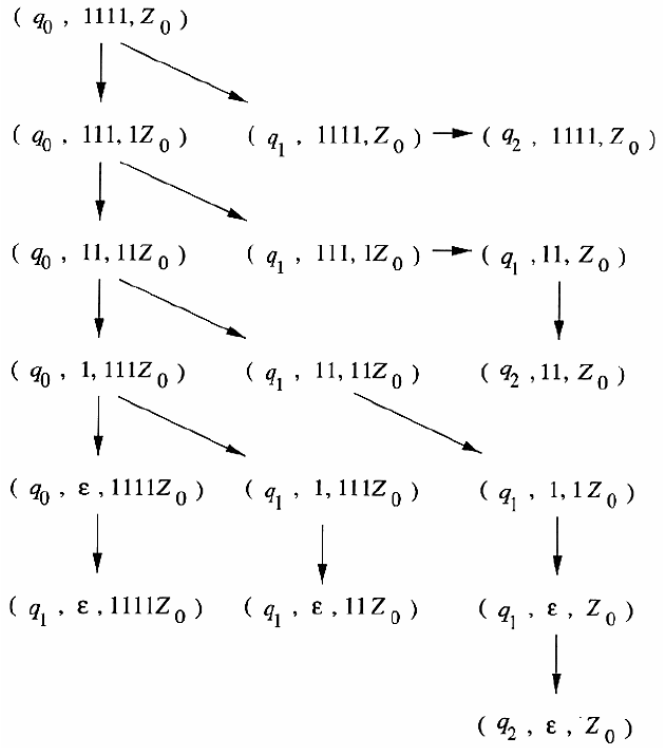
Sea $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ un PDA. Entonces $\forall w \in \Sigma^*, \beta \in \Gamma^*$: $(p, \alpha) \in \delta(q, a, X) \Rightarrow (q, aw, X\beta) \vdash (p, w, \alpha\beta)$.

Definimos como la cerradura reflexiva-transitiva de: $\vdash$

Ejemplo: Con la entrada 1111 el PDA:



Tiene la siguiente secuencia:



Se cumplen las siguientes propiedades. Si una secuencia ID es una computación legal para un PDA:

1. Entonces también es legal la secuencia que se obtiene al añadir una cadena al final del segundo componente.
2. Entonces también es legal la secuencia que se obtiene al añadir una cadena hasta abajo del tercer componente.
3. Y el final de una entrada no es consumida, el eliminar ese final de todos los ID's también resulta en una secuencia legal.

Teoremas: $\forall w \in \Sigma^*, \beta \in \Gamma^*\ (q, x, \alpha) \overset{*}{\vdash} (p, y, \beta) \Rightarrow (q, xw, \alpha\gamma) \overset{*}{\vdash} (p, yw, \beta\gamma)$

que cubre el primer principio si $\gamma = \epsilon$ y el segundo si $w = \epsilon$. Y: $(q, xw, \alpha) \overset{*}{\vdash} (p, yw, \beta) \Rightarrow (q, x, \alpha\gamma) \overset{*}{\vdash} (p, y, \beta)$

que cubre el tercer principio.

Existen dos formas equivalentes de PDA que aceptan un cierto lenguaje L:

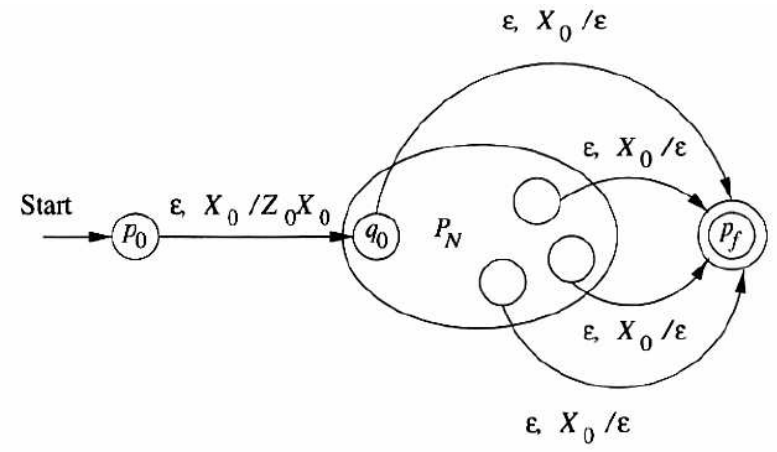
1. Aceptar por el estado final: $L(P) = \{w : (q_0, w, Z_0) \overset{*}{\vdash} (q, \epsilon, \alpha), q \in F\}$
2. Aceptar por el *stack* vacío: $N(P) = \{w : (q_0, w, Z_0) \overset{*}{\vdash} (q, \epsilon, \epsilon)\}$

Por ejemplo, podemos ver una secuencia legal para aceptar por estado final un palíndrome: $(q_0, ww^R, Z_0) \overset{*}{\vdash} (q_0, w^R, w^R Z_0) \vdash (q_1, w^R, w^R Z_0) \overset{*}{\vdash} (q_1, \epsilon, Z_0) \vdash (q_2, \epsilon, Z_0)$

Las dos representaciones son equivalentes y se puede pasar de una que acepta por *stack* vacío a una que acepta por estado final y viceversa.

6.1.1 De *stack* vacío a estado final

Se ilustra en la siguiente figura:

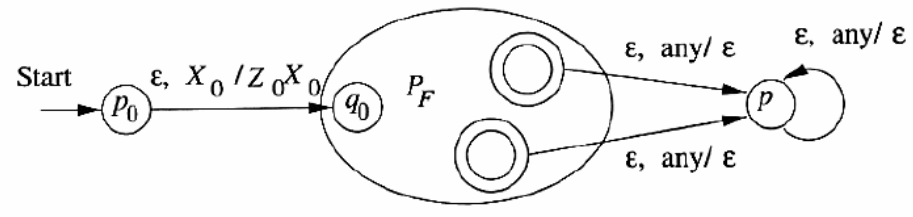


La idea, es usar un símbolo nuevo como marca para señalizar el final del *stack* y un nuevo estado cuyo único objetivo es poner Z0 arriba de este símbolo especial. Después se simula la misma transición de estados hasta que se vacíe el *stack*. Finalmente, añadimos un nuevo estado que es el de aceptación al que se llega cada vez que se vacía el *stack*.

Lo anterior lo podemos expresar como sigue: $(q_0, w, X_0) \vdash_F (q_0, w, Z_0X_0) \overset{*}{\vdash}_F (q, \epsilon, X_0) \vdash_F (p_f, \epsilon, \epsilon)$

6.1.2 De un estado final a un *stack* vacío

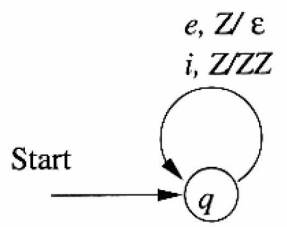
Se ilustra en la siguiente figura:



La idea es la siguiente: Cada vez que se llega a un estado final, se hace una transición vacía a un nuevo estado en donde se vacía el *stack* sin consumir símbolos de entrada. Para evitar simular una situación en donde se vacía el *stack* sin aceptar la cadena, de nuevo se introduce al principio un nuevo símbolo al *stack*.

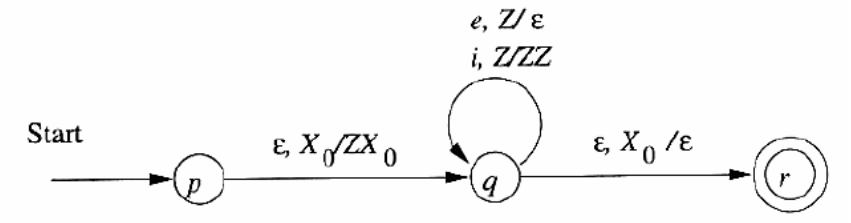
Lo anterior lo podemos expresar como sigue: $(p_0, w, X_0) \vdash_N (q_0, w, Z_0X_0) \vdash_N^* (q, \epsilon, \alpha X_0) \vdash_N^* (p, \epsilon, \epsilon)$

Ejemplo: podemos pasar del siguiente PDA que acepta por *stack* vacío a un PDA que acepta por estado final:



$P_N = (\{q\}, \{i, e\}, \{Z\}, \delta_N, q, Z)$, donde δ_N es:

$\delta_N(q, i, Z) = \{(q, ZZ)\}$
$\delta_N(q, e, Z) = \{(q, \epsilon)\}$



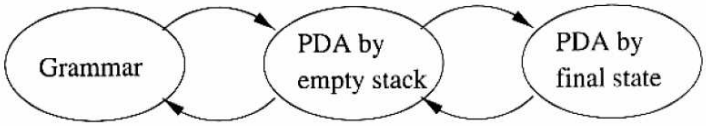
$P_F = (\{p, q, r\}, \{i, e\}, \{Z, X_0\}, \delta_F, p, X_0, r)$, donde δ_F es:

$\delta_F(p, \epsilon, X0) = \{(q, ZX_0)\}$
$\delta_F(q, i, Z) = \{(q, ZZ)\}$
$\delta_F(q, e, Z) = \{(q, \epsilon)\}$
$\delta_F(q, \epsilon, X_0) = \{(r, \epsilon)\}$

6.2 Equivalencia entre PDAs y CFGs

Los lenguajes definidos por los PDA son los lenguajes libres de contexto.

Un lenguaje es generado por un CFG si y solo si es aceptado por un PDA con *stack* vacío si y solo si es aceptado por un PDA con estado final.



La última parte ya la sabemos y sólo nos falta demostrar lo primero.

6.2.1 De un CFG a un PDA

Dada una gramática G vamos a construir un PDA que simula $\Rightarrow^*_{lm}$

Cualquier forma de sentencia izquierda que no es una cadena terminal la podemos escribir como $xA\alpha$ donde A es la variable más a la izquierda, x son los símbolos terminales que aparecen a la izquierda, y α es la cadena de símbolos terminales y variables que aparecen a la derecha de A.

La idea para construir un PDA a partir de una gramática, es que el PDA simula las formas de sentencia izquierdas que usa la gramática para generar una cadena w.

$A\alpha$ que es la "cola" de la forma de sentencia izquierda que va a aparecer en el *stack* con A como primer elemento.

Sea $xA\alpha \Rightarrow_{lm} x\beta\alpha$. Usa o adivina que se usa la producción $A \to \beta$. Esto corresponde a un PDA que consume primero a x con $A\alpha$ en el *stack* y luego con ϵ saca (*pops*) A y mete (*pushes*) β. O de otra forma, sea $w = xy$, entonces el PDA va en forma no-determinísta de la configuración $(q, y, A\alpha)$ a la configuración $(q, y, \beta\alpha)$.

En $(q, y, \beta\alpha)$ el PDA se comporta como antes, a menos que sean símbolos terminales en el prefijo de β, en cuyo caso el PDA los saca (*pops*) del *stack*,

si puede consumir símbolos de entrada.

Si todo se adivina bien, el PDA acaba con un *stack* vacío y con la cadena de entrada. Nótese que el PDA tiene un solo estado.

Formalmente, sea $G = (V, T, Q, S)$ un CFG. Se define un P_G como $(\{q\}, T, V \cup T, \delta, q, S)$ donde: $\delta(q, \epsilon, A) = \{(q, \beta) : A \to \beta \in Q\}$ para $A \in V$, y $\delta(q, a, a) = \{(q, \epsilon)\}$ para $a \in T$.

Ejemplo: convirtamos la siguiente gramática en un PDA. $I \to a|b|Ia|Ib|I0|I1$
$E \to I|E * E|E + E|(E)$

Los símbolos terminales del PDA son: $\{a, b, 0, 1, (,), +, *\}$

La función de transición del PDA es:
$\delta(q, \epsilon, I) = \{(q, a), (q, b), (q, Ia), (q, Ib), (q, I0), (q, I1)\}$
$\delta(q, \epsilon, E) = \{(q, I), (q, E + E), (q, E * E), (q, (E))\}$
$\delta(q, a, a) = \{(q, \epsilon)\}, \delta(q, b, b) = \{(q, \epsilon)\},$
$\delta(q, 0, 0) = \{(q, \epsilon)\}, \delta(q, 1, 1) = \{(q, \epsilon)\},$
$\delta(q, (, () = \{(q, \epsilon)\}, \delta(q,),)) = \{(q, \epsilon)\},$
$\delta(q, +, +) = \{(q, \epsilon)\}, \delta(q, *, *) = \{(q, \epsilon)\}.$

Teorema: $N(PG) = L(G)$.

Sea $w \in L(G)$, entonces, $S = \gamma_1 \Rightarrow_{lm} \gamma_2 \Rightarrow_{lm} \cdots \gamma_n = w$ Sea $\gamma_i = x_i\alpha_i$. Vamos a probar por inducción en i que si: $S \overset{*}{\Rightarrow}_{lm} \gamma_i$ entonces: $(q, w, S) \overset{*}{\vdash} (q, y_i, \alpha_i)$ Donde $w = x_i y_i$. El caso base es fácil, donde $i = 1, \gamma_1 = \epsilon$, y $y_1 = w$. Dado $(q, w, S) \overset{*}{\vdash} (q, y_i, \alpha_i)$ queremos probar para el siguiente paso, o sea: $(q, y_i, \alpha_i) \vdash (q, y_{i+1}, \alpha_{i+1})$

α_i empieza con una variable A y tenemos lo siguiente: $\underbrace{x_i A\chi}_{\gamma_i} \Rightarrow_{lm} \underbrace{x_{i+1}\beta\chi}_{\gamma_{i+1}}$

Por la hipótesis inductiva, $A\chi$ está en el *stack* y y_i no ha sido consumida. De la construcción de P_G sigue que podemos hacer el siguiente movimiento: $(q, y_i, A\chi) \vdash (q, y_i, \beta\chi)$.

