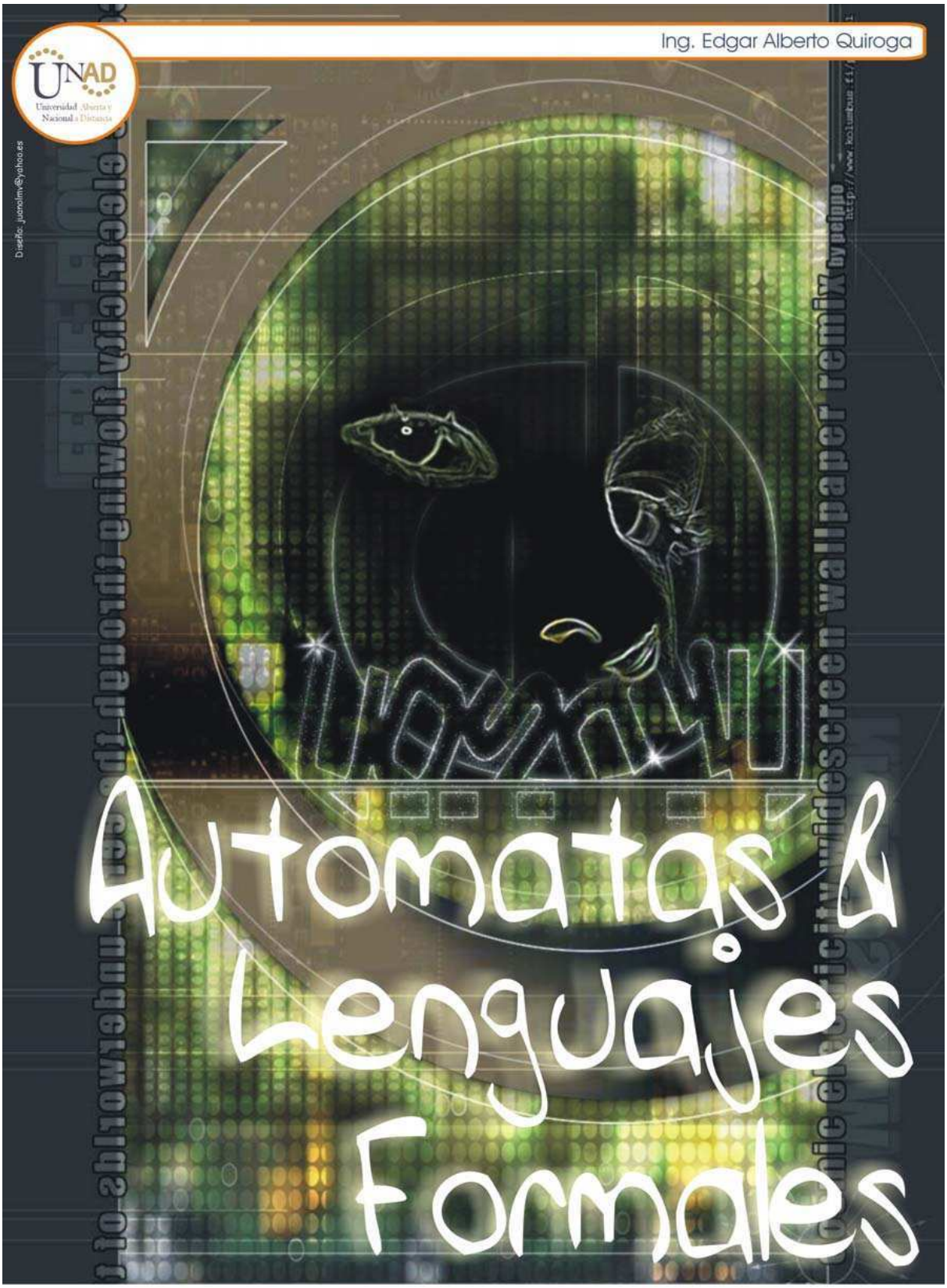
Ing. Edgar Alberto Quiroga

UNAD
Universidad Abierta y Nacional a Distancia

Diseño: juanolmv@yahoo.es



Automatas & Lenguajes Formales

MODULO

AUTÓMATAS Y LENGUAJES FORMALES

Edgar Alberto Quiroga Rojas

UNIVERSIDAD NACIONAL ABIERTA Y A DISTANCIA – UNAD
FACULTAD DE CIENCIAS BÁSICAS E INGENIERÍA
PROGRAMA INGENIERIA DE SISTEMAS
BOGOTÁ D.C., 2008

MODULO
AUTÓMATAS Y LENGUAJES FORMALES

ISBN

Autor: Edgar Alberto Quiroga Rojas
Diseño Portada Juan Olegario Monroy V.

2008
Centro Nacional de Medios para el Aprendizaje

TABLA DE CONTENIDO.

		35. Codificación de Máquinas de Turing

INTRODUCCIÓN

Autómatas y lenguajes formales es un curso de carácter teórico, que se inscribe en el campo de formación profesional básico del Programa de Ingeniería de Sistemas con un valor académico de tres créditos.

El estudiante en el desarrollo de este curso demuestra la asimilación de los conceptos y mecanismos fundamentales para la definición de lenguajes (expresiones regulares, gramáticas independientes del contexto y gramáticas generales), los tres tipos de máquinas correspondientes para su reconocimiento (autómatas finitos, autómatas a pila y maquinas de Turing) y las propiedades fundamentales de las familias de lenguajes por ellos definidas, también realiza el estudio de las condiciones necesarias para que un lenguaje sea de un tipo determinado.

El curso es principalmente teórico, jugando un papel secundario la implementación de algoritmos. Al final del curso el estudiante debe demostrar la asimilación de los conceptos fundamentales mediante la resolución de problemas acerca de los mismos, así como la realización de algunas prácticas en el computador.

Este curso toma como base el avance de los lenguajes de programación de alto y bajo nivel para propiciar la distinción entre lenguajes formales con reglas sintácticas y semánticas rígidas, concretas y bien definidas, de los lenguajes naturales como el inglés o el español, donde la sintaxis y la semántica no se pueden controlar fácilmente. Los intentos de formalizar los lenguajes naturales, lleva a la construcción de gramáticas, como una forma de describir estos lenguajes, utilizando para ello reglas de producción para construir las frases del lenguaje. Se puede entonces caracterizar un lenguaje mediante las reglas de una gramática adecuada.

Los temas sobre autómatas, computabilidad, e incluso la complejidad algorítmica fueron incorporándose al currículo de ciencias de la computación de diferentes universidades desde la década de los 60, esta incorporación puso de manifiesto que las ciencias de la computación habían usado gran cantidad de ideas de muy diferentes campos para su desarrollo, y que la investigación sobre aspectos básicos podía cooperar y aumentar los avances de la computación.

Como elemento determinante en el curso es importante que se tengan en cuenta los conceptos matemáticos básicos de teoría de conjuntos, funciones, relaciones y principios fundamentales de la lógica, ya que éstos temas no son tratados como temáticas en el módulo, pero que tienen gran importancia en el curso.

INTENCIONALIDADES FORMATIVAS

PROPÓSITOS

Facilitar la apropiación de conocimientos para que el estudiante pueda adquirir los conceptos básicos de la teoría de los lenguajes formales y la relación que existe con la teoría de autómatas.

Lograr que el estudiante entienda el alto nivel de abstracción de las máquinas secuenciales y los autómatas y que conozca los elementos y las técnicas necesarias para la construcción de las fases iniciales de un compilador.

OBJETIVOS

Estudiar los conceptos fundamentales de la teoría de autómatas y lenguajes formales, para la descripción de ellos.

Conocer la correspondencia entre máquinas, gramáticas y lenguajes, los problemas en los que las teorías tienen aplicación o que han motivado su construcción.

COMPETENCIAS

El estudiante conoce la jerarquía de modelos de máquinas computacionales y su funcionamiento, así como la jerarquía de las gramáticas formales y de los lenguajes correspondientes.

El estudiante conoce la correspondencia entre máquinas, gramáticas y lenguajes para poder construir compiladores.

El estudiante desarrolla la capacidad de abstracción y análisis teórico en relación con la teoría de lenguajes para adquirir herramientas para algunas asignaturas de la carrera.

METAS

Al terminar el curso el estudiante:

Debe demostrar la asimilación de los conceptos fundamentales mediante la resolución de problemas acerca de los mismos, y de la realización de algunas practicas con el apoyo del computador.

Desarrollará la capacidad de entender los problemas computacionales, y logrará una comprensión total de algunos tópicos de la ciencia de la computación; específicamente en modelos básicos de conmutabilidad y complejidad de problemas.

INTRODUCCIÓN

Los lenguajes pueden describirse como elementos que se generan, como cadenas a partir de cadenas sencillas, con el uso de operaciones de cadenas o el desarrollo del lenguaje mismo, que se puede generar con otros lenguajes más sencillos mediante operaciones de conjuntos.

Los Lenguajes más sencillos son los considerados lenguajes regulares, es decir, los que se pueden generar a partir de lenguajes de un elemento con la aplicación de ciertas operaciones estandar realizadas un número finito de veces.

Estos son pues los lenguajes que pueden reconocer los dispositivos llamados Autómatas finitos (AF) que son máquinas de cómputo con memoria muy restringida. En esta unidad se considera como segundo aspecto la idea de que un lenguaje no sea regular, ademas de proporcionar un modelo sencillo de computación que se puede generalizar en las unidades siguientes.

Con las caracterizaciones anteriores y otras de los lenguajes regulares se obtienen y estudian algoritmos para traducir una descripción de un lenguaje a otra descripción de un tipo distinto; se acumula experiencia en el uso de métodos formales para describir lenguajes y se intenta responder a preguntas acerca de ellos, son preguntas y ejercicios sencillos con sus respuestas y que permiten determinar la utilidad de los lenguajes regulares en aplicaciones del mundo real.

OBJETIVO GENERAL

Reconocer los lenguajes regulares, autómatas finitos y su aplicación.

OBJETIVOS ESPECIFICOS

Estudiar la aplicación de los lenguajes regulares y los autómatas finitos.

Adquirir las habilidades necesarias para desarrollar autómatas y máquinas que reconozcan lenguajes o computen funciones.

Distinguir los diferentes tipos de lenguajes formales existentes.

LECCION 1. - INTRODUCCION E HISTORIA[1]

Hoy en día parece que no existe ningún límite a lo que un computador puede llegar a hacer, y da la impresión de que cada vez se pueden resolver nuevos y más difíciles problemas.

El hombre ha tratado de buscar procedimientos y máquinas que le faciliten la realización de cálculos (aritméticos primero, y otros más complejos posteriormente).

El avance tecnológico para representar datos y/o información por un lado, y el diseño de nuevas formas de manejarlos, propician el desarrollo de dispositivos y máquinas de calcular.

Un aspecto importante en el desarrollo de los computadores, es sin duda, su aplicación para resolver problemas científicos y empresariales. Esta aplicación hubiese resultado muy difícil sin la utilización de procedimientos que permiten resolver estos problemas mediante una sucesión de pasos claros, concretos y sencillos, es decir algoritmos. El avance de las matemáticas permite la utilización de nuevas metodologías para la representación y manejo de la información.

Por otro lado, aparece el intento de los matemáticos y científicos para obtener un procedimiento general para resolver cualquier problema (matemático) claramente formulado. Es lo que podríamos llamar **El problema de la computación teórica**. El avance de la tecnología y de las matemáticas, y más en concreto de la teoría de conjuntos y de la lógica, permiten plantearse aspectos de la computación en 3 caminos.

a) **Computación teórica**. Autómatas, Funciones Recursivas, ...
b) **Computadores digitales**. Nuevas tecnologías, nuevos lenguajes, ...
c) **Intentos de modelizar el cerebro biológico**
    1. Redes Neuronales (intentan modelizar el "procesador")
    2. Conjuntos y Lógica Difusa (representar y manejar la información)

Uno de los principales factores determinantes de la profunda revolución experimentada en el ámbito de la ciencia, la técnica y la cultura de nuestros días es el desarrollo de la informática. La palabra 'informática' (**Infor**mación auto**mática**), es un nombre colectivo que designa un vasto conjunto de teorías y

[1] MORAL CALLEJÓN Serafín Teoría de autómatas y lenguajes formales, En: http://decsai.ugr.es/~smc/docencia/mci/automata.pdf
NAVARRETE SÁNCHEZ, Isabel y otros Teoría de autómatas y lenguajes formales En: http://perseo.dif.um.es/%7Eroque/talf/Material/apuntes.pdf

técnicas científicas desde la matemática abstracta hasta la ingeniería y la gestión administrativa cuyo objeto es el diseño y el uso de los computadores. Pero el núcleo teórico más sólido y fundamental de todo ese conjunto de doctrinas y prácticas es la llamada 'Teoría de la Computabilidad', formalmente elaborada en los años 30 y 40 gracias a los descubrimientos de lógicos matemáticos como Gödel, Turing, Post, Church, y Kleene, aunque sus orígenes más remotos datan de antiguo, con el planteamiento de la cuestión de saber si, al cabo de cierto esfuerzo, el hombre podría llegar a un extremo en la investigación en que, eventualmente, toda clase de problemas pudiera ser atacado por un procedimiento general de forma que no requiriera el más leve esfuerzo de imaginación creadora para llevarlo a cabo. Si todo queda determinado así en detalle, entonces sería obviamente posible abandonar la ejecución del método a una máquina, máxime si la máquina en cuestión es totalmente automática. Esta idea, ambiciosa sin duda, ha influido poderosamente en diferentes épocas el desarrollo de la ciencia.

El propósito inicial es hacer precisa la noción intuitiva de función calculable; esto es, una función cuyos valores pueden ser calculados de forma automática o efectiva mediante un algoritmo, y construir modelos teóricos para ello (de computación). Así podemos obtener una comprensión más clara de esta idea intuitiva; y solo de esta forma podemos explorar matemáticamente el concepto de computabilidad y los conceptos relacionados con ella, tales como decibilidad, etc...

La teoría de la computabilidad puede caracterizarse, desde el punto de vista de las Ciencias de la Computación, como la búsqueda de respuestas para las siguientes preguntas:

1)¿Qué pueden hacer los computadores (sin restricciones de ningún tipo)?
2) ¿Cuales son las limitaciones inherentes a los métodos automáticos de cálculo?.