Si β tiene un prefijo de terminales, podemos sacarlos (*pop*) con terminales correspondientes en un prefijo de y_i, acabando con la configuración

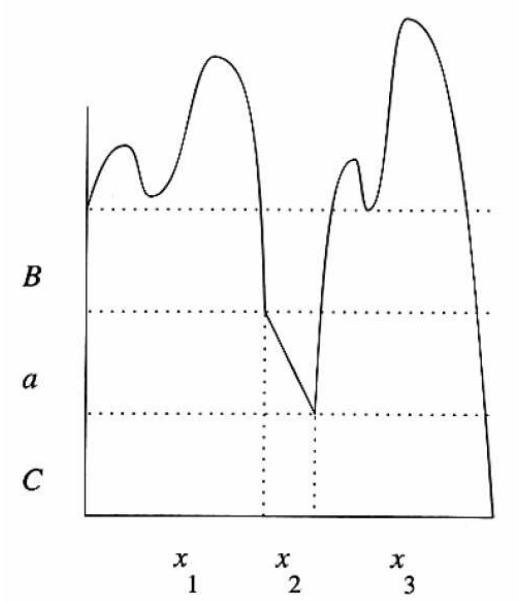
$(q, y_{i+1}, \alpha_{i+1})$, donde $\alpha_{i+1} = \beta\chi$ que es la parte final de: $x_{i+1}\beta\chi = \gamma_{i+1}$.

Finalmente, como $\gamma_n = w$, tenemos que $\alpha_n = \epsilon$, y $y_n = \epsilon$, y por lo tanto: $(q, w, S) \overset{*}{\vdash} (q, \epsilon, epsilon)$, por lo que $w \in N(P_G)$, osea que P acepta w con *stack* vacío.

Para la otra parte de la prueba, tenemos que probar que si: $(q, x, A) \overset{*}{\vdash} (q, \epsilon, \epsilon)$ entonces: $A \overset{*}{\Rightarrow} x$.

El caso base es simple, con $x = \epsilon$. Para el caso inductivo, tenemos que, como A es una variable, tenemos en general: $(q, x, A) \vdash (q, x, Y_1Y_2 \ldots Y_k) \vdash \ldots \vdash (q, \epsilon, \epsilon)$ donde $A \to Y_1Y_2 \ldots Y_k$ está en G.

Si escribimos x como $x_1x_2 \ldots x_k$, x_1 es la parte de la entrada que es consumida hasta que Y_1 es sacada (*popped*) del *stack*. Luego consumimos x_2 sacando a Y_2 del *stack*, y así sucesivamente. Podemos imaginarnos lo de la siguiente figura con $Y_1 = B, Y_2 = a$ y $Y_3 = C$.

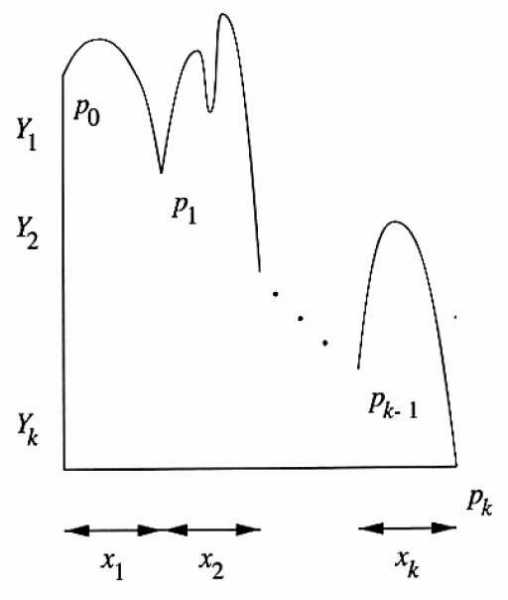


Aquí x es divido en 3 partes. La hipótesis inductiva nos dice que si $(q, x, A) \overset{*}{\vdash} (q, \epsilon, \epsilon)$ entonces $A \overset{*}{\Rightarrow} x$. Si ahora tenemos una x que la podemos descomponer en varias x_i, entonces podemos aplicar la hipótesis inductiva a

cada una de las partes y demostrar que: $A \Rightarrow Y_1Y_2 \dots Y_k \stackrel{*}{\Rightarrow} x_1Y_2 \dots Y_k \stackrel{*}{\Rightarrow} \dots \stackrel{*}{\Rightarrow} x_1x_2 \dots x_k$.

Para completar la prueba suponemos que $A = S$ y que $x = w$.

Se puede también demostrar como pasar de PDA's a CFG's. Osea como consumir una cadena $x = x_1x_2 \dots x_n$ y vaciar el *stack*.



La idea es definir una gramática con variables de la forma $[p_{i-1}Y_ip_i]$ que representan ir de p_{i-1} a p_i haciendo un *pop* de Y_i. Formalmente, sea $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ un PDA. Definimos una gramática $G = (V, \Sigma, R, S)$, como:

$V = \{pXq] : \{p, q\} \subseteq Q, Xin\Gamma\} \cup \{S\}$
$R = \{S \rightarrow [q_0Z_0p] : p \in Q \cup \{[qXr_k] \rightarrow a[rY_1r_1] \dots [r_{k-1}Y_kr_k] : a \in \Sigma \cup \{\epsilon\}, \{r_1, \dots, r_k\} \subseteq Q, (r, Y_1Y_2 \dots Y_k) \in \delta(q, a, X)$

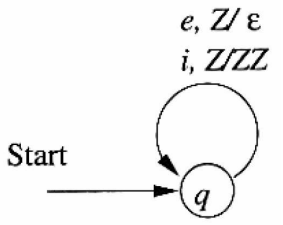
Osea las variables son el símbolo inicial S junto con todos los símbolos formados por pXq donde p y q son estados de Q y X es un símbolo del *stack*.

Las producciones se forman primero para todos los estados p, G tiene la producción $S \rightarrow [q_0Z_0p]$, que genera todas las cadenas w que causan a P

sacar (pop) a Z_0 del *stack* mientras se va del estado q_0 al estado p. Esto es $(q_0, w, Z_0) \overset{*}{\vdash} (p, \epsilon, \epsilon)$.

Por otro lado, se tienen las producciones que dicen que una forma de sacar (pop) a X e ir de un estado q a un estado r_k, es leer a a (que puede ser ϵ), usar algo de entrada para sacar (pop) a Y_1 del *stack*, mientras vamos del estado r al estado r_1, leer más entrada que saca Y_2 del *stack* y así sucesivamente.

Ejemplo: Supongamos el siguiente PDA



$P_N = (\{q\}, \{i, e\}, Z, \delta_N, q, Z)$, donde $\delta_N(q, i, Z) = \{(q, ZZ)\}$, y $\delta_N(q, e, Z) = \{(q, \epsilon)\}$. Podemos definir la siguiente gramática equivalente. $G = (V, \{i, e\}, R, S)$, donde $V = \{[qZq], S\}$ y $R = \{S \to [qZq], [qZq] \to i[qZq][qZq], [qZq] \to e\}$.

Asumimos que S es el símbolo de entrada para toda gramática. $[qZq]$ es la única tripleta que podemos formar con símbolos de entrada y símbolos del *stack*. A partir de S la única producción entonces que tenemos es: $S \to [qZq]$. Debido que tenemos la transición: $\delta_N(q, i, Z) = \{(q, ZZ)\}$ generamos la producción: $[qZq] \to i[qZq][qZq]$, y con $\delta_N(q, e, Z) = \{(q, \epsilon)\}$, generamos la producción: $[qZq] \to e$.

Si remplazamos a $[qZq]$ por A, nos queda: $S \to A$ y $A \to iAA|e$. De hecho podemos poner simplemente $S \to iSS|e$.

Ejemplo2: Sea $P_N = (\{p, q\}, \{0, 1\}, \{X, Z_0\}, \delta, q, Z_0)$, donde δ está dada por: $\delta(q, 1, Z_0) = \{(q, XZ_0)\}$
$\delta(q, 1, X) = \{(q, XX)\}$
$\delta(q, 0, X) = \{(p, X)\}$
$\delta(q, \epsilon, X) = \{(q, \epsilon)\}$
$\delta(p, 1, X) = \{(p, \epsilon)\}$
$\delta(p, 0, Z_0) = \{(q, Z_0)\}$

El CFG equivalente es: $G(V, \{0,1\}, R, S)$, donde:
$V = \{[pXp], [pXq], [pZ0p], [pZ0q], [qXq], [qXp], [qZ0q], [qZ0p], S\}$ y las reglas de producción R son:

- $S \to [qZ0q]|[qZ0p]$
- De la primera regla de transición:
  $[qZ0q] \to 1[qXq], [qZ0q]$
  $[qZ0q] \to 1[qXp], [pZ0q]$
  $[qZ0p] \to 1[qXq], [qZ0p]$
  $[qZ0p] \to 1[qXp], [pZ0q]$
- De la 2:
  $[qXq] \to 1[qXq], [qXq]$
  $[qXq] \to 1[qXp], [pXq]$
  $[qXp] \to 1[qXq], [qXp]$
  $[qXp] \to 1[qXp], [pXp]$
- De la 3:
  $[qXq] \to 0[pXq]$
  $[qXp] \to 0[pXp]$
- De la 4:
  $[qXq] \to \epsilon$
- De la 5:
  $[pXp] \to 1$
- De la 6:
  $[pZ0q] \to 0[qZ0q]$
  $[pZ0p] \to 0[qZ0p]$

Se puede probar que si G se construye como arriba a partir de un PDA P, entonces, $L(G) = N(P)$. La prueba se hace por inducción sobre las derivaciones. Se quiere probar que: Si $(q, w, X) \overset{*}{\vdash} (p, \epsilon, \epsilon)$ entonces: $[qXp] \overset{*}{\Rightarrow} w$.

El caso base, es sencillo cuando w es a o ϵ y $(p, \epsilon) \in \delta(q, w, X)$, por lo que por construcción de G tenemos $[qXp] \to w$ y por lo tanto $[qXp] \overset{*}{\Rightarrow} w$.

Para la parte de inducción tenemos que: $(q, w, X) \vdash (r_0, x, Y_1 Y_2 \dots Y_k) \vdash \dots \vdash (p, \epsilon, \epsilon)$. Donde $w = ax$ o $w = \epsilon x$, por lo que $(r_0, x, Y_1 Y_2 \dots Y_k) \in \delta(q, a, X)$. Entonces tenemos la siguiente producción: $[qXr_k] \to a[r_0 Y_1 r_1] \dots [r_{k-1} Y_k r_k]$ para toda $\{r_1, \dots, r_k\} \subset Q$.

Podemos escoger r_i ser el estado en la secuencia cuando Y_i sale del *stack* (*popped*). Sea $w = w_1 w_2 \dots w_k$, donde w_i es consumido cuando Y_i es sacado (*popped*). Entonces: $(r_{i-1}, w_i, Y_i) \overset{*}{\vdash} (r_i, \epsilon, \epsilon)$

Por la hipótesis de inducción tenemos: $[r_{i-1}, Y, r_i] \overset{*}{\Rightarrow} w_i$.

Por lo que obtenemos la siguiente secuencia de derivación:
$[qXr_k] \Rightarrow a[r_0 Y_1 r_1] \dots [r_{k-1} Y_k r_k] \overset{*}{\Rightarrow}$
$aw_1[r_1 Y_2 r_2][r_2 Y_3 r_3] \dots [r_{k-1} Y_k r_k] \overset{*}{\Rightarrow}$
$aw_1 w_2[r_2 Y_3 r_3] \dots [r_{k-1} Y_k r_k] \overset{*}{\Rightarrow}$
$\dots$
$aw_1 w_2 \dots w_k = w$

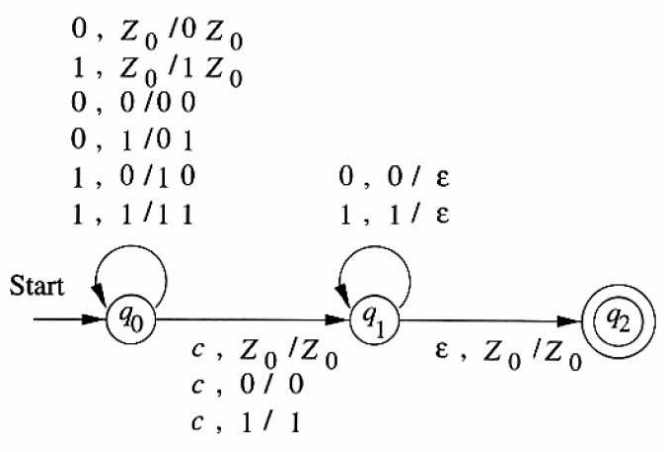
Para el *only-if* queremos probar que si: $[qXp] \overset{*}{\Rightarrow} w$ entonces: $(q, w, X) \overset{*}{\vdash} (p, \epsilon, \epsilon)$ La prueba sigue las mismas ideas que hemos estado viendo.