El primer paso en la búsqueda de las respuestas a estas preguntas está en el estudio de los modelos de computación. **Los comienzos de la Teoría. La Tesis de Church-Turing**

Los modelos abstractos de computación tienen su origen en los años 30, bastante antes de que existieran los computadores modernos, en el trabajo de los lógicos Church, Gödel, Kleene, Post, y Turing. Estos primeros trabajos han tenido una profunda influencia no solo en el desarrollo teórico de las Ciencias de la Computación, sino que muchos aspectos de la práctica de la computación que son ahora lugar común de los informáticos, fueron presagiados por ellos; incluyendo la existencia de computadores de propósito general, la posibilidad de interpretar programas, la dualidad entre software y hardware, y la representación de lenguajes por estructuras formales basados en reglas de producción.

El punto de partida de estos primeros trabajos fueron las cuestiones fundamentales que D. Hilbert formuló en 1928, durante el transcurso de un congreso internacional:

1.-¿Son completas las matemáticas, en el sentido de que pueda probarse o no cada aseveración matemática?
2.-¿Son las matemáticas consistentes, en el sentido de que no pueda probarse simultaneamente una aseveración y su negación?
3.-¿Son las matemáticas decidibles, en el sentido de que exista un método definido que se pueda aplicar a cualquier aseveración matemática, y que determine si dicha aseveración es cierta?.

La meta de Hilbert era crear un sistema matemático formal completo, consistente", en el que todas las aseveraciones pudieran plantearse con precisión. Su idea era encontrar un algoritmo que determinara la verdad o falsedad de cualquier proposición en el sistema formal. A este problema le llamó el ‘Entscheidungs problem’.

Por desgracia para Hilbert, en la década de 1930 se produjeron una serie de investigaciones que mostraron que esto no era posible. Las primeras noticias en contra surgen en 1931 con K. Gödel y su Teorema de Incompletitud: "Todo sistema de primer orden consistente que contenga los teoremas de la aritmética y cuyo conjunto de (números de Gödel de) axiomas sea recursivo no es completo."

Como consecuencia no será posible encontrar el sistema formal deseado por Hilbert en el marco de la lógica de primer orden, a no ser que se tome un conjunto no recursivo de axiomas, hecho que escapaba a la mente de los matemáticos. Una versión posterior y más general del teorema de Gödel elimina la posibilidad de considerar sistemas deductivos más potentes que los sistemas de primer orden, demostrando que no pueden ser consistentes y completos a la vez.

Un aspecto a destacar dentro del teorema de incompletitud de Gödel, fué la idea de codificación. Se indica un método (numeración de Gödel) mediante el cual se asigna un número de código (entero positivo) a cada fórmula bien formada del sistema (fbf) y a cada sucesión finita de fórmulas bien formadas, de tal modo que la fbf o sucesión finita de fbf se recupera fácilmente a partir de su número de código. A través de este código, los enunciados referentes a enteros positivos, pueden considerarse como enunciados referentes a números de código de expresiones, o incluso referentes a las propias expresiones. Esta misma idea fué posteriormente utilizada para codificar algoritmos como enteros positivos, y así poder considerar un algoritmo, cuyas entradas fuesen enteros positivos, como un algoritmo cuyas entradas fuesen algoritmos.

El siguiente paso importante lo constituye la aparición casi simultánea en 1936 de varias caracterizaciones independientes de la noción de calculabilidad efectiva, en los trabajos de Church, Kleene, Turing y Post. Los tres primeros mostraban problemas que eran efectivamente indecidibles; Church y Turing probaron además que el Entscheidungsproblem era un problema indecidible.

Church propuso la noción de función γ-definible como función efectivamente calculable. La demostración de teoremas se convierte en una transformación de

una cadena de símbolos en otra, en cálculo lambda, según un conjunto de reglas formales. Este sistema resultó ser inconsistente, pero la capacidad para expresar-calcular funciones numéricas como términos del sistema llamó pronto la atención de él y sus colaboradores.

Gödel habia recogido la idea de Herbrand de que una función f podría definirse por un conjunto de ecuaciones entre términos que incluian a la función f y a símbolos para funciones previamente definidas, y precisó esta idea requiriendo que cada valor de f se obtenga de las ecuaciones por sustitución de las variables por números y los términos libres de variables por los valores que ya se habian probado que designaban. Esto define la clase de 'las funciones recursivas de Herbrand-Gödel'.

En 1936, Church hace un esquema de la demostración de la equivalencia entre las funciones γ-definibles y las funciones recursivas de Herbrand-Gödel (esta equivalencia también había sido probada por Kleene); y aventura que estas iban a ser las únicas funciones calculables por medio de un algoritmo a través de la tesis que lleva su nombre, y utilizando la noción de función γ-definible, dió ejemplos de problemas de decisión irresolubles, y demostró que el Entscheidungsproblem era uno de esos problemas.

Por otra parte Kleene, pocos meses despues, demuestra formalmente la equivalencia entre funciones γ-definible y funciones recursivas de Herbrand-Gödel, y dá ejemplos de problemas irresolubles utilizando la noción de función recursiva.

La tercera noción de función calculable proviene del matemático inglés A. Turing, quién argumentó que la tercera cuestión de Hilbert (el Entscheidungsproblem) podía atacarse con la ayuda de una máquina, al menos con el concepto abstracto de máquina.

Turing señaló que había tenido éxito en caracterizar de un modo matemáticamente preciso, por medio de sus máquinas, la clase de las funciones calculables mediante un algoritmo, lo que se conoce hoy como *Tesis de Turing*.

Aunque no se puede dar ninguna prueba formal de que una máquina pueda tener esa propiedad, Turing dió un elevado número de argumentos a su favor, en base a lo cual presentó la tesis como un teorema demostrado. Además, utilizó su concepto de máquina para demostrar que existen funciones que no son calculables por un método definido y en particular, que el Entscheidungsproblem era uno de esos problemas.

Cuando Turing conoció los trabajos de Church-Kleene, demostró que los conceptos de función γ-definible y función calculable por medio de una máquina de Turing coinciden. Naturalmente a la luz de esto la Tesis de Turing resulta ser equivalente a la de Church.

Finalmente, cabe reseñar el trabajo de E. Post. Este estaba interesado en marcar la frontera entre lo que se puede hacer en matemáticas simplemente por procedimientos formales y lo que depende de la comprensión y el entendimiento. De esta forma, Post formula un modelo de procedimiento efectivo a través de los llamados sistemas deductivos normales. Estos son sistemas puramente formales en los que puede 'deducirse' sucesiones finitas de símbolos como consecuencia de otras sucesiones finitas de símbolos por medio de un tipo normalizado de reglas y a partir de un conjunto de axiomas. Así pues, dada una sucesión finita de símbolos como entrada, las reglas permiten convertirla en una sucesión finita de salida. En su artículo, Post demostró resultados de incompletitud e indecibilidad en estos sistemas.

Los resultados hasta ahora citados, se refieren a funciones totales. La existencia de algoritmos que con determinadas entradas nunca terminan, condujo de forma natural a considerar funciones parciales. Kleene fué el primero en hacer tal consideración en 1938. El estudio de estas funciones ha mostrado la posibilidad de generalizar todos los resultados anteriores a funciones parciales. Por otro lado, el estudio de las funciones parciales calculables ha resultado esencial para el posterior desarrollo de la materia.

Posteriormente, se demostró la equivalencia entre lo que se podía calcular mediante una máquina de Turing y lo que se podía calcular mediante un sistema formal en general.

A la vista de estos resultados, la Tesis de Church-Turing es aceptada como un axioma en la teoría de la computación, y ha servido como punto de partida en la investigación de los problemas que se pueden resolver mediante un algoritmo.

1.1.1. Problemas no computables

Usando la codificación de Gödel, se demostró que era posible construir una máquina de propósito general, es decir, capaz de resolver cualquier problema que se pudiese resolver mediante un algoritmo. Dicha máquina tendría como entrada el entero que codificaría el algoritmo solución del problema y la propia entrada del problema, de tal forma, que la máquina aplicaría el algoritmo codificado a la entrada del problema. Esta hipotética máquina puede considerarse como el padre de los actuales computadores de proposito general.

Una de las cuestiones más estudiadas en la teoría de la computabilidad ha sido la posibilidad de construir algoritmos que nos determinen si un determinado algoritmo posee o no una determinada propiedad. Así, sería interesante responder de forma automática a cuestiones como:

- ¿Calculan los algoritmos A y B la misma función? (Problema de la equivalencia)
- ¿Parará el algoritmo A para una de sus entradas? (Problema de la parada)
- ¿Parará el algoritmo A para todas sus entradas? (Problema de la totalidad)

- ¿Calcula el algoritmo A la función f? (Problema de la verificación?)
- etc . . .

En un principio se fueron obteniendo demostraciones individuales de la no computabilidad de cada una de estas cuestiones, de forma que se tenía la sensación de que casi cualquier pregunta interesante acerca de algoritmos era no computable. A pesar de esto, y como consecuencia de la existencia de un programa universal hay otras muchas cuestiones interesantes que se han demostrado computables.

El identificar los problemas que son computables y los que no lo son tiene un considerable interés, pues indica el alcance y los límites de la computabilidad, y así demuestra los límites teóricos de los computadores. Además de las cuestiones sobre algoritmos, se han encontrado numerosos problemas menos "generales"que han resultado ser no computables. Como ejemplo se cita: Décimo problema de Hilbert. Una ecuación diofántica es la ecuación de los ceros enteros de un polinomio con coeficientes enteros. Se pregunta si hay un procedimiento efectivo que determine si una ecuación diofántica tiene o no solución.

Por otro lado, son muchos los problemas interesantes que se han demostrado computables. Todas las funciones construidas por recursividad primitiva o minimalización a partir de funciones calculables resultan ser calculables como consecuencia de los trabajos de Church y Turing. Pero además, otras funciones más complejamente definidas también son computables. Como ejemplo más interesante de aplicación de este tipo de recursión tenemos la función de Ackermann

$$\psi:$$

$$\psi(0, y) = y + 1;$$

$$\psi(x + 1, 0) = \psi(x, 1);$$

$$\psi(x + 1, y + 1) = \psi(x, \psi(x + 1, y)).$$

LECCION 2. - DIFERENTES MODELOS DE COMPUTACION[2]

Consideraremos las Ciencias de la Computación como un cuerpo de conocimiento cuyo principal objetivo es la resolución de problemas por medio de un computador.
Se pueden citar las siguientes definiciones:

a) La ACM (Asociation Computing Machinering): 'la disciplina Ciencias de la Computación es el estudio sistemático de los procesos algorítmicos que

[2] MORAL CALLEJÓN Serafín Teoría de autómatas y lenguajes formales, En: http://decsai.ugr.es/~smc/docencia/mci/automata.pdf
NAVARRETE SÁNCHEZ, Isabel y otros Teoría de autómatas y lenguajes formales En: http://perseo.dif.um.es/%7Eroque/talf/Material/apuntes.pdf

describen y transforman información: teoría, análisis, diseño, eficiencia, implementación, y aplicación.'

b) Norman E. Gibbs y Allen B. Tucker (1986) indican que: 'no debemos entender que el objetivo de las Ciencias de la Computación sea la construcción de programas sino el estudio sistemático de los algoritmos y estructura de datos, específicamente de sus propiedades formales'.

Para ser más concretos (A. Berztiss 1987), se consideran las Ciencias de la computación, como un cuerpo de conocimiento cuyo objetivo es obtener respuestas para las siguientes cuestiones:

A) ¿Qué problemas se pueden resolver mediante un computador?
B) ¿Cómo puede construirse un programa para resolver un problema?
C) ¿Resuelve realmente nuestro programa el problema?
D) ¿Cuanto tiempo y espacio consume nuestro problema?

Al Analizar en profundidad los 4 puntos anteriores se llega a descubrir explícitamente los diferentes contenidos abarcados por las Ciencias de la Computación.

El planteamiento de la primera cuestión conduce a precisar el concepto de problema y de lo que un computador es capaz de realizar.

Durante muchos años se creyó que si un problema podía enunciarse de manera precisa, entonces con suficiente esfuerzo y tiempo sería posible encontrar un 'algoritmo' o método para encontrar una solución (o tal vez podría proporcionarse una prueba de que tal solución no existe). En otras palabras, se creía que no había problema que fuera tan intrínsecamente difícil que en principio nunca pudiera resolverse. Uno de los grandes promotores de esta creencia fué el matemático David Hilbert (1862 - 1943), quien en un congreso mundial afirmó:

"Todo problema matemático bien definido debe ser necesariamente susceptible de un planteamiento exacto, ya sea en forma de una respuesta real a la pregunta planteada o debido a la constatación de la imposibilidad de resolverlo, a lo que se debería el necesario fallo de todos los intentos... "

El principal obstáculo que los matemáticos de principios de siglo encontraban al plantearse estas cuestiones era concretar con exactitud lo que significa la palabra algoritmo como sinónimo de método para encontrar una solución. La noción de algoritmo era intuitiva y no matemáticamente precisa. Las descripciones dadas por los primeros investigadores tomaron diferentes formas, que pueden clasificarse ampliamente del siguiente modo:

(a) máquinas computadoras abstractas (definidas de modo preciso),
(b) construcciones formales de procedimientos de cómputo, y
(c) construcciones formales productoras de clases de funciones.

Las dos primeras caracterizaciones se refieren a la propia noción de algoritmo (en principio no hay gran diferencia entre ambas). La última dá descripciones de la clase de funciones computables mediante un algoritmo.

Ejemplos de (a) son los Autómatas y las *máquinas de Turing*, (diseñadas por Turing en los años 30). Un ejemplo de (b) son los *sistemas de Thue*. Por último, las *funciones recursivas* constituyen el ejemplo clásico de (c).

El resultado crucial es que las diversas caracterizaciones de las funciones (parciales) computables mediante un algoritmo condujeron todas a una misma clase, a saber, la clase de las funciones parciales recursivas. Esto es algo suceptible de demostración, y que ha sido demostrado. Lo que no es suceptible de demostración es que la clase de las funciones parciales recursivas coincida con la clase de las funciones computables mediante un algoritmo. No obstante, a la luz de las evidencias a favor y de la falta de evidencias en contra, aceptamos la Tesis de Church que afirma la equivalencia de ambas clases.

Se clasifican los problemas según que siempre sea posible encontrar la solución por medio de un algoritmo (problemas computables) ó que no existan algoritmos que siempre produzcan una solución (problemas no computables).

Surge de modo inmediato la cuestión B) de como diseñar un programa (algoritmo especificado para poder ser ejecutado por un computador) que resuelva un problema dado. En la primera época del desarrollo informático los programas dependían intrínsecamente del computador utilizado, pues se expresaban en lenguaje máquina, directamente interpretable por el computador.

Surgió entonces la necesidad de idear otros mecanismos para construir y expresar los programas. El hilo conductor de tales mecanismos fué la abstracción: separar el programa del computador y acercarlo cada vez más al problema.

Los subprogramas empezaron ya a usarse a principios de los 50, dando lugar posteriormente al primer tipo de abstracción, la procedimental. A principios de los 60, se empezaron a entender los conceptos abstractos asociados a estructuras de datos básicas pero aún no se separaban los conceptos de las implementaciones. Con el nacimiento en esta época de los primeros lenguajes de alto nivel, Fortran p.ej., se llegó a la abstracción sintáctica, al abstraerse la semántica de las expresiones matemáticas y encapsular el acceso a ellas a través de la sintaxis propia del lenguaje. En cualquier caso con el desarrollo de estos lenguajes de alto nivel se solventaron los problemas de flexibilidad en la comunicación con el computador, y se empezaron a estudiar los algoritmos de forma independiente del computador concreto en que se probaran y del lenguaje concreto en que se expresaran.

Aparece la necesidad de traducir los programas escritos en lenguajes de alto nivel al lenguaje máquina, de forma automática, y se buscan máquinas o procedimientos que puedan reconer el léxico y la sintáxis de dichos lenguajes.

Hay que comentar que no hay un algoritmo para enseñar a diseñar algoritmos, y que muchas veces el proceso de construcción puede llegar a ser muy poco disciplinado. No obstante, existen técnicas de diseño de algoritmos, que vienen a ser modelos abstractos de los mismos aplicables a gran variedad de problemas reales.

Una vez construido un programa para un problema, surge la cuestión C) de si lo resuelve realmente. Normalmente los programadores prueban sus programas sobre una gran cantidad de datos de entrada para descubrir la mayoría de los errores lógicos presentes, aunque con este método (al que suele denominarse de prueba y depuración) no se puede estar completamente seguro de que el programa no contiene errores. Necesitaríamos para realizar la verificación formal, reglas que describan de forma precisa el efecto que cada instrucción tiene en el estado actual del programa, para, aplicando dichas reglas demostrar rigurosamente que lo que hace el programa coincide con sus especificaciones. En cualquier caso y cuando la prueba formal resulte muy complicada, podemos aumentar la confianza en nuestro programa realizando en el mismo los "test¨ cuidadosos de que hablábamos al principio.

Alcanzado este punto, ya tenemos un programa que en principio es solución de un problema.

Se plantea entonces la duda de que hacer en caso de que para el mismo problema seamos capaces de construir otro programa que también lo resuelva. ¿Cómo decidirnos por una u otra solución? o más aún, ¿qué ocurre si el programa aún siendo correcto consume demasiados recursos y es inaceptable?. La respuesta viene dada a través del punto D) en nuestro recorrido: el análisis del tiempo y espacio que necesita una solución concreta; en definitiva, el estudio de la eficiencia de los programas, midiendo la complejidad en espacio, por el número de variables y el número y tamaño de las estructuras de datos que se usan, y la complejidad en tiempo por el número de acciones elementales llevadas a cabo en la ejecución del programa.

Los problemas computables fueron entonces clasificados en dos tipos: problemas eficientemente computables, para los que existía un algoritmo eficiente; y problemas intratables, para los que no existen algoritmos eficientes. La existencia de problemas intratables no ha sido probada, si bién se han encontrado muchas evidencias a su favor.

Otra clase de problemas a considerar es la clase NP[3] de los problemas para los que existia un algoritmo no determinístico en tiempo polinomial, y dentro de ella, los problemas NP- completos.

[3] la N de “no determinista”; la P de “polinómico”. Problemas que no pueden ser resueltos por la máquinas determinísticas o computadores actuales.

Los intentos (desde los años 40) de construir máquinas para modelizar algunas de las funciones del cerebro biológico, ha permitido desarrollar máquinas capaces de 'aprender' (y reproducir) funciones (o sistemas) cuya forma (o comportamiento) se desconoce, pero sí conocemos una serie de ejemplos que reflejan esta forma (o comportamiento). Estas máquinas llamadas Redes Neuronales Artificiales también aportan su granito de arena al desarrollo de la computación.

A menudo se utiliza la técnica de reducir un problema a otro para comprobar si tiene o no solución efectiva. La estrategia en el caso de la respuesta negativa es la siguiente, si se reduce de forma efectiva un problema sin solución efectiva a otro problema (mediante una función calculable), entonces este nuevo problema tampoco tendrá solución efectiva. La razón es muy simple, si tuviese solución efectiva, componiendo el algoritmo solución con el algoritmo de transformación obtendríamos una solución para el problema efectivamente irresoluble. En sentido inverso, si se reduce un problema a otro para el que se conoce una solución efectiva, entonces componiendo se obtiene una solución para el primer problema. Esta técnica es muy útil y se utiliza a menudo. Por otro lado, esta mísma técnica es muy empleada en el campo de la complejidad algorítmica.

La Complejidad Algorítmica trata de estudiar la relativa dificultad computacional de las funciones computables. Rabin (1960) fué de los primeros en plantear la cuestión ¿Qué quiere decir que *f* sea más difícil de computar que *g*?

J. Hartmanis and R.E. Stearns, en *On the computational complexity of algorithms* (1965) introducen la noción fundamental de medida de complejidad definida como el tiempo de computación sobre una máquina de Turing multicinta.

Después surge la definición de funciones computables en tiempo polinomial, y se establece una jerarquía de complejidad, los problemas NP, NP-duros y NP-completos

LECCION 3. - AUTOMATAS Y LENGUAJES

El desarrollo de los ordenadores en la década de los 40, con la introducción de los programas en la memoria principal, y posteriormente con los lenguajes de programación de alto nivel, propician la distinción entre lenguajes formales, con reglas sintácticas y semánticas rígidas, concretas y bién definidas, de los lenguajes naturales como el inglés, donde la sintaxis y la semántica no se pueden controlar fácilmente. Los intentos de formalizar los lenguajes naturales, lleva a la construcción de gramáticas, como una forma de describir estos lenguajes, utilizando para ello reglas de producción para construir las frases del lenguaje. Se puede entonces caracterizar un Lenguaje, mediante las reglas de una gramática adecuada.

Los trabajos de McCulloch y Pitts (1943) describen los cálculos lógicos inmersos en un dispositivo (neurona artificial) que habían diseñado para simular la actividad

de una neurona biológica. El dispositivo recibía o no, una serie de impulsos eléctricos por sus entradas que se ponderaban, y producía una salida binaria (existe pulso eléctrico o no). Las entradas y salidas se podían considerar como cadenas de 0 y 1, indicando entonces la forma de combinar la cadena de entrada para producir la salida. La notación utilizada es la base para el desarrollo de expresiones regulares en la descripción de conjuntos de cadenas de caracteres.

C. Shannon (1948) define los fundamentos de la teoría de la información, y utiliza esquemas para poder definir sistemas discretos, parecidos a los autómatas finitos, relacionándolos con cadenas de Markov, para realizar aproximaciones a los lenguajes naturales.

J. Von Neumann (1948) introduce el termino de teoría de autómatas, y dice sobre los trabajos de McCulloch-Pitts: *... el resultado más importante de McCulloch-Pitts, es que cualquier funcionamiento en este sentido, que pueda ser definido en todo, lógicamente, estríctamente y sin ambiguedad, en un número finito de palabras, puede ser realizado también por una tal red neuronal formal.*

La necesidad de traducir los algorítmos escritos en lenguajes de alto nivel al lenguaje máquina, propicia la utilización de máquinas como los autómatas de estados finitos, para reconocer si una cadena determinada pertenece (es una frase de) a un lenguaje concreto, usando para ello la función de transición de estados, mediante un diagrama de transición o una tabla adecuada. Tenemos así otra forma de caracterizar los lenguajes, de acuerdo con máquinas automáticas que permitan reconocer sus frases.

S.C. Kleene, en 1951, realiza un informe (solicitado por la RAND Corporation) sobre los trabajos de McCulloch-Pitts, que se publica en 1956. En este informe, Kleene demuestra la equivalencia entre lo que él llama "dos formas de definir una misma cosa", que son los *sucesos regulares* (que se pueden describir a partir de sucesos bases y los operadores unión, concatenación e iteración (*)), es decir, expresiones regulares, y sucesos especificados por un autómata finito.

Rabin y Scott (1960) obtienen un modelo de computador con una cantidad finita de memoria, al que llamaron autómata de estados finitos. Demostraron que su comportamiento posible, era básicamente el mismo que el descrito mediante expresiones regulares, desarrolladas a partir de los trabajos de McCulloch y Pitts. No obstante lo dicho, para un alfabeto concreto, no todos los lenguajes que se pueden construir son regulares. Ni siquiera todos los interesantes desde el punto de vista de la construcción de algorítmos para resolver problemas. Hay entonces muchos problemas que no son calculables con estos lenguajes. Esto pone de manifiesto las limitaciones de los autómatas finitos y las gramáticas regulares, y propicia el desarrollo de máquinas reconocedoras de otros tipos de lenguajes y de las gramáticas correspondientes, asociadas a los mismos.

En 1956, la Princenton Univ. Press publica el libro *Automata Studies*, editado por C. Shannon y J. McCarthy, donde se recogen una serie de trabajos sobre autómatas y lenguajes formales.

D. A. Huffman (1954) ya utiliza conceptos como *estado de un autómata y tabla de transiciones*.

N. Chomsky (1956) propone tres modelos para la descripción de lenguajes, que son la base de su futura jerarquía de los tipos de lenguajes, que ayudó también en el desarrollo de los lenguajes de programación. Para ello intentó utilizar autómatas para extraer estructuras sintácticas (*....el inglés no es un lenguaje de estados finitos.*) y dirige sus estudios a las gramáticas, indicando que la diferencia esencial entre autómatas y gramáticas es que la lógica asociada a los autómatas (p.e., para ver la equivalencia entre dos de ellos) es Decidible, mientras que la asociada a las gramáticas no lo es. Desarrolla el concepto de gramática libre del contexto, en el transcurso de sus investigaciones sobre la sintáxis de los lenguajes naturales.

Backus y Naur desarrollaron una notación formal para describir la sintáxis de algunos lenguajes de programación, que básicamente se sigue utilizando todavía, y que podía considerarse equivalente a las gramáticas libres del contexto.

Consideramos entonces los lenguajes libres (independientes) del contexto, y las gramáticas libres del contexto y los autómatas con pila, como forma de caracterizarlos y manejarlos. Los distintos lenguajes formales que se pueden construir sobre un alfabeto concreto pueden clasificarse en clases cada vez más amplias que incluyen como subconjunto a las anteriores, de acuerdo con la jerarquía establecida por Chomsky en los años 50.

Se puede llegar así, de una forma casi natural a considerar las máquinas de Turing, establecidas casi 20 años antes, como máquinas reconocedoras de los lenguajes formales dependientes del contexto o estructurados por frases, e incluso a interpretar la Tesis de Turing como que un sistema computacional nunca podrá efectuar un análisis sintáctico de aquellos lenguajes que están por encima de los lenguajes estructurados por frases, según la jerarquía de Chomsky".

En consecuencia, podemos utilizar la teoría de autómatas y los conceptos relativos a gramáticas sobre distintos tipos de lenguajes, para decidir (si se puede) si una función (o problema) es calculable, en base a que podamos construir un algoritmo solución mediante un lenguaje que puede ser analizado mediante alguna máquina de las citadas anteriormente.

Los temas sobre autómatas, computabilidad, e incluso la complejidad algorítmica fueron incorporándose a los curriculum de ciencias de la computación de diferentes universidades, mediada la década de los 60. Esta incorporación puso de manifiesto que las ciencias de la computación habían usado gran cantidad de

ideas de muy diferentes campos para su desarrollo, y que la investigación sobre aspectos básicos podía cooperar y aumentar los avances de la computación.

1.3.1. Qué es un lenguaje formal?[4]

En matemáticas, lógica, y las ciencias computacionales, un lenguaje formal es un conjunto de palabras (cadenas de caracteres) de longitud finita formadas a partir de un alfabeto (conjunto de caracteres) finito.

Informalmente, el término lenguaje formal se utiliza en muchos contextos (en las ciencias, en derecho, etc.) para referirse a un modo de expresión más cuidadoso y preciso que el habla cotidiana. Hasta finales de la década de 1990, el consenso general era que un lenguaje formal, era en cierto modo la versión «límite» de este uso antes mencionado: un lenguaje tan formalizado que podía ser usado en forma escrita para describir métodos computacionales. Sin embargo, hoy en día, el punto de vista de que la naturaleza esencial de los lenguajes naturales (sin importar su grado de «formalidad» en el sentido informal antes descrito) difiere de manera importante de aquella de los verdaderos lenguajes formales, gana cada vez más adeptos.

Un posible alfabeto sería, digamos, {a, b}, y una cadena cualquiera sobre este alfabeto sería, por ejemplo, ababba. Un lenguaje sobre este alfabeto, que incluyera esta cadena, sería: el conjunto de todas las cadenas que contienen el mismo número de símbolos a que b, por ejemplo.

La palabra vacía (esto es, la cadena de longitud cero) es permitida y frecuentemente denotada mediante ε o λ. Mientras que el alfabeto es un conjunto finito y cada palabra tiene una longitud también finita, un lenguaje puede bien incluir un número infinito de palabras.

Algunos ejemplos varios de lenguajes formales:

- el conjunto de todas las palabras sobre {a, b}
- el conjunto {an: n es un número primo}
- el conjunto de todos los programas sintácticamente válidos en un determinado lenguaje de programación
- el conjunto de entradas para las cuales una particular máquina de Turing se detiene.