6.3 PDA's determinísticos

Un PDA $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ es *determinístico* si y solo si:

1. $\delta(q, a, X)$ es siempre vacío o es un singleton.
2. Si $\delta(q, a, X)$ no es vacío, entonces $\delta(q, \epsilon, X)$ debe de ser vacío.

Ejemplo: Sea $L_{wcw^r} = \{wcw^R : w \in \{0,1\}^*\}$. Entonces L_{wcw^r} es reconocido por el siguiente DPDA:



Se puede demostrar que $RE \subset L(DPDA) \subset CFL$. La primera parte es fácil. Si es RE se puede construir un DFA, por lo que construir un DPDA es trivial a partir de este. De hecho podemos ignorar el *stack*.

6.3.1 Algunas propiedades

$RE \subset L(DPDA) \subset CFL$

Por ejemplo: L_{wcw^r} es reconocido por $L(DPDA)$ pero no por RE.

L_{ww^r} es reconocido por un lenguaje de un CFG pero no por $L(DPDA)$.

Si $L = L(P)$ para algún DPDA P, entonces L tiene un CFG no ambiguo.

Capítulo 7

Propiedades de los Lenguajes Libres de Contexto

- Simplificación de CFG's. Esto facilita la vida, porque podemos decir que si un lenguaje es CF, entonces tiene una gramática de una forma especial.
- Lema de Pumping para CFL's. Similar a el caso regular. No se cubre en este curso.
- Propiedades de cerradura. Algunas, pero no todas, de las propiedades de cerradura de los lenguajes regulares que se acarrean a los CFL's.
- Propiedades de decisión. Podemos probar la membresía y vacío, pero por ejemplo, la equivalencia de CFL's no es decidible.

7.1 Forma Normal de Chomsky

Queremos mostrar que todo CFL (sin ϵ) se genera por una CFG donde todas las producciones son de la forma

$A \to BC$, o $A \to a$

Donde A, B, y C son variables, y a es un terminal. A esto se le conoce como CNF, y para llegar a ella debemos:

1. Eliminar símbolos no-útiles, aquellos que no aparecen en ninguna derivación $S \stackrel{*}{\Rightarrow} w$, para el símbolo de inicio S y el terminal w.
2. Eliminar las producciones-ϵ, es decir, producciones de la forma $A \to \epsilon$.
3. Eliminar producciones unitarias, es decir, producciones de la forma $A \to B$, donde A y B son variables.

7.1.1 Eliminando Símbolos no-Útiles

Un símbolo X es útil para una gramática $G = (V, T, P, S)$, si hay una derivación: $S \stackrel{*}{\Rightarrow}_G \alpha X \beta \stackrel{*}{\Rightarrow}_G w$ para una cadena terminal w. A los símbolos que no son útiles se les denomina *inútiles*.

Un símbolo X es *generador* si $X \stackrel{*}{\Rightarrow}_G w$, para alguna $w \in T^*$.

Un símbolo X es *alcanzable* si $S \stackrel{*}{\Rightarrow}_G \alpha X \beta$, para algún $\alpha, \beta \subseteq (V \cup T)^*$.

Cabe notar que si eliminamos a los símbolos no-generadores primero, y luego a los no-alcanzables, nos quedamos únicamente con símbolos útiles.

Ejemplo: Sea G: $S \to AB|a, A \to b$

S y A son generadores, B no lo es. Si eliminamos B tenemos que eliminar $S \to AB$, dejando la gramática $S \to a, A \to b$

Ahora sólo S es alcanzable. Eliminando A y b nos deja con $S \to a$

Con el lenguaje $\{a\}$.

De otra manera (para este ejemplo), si eliminamos primero los símbolos no-alcanzables, nos damos cuenta de que todos los símbolos son alcanzables.

A partir de: $S \to AB|a, A \to b$

Después eliminamos B como no-generador, y nos quedamos con $S \to a, A \to b$

que todavía contiene símbolos inútiles.

Teorema 7.2: Sea $G = (V, T, P, S)$ una CFG tal que $L(G) \neq \emptyset$. Sea $G1 = (V_1, T_1, P_1, S)$ la gramática obtenida:

1. Eliminando todos los símbolos no-generadores y las producciones en las que ocurren. Sea la nueva gramática $G_2 = (V_2, T_2, P_2, S)$.
2. Eliminando de G_2 todos los símbolos no-alcanzables y las producciones en que ocurren.

G_1 no tiene símbolos inútiles, y $L(G_1) = L(G)$.

Prueba: Primero probamos que G_1 no tiene símbolos inútiles:

Si $X \in (V_1 \cup T_1)$. Así $X \stackrel{*}{\Rightarrow} w$ en G_1, para algún $w \in T^*$. Además, cada símbolo utilizado en esta derivación también es generador. Así que $X \stackrel{*}{\Rightarrow} w$ en G_2 también.

Como X no se eliminó en el paso 2, hay una α y una β, tal que $S \stackrel{*}{\Rightarrow} \alpha X \beta$ en G_2. Aún más, cada símbolo utilizado en esta derivación también es alcanzable, así $S \stackrel{*}{\Rightarrow} \alpha X \beta$ en G_1.

Sabemos que cada símbolo en $\alpha X \beta$ es alcanzable y que están en $V_2 \cup T_2$, entonces cada uno de ellos es generador en G_2.

La derivación terminal $\alpha X \beta \stackrel{*}{\Rightarrow} xwy$ en G_2 solo involucra símbolos que son alcanzables desde S, porque son alcanzados por símbolos en $\alpha X \beta$. De este modo, la derivación terminal también es una derivación de G_1, i.e.,

$S \Rightarrow \alpha X \beta \stackrel{*}{\Rightarrow} xwy$ en G_1.

Ahora mostramos que $L(G_1) = L(G)$.

Como sólo eliminamos símbolos y producciones de G a G_1 $(O_1) \subseteq P)$, entonces tenemos que $L(G_1) \subseteq L(G)$.

Entonces, sea $w \in L(G)$. Así $S \stackrel{*}{\Rightarrow}_G w$. Cada símbolo en esta derivación es evidentemente alcanzable y generador, entonces esta es también una derivación de G_1. Así $w \in L(G_1)$.

7.1.2 Cálculo de Símbolos Generadores y Alcanzables

Necesitamos algoritmos para calcular los símbolos generadores y alcanzables de $G = (V, T, P, S)$.

Los símbolos generadores $g(G)$ se calculan con el siguiente algoritmo de cerradura:

Base: Todo símbolo de T es generador, se genera a sí mismo.

Inducción: Suponemos que tenemos la producción $A \to \alpha$, y cada símbolo de α es generador. Entonces A es generador (esto incluye $\alpha = \epsilon$, las reglas que tienen a ϵ en el cuerpo son generadoras).

Ejemplo: Sea G: $S \to AB|a, A \to b$

Entonces, primero $g(G) = \{a, b\}$.

Como $S \to a$ ponemos a S en $g(G)$, y porque $A \to b$ añadimos también a A, y eso es todo, el conjunto de símbolos generadores es $\{a, b, A, S\}$.

Teorema 7.4: El algoritmo anterior encuentra todos y sólo los símbolos generadores de G.

Prueba: Se mostrara con una inducción sobre la fase en la cual un símbolo X se añade a $g(G)$ y que X es en verdad generador.

Entonces suponemos que X es generador. De este modo $X \stackrel{*}{\Rightarrow}_G w$, para alguna $w \in T^*$. Ahora probamos por inducción sobre esta derivación que $X \in g(G)$.

Base: Cero pasos. Entonces X es terminal y se añade en la base del algoritmo de cerradura.

Inducción: La derivación toma $n > 0$ pasos. Sea la primera producción utilizada $X \to \alpha$. Entonces

$X \Rightarrow \alpha \stackrel{*}{\Rightarrow} w$

y $\alpha \stackrel{*}{\Rightarrow} w$ en menos de n pasos y por la hipótesis de inducción $\alpha \in g(G)$. De la parte inductiva del algoritmo se concluye que $X \in g(G)$ porque $X \to \alpha$.

El conjunto de símbolos alcanzables $r(G)$ de $G = (V, T, P, S)$ se calcula con el siguiente algoritmo de cerradura:

Base: $r(G) = \{S\}$, S es alcanzable.

Inducción: Si la variable $A \in r(G)$ y $A \to \alpha \in P$ entonces se añaden todos los símbolos de α a $r(G)$.

Ejemplo: Sea $G : S \to AB|a, A \to b$

Entonces, primero $r(G) = \{S\}$.

Con base en la primera producción añadimos $\{A, B, a\}$ a $r(G)$.

Con base en la segunda producción añadimos $\{b\}$ a $r(G)$ y eso es todo.

Teorema 7.6: El algoritmo anterior encuentra todos y solo los símbolos alcanzables de G.

Prueba: Esta prueba se hace con un par de inducciones parecidas a las del teorema 7.4.

7.2 Eliminando Producciones-ϵ

Aunque las producciones ϵ son convenientes, no son esenciales.

Se probará que si L es CF, entonces $L - \{\epsilon\}$ tiene una CFG sin producciones ϵ.

La estrategia consiste en descubrir cuáles variables son *nulificables*. Se dice que la variable A es *nulificable* si $A \stackrel{*}{\Rightarrow} \epsilon$.

Sea A nulificable. Reemplazaremos una regla como

$A \to BAD$

con

$A \to BAD, A \to BD$

Y borraremos cualquier regla con cuerpo ϵ.

El siguiente algoritmo calcula $n(G)$, el conjunto de símbolos nulificables de una gramática $G = (V, T, P, S)$ como sigue:

Base: $n(G) = \{A : A \to \epsilon \in P\}$

Inducción: Si $\{C_1, C_2, \ldots, C_k\} \subseteq n(G)$ y $A \to C_1, C_2, \ldots, C_k \in P$, entonces $n(G) = n(G) \cup \{A\}$. Nota, cada $C - i$ debe ser una variable para ser nulificable, entonces se consideran sólo las producciones con cuerpos conformados de variables.

Teorema 7.7: En cualquier gramática G, los únicos símbolos nulificables son las variables encontradas por el algoritmo anterior.

Prueba: La inducción es sencilla en ambas direcciones.

Base: Un paso. $A \to \epsilon$ debe ser una producción y se encuentra en la parte base del algoritmo.

Inducción: Suponemos que $A \stackrel{*}{\Rightarrow} \epsilon$ en n pasos donde $n > 1$. El primer paso se ve como $A \Rightarrow C_1C_2 \ldots C_k \stackrel{*}{\Rightarrow} \epsilon$, donde cada C_i deriva ϵ en una secuencia de menos de n pasos.

Por la HI, cada C_i es descubierta por el algoritmo como nulificable. Entonces, por el paso de inducción, se descubre que A es nulificable, por la producción $A \to C_1C_2 \ldots C_k$

Una vez que conocemos los símbolos nulificables, podemos transformar G en G_1 como sigue:

- Para cada $A \to X_1X_2 \ldots X_k \in P$ con $m \leq k$ símbolos nulificables, reemplazar por 2^m reglas, una con cada sub-lista de los símbolos nulificables ausentes.

  Excepción: Si $m = k$ no añadimos la regla donde borramos todos los m símbolos nulificables.

- Borrar todas las reglas de la forma $A \to \epsilon$.

Ejemplo: Sea $G : S \to AB, A \to aAA|\epsilon, B \to bBB|\epsilon$

Ahora $n(G) = \{A, B, S\}$. La primer regla se convertirá en $S \to AB|A|B$
la segunda: $A \to aAA|aA|aA|a$
la tercera $B \to bBB|bB|bB|b$

Después borramos reglas con cuerpos-ϵ, y terminamos con la gramática G_1: $S \to AB|A|B, A \to aAA|aA|aA|a, B \to bBB|bB|bB|b$

La gramática final es: $S \to AB|A|B, A \to aAA|aA|a, B \to bBB|bB|b$.

Se concluye que la eliminación de producciones ϵ con la construcción anterior no cambia el lenguaje, excepto que ϵ ya no está presente si lo estaba en el lenguaje de G. Ahora la prueba de que para cada CFG G, hay una gramática G_1 sin producciones ϵ tal que: $L(G1) = L(G) - \{\epsilon\}$

Teorema 7.9: Si la gramática G_1 se construye a partir de G con la construcción anterior para eliminar producciones ϵ, entonces $L(G1) = L(G) - \{\epsilon\}$.

Prueba: Se probará el enunciado más fuerte:

$A \stackrel{*}{\Rightarrow} w$ en G_1 sí y solo si $w \neq \epsilon$ y $A \stackrel{*}{\Rightarrow} w$ en G.

Dirección (*sólo si*): Suponemos que $A \stackrel{*}{\Rightarrow} w$ en G_1. Entonces claramente $w \neq \epsilon$, porque G_1 no tiene producciones-ϵ.

Ahora mostraremos por una inducción sobre la longitud de la derivación que $A \stackrel{*}{\Rightarrow} w$ también en G.

Base: Un paso. Entonces existe $A \to w$ en G_1. A partir de la construcción de G_1 tenemos que existe $A \to \alpha$ en G, donde α es w más algunas variables nulificables esparcidas.

Entonces: $A \Rightarrow \alpha \stackrel{*}{\Rightarrow} w$ en G.

Inducción: La derivación toma $n > 1$ pasos.