Los lenguajes formales pueden ser especificados en una amplia variedad de maneras, como:

- cadenas producidas por una gramática formal (ver Jerarquía de Chomsky)
- cadenas producidas por una expresión regular

[4] Definición de Wikipedia, En http://es.wikipedia.org/wiki/Lenguaje_formal

- cadenas aceptadas por un autómata, tal como una máquina de Turing

Una pregunta que se hace típicamente sobre un determinado lenguaje formal L es cuán difícil es decidir si incluye o no una determinada palabra v. Este tema es del dominio de la teoría de la computabilidad y la teoría de la complejidad computacional.

Por contraposición al lenguaje propio de los seres vivos y en especial el lenguaje humano, considerados lenguajes naturales, se denomina lenguaje formal a los lenguajes «artificiales» propios de las matemáticas o la informática, los lenguajes artificiales son llamados lenguajes formales (incluyendo lenguajes de programación). Sin embargo, el lenguaje humano tiene una característica que no se encuentra en los lenguajes de programación: la diversidad.

En 1956, Noam Chomsky creó la Jerarquía de Chomsky para organizar los distintos tipos de lenguaje formal.

Un Lenguaje normal o natural, como por ejemplo el lenguaje español u inglés, son la clase de lenguajes que han evolucionado con el paso del tiempo y tienen por fin la comunicación humana. Este tipo de lenguajes están en constante evolución y sus reglas gramaticales solo pueden ser explicadas y no determinadas en cuanto a la estructura del lenguaje.

En contraste, un lenguaje formal esta definido por reglas preestablecidas y se ajustan con rigor a ellas, ejemplos son los lenguajes computacionales como C o Pascal.

Como especificar la sintaxis de un lenguaje?: Se utiliza la jerarquía de chomsky; la **jerarquía de Chomsky** es una clasificación jerárquica de distintos tipos de gramáticas formales que generan lenguajes formales. Esta jerarquía fue descrita por Noam Chomsky en 1956.

Define cuatro familias de gramáticas y lenguajes estas son: 0,1,2,3 gramáticas sin restricciones, de contexto, de contexto libre y regulares.

- Gramáticas de tipo 0 (sin restricciones), que incluye a todas las gramáticas formales. Estas gramáticas generan todos los lenguajes capaces de ser reconocidos por una máquina de Turing. Los lenguajes son conocidos como lenguajes recursivamente enumerables. Nótese que esta categoría es diferente de la de los lenguajes recursivos, cuya decisión puede ser realizada por una máquina de Turing que se detenga.

- Gramáticas de tipo 1 (gramáticas sensibles al contexto) generan los lenguajes sensibles al contexto. Estas gramáticas tienen reglas de la forma $\alpha A \beta \rightarrow \alpha \gamma \beta$ con *A* un no terminal y α, β y γ cadenas de terminales y no terminales. Las cadenas α y β pueden ser vacías, pero γ no puede serlo. La regla $S \rightarrow \epsilon$ está permitida si *S* no aparece en la parte derecha de ninguna

regla. Los lenguajes descritos por estas gramáticas son exactamente todos aquellos lenguajes reconocidos por una máquina de Turing no determinista cuya cinta de memoria está acotada por un cierto número entero de veces sobre la longitud de entrada.

- Gramáticas de tipo 2 (gramáticas libres del contexto) generan los lenguajes independientes del contexto. Las reglas son de la forma $A \rightarrow \gamma$ con *A* un no terminal y γ una cadena de terminales y no terminales. Estos lenguajes son aquellos que pueden ser reconocidos por un autómata con pila.

- Gramáticas de tipo 3 (gramáticas regulares) generan los lenguajes regulares. Estas gramáticas se restringen a aquellas reglas que tienen en la parte izquierda un no terminal, y en la parte derecha un solo terminal, posiblemente seguido de un no terminal. La regla $S \rightarrow \epsilon$ también está permitida si *S* no aparece en la parte derecha de ninguna regla. Estos lenguajes son aquellos que pueden ser aceptados por un autómata finito. También esta familia de lenguajes pueden ser obtenidas por medio de expresiones regulares.

LECCION 4. – LENGUAJES REGULARES[5]

Los lenguajes regulares se llaman así porque sus palabras contienen "regularidades" o repeticiones de los mismos componentes, como por ejemplo en el lenguaje L_1 siguiente: L_1 = {ab, abab, ababab, abababab, . . .} En este ejemplo se aprecia que las palabras de L1 son simplemente repeticiones de "ab" cualquier número de veces. Aquí la "regularidad" consiste en que las palabras contienen "ab" algún número de veces.

Otro ejemplo más complicado sería el lenguaje L_2: L_2 = {abc, cc, abab, abccc, ababc, . . .}

La regularidad en L_2 consiste en que sus palabras comienzan con repeticiones de "ab", seguidas de repeticiones de "c". Similarmente es posible definir muchos otros lenguajes basados en la idea de repetir esquemas simples. Esta es la idea básica para formar los lenguajes Regulares.

Adicionalmente a las repeticiones de esquemas simples, vamos a considerar que los lenguajes finitos son también regulares por definición. Por ejemplo, el lenguaje L_3 = {anita, lava, la, tina} es regular.

Finalmente, al combinar lenguajes regulares uniéndolos o concatenándolos, también se obtiene un lenguaje regular. Por ejemplo, $L_1 \cup L_3$ = {anita, lava, la, tina, ab, abab, ababab, abababab, . . .} es regular.

[5] BRENA PIÑERO, Ramon F. Autómatas y lenguajes un enfoque de diseño (2003) ITESM, En: http://lizt.mty.itesm.mx/~rbrena/AyL.html

También es regular una concatenación como L_3L_3 = {anitaanita, anitalava, anitala, anitatina, lavaanita, lavalava, lavala, lavatina, . . .}

Las cadenas no nulas, en un alfabeto $\sum$ se crean por concatenacion de cadenas sencillas, las de longitud 1. Tambien es posible ver la concatenación como una operación en lenguajes, de modo que se podrían considerar los lenguajes obtenidos por concatenación de lenguajes sencillos de la forma {a}, donde a $\in \sum$, No obstante si la concatenación es la única operación permitida, entonces solo pueden obtenerse cadenas o lenguajes que contengan cadenas sencillas, agregar la operación de conjuntos de unión posibilita tener lenguajes de varios elementos y si se añade la operación * cerradura o estrella de Kleene que se deriva naturalmente de la concatenación, tambien es factible producir lenguajes infinitos.

Se adicionan dos lenguajes más a los lenguajes sencillos de la forma {a}: el lenguaje vacio ϕ y el lenguaje {λ} cuyo único elemento es la cadena nula.

Un lenguaje regular en un alfabeto $\sum$ es uno que puede obtenerse de esos lenguajes básicos con las operaciones de unión, concatenación y * de Kleene. Así pues es factible describir un lenguaje regular como una fórmula explícita. Es comun que ésta última se simplifique un poco, al omitir las llaves {} o substituirlas con paréntesis y al reemplazar $\cup$ con +, de lo cual resulta una expresión regular.

1.4.1. Definición formal de Lenguaje Regular

Un lenguaje L es regular si y sólo si se cumple al menos una de las condiciones siguientes:

- L es finito
- L es la unión o la concatenación de otros lenguajes regulares R_1 y R_2, L = $R_1 \cup R_2$
- L = R1R2 respectivamente.
- L es la cerradura o estrella de Kleene de algún lenguaje regular, L = R^*.

Sea $\sum$ un alfabeto. El conjunto de lenguajes regulares sobre $\sum$ se define como:

(a) ϕ es un lenguaje regular.
(b) {λ} es un lenguaje regular.
(c) Para todo a $\in \sum$, {a} es un lenguaje regular.
(d) Si A y B son lenguajes regulares, entonces A$\cup$B, A•B y A* son lenguajes regulares.
(e) Ningún otro lenguaje sobre $\sum$ es regular.

Por la definición anterior, el conjunto de los lenguajes regulares sobre $\sum$ esta formado por el lenguaje vacío, los lenguajes unitarios incluido {λ} y todos los lenguajes obtenidos a partir de la unión, concatenación y cerradura o estrella de Kleene.

Ejemplo: Sea $\sum$ = {a, b}, lo siguiente es cierto:

(i) ϕ y {λ} son lenguajes regulares
(ii) {a} y {b} son lenguajes regulares.
(iii) {a, b} es regular pues resulta de la unión de {a} y {b}.
(iv) {ab} es regular pues resulta de la concatenación de {a} y {b}.
(v) {a, ab, b} es regular (unión de ii) y iv)).
(vi) $\{a^i \mid i \geq 0\}$ es regular.
(vii) $\{a^i b^j \mid i \geq 0 \text{ y } j \geq 0\}$ es regular.
(viii) $\{(ab)^i \mid i \geq 0\}$ es regular.

LECCION 5. - AUTOMATA[6]

La palabra autómata evoca algo que pretende imitar las funciones propias de los seres vivos, especialmente relacionadas con el movimiento, por ejemplo el típico robot antropomorfo. En el campo de los Traductores, Procesadores, Compiladores e Intérpretes, lo fundamental no es la simulación del movimiento, sino la simulación de procesos para tratar información.

La información se codifica en cadenas de símbolos, y un autómata es un dispositivo que manipula cadenas de símbolos que se le presentan a su entrada, produciendo otras tiras o cadenas de símbolos a su salida.

El autómata recibe los símbolos de entrada, uno detrás de otro, es decir secuencialmente. El símbolo de salida que en un instante determinado produce un autómata, no sólo depende del último símbolo recibido a la entrada, sino de toda la secuencia o cadena, que ha recibido hasta ese instante.

Todo lo anterior conduce a definir un concepto fundamental: estado de un autómata.

El estado de un autómata es toda la información necesaria en un momento dado, para poder deducir, dado un símbolo de entrada en ese momento, cual será el símbolo de salida. Es decir, conocer el estado de un autómata, es lo mismo que conocer toda la historia de símbolos de entrada, así como el estado inicial, estado en que se encontraba el autómata al recibir el primero de los símbolos de entrada. El autómata tendrá un determinado número de estados (pudiendo ser infinitos), y se encontrará en uno u otro según sea la historia de símbolos que le han llegado.

Se define configuración de un autómata a su situación en un instante. Se define movimiento de un autómata como el transito entre dos configuraciones. Si un autómata se encuentra en un estado determinado, recibe un símbolo también

6 CUEVAS LOVALLE, Juan Manuel LENGUAJES, GRAMÁTICAS Y AUTÓMATAS. Segunda Edición, (España), 2001.

determinado, producirá un símbolo de salida y efectuará un cambio o transición a otro estado (también puede quedarse en el mismo estado).

El campo de estudio de los Traductores, Procesadores e Intérpretes son los lenguajes y las gramáticas que los generan. Los elementos del lenguaje son sentencias, palabras, etc... Formadas a partir de un alfabeto o vocabulario, que no es otra cosa que un conjunto finito de símbolos. Establecidas las reglas gramaticales, una cadena de símbolos pertenecerá al correspondiente lenguaje si tal cadena se ha formado obedeciendo esas reglas. Entonces un autómata reconocedor de ese lenguaje, funciona de tal forma que cuando reciba a su entrada una determinada cadena de símbolos indica si dicha cadena pertenece o no al lenguaje. También se mostrará como existe un tipo de autómata para reconocer cada uno de los tipos de lenguajes generados por las correspondientes gramáticas.

1.5.1. Definición formal de autómata

Un autómata es una quíntupla A = (E, S, Q, f, g) donde :

E = {conjunto de entradas o vocabulario de entrada}
S = {conjunto de salidas o vocabulario de salida}
Q = {conjunto de estados}

E es un conjunto finito, y sus elementos se llaman entradas o símbolos de entrada.
S es un conjunto finito, y sus elementos se llaman salidas o símbolos de salida.
Q es el conjunto de estados posibles, puede ser finito o infinito.

f es la función de transición o función del estado siguiente, y para un par del conjunto
E Q devuelve un estado perteneciente al conjunto Q.
E Q es el conjunto producto cartesiano de E por Q.
g es la función de salida, y para un par del conjunto E Q, devuelve un símbolo de salida del conjunto S.

Representación de autómatas

Los autómatas se pueden representar mediante:
- Tabla de transiciones.
- Diagrama de Moore.

Tabla de transiciones. Las funciones f y g pueden representarse mediante una tabla, con tantas filas como estados y tantas columnas como entradas. Así por ejemplo se puede representar el autómata A = (E, S, Q, f, g) donde E = {a,b}, S = {0,1}, Q = {q1,q2,q3} y las funciones f y g se pueden representar por :

f	a	b
q_1	Q_1	q_2
q_2	Q_3	q_2
q_3	Q_3	q_1

g	a	b
q_1	0	1
q_2	0	0
q_3	1	0

Así se tiene que $f(a, q_1)=q_1$; $g(a, q_1)=0$; o también $f(a, q_2)=q_3$; y $g(a, q_3)=1$.

Ambas funciones también se pueden representar en una misma tabla de la siguiente forma:

f / g	a	b
q_1	q1/0	q2/1
q2	q3/0	q2/0
q3	q3/1	q1/0

Diagramas de Moore. Los diagramas de Moore son otra forma de representar las funciones de transición y salida de un autómata.

El diagrama de Moore es un grafo orientado en el que cada nodo corresponde a un estado; y si $f(\varepsilon, q_i) = q_j$ y $g(\varepsilon, q_i) = s$ existe un arco dirigido del nodo q_i al correspondiente q_j, sobre el que se pone la etiqueta ε / s, tal y como se muestra en la figura 1.

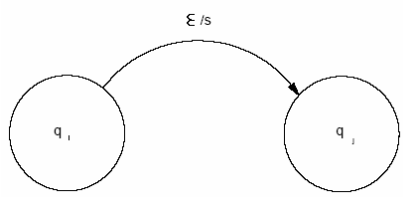


Figura 1: Diagrama de Moore

Así continuando con el ejemplo, el autómata se representa con el diagrama de Moore de la figura 2.

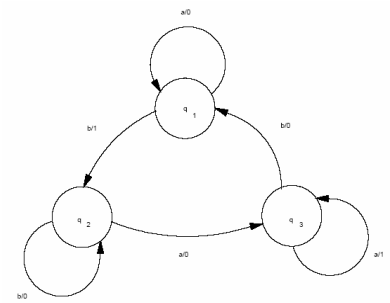


Figura 2: Ejemplo de Diagrama de Moore

AUTOMATAS FINITOS

Para comprender el significado de Autómata Finito[7], tendremos en cuenta el término máquina, que evoca algo hecho en metal, usualmente ruidoso y grasoso, que ejecuta tareas repetitivas que requieren de mucha fuerza o velocidad o precisión. Ejemplos de estas máquinas son las embotelladoras automáticas de refrescos. Su diseño requiere de conocimientos en mecánica, resistencia de materiales, y hasta dinámica de fluidos. Al diseñar tal máquina, el plano en que se le dibuja hace abstracción de algunos detalles presentes en la máquina real, tales como el color con que se pinta, o las imperfecciones en la soldadura.

El plano de diseño mecánico de una máquina es una abstracción de ésta, que es útil para representar su forma física. Sin embargo, hay otro enfoque con que se puede modelar la máquina embotelladora: cómo funciona, en el sentido de saber qué secuencia de operaciones ejecuta. Así, la parte que introduce el líquido pasa por un ciclo repetitivo en que primero introduce un tubo en la botella, luego descarga el líquido, y finalmente sale el tubo para permitir la colocación de la cápsula (“corcholata”). El orden en que se efectúa este ciclo es crucial, pues si se descarga el líquido antes de haber introducido el tubo en la botella, el resultado no será satisfactorio.

Las máquinas que se estudian son abstracciones matemáticas que capturan solamente el aspecto referente a las secuencias de eventos que ocurren, sin tomar en cuenta ni la forma de la máquina ni sus dimensiones, ni tampoco si efectúa movimientos rectos o curvos, etc.

En esta parte se estudian las máquinas abstractas más simples, los autómatas finitos, las cuales están en relación con los lenguajes regulares, como veremos a continuación.

LECCION 6. - DEFINICIÓN FORMAL DE AUTÓMATAS FINITOS

Al describir una máquina de estados finitos en particular, debemos incluir las informaciones que varían de un autómata a otro; es decir, no tiene sentido incluir descripciones generales aplicables a todo autómata. Estas informaciones son exactamente las que aparecen en un diagrama de estados y transiciones, como se presenta más adelante.

[7] BRENA PIÑERO, Ramon F. Autómatas y lenguajes un enfoque de diseño (2003) ITESM, En: http://lizt.mty.itesm.mx/~rbrena/AyL.html

Un autómata finito es una quintupla

M =(Q, A, δ, q_0, F) en que:
– Q es un conjunto finito llamado conjunto de estados
– A es un alfabeto llamado alfabeto de entrada
– δ es una aplicación llamada función de transición

$$\delta : Q \times A \rightarrow Q$$

–q_0 es un elemento de Q, llamado estado inicial
– F es un subconjunto de Q, llamado conjunto de estado finales.

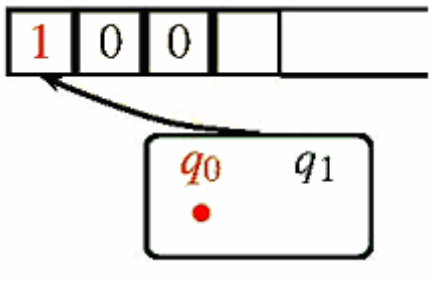


Figura 3: Automata Finito

Desde el punto de vista intuitivo, podemos ver un autómata finito como una caja negra de control (ver Figura 3), que va leyendo símbolos de una cadena escrita en una cinta, que se puede considerar ilimitada por la derecha. Existe una cabeza de lectura que en cada momento está situada en una casilla de la cinta. Inicialmente, esta se sitúa en la casilla de más a la izquierda. El autómata en cada momento está en uno de los estado de *Q*. Inicialmente se encuentra en q_0.

En cada paso, el autómata lee un símbolo y según el estado en que se encuentre, cambia de estado y pasa a leer el siguiente símbolo. Así sucesivamente hasta que termine de leer todos los símbolos de la cadena. Si en ese momento la máquina está en un estado final, se dice que el autómata acepta la cadena. Si no está en un estado final, la rechaza.

Otra definición:
Una máquina de estados finitos M es un quíntuplo (K,Σ, δ, s, F), donde:
K es un conjunto de identificadores (símbolos) de estados;
Σ es el alfabeto de entrada;
$s \in K$ es el estado inicial;
$F \in K$ es un conjunto de estados finales;
δ: K x Σ → K es la función de transición, que a partir de un estado y un símbolo del Alfabeto obtiene un nuevo estado. (que puede ser el mismo en que se encontraba)

Otra Definición:
Un Autómata Finito se define por una quíntupla: $< Q, \Sigma, q_0, \delta, A >$, en donde

Q es el conjunto de estados
Σ es el alfabeto del lenguaje
q_0 es el estado inicial
δ es la función de transición
A es el conjunto de estados de aceptación.

Ejemplo: Tomando el diagrama anterior:

$Q = \{ \lambda , 0 , 1 , No \}$; $\Sigma = \{0,1\}$ $q_0 = (\lambda)$; $A = \{(1)\}$

Función de transición: $\delta: Q \times \Sigma \rightarrow Q$

		ENTRADAS	
	δ	0	1
E S T A D O S	λ	0	1
	0	NO	1
	1	0	1
	NO	NO	NO

La función de transición indica a qué estado se va a pasar sabiendo cuál es el estado actual y el símbolo que se está leyendo. Es importante notar que δ es una función y no simplemente una relación; esto implica que para un estado y un símbolo del alfabeto dados, habrá un y sólo un estado siguiente. Esta característica, que permite saber siempre cuál será el siguiente estado, se llama determinismo.

Los autómatas finitos son capaces de reconocer solamente, un determinado tipo de lenguajes, llamados Lenguajes Regulares, que pueden ser caracterizados también, mediante un tipo de gramáticas llamadas también regulares. Una forma adicional de caracterizar los lenguajes regulares, es mediante las llamadas expresiones regulares, que son las frases del lenguaje, construidas mediante operadores sobre el alfabeto del mismo y otras expresiones regulares, incluyendo el lenguaje vacío.

Estas caracterizaciones de los lenguajes regulares se utilizan en la práctica, según que la situación concreta esté favorecida por la forma de describir el lenguaje de cada una de ellas. Los autómatas finitos se utilizan generalmente para verificar que las cadenas pertenecen al lenguaje, y como un analizador en la traducción de algorítmos al computador.

Las gramáticas y sus reglas de producción se usan frecuentemente en la descripción de la sintaxis de los lenguajes de programación que se suele incluir en los manuales correspondientes. Por otro lado, las expresiones regulares proporcionan una forma concisa y relativamente sencilla (aunque menos intuitiva) para describir los lenguajes regulares, poniendo de manifiesto algunos detalles de su estructura que no quedan tan claros en las otras caracterizaciones. Su uso es habitual en editores de texto, para búsqueda y sustitución de cadenas.

En definitiva, las caracterizaciones señaladas de los lenguajes (formales) regulares, y por tanto ellos mismos, tienen un uso habitual en la computación práctica actual.

Ejemplo de autómata finito: Se va a diseñar un autómata que reconozca el paso de un alumno por un curso, por ejemplo, Autómatas y lenguajes formales. Representar las distintas decisiones que se realizan y si se aprueba o aplaza el curso. Se controla que no haya más de dos convocatorias por año y se termina cuando se aprueba el curso.Habrá un alfabeto de entrada contendrá los siguientes elementos:

P: El alumno se presenta al examen.
N: El alumno no se presenta al examen.
A: El alumno aprueba el examen.
S: El alumno aplaza un examen.

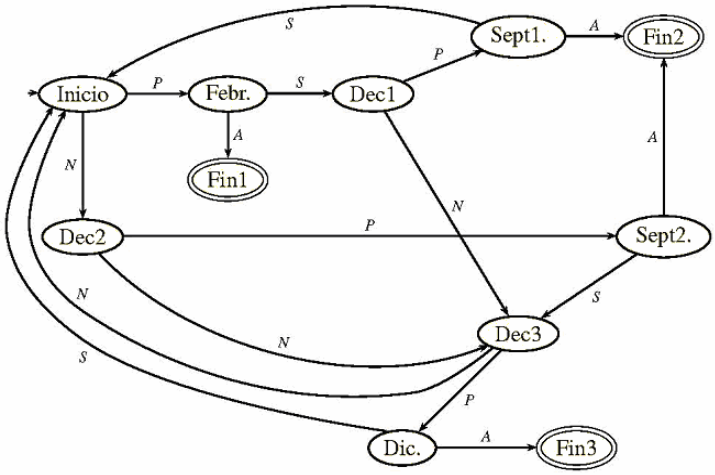


Figura 4: Recorrido de un alumno por una asignatura

La secuencia se ilustra en la figura 4. Comienza en un estado, Inicio. A continuación decide si presenta en Febrero o no. Si no presenta y aprueba, termina. Si no presenta o aplaza, se decide si se presenta en septiembre, pero como hay que controlar que un estudiante no se presente a tres convocatorias en un año, los estados son distintos en ambos casos. Si en septiembre aprueba,

termina. Si suspende o aplaza y ya se había presentado en febrero, comienza de nuevo. En otro caso, puede decidir si presenta en diciembre. Si aprueba, termina y si suspende, empieza de nuevo.

Este esquema corresponde a un autómata finito. Se caracteriza por una estructura de control que depende de un conjunto finito de estados. Se pasa de unos a otros leyendo símbolos del alfabeto de entrada. Este autómata representa una versión simplificada del problema real, ya que no controla el número total de convocatorias. Un autómata para todos los cursos se puede construir uniendo autómatas para cada una de los cursos, pero teniendo en cuenta relaciones como requisitos entre los mismos.

Ejemplo Supongamos el autómata $M = (Q, A, q_0, \delta, F)$ donde
–$Q = \{q_0, q_1, q_2\}$
–$A = \{a, b\}$

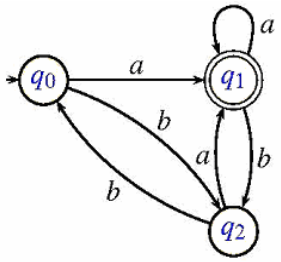


Figura 5: Diagrama de transición, Autómata de estado finito

– La función de transición δ está definida por las siguientes igualdades:

$\delta(q_0, a)=q_1$
$\delta(q_0, b)=q_2$
$\delta(q_1, a)=q_1$
$\delta(q_1, b)=q_2$
$\delta(q_2, a)=q_1$
$\delta(q_2, b)=q_0$

–F ={q1} El diagrama de transición viene expresado en la Figura 5.

LECCION 7. - AUTOMATAS FINITOS DETERMINISTICOS[8] (AFD)

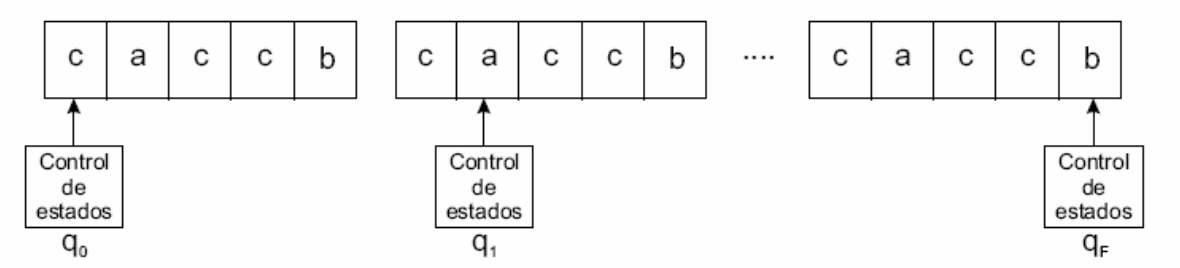
Un autómata finito determinístico (AFD) se define como una quíntupla $M=(Q,V,\delta,q_0,F)$,donde:

Q es un conjunto finito de estados
V es el alfabeto de entrada
q_0 es el estado inicial
$F \in Q$ es el conjunto de estados finales
$\delta : Q \times V \rightarrow Q$ es la función de transición

El nombre "determinista" viene de la forma en que está definida la función de transición: si en un instante t la máquina está en el estado q y lee el símbolo a entonces, en el instante siguiente t + 1 la máquina cambia de estado y sabemos con seguridad cual es el estado al que cambia, que es precisamente δ(q, a).

El AFD es inicializado con una palabra de entrada w como sigue:

1. w se coloca en la cinta de entrada, con un símbolo en cada celda
2. el cabezal de lectura se apunta al símbolo mas a la izquierda de w
3. el estado actual pasa a ser q_0



Una vez que se ha inicializado el AFD, comienza su "ejecución" sobre la palabra de entrada.
Como cualquier computador tiene un ciclo de ejecución básico:

1. se lee el símbolo actual, que es el apuntado por el cabezal de lectura. Si el cabezal apunta a una celda vacila entonces el AFD termina su ejecución, aceptando la palabra en caso de que el estado actual sea final y rechazando la palabra en caso contrario. Esto ocurre cuando se ha leído toda la palabra de entrada, y se produce una situación similar a tener una condición "fin de fichero" en la ejecución de un programa.

[8] MARIN MORALES, Roque y otros, Teoría de autómatas y lenguajes formales En: Universidad de Murcia http://perseo.dif.um.es/%7Eroque/talf/Material/apuntes.pdf

2. se calcula el estado siguiente a partir del estado actual y del símbolo actual según la función de transición, esto es, δ(estado actual, símbolo actual) = estado siguiente
3. el cabezal de lectura se mueve una celda a la derecha
4. el estado siguiente pasa a ser el estado actual y vuelve al paso 1

La función de transición de un AFD se puede representar de dos formas: mediante una **tabla de transición** o mediante un **diagrama de transición**.

Tabla de transición	Cada fila corresponde a un estado $q \in Q$ El estado inicial se precede del símbolo → Cada estado final se precede del símbolo # Cada columna corresponde a un símbolo de entrada $a \in V$ En la posición (q, a) esta el estado que determine δ(q, a)
Diagrama de transición	Los nodos se etiquetan con los estados El estado inicial tiene un arco entrante no etiquetado Los estados finales están rodeados de un doble círculo Habrá un arco etiquetado con a desde el nodo q_i al q_j si $\delta(q_i, a) = q_j$

Ejemplo: Suponga que se tiene el autómata finito determinista dado por

$$M = (\{q_0, q_1, q_2\}, \{0, 1\}, \delta, q_0, \{q_1\})$$

donde la función $\delta : \{q_0, q_1, q_2\} \times \{0, 1\} \rightarrow \{q_0, q_1, q_2\}$ viene dada por:

$$\delta(q_0, 0) = q_0 \quad \delta(q_0, 1) = q_1$$
$$\delta(q_1, 0) = q_0 \quad \delta(q_1, 1) = q_2$$
$$\delta(q_2, 0) = q_2 \quad \delta(q_2, 1) = q_1$$

La tabla de transición correspondiente a este autómata será:

δ	0	1
→ q_0	q_0	q_1
# q_1	q_0	q_2
q_2	q_2	q_1

y el diagrama de transición correspondiente se muestra a continuación.