Entonces $A \Rightarrow X_1X_2 \ldots X_k \stackrel{*}{\Rightarrow} w$ en G_1. y la primera derivación debió venir de una producción:

$A \to Y_1Y_2 \ldots Y_m$.

donde $m \geq k$, algunas Y_i's son X_j's y los otros son símbolos nulificables

de G.

Más aún, $w = w_1 w_2 \dots w_k$, y $X_i \stackrel{*}{\Rightarrow} w_i$ en G_1 en menos de n pasos.

Por la hipótesis de inducción tenemos que $X_i \stackrel{*}{\Rightarrow} w_i$ en G. Ahora tenemos:

$A \Rightarrow_G Y_1 Y_2 \dots Y_m \stackrel{*}{\Rightarrow}_G X_1 X_2 \dots X_k \stackrel{*}{\Rightarrow}_G w_1 w_2 \dots w_k = w$

Dirección (*si*): Sea $A \stackrel{*}{\Rightarrow}_G w$, y $w \neq \epsilon$. Se mostrará por inducción de la longitud de la derivación que $A \stackrel{*}{\Rightarrow} w$ en G_1.

Base: La longitud es uno. Entonces $A \to w$ esta en G, y como $w \neq \epsilon$ la regla también esta en G_1.

Inducción: La derivación toma $n > 1$ pasos. Entonces esto se ve como:

$A \Rightarrow_G Y_1 Y_2 \dots Y_m \stackrel{*}{\Rightarrow}_G w$

Ahora $w = w_1 w_2 \dots w_m$, y $Yi \stackrel{*}{\Rightarrow}_G w_i$ en menos de n pasos.

Sean $X_1 X_2 \dots X_k$ aquellos Y_j's en orden, tal que $w_j \neq \epsilon$. Entonces $A \to X_1 X_2 \dots X_k$ es una regla de G_1.

Ahora $X_1 X_2 \dots X_k \stackrel{*}{\Rightarrow}_G w$, las únicas Y_j's no presentes entre las X's son las que derivan ϵ.

Cada $X_j / Y_j \stackrel{*}{\Rightarrow}_G w_j$ en menos de n pasos, entonces, por la hipótesis de inducción tenemos que si $w \neq \epsilon$ entonces $Y_j \stackrel{*}{\Rightarrow} w_j$ en G_1.

Así $A \Rightarrow X_1 X_2 \dots X_k \stackrel{*}{\Rightarrow} w$ en G_1.

la demanda del teorema ahora sigue del enunciado: $A \stackrel{*}{\Rightarrow} w$ en G_1 sí y solo si $w \neq \epsilon$ y $A \stackrel{*}{\Rightarrow} w$ en G (presentado en la prueba del teorema 7.9) al elegir $A = S$.

7.3 Eliminando Producciones Unitarias

$A \to B$ es una producción unitaria, cuando A y B son variables.

Las producciones unitarias se pueden eliminar.

Veamos la gramática: $I \to a|b|Ia|Ib|I0|I1$
$F \to I|(E)$
$T \to F|T * F$
$E \to T|E + T$

tiene las producciones unitarias $E \to T, T \to F$, y $F \to I$.

Ahora expandiremos la regla $E \to T$ y obtendremos las reglas. Sustituimos en $E \to T$ a la T por F y luego por $(T * F)$:

$E \to F, E \to T * F$

después expandemos $E \to F$ y obtenemos:

$E \to I|(E)|T * F$

Finalmente expandemos $E \to I$ y obtenemos:

$E \to a|b|Ia|Ib|I0|I1|(E)|T * F$

El método de expansión trabaja siempre y cuando no haya ciclos en las reglas, por ejemplo en: $A \to B, B \to C, C \to A$.

El siguiente método basado en *pares unitarios* trabajará para todas las gramáticas.

(A, B) es un par unitario si $A \stackrel{*}{\Rightarrow} B$ utilizando únicamente producciones unitarias.

Nota: En $A \to BC, C \to \epsilon$ tenemos $A \stackrel{*}{\Rightarrow} B$, pero no utilizamos solo producciones unitarias.

Para calcular $u(G)$, el conjunto de todos los pares unitarios de $G = (V, T, P, S)$ utilizamos el siguiente algoritmo de cerradura.

Base: (A, A) es un par unitario para la variable A. Esto es, $A \stackrel{*}{\Rightarrow} A$ en cero pasos. $u(G) = \{(A, A) : A \in V\}$

Inducción: Suponemos que $(A, B) \in u(G)$ y que $B \to C \in P$ donde C es una variable. Entonces añadimos (A, C) a $u(G)$.

Teorema 7.11: El algoritmo anterior encuentra todos y solo los pares unitarios de una CFG G.

Prueba: En una dirección, hacemos una inducción sobre el orden en que se descubren los pares, si encontramos que (A, B) es un par uitario, entonces $A \stackrel{*}{\Rightarrow}_G B$ utilizando únicamente producciones unitarias (prueba omitida).

En la otra dirección, suponemos que $A \stackrel{*}{\Rightarrow}_G B$ usando únicamente producciones unitarias. Podemos mostrar por inducción de la longitud de la derivación que encontraremos el par (A, B).

Base: Cero pasos. Entonces $A = B$, y añadimos el par (A, B) en la base.

Inducción: Suponemos que $A \stackrel{*}{\Rightarrow} B$ en n pasos, para alguna $n > 0$, cada paso consiste en aplicar una producción unitaria. La derivación luce como:

$A \stackrel{*}{\Rightarrow} C \Rightarrow B$

$A \stackrel{*}{\Rightarrow} C$ toma $n-1$ pasos, y por la hipótesis inductiva, descubrimos el par (A, C).

La parte inductiva del algoritmo combina el par (A, C) con la producción $C \rightarrow B$ para inferir el par (A, B).

Para eliminar producciones unitarias, procedemos de la siguiente manera. Dada $G = (V, T, P, S)$, podemos construir $G_1 = (V, T, P_1, S)$:

1. Encontrando todos los pares unitarios de G.
2. Para cada par unitario (A, B), añadimos a P_1 todas las producciones $A \rightarrow \alpha$, donde $B \rightarrow \alpha$ es una producción no unitaria en P. Note que es posible tener $A = B$; de esta manera, P_1 contiene todas las producciones unitarias en P.

   $P_1 = \{A \rightarrow \alpha : \alpha \notin V, B \rightarrow \alpha \in P, (A, B) \in u(G)\}$

Ejemplo: A partir de la gramática:
$I \rightarrow a|b|Ia|Ib|I0|I1$
$F \rightarrow I|(E)$
$T \rightarrow F|T * F$
$E \rightarrow T|E + T$

podemos obtener:

Par	Producción
(E, E)	$E \to E + T$
(E, T)	$E \to T * F$
(E, F)	$E \to (E)$
(E, I)	$E \to a\|b\|Ia\|Ib\|I0\|I1$
(T, T)	$T \to T * F$
(T, T)	$T \to (E)$
(T, F)	$T \to a\|b\|Ia\|Ib\|I0\|I1$
(F, F)	$F \to (E)$
(F, I)	$F \to a\|b\|Ia\|Ib\|I0\|I1$
(I, I)	$I \to a\|b\|Ia\|Ib\|I0\|I1$

La gramática resultante es equivalente a la original (prueba omitida).

Resumen

Para "limpiar" una gramática podemos:

1. Eliminar producciones-ϵ
2. Eliminar producciones unitarias
3. Eliminar símbolos inútiles

en este orden.

7.4 Forma Normal de Chomsky, CNF

Ahora se mostrará que cada CFL no vacío sin ϵ tiene una gramática G sin símbolos inútiles, y de tal manera que cada producción tenga la forma:

$A \to BC$, donde $\{A, B, C\} \subseteq T$, o $A \to \alpha$, donde $A \in V$, y $\alpha \in T$.

Para lograr esto, iniciamos con alguna gramática para el CFL, y:

1. "Limpiamos la gramática".

2. Hacemos que todos los cuerpos de longitud 2 o más consistan solo de variables.

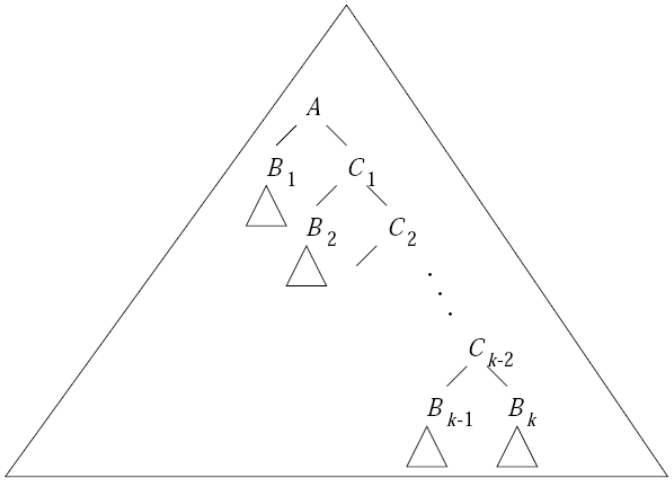
3. Dividimos los cuerpos de longitud 3 o más en una cascada de producciones con cuerpos-de-dos-variables.

Para el paso 2, por cada terminal a que aparece en un cuerpo de longitud ≥ 2, creamos una nueva variable, como A, y reemplazamos a a por A en todos los cuerpos. Después añadimos una nueva regla $A \to a$.

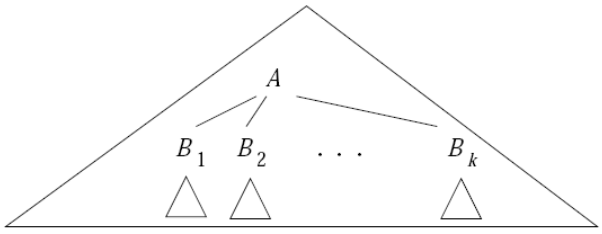
Para el paso 3, por cada regla de la forma $A \to B_1 B_2 \dots B_k$, $k \geq 3$, introducimos variables nuevas $C_1, C_2, \dots C_{k-2}$, y reemplazamos la regla con:

$A \to B_1 C_1$
$C_1 \to B_2 C_2$
$\dots$
$C_{k-3} \to B_{k-2} C_{k-2}$
$C_{k-2} \to B_{k-1} B_k$

Ilustración del efecto del paso 3:



(a)



(b)

7.4.1 Ejemplo de una Conversión CNF

Iniciamos con la gramática (el paso 1 ya esta hecho):
$E \to E + T|T * F|(E)|a|b|Ia|Ib|I0|I1$
$T \to T * E|(E)|a|b|Ia|Ib|I0|I1$
$F \to (E)|a|b|Ia|Ib|I0|I1$
$I \to a|b|Ia|Ib|I0|I1$

Para el paso 2, necesitamos las reglas:

$A \to a, B \to b, Z \to 0, O \to 1$
$P \to +, M \to *, L \to (, R \to)$

y al reemplazar obtenemos la gramática:
$E \to EPT|TMF|LER|a|b|Ia|Ib|I0|I1$
$T \to TPE|LEL|a|b|Ia|Ib|I0|I1$
$F \to LER|a|b|Ia|Ib|I0|I1$
$I \to a|b|Ia|Ib|I0|I1$
$A \to a, B \to b, Z \to 0, O \to 1$
$P \to +, M \to *, L \to (, R \to)$

Para el paso 3, reemplazamos:
$E \to EPT$ por $E \to EC_1, C : 1 \to PT$
$E \to TMF, T \to TMF$ por $E \to TC_2, T \to TC_2, C_2 \to MF$
$E \to LER, T \to LER, F \to LER$ por
$E \to LC_3, T \to LC_3, F \to LC_3, C_3 \to ER$

La gramática CNF final es:
$E \to EC_1|TC_2|LC_3|a|b|IA|IB|IZ|IO$
$T \to TC_2|LC_3|a|b|IA|IB|IZ|IO$
$F \to LC_3|a|b|IA|IB|IZ|IO$
$I \to a|b|IA|IB|IZ|IO$
$C_1 \to PT, C_2 \to MF, C_3 \to ER$
$A \to a, B \to b, Z \to 0, O \to 1$
$P \to +, M \to *, L \to (, R \to)$

7.5 Propiedades de Cerradura de los CFL's

Considérese el mapeo: $s : \Sigma \to 2^{\triangle^*}$ donde Σ y $\triangle$ son alfabetos finitos. Sea $w \in \Sigma^*$, donde $w = a_1 a_2 \dots a_n$, y se define:
$s(a_1 a_2 \dots a_n) = s(a_1).s(a_2).\dots.s(a_n)$ y, para $L \subseteq \Sigma*$, $s(L) = \bigcup_{w \in L} s(w)$

Tal mapeo s se llama una *substitución.*

Ejemplo: $\Sigma = \{0, 1\}, \triangle = \{a, b\}$, $s(0) = \{a^n b^n : n \geq 1\}, s(1) = \{aa, bb\}$.

Sea $w = 01$. Entonces $s(w) = s(0).s(1) = \{a^n b^n aa : n \geq 1\} \cup \{a^n b^{n+2} :$

$n \geq 1\}$

Sea $L = \{0\}^*$. Entonces $s(L) = (s(0))^* = \{a^{n_1}b^{n_1}a^{n_2}b_{n^2}\ldots a^{n_k}b^{n_k} : k \geq 0, n_i \geq 1\}$

Teorema 7.23: Sea L un CFL sobre Σ, y s una substitución, tal que $s(a)$ sea un CFL, $\forall a \in \Sigma$. Entonces $s(L)$ es un CFL.