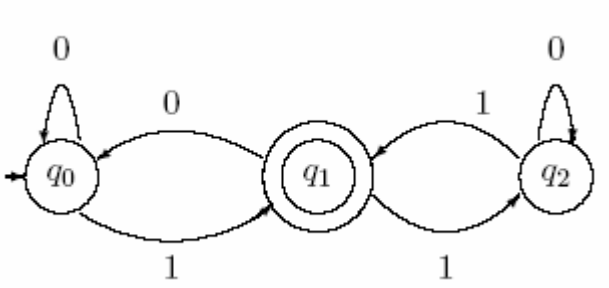


Figura 6: Diagrama de transición del ejemplo

Nota El diagrama de transición de un AFD tiene por cada nodo un solo arco etiquetado con cada uno de los símbolos del alfabeto. Algunos autores consideran que la función de transición puede ser parcial, es decir, no estar definida para algún δ(q, a). En ese caso se dice que el AFD es incompleto, y en el diagrama de trancisión faltan entonces los arcos correspondientes a los casos no definidos de la función de transición. En este módulo se Nosotros considera que los AFDs son completos.

Ejemplo: Se considera la expresión regular r=(11 + 110)*0 Se intenta construir un AF que acepte el lenguaje correspondiente L que corresponde a la expresión regular (a+b)*(ab+bba)(a+b)*)

La solucion del ejercicio luego de los analisis correspondientes, en su diagrama de transición queda de la siguiente forma

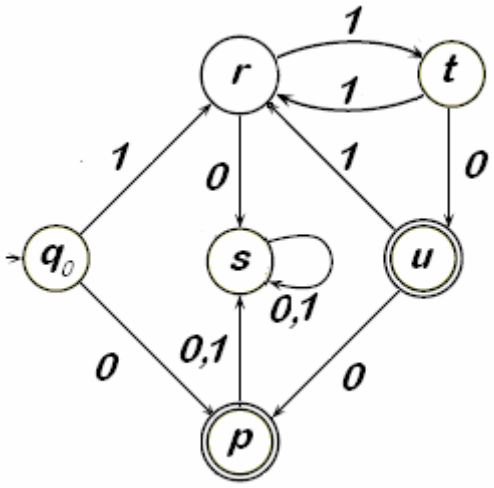


Figura 7: Autómata finito que acepta L_3

LECCION 8. – AUTOMATAS FINITOS NO DETERMINISTICOS (AFND)

Una extensión a los autómatas finitos deterministas es la de permitir que de cada nodo del diagrama de estados salga un número de flechas mayor o menor que Así, se puede permitir que falte la flecha correspondiente a alguno de los símbolos del alfabeto, o bien que haya varias flechas que salgan de un solo nodo con la misma etiqueta. Inclusive se permite que las transiciones tengan como etiqueta palabras de varias letras o hasta la palabra vacía. A estos autómatas finitos se les llama no Determinísticos o no deterministas (abreviado AFND), Al retirar algunas de las restricciones que tienen los autómatas finitos Determinísticos, su diseño para un lenguaje dado puede volverse más simple.

Hacemos notar en este punto que, dado que los AFN tienen menos restricciones que los AFD, resulta que los AFD son un caso particular de los AFND, por lo que todo AFD es de hecho un AFND.

Los autómatas finitos no Determinísticos (AFND) aceptan exactamente los mismos lenguajes que los autómatas determinísticos. Sin embargo, serán importantes para demostrar teoremas y por su más alto poder expresivo.

Un autómata finito no deterministico (AFND) es una quíntupla $M = (Q, V, \Delta, q_0, F)$ donde todos los componentes son como en los AFDs, excepto la función de transición que se define ahora como:

$$\Delta : Q \times V \longrightarrow P(Q)$$

donde P(Q) denota el conjunto de las partes de Q (o conjunto potencia 2^Q).

“No determinismo” (codominio P(Q)): a partir del estado actual y del símbolo actual de entrada no se puede determinar de forma exacta cuál será el estado siguiente.

Por ejemplo, $\Delta (q, a) = \{q_1, q_2,...,q_m\}$ indica que para el estado actual q y el símbolo de entrada a, el estado siguiente puede ser cualquier estado entre q_1 y q_m.

También puede darse $\Delta(q, a) = 0$: el estado siguiente no está definido, La interpretación intuitiva es que ahora el autómata, ante una entrada y un estado dado, puede evolucionar a varios estados posibles (incluyendo un solo estado o ninguno si $\Delta (q, a)= 0$). Es decir es como un algoritmo que en un momento dado nos deja varias opciones posibles o incluso puede no dejarnos ninguna.

Un AFND acepta una palabra de entrada w siempre que sea posible comenzar por el estado inicial y que exista una secuencia de transiciones que nos lleven a consumir la palabra y acabe el autómata en un estado final, es decir Una palabra se dice aceptada por un AFND si, siguiendo en cada momento alguna de las opciones posibles, llegamos a un estado final.

Los AFND también se representan mediante tablas o diagramas de transición

• En el diagrama de transición, hay algún nodo del que parten dos o más arcos etiquetados con el mismo símbolo del alfabeto, o falta algún arco para algún símbolo del alfabeto

• En la tabla de transición, alguna celda contiene 0 o un conjunto no unitario

Δ	a	b
Q_0	$\{q_0,q_3\}$	$\{q_0,q_1\}$
Q_1	0	$\{q_2\}$
$\#Q_2$	$\{q_2\}$	$\{q_2\}$
Q_3	$\{q_4\}$	0
$\#q_4$	$\{q_4\}$	$\{q_4\}$

Tabla de transición

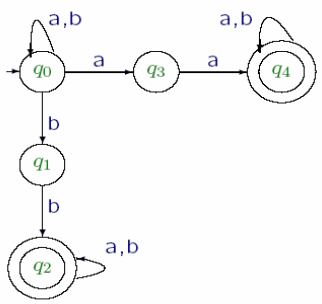


Diagrama de transicion

Figura 8: Representación de AFND

2.8.1 Diagramas de Transición

Los diagramas de transición de los AFND son totalmente análogos a los de los autómatas determinísticos. Solo que ahora no tiene que salir de cada vértice un y solo un arco para cada símbolo del alfabeto de entrada. En un autómata no determinístico, de un vértice pueden salir una, ninguna o varias flechas con la misma etiqueta.

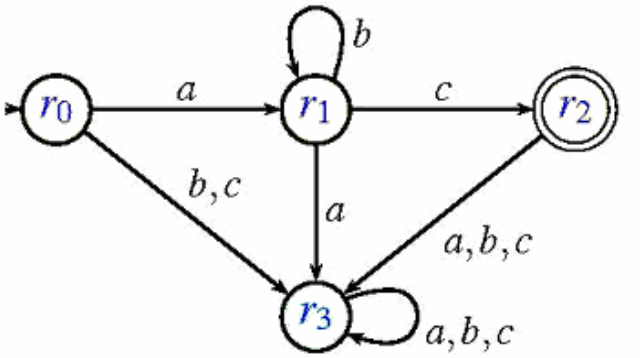


Figura 9: Autómata Finito no Deterministico

Este es un autómata no-determinístico ya que hay transiciones no definidas. En general, los autómatas no-determinísticos son más simples que los determinísticos.

(Reconocimiento de Patrones).-Supongamos un ejemplo de tranmisión de datos

entre barcos. El receptor de un barco debe de estar siempre esperando la trasmisión de datos que puede llegar en cualquier momento. Cuando no hay trasmisión de datos hay un ruído de fondo (sucesión aleatoria de 0, 1). Para comenzar la trasmisión se manda una cadena de aviso, por ejemplo. 010010. Si esa cadena se reconoce hay que registrar los datos que siguen. El programa que reconoce esta cadena puede estar basado en un autómata finito. La idea es que este no pueda llegar a un estado no final mientras no se reciba la cadena inicial. En ese momento el autómata pasa a un estado final. A partir de ahí todo lo que llegue se registra. el propósito es hacer un autómata que llegue a un estado final tan pronto como se reconozca 010010. Intentar hacer un autómata finito determinístico directamente puede ser complicado, pero es muy fácil el hacer un AFND, como el de la Figura 10.

Hay que señalar que esto sería solamente el esquema de una sola parte de la trasmisión. Se podría complicar incluyendo también una cadena para el fin de la trasmisión.

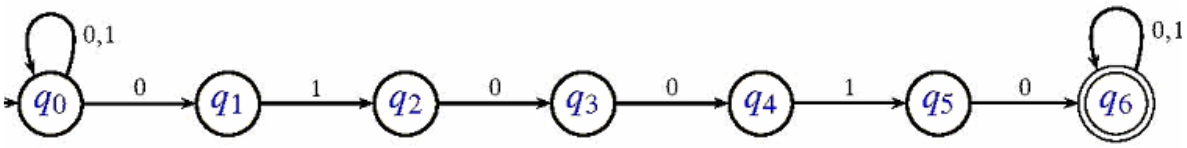


Figura 10: Autómata No-Deterministico que reconoce la cadena 010010.

Ejemplo: Usando los automatas No-Deterministicos se puede simplificar el ejercicio de la figura 7 y representar con más claridad la estructura de la expresión regular (11 + 110)*0 se presentan las 2 figuras para verificar la similitud.

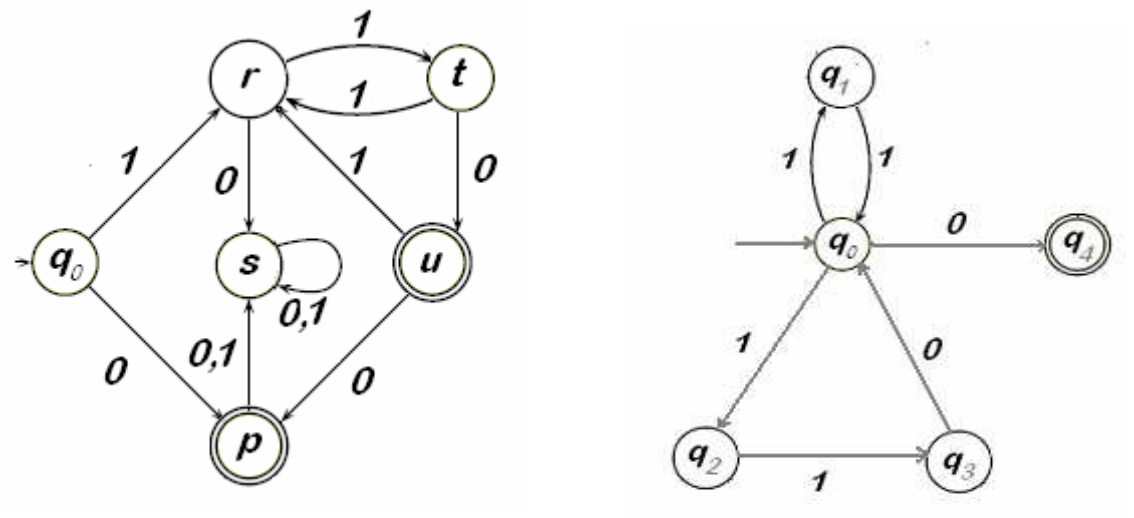


Figura 11: Automata finito y automata finito no deterministico

LECCION 9. - AUTOMATA FINITO CON λ-TRANSICIONES.

Un automata finito con λ-transiciones (AFND-λ) es un AFND al que se le permite cambiar de estado sin necesidad de consumir un símbolo de entrada. La función se define como:

$\Delta: Q \times V \cup \{\lambda\} \rightarrow P(Q)$

La tabla de transicion de un AFND-λ es como la de un AFND excepto que se le añade una columna correspondiente a λ

Ejemplo AFND-λ cuyo diagrama de transicion es:

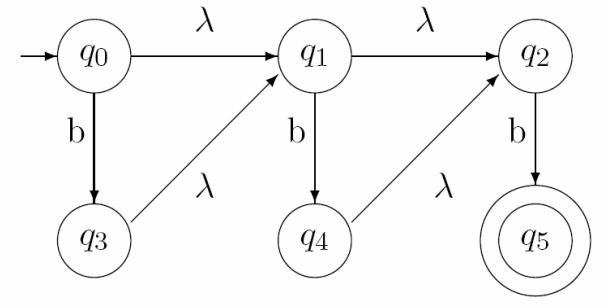


Figura 12: AFND-λ

En el instante actual está en q_1 y lee b:
en el instante siguiente, el autómata puede decidir de forma no determinista entre:
"leer el símbolo b y cambiar al estado q_4"
"cambiar al estado q_2 sin mover el cabezal de lectura"

El conjunto de cadenas que es capaz de aceptar este automata es {b, bb, bbb}

LECCION 10. - LENGUAJE ACEPTADO POR UN AF

Un autómata finito sirve para reconocer cierto tipo de lenguajes. Antes de definir formalmente el concepto de lenguaje aceptado por un AF necesitamos definir los conceptos de configuración y cálculo en un autómata finito.

La configuración de un autómata finito (sin importar el tipo) en cierto instante viene dada por el estado del autómata en ese instante y por la porción de cadena de entrada que le queda por leer o procesar. La porción de cadena leída hasta llegar al estado actual no tiene influencia en el comportamiento futuro de la máquina. En este sentido podemos decir que un AF es una máquina sin memoria externa; son los estados los que resumen de alguna forma la información procesada.

Formalmente una configuración de un AF es un elemento (q,w) $\in$ (Q × V*). Algunos tipos de configuraciones especiales son:

Configuración inicial : (q_0,w), donde q_0 es el estado inicial y w la palabra de entrada.

Configuración de parada: cualquier configuración en la que el autómata puede parar su ejecución, bien porque se haya procesado toda la entrada o bien porque se haya llegado a una situación donde no es aplicable ninguna transición.

Configuración de aceptación: (q_F , λ), donde q_F es un estado final del autómata.

Una vez alcanzada esta configuración el autómata puede aceptar la palabra.

Si consideramos el conjunto de las configuraciones de un autómata finito, podemos definir una relación binaria $\vdash \subseteq (Q \times V^*) \times (Q \times V^*)$ que llamaremos relación de cálculo en un paso.

Intuitivamente si dos configuraciones C_i y C_j están relacionadas mediante la relación $\vdash$ y lo notamos como Ci $\vdash$ Cj , quiere decir que podemos pasar de la configuracin C_i a la C_j aplicando una sola transición y diremos que “la configuración C_i alcanza en un paso la configuración C_j”.

Para definir formalmente la relación de cálculo en un paso $\vdash$, distinguiremos tres casos correspondientes a los tres tipos de autómatas que hemos visto:

Si tenemos un AFD, la relación de cálculo en un paso se define de la siguiente forma:

$$(q,w) \vdash (q',w') \Leftrightarrow \begin{cases} w = aw', \text{ donde } a \in V \\ q' = \delta(q,a) \end{cases}$$

Si tenemos un AFND, la relación de cálculo en un paso la se define:

$$(q,w) \vdash (q',w') \Leftrightarrow \begin{cases} w = aw', \text{ donde } a \in V \\ q' \in \Delta(q,a) \end{cases}$$

Si tenemos un AFND-λ, la relación de cálculo en un paso se define:

$$(q,w) \vdash (q',w') \Leftrightarrow \begin{cases} w = \sigma w', \text{ donde } \sigma \in V \cup \{\lambda\} \\ q' \in \Delta(q,\sigma) \end{cases}$$

Cuando queramos distinguir el autómata M al que refiere la relación, se usaría $\vdash_M$.

La clausura reflexiva y transitiva de la relación $\vdash$ es otra relación binaria

$\vdash^* \subseteq (Q \times V^*) \times (Q \times V^*)$, que llamaremos relación de cálculo. Diremos que la "configuración C_i alcanza (en cero o más pasos) la configuración C_j", y lo notamos como $Ci \vdash^* Cj$, si se cumple una de las dos condiciones siguientes:

1. $C_i = C_j$, o bien,
2. $\exists C_0, C_1, \ldots C_n$, tal que $C_0 = C_i$, $C_n = C_j$, y $\forall\ 0 \leq k \leq n-1$ se cumple que $Ck \vdash Ck+1$

A una secuencia del tipo $C_0 \vdash C_1 \vdash \ldots \vdash C_n$ la llamaremos cálculo en n pasos, abreviadamente $C_1 \vdash^*_{\text{n pasos}} C_n$.

Ejemplo: Considerando el AFD de la figura 6 podemos decir que $(q_0, 01) \vdash (q_0, 1)$, $(q_0, 1) \vdash (q_1, \lambda)$ y por tanto $(q_0, 01) \vdash^* (q_1, \lambda)$. También $(q_1, 101) \vdash (q_2, 01)$ y en varios pasos $(q_2, 0011) \vdash^* (q_1, 1)$.

Por otra parte para el AFND de la figura 8 tenemos, por ejemplo, que $(q_0, abb) \vdash (q_0, bb)$ y también $(q_0, abb) \vdash (q_3, bb)$. Al ser el autómata no determinista vemos que a partir de una misma configuración, en este caso (q_0, abb), se puede llegar en un paso de cálculo a dos o más configuraciones distintas. Esta situación no puede producirse en un AFD.

Para el AFND-λ de la figura 12 el cálculo $(q_1, bb) \vdash (q_2, bb)$ es un ejemplo donde se produce una transición que implica un cambio de estado sin consumir símbolos de entrada. Esto es posible porque $q_2 \in \Delta(q_1, \lambda)$.

Si tenemos un autómata finito $M = (Q, V, \delta, q_0, F)$, se define el lenguaje aceptado por M y lo notamos L(M), como:

$L(M) = \{w \in V^* \mid (q_0, w) \vdash^* (q_F, \lambda) \text{ donde } q_F \in F\}$

Es decir, una palabra w Será aceptada por el autómata M, si partiendo de la configuración inicial con w en la cinta de entrada, el autómata es capaz de alcanzar una configuración de aceptación. Dependiendo del tipo de autómata de que se trate, $\vdash^*$ hará referencia a la clausura reflexiva y transitiva de la relación $\vdash$ en un AFD, en un AFND o en un AF con λ-transiciones.

En un autómata finito determinista, el hecho de que una palabra w sea aceptada por el autómata nos asegura que existe un único camino en el diagrama de transición que nos lleva del nodo etiquetado con el estado inicial al nodo etiquetado con el estado final y cada arco que se recorre en este camino ésta etiquetado con un símbolo de la palabra. Podríamos simular la ejecución de un autómata finito determinista mediante un programa que codifique la función de transición y simule los cambios de estado. Si |w| = n entonces el programa puede determinar si la palabra es aceptada o no en O(n).

En el caso de un AFND o un AFND-λ no podemos asegurar que exista un único camino en el diagrama que nos lleve del estado inicial a un estado final consumiendo los símbolos de la palabra. Incluso puede que para una palabra $w \in L(M)$ podamos tener una camino que no acabe en estado final o que llegue a un estado desde el que no se pueda seguir leyendo símbolos. Esto es debido al no determinismo, que hace que los cálculos en estos autómatas no estén perfectamente determinados. Si quisiéramos simular un autómata no determinista para decidir si una palabra es aceptada o no, tendríamos que usar alguna técnica de retroceso o backtracking para explorar distintas posibilidades hasta encontrar un cálculo correcto que reconozca la palabra o determinar que la palabra no es aceptada si se han explorado todos los posibles cálculos y ninguno de ellos conduce a un estado final. Esto nos llevaría a un algoritmo de tiempo exponencial para reconocer una palabra. De ahí que a efectos prácticos, como en la construcción de analizadores léxicos o reconocimiento de patrones en un texto, lo deseable es tener un autómata finito determinista.

Ejemplo Recordemos los AFs ya vistos y veamos ahora cual es el lenguaje aceptado por ellos. El diagrama de la figura 8 correspondiente a un AFND permite ver que L(M) es el lenguaje descrito por la expresión regular (a + b)*(aa + bb)(a + b)* que consiste en aquellas cadenas sobre el alfabeto V = {a, b} que contienen al menos una ocurrencia de la subcadena aa o bb. Por ejemplo, la cadena abb es aceptada, ya que tenemos el cálculo:

$(q_0, abb) \vdash (q_0, bb) \vdash (q_1, b) \vdash (q_2, \lambda)$, y $q_2 \in F$

Sin embargo podemos tener otro cálculo que no conduce a estado final:

$(q_0, abb) \vdash (q_0, bb) \vdash (q_0, b) \vdash (q_1, \lambda)$, $q_1 \notin F$

e incluso un cálculo que no llega a consumir la palabra:

$(q_0, abb) \vdash (q_3, bb) \vdash$ (y no puede seguir)

A partir del diagrama del AFD de la figura 6 no es tan sencillo ver cual es el lenguaje aceptado. Pero, según se vera mas adelante con el teorema de kleene se tiene un método exacto para encontrar este lenguaje. En este caso el lenguaje aceptado es el descrito por la expresión regular $(0 + 1\ (10^*1)^*)^*\ 1\ (10^*1)^*$

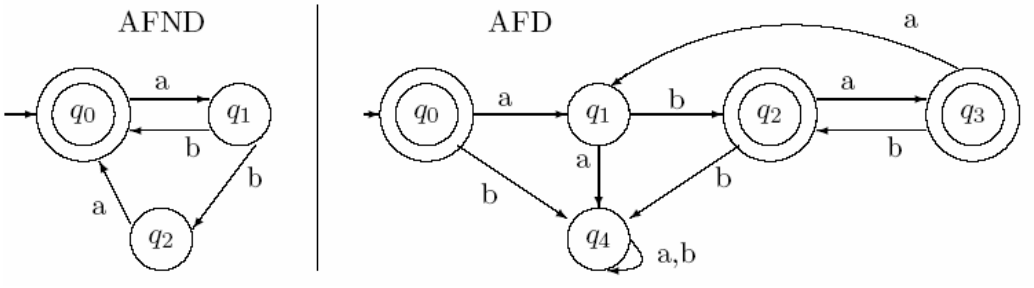


Figura 13: AFs que aceptan L(α) donde α = (ab + aba)□

EXPRESIONES Y LENGUAJES REGULARES

LECCION 11. - EXPRESIONES REGULARES[9]

La notación de conjuntos nos permite describir los lenguajes regulares, pero se utiliza una notación en que las representaciones de los lenguajes son simplemente
texto (cadenas de caracteres). Así las representaciones de los lenguajes regulares son simplemente palabras de un lenguaje (el de las representaciones correctamente formadas). Con estas ideas se va a definir un lenguaje, el de las expresiones regulares, en que cada palabra va a denotar un lenguaje regular.

Asi mismo una expresión regular es la representación de la cadena más característica del lenguaje respectivo, por ejemplo 1*10 es una cadena consistente en la subcadena 10 precedida de cualquier numero de unos.

La definición de expresión regular es en realidad un poco más restringida en varios aspectos de los que se necesita en la práctica, se usa una notación como L^2 para lenguajes y es razonable denotar en forma similar las expresiones regulares, entonces en ocasiones se escribe (r^2) para indicar la expresión regular (rr), (r^+) para la expresión regular $((r^*)r)$, y asi sucesivamente.

Una expresión regular R para un alfabeto $\sum$ se define como sigue:
ϕ y λ son expresiones regulares.
A es una expresión regular para todo a $\in \sum$.
Si a y b son expresiones regulares, entonces a $\cup$ b, a•b, a* y b* son expresiones regulares.
Ninguna otra secuencia de símbolos de $\sum$ es una expresión regular.

Sea $\sum$ un alfabeto. El conjunto ER de las expresiones regulares sobre $\sum$ contiene las cadenas en el alfabeto $\sum \cup$ {"^", "+", "•", "*", "(", ")", "ϕ"} que cumplen con lo siguiente:

1. "^" y "ϕ" $\in$ ER
2. Si $\sigma \in \sum$, entonces $\sigma \in$ ER.
3. Si $E_1, E_2 \in$ ER, entonces "("E_1"+"E_2")" $\in$ ER, "("E_1"•"E_2")" $\in$ ER, "("E_1")*" $\in$ ER.

Las comillas " " enfatizan el hecho de que estamos definiendo cadenas de texto,

[9] BRENA PIÑERO, Ramon F. Autómatas y lenguajes un enfoque de diseño (2003) ITESM, En: http://lizt.mty.itesm.mx/~rbrena/AyL.html

no expresiones matemáticas (es el caso de las expresiones de conjuntos para describir los conjuntos regulares.) Es la misma diferencia que hay entre el caracter ASCII “0”, que se puede teclear en una terminal, y el número 0, que significa que se cuenta un conjunto sin ningún elemento.

Ejemplo:
Son ER : en {a, b, c} las siguientes: “a”, “((a+b))*”, “((a •b) •c)”.
No son ER : “ab”, “((a • b(c)*)”.

LECCION 12. - SIGNIFICADO DE LAS EXPRESIONES REGULARES

Las ER son simplemente fórmulas cuyo propósito es representar cada una de ellas un lenguaje. Así, el significado de una ER es simplemente el lenguaje que ella representa.

Por ejemplo, la ER “ϕ” representa el conjunto vacío $\{\}$.

Para comprender intuitivamente la manera en que las ER representan lenguajes, consideremos el proceso de verificar si una palabra dada w pertenece o no al lenguaje representado por una ER dada. Vamos a decir que una palabra “empata” con una expresión regular si es parte del lenguaje que esta representa.

La palabra vacía "ε “empata” con la ER ^.

Una palabra de una letra como “a” empata con una ER consistente en la misma letra “a”, “b” empata “b”, Luego, una palabra w = uv, esto es w está formada de dos pedazos u y v, empata con una expresión (U • V) a condición de que u empate con U y v empate con V.

Por ejemplo, abc empata con (a • (b • c)) porque abc puede ser dividida en a y bc, y a empata con a en la ER, mientras que bc empata con (b • c) separando b y c de la misma manera.

Similarmente, cuando la ER es de la forma (U + V), puede empatar con una palabra w cuando esta empata con U o bien cuando empata con V . Por ejemplo, bc empata (a+(b•c)).

Una palabra w empata con una expresión U* cuando w puede ser partida en pedazos w = w1w2, . . . de tal manera que cada pedazo wi empata con U.

Por ejemplo, caba empata con (((c + b) • a))_ porque puede partirse en los pedazos ca y ba, y ambos empatan con ((c + b) • a), lo cual es fácil de verificar.

Para simplificar la especificación de un lenguaje se utilizan las expresiones regulares y por esto conviene escribir a en lugar de {a}:
a $\cup$ b denota {a,b} = {a} $\cup$ {b}
ab denota {a,b}
a* denota {a}*
a+ denota {a}+

El orden de precedencia de los operadores *, $\cup$ y $\bullet$ es: Primero *; luego $\bullet$ y por último $\cup$.

Por ejemplo, una expresión dada por ({a}*{b})$\cup${c} se reduce a a*b$\cup$c.

Otro ejemplo: El lenguaje de todas las cadenas sobre {a,b,c} que no tienen ninguna subcadena ac se denota por c*(a $\cup$ bc*)*.

Esta definición nos permite construir expresiones en la notación de conjuntos que representan lenguajes regulares.

Ejemplo.- Sea el lenguaje L de palabras formadas por a y b, pero que empiezan con a, como aab, ab, a, abaa, etc. Probar que este lenguaje es regular, y dar una expresión de conjuntos que lo represente.

Solución.- El alfabeto es $\sum$ = {a, b}. El lenguaje L puede ser visto como la concatenación de una a con cadenas cualesquiera de a y b; ahora bien, éstas últimas son los elementos de {a, b}*, mientras que el lenguaje que sólo contiene la palabra a es {a}. Ambos lenguajes son regulares.

Entonces su concatenación es {a}{a, b}*, que también es regular.

NOTA: La concatenación de dos lenguajes L1 y L2 se define como el conjunto de las palabras formadas concatenando una de L1 con una de L2.