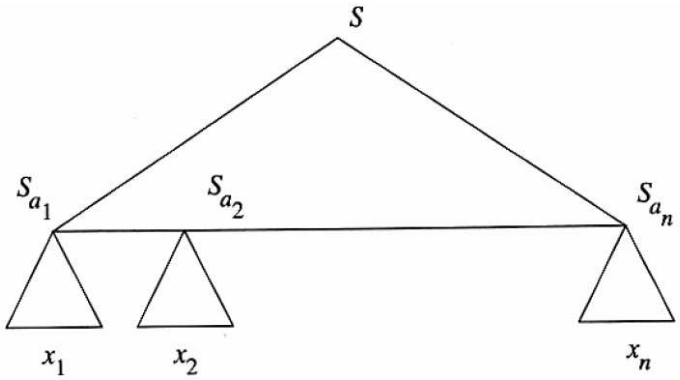
Iniciamos con las gramáticas: $G = (V, \Sigma, P, S)$ para L, y $G = (V_a, T_a, P_a, S_a)$ para cada $s(a)$. Construimos: $G = (V', T', P', S')$, donde:

$V' = (U_{a\in\Sigma}T_a) \cup V, T' = U_{a\in\Sigma}T_a, P' = U_{a\in\Sigma}P_a$ más las producciones de P con cada a en un cuerpo reemplazado con el símbolo S_a.

Ahora debemos mostrar que $L(G') = s(L)$.

Sea $w \in s(L)$. Entonces $\exists x = a_1a_2\ldots a_n$ en L, y $\exists x_i \in s(a_i)$, tal que $w = x_1x_2\ldots x_n$.

Un árbol de derivación en G' se verá como:



Así podemos generar $S_{a_1}S_{a_2}\ldots S_{a_n}$ en G' y de ahí generamos $x_1x_2\ldots x_n = w$. De este modo $w \in L(G')$.

Después, sea $w \in L(G')$. Entonces el árbol de parseo para w debe verse como el de arriba.

Ahora borramos los sub-árboles que cuelgan. Ahora se tiene la producción: $S_{a_1}S_{a_2}\ldots S_{a_n}$, donde $a_1a_2\ldots a_n \in L(G)$. Ahora w es también igual a $s(a_1a_2\ldots a_n)$, lo cual esta en $S(L)$.

7.6 Aplicaciones del Teorema de Substitución

Teorema 7.24: Los CFL's son cerrados bajo (i): unión, (ii): concatenación, (iii): Cerradura de Kleene y cerradura positiva +, y (iv): homomorfismo.

Prueba:

(i) : Sean L_1 y L_2 CFL's, sea $L = \{1, 2\}$, y $s(1) = L_1, s(2) = L_2$. Entonces $L_1 \cup L_2 = s(L)$.

(ii) : Aquí elegimos $L = \{1, 2\}$ y s como antes. Entonces $L_1 \cdot L_2 = s(L)$.

(iii) : Suponemos que L_1 es CF. Sea $L = 1*, s(1) = L_1$. Ahora $L_1^* = s(L)$. Prueba similar para +.

(iv) : Sea L_1 un CFL sobre Σ, y h un homomorfismo sobre Σ. Entonces definimos s por $a \mapsto \{h(a)\}$

Entonces $h(L) = s(L)$.

Teorema: Si L es CF, entonces también en L^R.

Prueba: Suponemos que L es generado por $G = (V, T, P, S)$. Construimos $G^R = (V, T, P^R, S)$, donde: $P^R = \{A \to \alpha^R : A \to \alpha \in P\}$

Mostrar (en casa) por inducción sobre las longitudes de las derivaciones en G (para una dirección) y en G^R (para la otra dirección) que $(L(G))^R = L(G^R)$.

Sea $L_1 = \{0^n 1^n 2^i : n \geq 1, i \geq 1\}$. L_1 es CF con la gramática:
$S \to AB$
$A \to 0A1|01$
$B \to 2B|2$

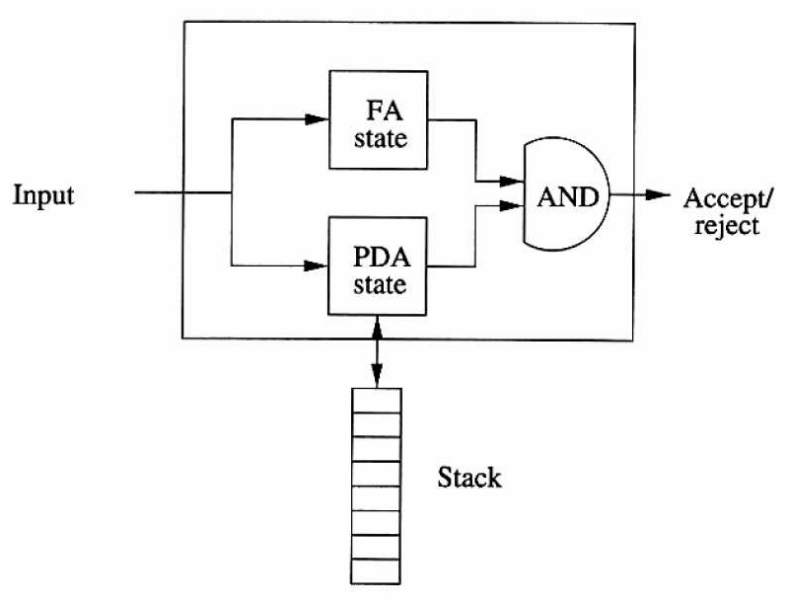
Además, $L_2 = \{0^i 1^n 2^n : n \geq 1, i \geq 1\}$ es CF con la gramática
$S \to AB$
$A \to 0A|0$
$B \to 1B2|12$

Sin embargo, $L_1 \cap L_2 = \{0^n 1^n 2^n : n \geq 1\}$ lo cual no es CF.

Teorema 7.27: Si L es CR, y R regular, entonces $L \cap R$ es CF.

Prueba: Sea L aceptado por el PDA: $P = (Q_P, \Sigma, \Gamma, \delta_P, q_P, Z_0, F_P)$, por el estado final, y R aceptado por el DFA: $A = (Q_A, \Sigma, \Gamma, \delta_A, q_A, Z_0, F_A)$

Construiremos un PDA para $L \cap R$ de acuerdo a la figura:



Formalmente, definimos: $P = (Q_P \times Q_A, \Sigma, \Gamma, \delta, (q_P, q_A), Z_0, F_P \times F_A)$ donde: $\delta((q,p),a,X) = \{((r, \hat{\delta}_A(p,a)), \gamma) : (r,\gamma) \in \delta_P(q,a,X)\}$

Probar en casa por inducción de $\overset{*}{\vdash}$, para P y para P' que: $(q_P, w, Z_0) \overset{*}{\vdash} (q, \epsilon, \gamma)$ en P sí y solo si $((q_P, q_A), w, Z_0) \overset{*}{\vdash} ((q, \hat{\delta}(p_A, w)), \epsilon, \gamma)$ en P'

porqué?

Teorema 7.29: Sean L, L_1, L_2 CFL's y R regular. Entonces:

1. L R es CF
2. $\overline{L}$ no es necesariamente CF

3. $L_1\ L_2$ no es necesariamente CF

Prueba:

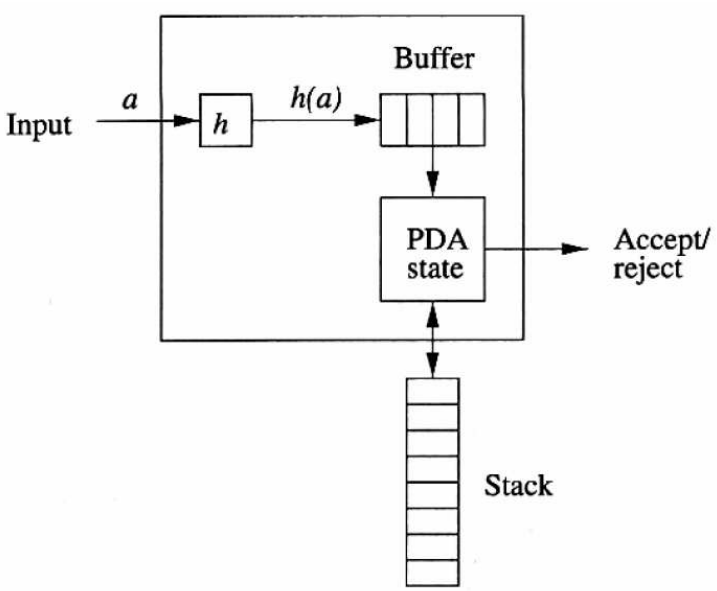
1. $\overline{R}$ es regular, $L \cap \overline{R}$ es regular, y $L \cap \overline{R} = L\ R$.
2. Si $\overline{L}$ siempre fue CF, se tiene que dar que: $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$ siempre sea CF.
3. Note que Σ^* es CF, de este modo, si $L_1\ L_2$ siempre fue CF, entonces también lo sería $\Sigma *\ L = \overline{L}$.

7.7 Homomorfismo Inverso

Sea $h : \Sigma \to \Theta^*$ un homomorfismo. Sea $L \subseteq \Theta^*$, y definimos: $h^{-1}(L) = \{w \in \Sigma^* : h(w) \in L\}$ ahora tenemos:

Teorema 7.30: Sea L un CFL, y h un homomorfismo. Entonces $h^{-1}(L)$ es un CFL.

Prueba: El plan de la prueba es



Sea L aceptado por el PDA: $P = (Q, \Theta, \Gamma, \delta, q_0, Z_0, F)$, construimos un nuevo PDA $P' = (Q', \Theta, \Gamma, \delta', (q_0, \epsilon), Z_0, F \times \{\epsilon\})$ donde:
$Q' = \{(q, x) : q \in Q, x \in suffix(h(a)), a \in \Sigma\}$
$\delta'((q, \epsilon), a, X) = \{((q, h(a)), X) : \epsilon \neq a \in T \cup \{\epsilon\}, q \in Q, X \in \Gamma\}$

Mostrar en casa con inducciones que:
$(q_0, h(w), Z_0) \overset{*}{\vdash} (p, \epsilon, \gamma)$ in P si y solo si $((q_0, \epsilon), w, Z_0) \overset{*}{\vdash} ((p, \epsilon), \epsilon, \gamma)$ en P'.

7.8 Propiedades de Decisión de los CFL's

Se verán los siguientes temas:

- Complejidad de convertir entre CFA's y PDAQ's
- Conversión de un CFG a CNF
- Probar $L(G) \neq \emptyset$, para una G dada

- Probar $w \in L(G)$, para una w dada y una G fija.
- Avances de problemas no decidibles de CFL

7.8.1 Convirtiendo entre CFA's y PDA's

El tamaño de la entrada es n.

Lo siguiente trabaja en tiempo $O(n)$:

1. Convertir una CFG a un PDA
2. Convertir un PDA de "estado final" a un PDA de "pila vacía"
3. Convertir un PDA de "pila vacía" a un PDA de "estado final"

7.8.2 Explosión Exponencial Evitable

Para convertir un PDA a una CFG tenemos:

Formalmente, sea $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0)$ un PDA. Definimos $G = (V, \Sigma, R, S)$, donde:

$$\begin{aligned} V = & \quad \{[pXq] : \{p, q\} \subseteq Q, X \in \Gamma\} \cup \{S\} \\ R = & \quad \{S \to [q_0 Z_0 p] : p \in Q\} \cup \{[qXr_k] \to a[rY_1r_1] \dots [r_{k-1}Y_k r_k] : \\ & \quad a \in \Sigma \cup \{\epsilon\}, \{r_1, \dots, r_k\} \subseteq Q, (r, Y_1 Y_2 \dots Y_k) \in \delta(q, a, X)\} \end{aligned}$$

cuando mucho n^3 variables de la forma $[pXq]$.

Si $(r, Y1Y2 \dots Yk) \in \delta(q, a, X)$, tendremos $O(n^n)$ reglas de la forma: $[qXr_k] \to a[rY_1r_1] \dots [r_{k-1}Y_k r_k]$

Introduciendo $k-2$ estados nuevos podemos modificar el PDA para hacer un *push* de a lo más un símbolo por transición.

Ahora, k será ≤ 2 para todas las reglas.

La longitud total de todas las transiciones es todavía $O(n)$.

Ahora, cada transición genera a lo más n^2 producciones.

El tamaño total (y tiempo para calcular) la gramática es entonces $O(n^3)$.

7.8.3 Conversión a CNF

Buenas noticias:

1. Calcular $r(G)$ y $g(G)$ y eliminar símbolos no-útiles toma tiempo $O(n)$. Esto se mostrará pronto.
2. El tamaño de $u(G)$ y la gramática resultante con producciones P_1 es $O(n^2)$
3. Arreglando que los cuerpos consistan solo de variables lleva $O(n)$
4. Dividir los cuerpos lleva $O(n)$

Malas noticias:

1. La eliminación de los símbolos nulificables puede hacer que la nueva gramática tenga un tamaño de $O(2^n)$

La mala noticia es evitable: Dividir los cuerpos primero antes de la eliminación de símbolos nulificables.

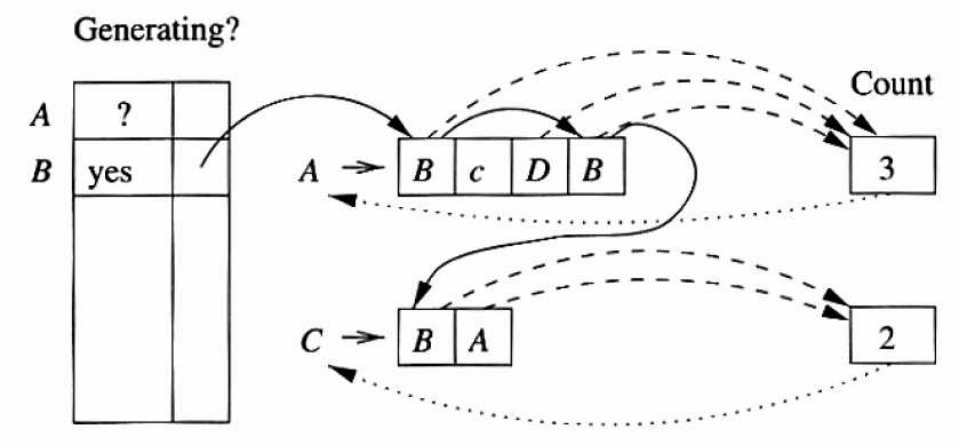
La conversión a CNF lleva $O(n^2)$

7.8.4 Probando si los CFL's son Vacíos

$L(G)$ es no-vacío si el símbolo de inicio S es generador.

Una implementación ingenua sobre $g(G)$ tome tiempo $O(n^2)$.

$g(G)$ se puede calcular en tiempo $O(n)$ como sigue:



La creación e inicialización del arreglo lleva $O(n)$

La creación e inicialización de las ligas y contadores lleva $O(n)$

Cuando un contador va a cero, tenemos que:

1. Encontrar la variable cabeza A, verificar si ya existe "yes" en el arreglo, y si no, ponerlo en la cola lleva $O(1)$ por producción. Total $O(n)$
2. Seguir las ligas para A, y decrementar los contadores. Toma un tiempo de $O(n)$.

El tiempo total es $O(n)$.

7.8.5 $w \in L(G)$?

Modo ineficiente:

Suponemos que G esta en CNF, probamos que la cadena sea w con $|w| = n$. Como el árbol de parseo es binario, hay $2n - 1$ nodos internos.

La generación de todos los árboles de parseo binarios de G con $2n - 1$ nodos internos.

Verificar si algún árbol de parseo genera w.

Algoritmo CYK para Probar Membresía

La gramática G es fija

La entrada es $w = a_1 a_2 \dots a_n$

Construimos una tabla triangular, donde X_{ij} contiene todas las variables A, tal que: $A \stackrel{*}{\Rightarrow}_G a_i a_{i+1} \dots a_j$:

$$\begin{array}{lllll} X_{15} & & & & \\ X_{14} & X_{25} & & & \\ X_{13} & X_{24} & X_{35} & & \\ X_{12} & X_{23} & X_{34} & X_{45} & \\ X_{11} & X_{22} & X_{33} & X_{44} & X_{55} \end{array}$$

Para llenar la tabla trabajamos renglón-por-renglón, hacia arriba

El primer renglón se calcula en la base, los subsecuentes en la inducción.

Base: $X_{ii} == \{A : A \to a_i \in G\}$

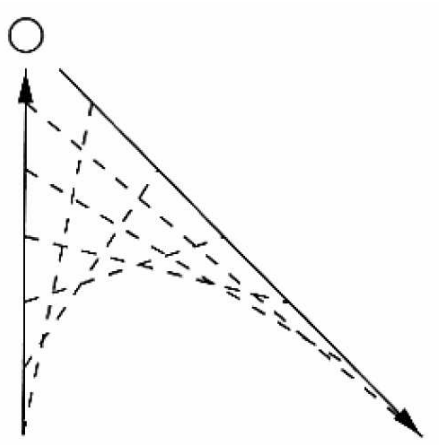
Inducción: Deseamos calcular X_{ij}, el cual esta en el renglón $j - i + 1$.

$A \in X_{ij}$ si $A \stackrel{*}{\Rightarrow} a_i a_{i+1} \dots a_j$ si para algúna $k < j$ y $A \to BC$, tenemos $B \stackrel{*}{\Rightarrow} a_i a_{i+1} \dots a_k$ y $C \stackrel{*}{\Rightarrow} a_{k+1} a_{k+2} \dots a_j$ si $B \in X_{ik}$ y $C \in X_{kj}$

Ejemplo: G tiene las producciones:
$S \to AB|BC$
$A \to BA|a$
$B \to CC|b$
$C \to AB|a$

Para calcular X_{ij} necesitamos comparar a lo más n pares de conjuntos previamente calculados: $(X_{ii}, X_{i+1,j}), (X_{i,i+1}, X_{i+1,j}), \dots, (X_{i,j-1}, X_{jj})$ como

se sugiere abajo:



Para $w = a_1 \dots a_n$, hay que calcular $O(n^2)$ entradas X_{ij}.

Para cada X_{ij} necesitamos comparar a lo más n pares $(X_{ik}, X_{k+1,j})$.

El trabajo total es $O(n^3)$.

7.8.6 Muestra de Problemas CFL No-Decidibles

Los siguientes son problemas no-decidibles:

1. Es una CFG G dada ambigua?
2. Es un CFL dado inherentemente ambiguo?
3. Es la intersección de dos CFL's vacía?
4. Son dos CFL's el mismo?
5. Es un CFL dado universal (igual a Σ^*)?

Capítulo 8

Máquinas de Turing

8.1 Introducción

Hasta ahora hemos visto clases de lenguajes relativamente simples. Lo que vamos a ver ahora es preguntarnos qué lenguajes pueden definirse por cualquier equipo computacional.

Vamos a ver qué pueden hacer las computadoras y los problemas que no pueden resolver, a los que llamaremos *indecidibles*.

Por ejemplo, podemos pensar en un programa sencillo de computadora que imprima "hola". De igual forma, podemos pensar en otro programa que imprima "hola" cuando encuentre un entero positivo $n > 2$, que cumpla: $x^n + y^n = z^n$, para x, y y z enteros positivos.

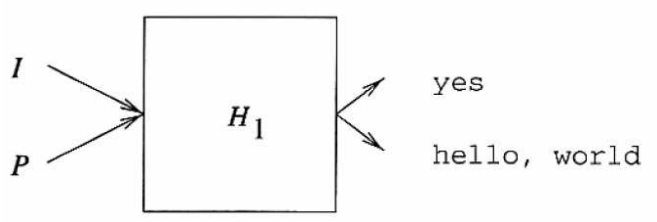
La solución entera de la ecuación de arriba se conoce como el último teorema de Fermat, que llevo a los matemáticos 300 años resolver.

El poder analizar cualquier programa de computadora y decidir si va a imprimir un letrero como "hola" es en general indecidible.

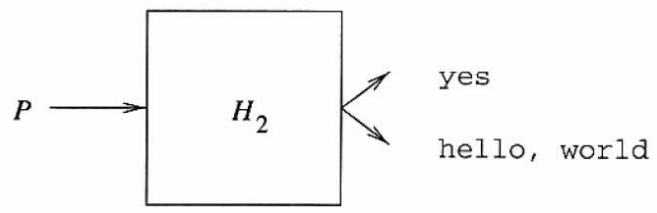
La idea de la prueba es relativamente simple. Necesitamos tener un programa H que toma de entrada otro programa P y una entrada a ese programa I y regresa "si" o "no" dependiendo de si el programa P imprime "hola" o

no.

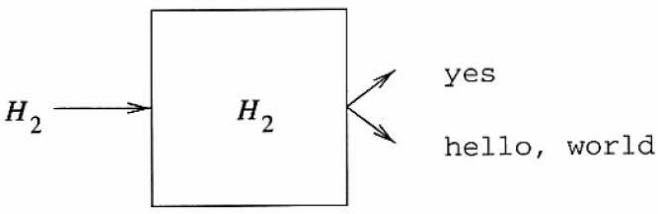
Podemos pensar igualmente en un programa $H1$ que imprima "si" cuando el programa P imprima "hola" e imprima "hola" cuando no.



Ahora podemos pensar en otro programa $H2$ que toma solo de entrada a P e imprime "si" cuando P imprime "hola" e imprime "hola" cuando P no imprime "hola".



Ahora si le damos $H2$ como entrada a $H2$ llegamos a una contradicción.



Muchas veces para probar si un problema es indecidible, se transforma a otra del cual ya se sabe que es indecidible.

Por ejemplo, si queremos probar que un programa va a llamar a una función "foo" es o no es indecidible. La idea es diseñar un programa que con cierta entrada llame a la función "foo" cuando otro programa imprima "hola".

8.2 Máquina de Turing

El propósito de la teoría de indecibilidad no es sólo establecer cuales problemas son indecidibles, sino también dar una guía sobre qué es lo que se puede hacer o no con programación.

También tiene que ver con problemas, que aunque decidibles, son intratables.

A finales del s. XIX y principios del s. XX, D. Hilbert lanzó la pregunta abierta, si era posible encontrar un algoritmo que determinara el valor de verdad de una fórmula en lógica de primer orden aplicada a los enteros.

En 1931, K. Gödel probó su teorema de incompletes usando un argumento parecido al de H2 que vimos arriba, para probar que no se podía construir dicho algoritmo.

En 1936, A. Turing publicó su máquina de Turing como un modelo para cualquier tipo de computación (aunque todavía no existían las computadoras). La hipótesis de Church o la tésis de Church-Turing dice que lo que las máquinas de Turing (y para tal caso las computadoras modernas) pueden computar son las funciones recursivamente enumerables.

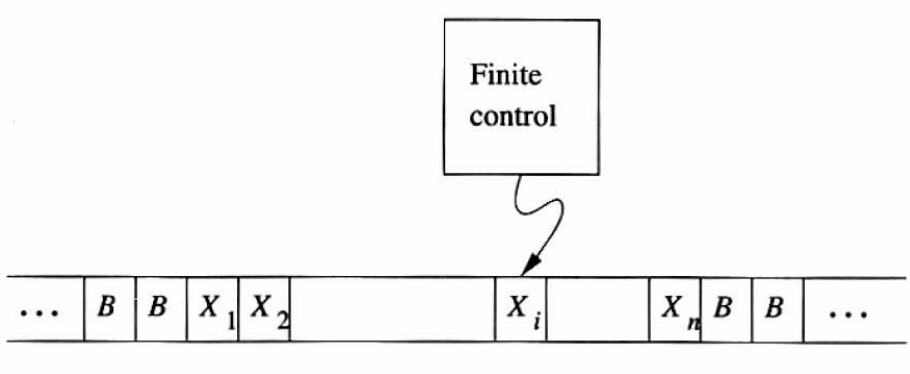
Una máquina de Turing consiste de un control finito que puede estar en cualquier estado de un conjunto finito de estados.

Se tiene una cinta dividida en celdas, cada celda con un símbolo. Inicialmente, la entrada (cadena finita de símbolos del alfabeto) se coloca en la cinta, el resto de las celdas tienen el símbolo especial vacío.

La cabeza de la cinta está siempre sobre una celda y al principio está sobre la celda más a la izquierda con el primer símbolo de la cadena de entrada.

Un movimiento o transición puede cambiar de estado (o quedarse en el estado actual), escribir un símbolo (reemplazando el símbolo que existía o

dejando el mismo) y mover la cabeza a la izquierda o derecha.



Formalmente, una máquina de Turing es una séptupla: $M = (Q, \Sigma, \Gamma, \delta, q0, B, F)$, donde:

- Q: es un conjunto finito de estados
- Σ: es un conjunto finito de símbolos de entrada
- Γ: es el conjunto de símbolos de la cinta. Σ es siempre un subconjunto de Γ
- δ: la función de transición $\delta(q, X) = (p, Y, D)$, donde p es el siguiente estado en Q, Y es el símbolo en Γ que se escribe en la celda que está viendo la cabeza de la cinta y D es la dirección (izq. o der.).
- q_0: es el estado inicial
- B: es el símbolo de vacío, que esta en Γ pero no en Σ
- F: es el conjunto de estados finales o de aceptación.

8.2.1 Descripciones instantáneas o IDs para las máquinas de Turing

Como la cinta es infinita, se representan sólo los símbolos entre los B's (a veces se pueden incluir algunos B's) y se incluye un símbolo especial para

indicar la posición de la cabeza. Por ejemplo: $X_1X_2\ldots X_{i-1}qX_iX_{i+1}\ldots X_n$ representa in ID donde:

q es el estado de la máquina de Turing.

La cabeza de la cinta está viendo el i-ésimo símbolo a la izquierda $X_1X_2\ldots X_n$ es el pedazo de cinta entre los símbolos más a la izquierda y más a la derecha que no son vacíos.

Usamos la misma notación de ID que para los PDAs: $\vdash$ y $\overset{*}{\vdash}$.

Supongamos que $\delta(q, X_i) = (p, Y, L)$, el siguiente movimiento es a la izquierda (L). Entonces:

$X_1X_2\ldots X_{i-1}qX_iX_{i+1}\ldots X_n \vdash X_1X_2\ldots X_{i-2}pX_{i-1}YX_{i+1}\ldots X_n$

Excepciones:

1. Si $i = 1$ entonces M se mueve al B a la izquierda de X_1 : $qX_1X_2\ldots X_n \vdash pBYX_2\ldots X_n$
2. Si $i = n$ y $Y = B$, entonces el símbolo que se re-escribe sobre X_n se une a la cadena infinita de B's y no se escribe: $X_1X_2\ldots X_{n-1}qX_n \vdash X_1X_2\ldots X_{n-2}pX_{n-1}$

Ahora, supongamos que $\delta(q, X_i) = (p, Y, R)$, movimiento hacia la derecha (R): $X_1X_2\ldots X_{i-1}qX_iX_{i+1}\ldots X_nX_1X_2\ldots X_{i-1}pYX_{i+1}\ldots X_n$

Excepciones:

1. Si $i = n$ entonces la $i + 1$ celda tiene un B que no era parte de la ID anterior: $X_1X_2\ldots X_{n-1}qX_n \vdash X_1X_2\ldots X_{n-1}YpB$
2. Si $i = 1$ y $Y = B$, entonces el símbolo que se re-escribe sobre X_1 se une a la cadena infinita de B's y no se escribe: $qX_1X_2\ldots X_n \vdash pX_2\ldots X_n$

Ejemplo: una TM que acepta: $\{0^n1^n|n \geq 1\}$

Nos queda lo siguiente: $M = (\{q_0, q_1, q_2, q_3, q_4\}, \{0, 1\}, \{0, 1, X, Y, B\}, \delta, q_0, B, \{q_4\})$

Con la siguiente tabla de transición:

	Símbolo				
Estado	0	1	X	Y	B
q_0	(q_1, X, R)	–	–	(q_3, Y, R)	–
q_1	$(q_1, 0, R)$	(q_2, Y, L)	–	(q_1, Y, R)	–
q_2	$(q_2, 0, L)$	–	(q_0, X, R)	(q_2, Y, L)	–
q_3	–	–	–	(q_3, Y, R)	(q_4, B, R)
q_4	–	–	–	–	–

Por ejemplo, si le damos la entrada 0011 sigue las siguientes transiciones:

$q_0 0011 \vdash X q_1 011 \vdash X 0 q_1 11 \vdash X q_2 0 Y 1 \vdash q_2 X 0 Y 1 \vdash X q_0 0 Y 1 \vdash X X q_1 Y 1 \vdash X X Y q_1 1 \vdash X X q_2 Y Y \vdash X q_2 X Y Y \vdash X X q_0 Y Y \vdash X X Y q_3 Y \vdash X X Y Y q_3 B \vdash X X Y Y B q_4 B$

Mientras que para la cadena 0010, tenemos lo siguiente:

$q_0 0010 \vdash X q_1 010 \vdash X 0 q_1 10 \vdash X q_2 0 Y 0 \vdash q_2 X 0 Y 0 \vdash X q_0 0 Y 0 \vdash X X q_1 Y 0 \vdash X X Y q_1 0 \vdash X X Y 0 q_1 B$

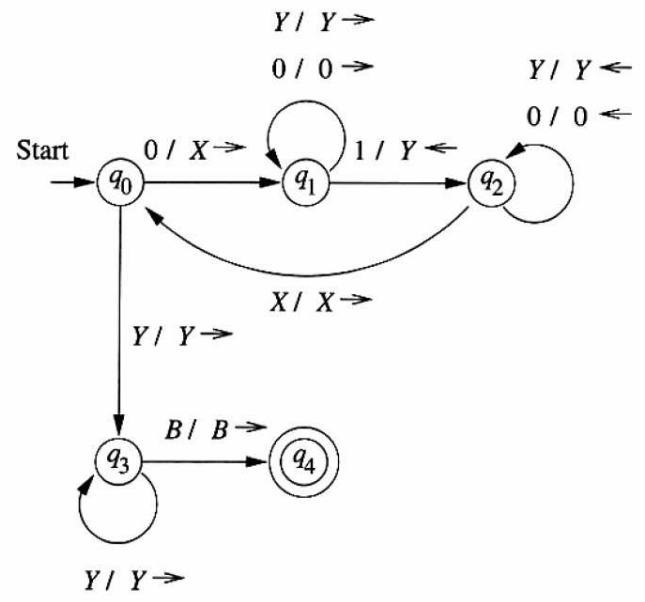
8.2.2 Diagramas de transición para TMs

En un diagrama de transición para TMs los nodos son los estados y los arcos tienen etiquetas de la forma X/YD donde X y Y son símbolos de la cinta y D es la dirección.

Para cada $\delta(q, X) = (p, Y, D)$, tenemos un arco con etiqueta: X/YD que va del nodo q al nodo p.

Lo único que falta es el símbolo vacío que asumimos que es B.

Por ejemplo, el diagrama de transición para la TM anterior es:

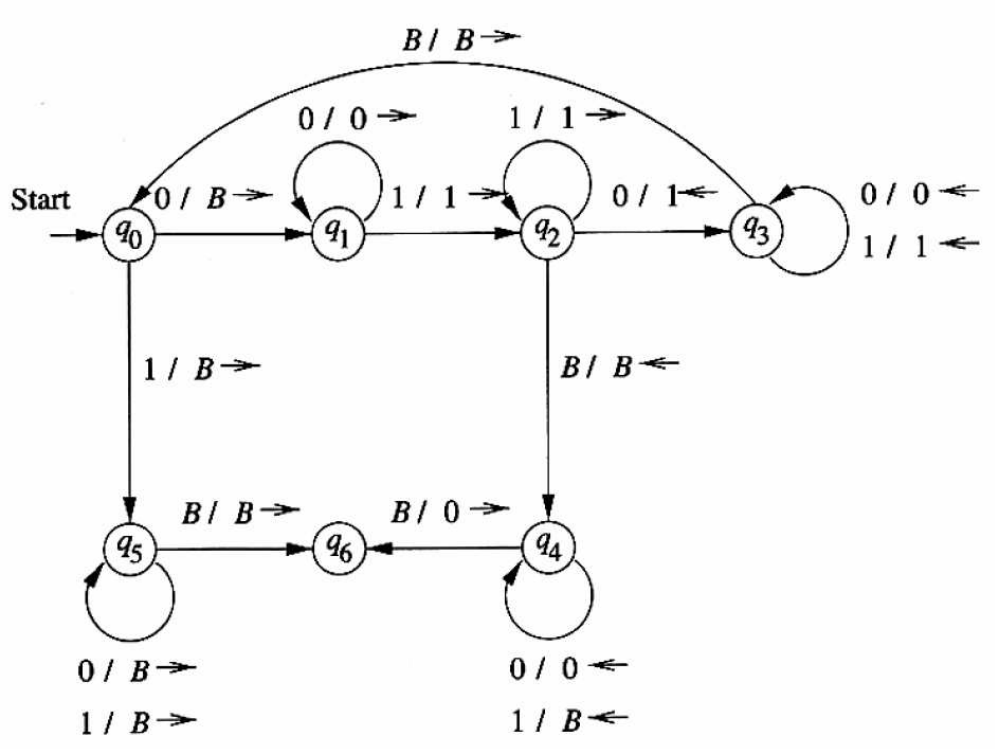


Ejemplo: Diseñar una TM que calcula la función $\dot{-}$ llamada *monus* o substracción propia, que se define como: $m \dot{-} n = max(m - n, 0)$.

La siguiente tabla y diagrama de transición lo definen:

	Símbolo		
Estado	0	1	B
q_0	(q_1, B, R)	(q_5, B, R)	$-$
q_1	$(q_1, 0, R)$	$(q_2, 1, R)$	$-$
q_2	$(q_3, 1, L)$	$(q_2, 1, R)$	(q_4, B, L)
q_3	$(q_3, 0, L)$	$(q_3, 1, L)$	(q_0, B, R)
q_4	$(q_4, 0, L)$	(q_4, B, L)	$(q_6, 0, R)$
q_5	(q_5, B, R)	(q_5, B, R)	(q_6, B, R)
q_6	$-$	$-$	$-$

Diagrama de transición:



8.2.3 Lenguaje de una TM

El lenguaje que aceptan las TMs es el conjunto de cadenas $w \in \Sigma*$ tales que $q_0 w \overset{*}{\vdash} \alpha p \beta$ para un estado p en F y cualesquiera cadenas en la cinta α y β.

Los lenguajes que aceptan las TMs se llama lenguajes *recursivamente enumerables* o lenguajes RE.

Halting: otra forma de aceptar cadenas usado normalmente en las TMs es aceptar por paro (*halting*). Decimos que una TM se para (*halts*) si entra a un estado q leyendo el símbolo de la cinta X y no existe ningún movimiento, i.e., $\delta(q, X)$ no está definido.

Por ejemplo la máquina de Turing que calcula $\dot{-}$, se para (*halts*) con

todas las cadenas de 0's y 1's ya que eventualmente alcanza el estado q_6.

Asumimos que una TM siempre se para en un estado de aceptación.

Ejercicios:

Extensiones a TMs:

Una TM es tan poderosa como una computadora convencional.

Se pueden realizar varias extensiones a una TM que permiten hacer especificaciones más simples, aunque no aumentan su expresividad (reconocen los mismos lenguajes).

8.2.4 Almacenar información en un estado

Podemos usar el control finito de una TM no sólo para indicar la posición actual, sino también para almacenar una cantidad finita de datos.

No se extiende el modelo, lo que hacemos es que pensamos en el estado como una tupla.

Por ejemplo, si queremos aceptar $01^* + 10^*$ podemos almacenar el primer elemento que se lee y asegurarnos que ese elemento no aparezca en el resto de la cadena.

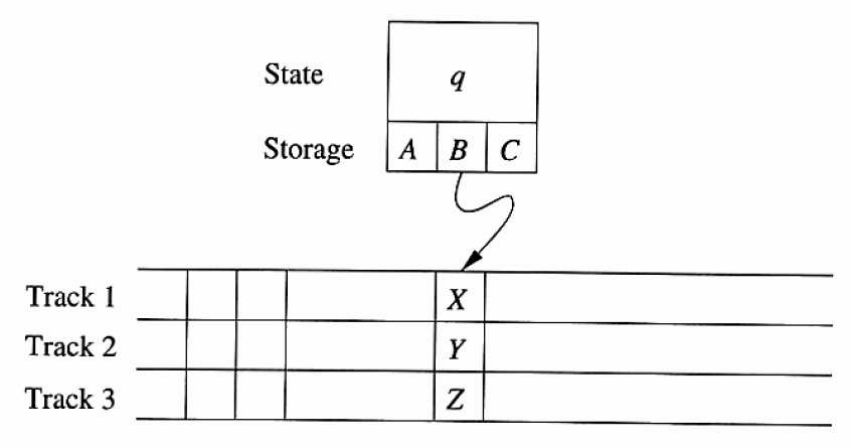
El $M = (Q, \{0,1\}, \{0,1,B\}, \delta, [q_0, B], \{[q_1, B]\})$

Su función de transición es:
$\delta([q_0, B], a) = ([q_1, a], a, R)$ para $a = 0$ o $a = 1$.
$\delta([q_1, a], ca) = ([q_1, a], ca, R)$ donde ca es el complento de a. Si $a = 0$ entonces $ca = 1$, y viceversa.
$\delta([q_1, a], B) = ([q_1, B], B, R)$ para $a = 0$ o $a = 1$, llegando al estado de aceptación: $[q_1, B]$.

8.2.5 Tracks múltiples

Otro truco es pensar que tenemos varios caminos paralelos. Cada una tiene un símbolo y el alfabeto de la cinta de la TM es una tupla con un componente por cada camino.

El siguiente dibujo ilustra una TM con almacenamiento de información en el control y con caminos múltiples:



Un uso común de esto es usar un camino para guardar información y otro para guardar una marca. Por ejemplo, si queremos reconocer: $L_{wcw} = \{wcw | w \in (0+1)^+\}$

$M = (Q, \Sigma, \Gamma, \delta, [q_1, B], [B, B], \{[q_9, B]\})$

- Q: el conjunto de estados en $\{q_0, q_1, \ldots, q_9\} \times \{0, 1, B\}$, osea pares que tienen un estado de control (q_i) y un componente de dato ($0, 1$ o *blank*).
- Γ: los símbolos de la cinta son $\{B, *\} \times \{0, 1, c, B\}$, donde el primer componente es vacío (*blank*) o $*$ (para marcar símbolos del primer y segundo grupo de 0's y 1's).
- Σ: los símbolos de entrada son $[B, 0]$ y $[B, 1]$.
- δ: la función de transición es (donde a y b pueden ser 0 o 1):
  1. $\delta([q_1, B], [B, a]) = ([q_2, a], [*, a], R)$

2. $\delta([q_2, a], [B, b]) = ([q_2, a], [B, b], R)$
3. $\delta([q_2, a], [B, c]) = ([q_3, a], [B, c], R)$
4. $\delta([q_3, a], [*, b]) = ([q_3, a], [*, b], R)$
5. $\delta([q_3, a], [B, a]) = ([q_4, B], [*, a], L)$
6. $\delta([q_4, B], [*, a]) = ([q_4, B], [*, a], L)$
7. $\delta([q_4, B], [B, c]) = ([q_5, B], [B, c], L)$
8. $\delta([q_5, B], [B, a]) = ([q_6, B], [B, a], L)$
9. $\delta([q_6, B], [B, a]) = ([q_6, B], [B, a], L)$
10. $\delta([q_6, B], [*, a]) = ([q_1, B], [*, a], R)$
11. $\delta([q_5, B], [*, a]) = ([q_7, B], [*, a], R)$
12. $\delta([q_7, B], [B, c]) = ([q_8, B], [B, c], R)$
13. $\delta([q_8, B], [*, a]) = ([q_8, B], [*, a], R)$
14. $\delta([q_8, B], [B, B]) = ([q_9, B], [B, B], R)$

8.2.6 Subrutinas

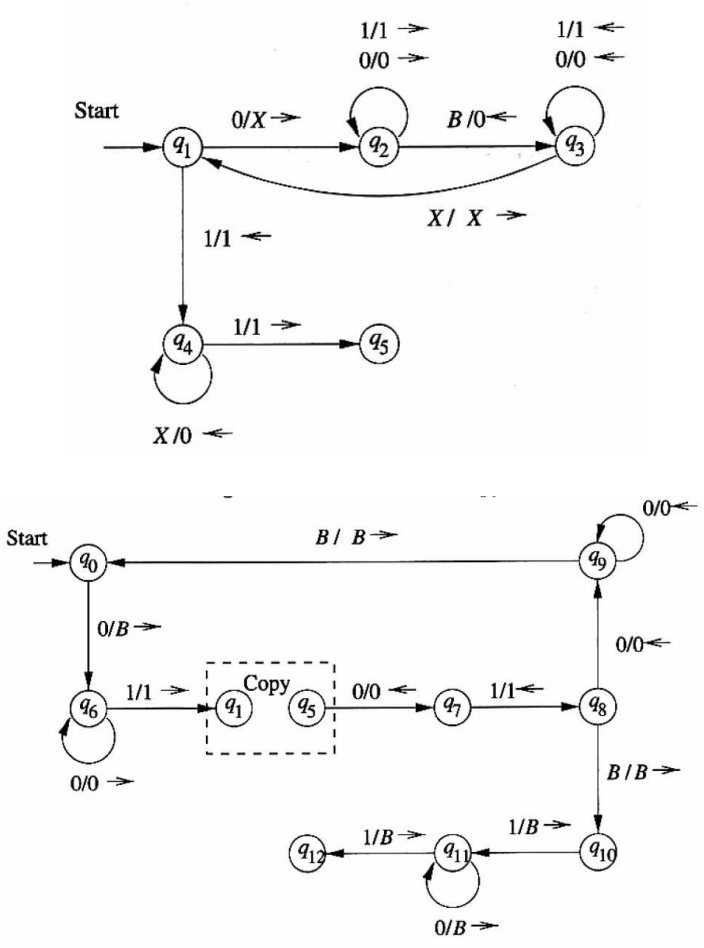
Una subrutina de una TM es un conjunto de estados que realiza algún proceso útil. Su llamado ocurre cuando se hace una transición a su estado inicial. Si se requiere "llamar" varias veces desde diferentes estados, se tienen que hacer varias copias con estados diferentes.

Por ejemplo, la siguiente TM implementa la multiplicación, y empieza con una cadena 0^m10^n1 en su cinta y termina con la cadena 0^{mn} en la cinta.

El núcleo es una subrutina llamada *Copia* que copia un bloque de 0's al final. Copia convierte una ID de forma $0^{m-k}1_1^q0^n10^{(k-1)n}$ a la siguiente ID $0^{m-k}1q_50^n10^{kn}$.

Las siguientes figuras ilustran la subrutina *Copia* y la TM completa para

multiplicación:



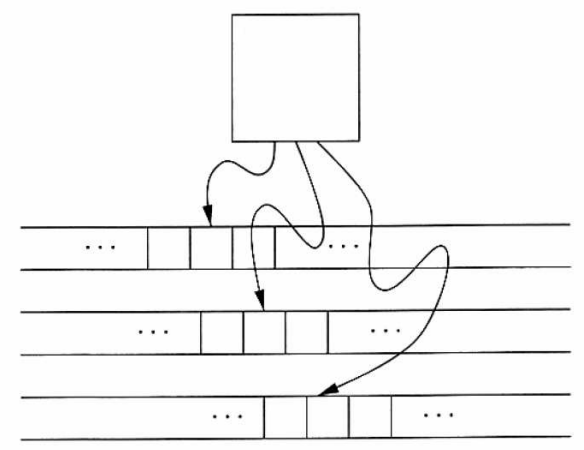
8.2.7 TM con múltiples cintas

Tiene un número finito de cintas. En el estado inicial los símbolos de entrada son colocados en la primera cinta y la cabeza del resto de las cintas en un B arbitrario.

En un movimiento, el control entra a un nuevo estado y en cada cinta se escribe un símbolo en la celda que está leyendo, cada cabeza de cada cinta hace un movimiento que puede ser a la izquierda, derecha o quedarse donde

está.

Las TM con cintas múltiples aceptan entrando a un estado de aceptación.



Se puede demostrar que todo lenguaje aceptado por una TM con múltiples cintas es recursivamente enumerable.

Básicamente para una TM con k cintas se simula una TM de una cinta con $2k$ caminos, donde la mitad de los caminos guardan el contenido de las cintas y la otra mitad guardan una marca que indica la posición de las cabezas de cada cinta.

8.2.8 TM no determinista

En una TM no determinista (NTM) la función de transición es ahora un conjunto de tripletas y puede seleccionar cualquier elemento de ese conjunto en cada transición. $\delta(q, X) = \{(q_1, Y_1, D_1), (q_2, Y_2, D_2), \ldots, (q_k, Y_k, D_k)\}$

Una NTM acepta una cadena w si existe una secuencia de selecciones de movimientos que nos llevan a un ID con un estado de aceptación.

De nuevo se puede probar que para cada NTM existe una TM (determinista).

Básicamente se siguen las "m" posibles opciones en cada movimiento

siguiendo un esquema tipo *breadth-first*.

Ejercicios:

8.3 Máquinas de Turing restringidas

Se pueden imponer ciertas restricciones a la TM y de todos modos mostrar que aceptan el mismo lenguaje.

8.3.1 TM con cinta semi-infinita

La cinta es infinita en un sentido, la cadena de entrada se coloca al principio de la cinta la cual no continua a la izquierda. También se incluye la restricción de no poder escribir B (*blank*) en la cinta.

Se puede demostrar que un TM normal se puede simular con una TM semi-infinita usando dos caminos, uno que simula la cinta a la izquierda de la cadena de entrada y otro que simula la otra parte de la cinta.

X_0	X_1	X_2	$\ldots$
$*$	X_{-1}	X_{-2}	$\ldots$

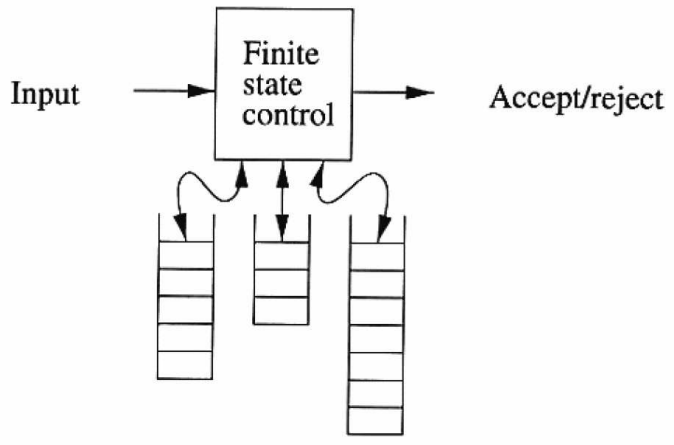
Los estados en la TM semi-infinita son los que tiene la TM original junto con U o L para representar arriba (U) o abajo (L), además de un par de estados para preparar la TM semi-infinita.

Las transiciones en la cinta de arriba son iguales a las de la TM original, y las de abajo son contrarias (si la TM original se mueve a la derecha, la TM semi-infinita inferior se mueve a la izquierda y al revés, si se mueve a la izquierda la TM original la otra se mueve a la derecha).

Sólo se tiene que tener cuidado en las situaciones en donde se cambia de un camino al otro.

8.3.2 Máquinas multistack

Podemos pensar en una generalización de los PDAs cuando consideramos varios *stacks*.



Una máquina de *k-stacks* es un PDA determinista con k *stacks*. Un movimiento en esta máquina, cambia el estado y reemplaza el símbolo de arriba de cada *stack* con una cadena normalmente diferente para cada *stack*.

La transición sería: $\delta(q, a, X_1, X_2, \ldots, X_k) = (p, \gamma_1, \gamma_2, \ldots, \gamma_k)$. Donde en estado q con X_i hasta arriba de cada uno de los $i = 1, 2, \ldots, k$ *stacks*, se consume el símbolo a y se reemplaza cada X_i por γ_i.

Se puede demostrar que un PDA con 2 *stacks* puede simular una TM.

En la demostración se asume que al final de la cadena de entrada existe un símbolo especial que no es parte de la entrada.

Lo primero que se hace es que se copia la cadena al primer *stack*, se hace *pop* de este *stack* y *push* en el segundo *stack*, con esto el primer elemento de la cadena de entrada está hasta arriba del segundo *stack*, y luego se empiezan a simular las transiciones de estados.

El primer *stack* vacío nos representa todos los *blanks* a la izquierda de la cadena de entrada y en general lo que está a la izquierda de donde apunta la cabeza de la cinta de la TM.

Si TM reemplaza X por Y y se mueve a la derecha, PDA introduce (*pushes*) Y en el primer *stack* y saca (*pops*) X del segundo 'emphstack.

Si TM reemplaza X por Y y se mueve a la izquierda, PDA saca (*pops*) el primer elemento del primer *stack* (digamos Z) y reemplaza X por ZY en el segundo *stack*.

8.3.3 Máquinas contadoras (*counter machines*)

Una máquina contadora tiene la misma estructura que una máquina *multi-stack*, sólo que en lugar de *stacks* tiene contadores.

Un contador tiene algún entero positivo y sólo puede distinguir entre 0 (cero) o un contador diferente de cero.

En cada movimiento, cambia de estado y suma o resta 1 del contador (que no puede volverse negativo).

También podemos verlo como una máquina *multistack* restringida que tiene dos símbolos de stack Z_0 (marca el fondo) y X. El tener $X_i Z_0$ nos identifica al contador i.

Se puede demostrar que los lenguajes aceptados por una máquina de un contador son los CFL.

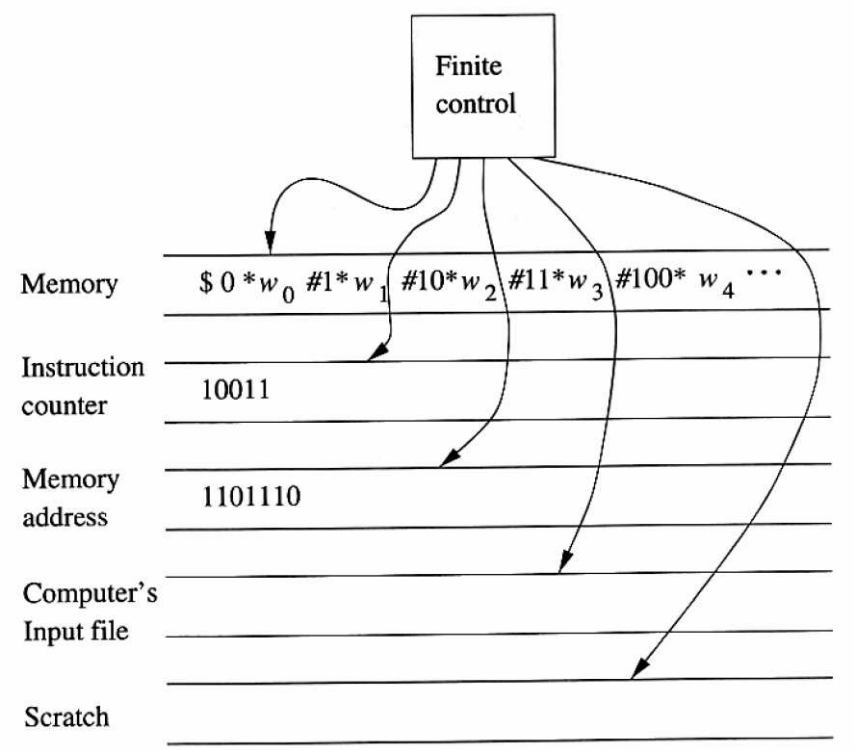
Se puede demostrar que cualquier lenguaje recursivamente enumerable puede ser aceptado por una máquina de dos contadores.

8.4 Máquinas de Turing y Computadoras

Una computadora puede simular una máquina de Turing. Aunque con un número muy grande de símbolos y cadenas en principio infinitas, se podrían tener problemas de memoria, se puede codificar la TM (TM universal) y simular en una computadora convencional sin problemas de memoria.

Una TM puede simular una computadora usando varias cintas para tomar

en cuenta los diferentes procesos que se realizan en una computadora (para representar la memoria de la computadora, la siguiente instrucción a realizar, si se requiere copiar cierto contenido de la memoria, cambiarlo, etc.).



Se puede demostrar que si una computadora tiene sólo instrucciones que incrementan la longitud del tamaño de las palabras en 1 y las instrucciones de tamaño w se pueden realizar en una TM con cintas múltiples en $O(k^2)$ pasos, entonces una TM parecida a la mostrada arriba pueden simular n pasos de la computadora en $O(n^3)$ pasos.