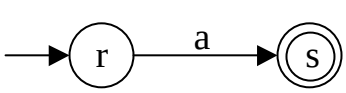
{a} es finito, por lo tanto regular, mientras que {a, b}* es la cerradura de {a, b}, que es regular por ser finito.

LECCION 13. - AUTÓMATAS FINITOS Y EXPRESIONES REGULARES[10]

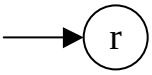
Hasta ahora, la relación entre los autómatas finitos y las expresiones regulares se ha tratado de una manera intuitiva. Ahora formalizaremos dicha relación.

[10] IOST F. Hans, Teoría de autómatas y lenguajes formales, CAPÍTULO 2, LENGUAJES REGULARES Y AUTOMATAS FINITOS (2001) en: http://iie.ufro.cl/~hansiost/automatas/Capitulo2.doc

Para un alfabeto Σ, se pueden construir los AFND (y los AFD) que acepten palabras unitarias. Por ejemplo:



(a)



(b)

El autómata de la Figura (a) acepta el lenguaje unitario {a}. El autómata de la Figura (b) acepta el lenguaje ϕ.

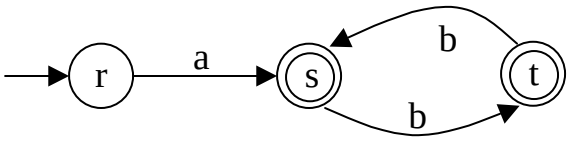
Si $M_1 = (Q_1,\Sigma_1,So_1,F_1,\delta_1)$ y $M_2 = (Q_2,\Sigma_2,So_2,F_2,\delta_2)$ son AFND, podemos unir M1 y M2 en un nuevo AFND que acepte $L(M_1) \cup L(M_2)$, añadiendo un nuevo estado inicial **So** y 2 transiciones λ, una de **So a So_1** y otra de **So a So_2**.Este nuevo AFND estará dado por:

M = (Q,Σ,So,F,δ), donde

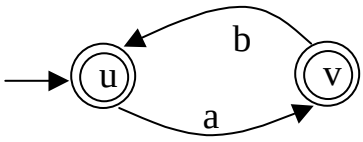
$\Sigma = \Sigma_1 \cup \Sigma_2$; $F = F_1 \cup F_2$; $Q = Q_1 \cup Q_2 \cup \{So\}$

y δ se define de tal forma que se incluyan todas las transiciones de δ1, δ2 y las transiciones λ de So a So1 y So2, es decir: $\delta = \delta 1 \cup \delta 2 \cup \{(So,\lambda,So1), (So,\lambda,So2)$

Ejemplo: Considere los siguientes AFND:

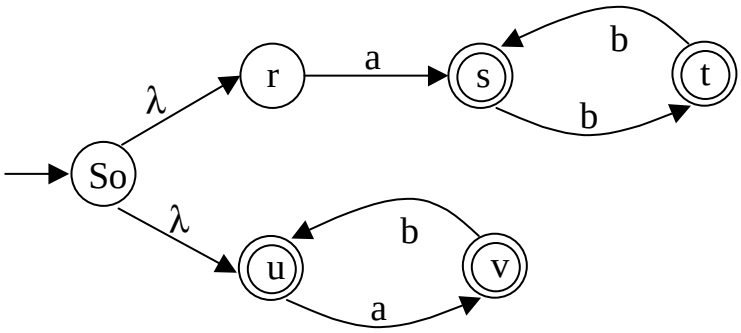


AFND que acepta ab*



AFND que acepta (ab)*

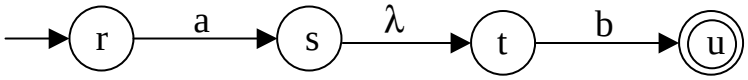
Ambos AFND se pueden unir de manera tal que el nuevo AFND acepte ab* ∪ (ab)* :



Por otra parte si tenemos 2 autómatas M1 y M2, podemos unirlos para formar un AFND que acepte L(M1) • L(M2). Se requiere un autómata que reconozca una cadena de L(M1) y a continuación una de L(M2). Esto se realiza pasando del estado final de M1 al estado inicial de M2 a través de una transición λ.

Por ejemplo:

Aceptan los lenguajes {a} y {b} respectivamente. Así, el autómata que acepta el lenguaje {ab} es:



Es importante resaltar que el autómata resultante tendrá como estado inicial, el estado inicial de M_1 y como conjunto de estados finales, el conjunto de estados finales de M_2. Por lo tanto el AFND que acepta $L(M_1) \bullet L(M_2)$ está dado por:

$Q = Q_1 \cup Q_2$
$So = So_1$
$F = F_2$
$\delta = \delta_1 \cup \delta_2 \cup (F_1 \times \{\lambda\} \times \{So_2\})$

La función de transición δ resultante, incluirá todas las transiciones presentes en ambos autómatas junto con todas las ternas de la forma (q,λ,So_2), donde q es un estado de aceptación de M_1, es decir $So_2 \in \delta(q,\lambda)$ para todo $q \in F_1$.

Por otra parte se puede deducir un procedimiento para construir un AFND que acepte **L(M)*** para un AFND M = (Q, Σ, So, F, δ), como sigue:

Primero, se agrega un nuevo estado inicial **s'**; además, este nuevo estado será de aceptación, con el fin de que se acepte λ. Se agrega una transición λ desde **s'** al antiguo estado inicial de M, So. Se agregará, además, una transición λ desde todos los estados de aceptación de M hasta **s'**. El autómata resultante será:

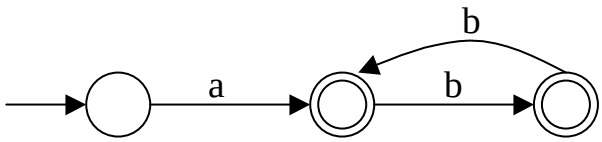
$M' = (Q', \Sigma, s', F', \delta')$

donde:

$Q' = Q \cup \{s'\}$
$F' = \{s'\}$
$\delta' = \delta \cup \{(s',\lambda,So)\} \cup (F \times \{\lambda\} \times \{s'\})$

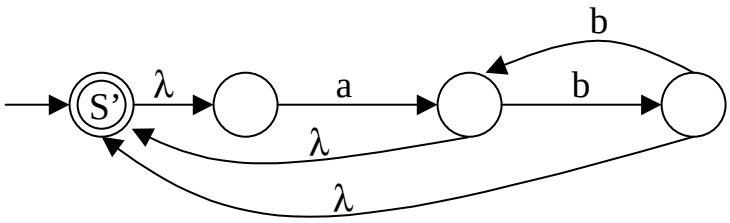
Ejemplo: Sea el AFND M que acepta el lenguaje dado por ab*:

M:



L(M)* se construye aplicando el procedimiento anterior:

M':



L(M') : (ab*)*

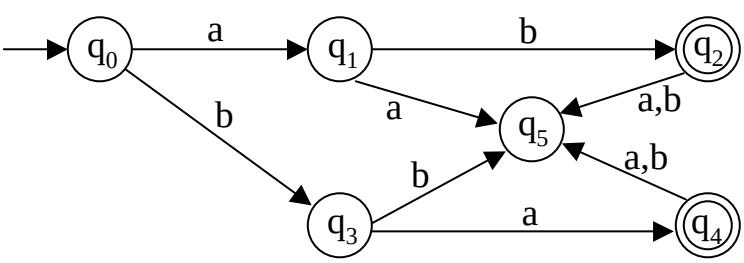
Teorema: El conjunto de lenguajes aceptados por un autómata finito sobre el alfabeto $\sum$ contiene el lenguaje ϕ y los lenguajes unitarios {a} para todo a $\in \sum$. Este conjunto es cerrado con respecto a la unión, concatenación y cerradura de estrella.

Este teorema implica que todo lenguaje regular es aceptado por un autómata finito.

Sea el autómata finito M = (Q, $\sum$,So,F,δ) y supongamos que So = q_0 es el estado incial. Se define:

$$A_i = \{w \in \sum^* \mid \delta e(q_i,w) \cap F \neq \phi\}$$

A_i es el conjunto de las cadenas sobre $\sum^*$ que hacen que M pase desde q_i hasta un estado de aceptación. Se dice que A_i es el conjunto de las cadenas aceptadas por el estado q_i. Es importante notar que A_0 = L(M). Además, es posible que $A_i = \phi$. Si $q_i \in$ F, entonces $\lambda \in$ Ai. Como ejemplo, considérese el siguiente AFND:



$A_5 = \phi$
$A_2 = \lambda$
$A_4 = \lambda$
$A_1 = b$
$A_3 = a$
$A_0 = ab \cup ba$

Del autómata anterior se deduce:

$\delta(q_0,a) = \{q_1\}$, es decir $q_1 \in \delta(q_0,a)$, por lo tanto A_0 contiene a: $a \bullet A_1$. En general, si: $q_j \in \delta(q_i,w)$ implica que A_i contiene a wA_j. De hecho se tiene que:

$$A_i = \bigcup \{wA_j \mid q_j \in \delta(q_i,w)\}$$

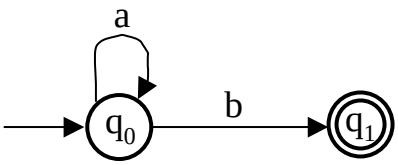
Esto proporciona las técnicas básicas para obtener una expresión regular a partir de un autómata finito. Consideremos el ejemplo anterior, tenemos:

$A_0 = a\ A_1 \cup b\ A_3$ $\qquad$ $A_3 = a\ A_4 \cup b\ A_5$
$A_1 = b\ A_2 \cup a\ A_5$ $\qquad$ $A_4 = \lambda \cup a\ A_5 \cup b\ A_5$
$A_2 = \lambda \cup a\ A_5 \cup b\ A_5$ $\qquad$ $A_5 = \phi$

Así, tenemos un sistema de ecuaciones que se cumplen para L(M). Sustituyendo se obtiene:

$$L(M) = ab \cup ba = A_0$$

Considérese ahora el siguiente autómata:



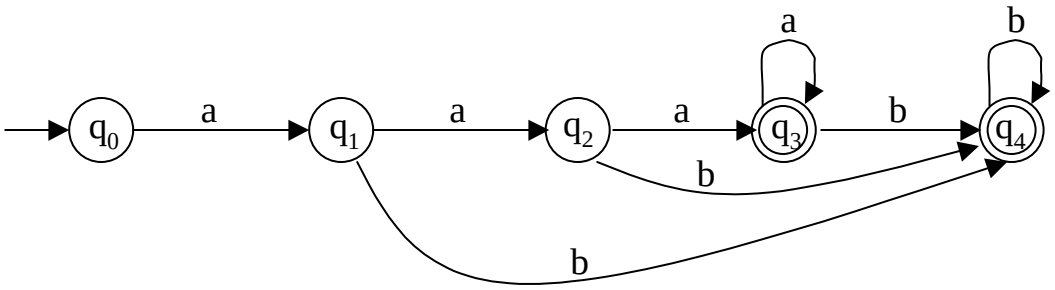
$\rightarrow A_0 = a\ A_0 \cup b\ A_1$; $A_1 = \lambda$

Resolviendo y sustituyendo resulta que $A_0 = a\ A_0 \cup b$, y no es posible simplificarlo más. El siguiente lema muestra como resolver este problema:

Lema de Arden: Una ecuación de la forma $X = AX \cup B$, donde $\lambda \in A$ tiene una solución única $X = A^*B$.

En el ejemplo anterior, al aplicar el lema de Arden queda $A_0 = a^*b$, que es lo que intuitivamente se deduce del diagrama.

Ejemplo: Considérese el siguiente autómata:



$A_0 = aA_1$; $A_1 = aA_2 \cup bA_4$; $A_2 = aA_3 \cup bA_4$

$A_3 = \lambda \cup aA_3 \cup bA_4$; $A_4 = \lambda \cup bA_4$

Sustituyendo y aplicando el lema de Arden, tenemos:

$A_4 = \lambda \cup bA_4 = bA_4 \cup \lambda = b^*\lambda = b^*$ (por lema de Arden)

$A_3 = \lambda \cup aA_3 \cup bb^* = aA_3 \cup \lambda \cup b^+ = a^*(\lambda \cup b^+) = a^*b^*$

$A_2 = aa^*b^* \cup bb^* = a^+b^* \cup b^+$

$A_1 = a(a^+b^* \cup b^+) \cup bb^* = aa^+b^* \cup ab^+ \cup b^+$

$A_0 = aA_1 = a(aa^+b^* \cup ab^+ \cup b^+)$

$A_0 = a^2a^+b^* \cup a^2b^+ \cup ab^+$

Lema: Sea M un autómata finito. Entonces existe una expresión r para la cual:

$$L(r) = L(M)$$

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

Podemos suponer que el autómata finito M no tiene λ-transiciones (si las tuviera ya sabemos que podemos encontrar autómata equivalente sin λ-transiciones). Sea $M = (Q, V, \Delta, q_0, F)$. A partir de este autómata podemos obtener un sistema de

ecuaciones de expresiones regulares que llamaremos ecuaciones características del autómata. Estas ecuaciones se obtienen a partir del diagrama de transición del autómata del siguiente modo:

A cada nodo qi le corresponde una ecuación y cada estado se puede considerar como una incógnita de la ecuación.

La ecuación para el estado qi tiene en el primer miembro el estado qi y el segundo miembro de la ecuación está formado por una suma de términos, de forma que por cada arco del diagrama de la forma $q_i \xrightarrow{a} q_j$ tenemos un término aq_j . Si el estado q_i es final, añadimos además el termino λ al segundo miembro.

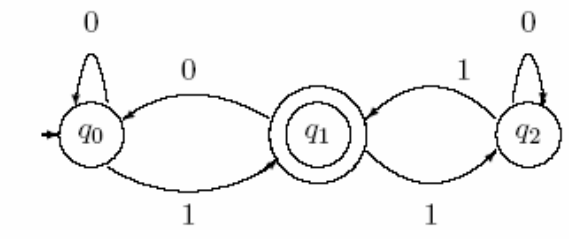
Cada incógnita qi del sistema de ecuaciones representa el conjunto de palabras que nos llevan del nodo qi a un estado final, en el diagrama de transición. Por tanto, si resolvemos el sistema de las ecuaciones características del autómata tendremos soluciones de la forma qi = α_i, donde α_i es una expresión regular sobre el alfabeto V y como hemos dicho el lenguaje descrito por esta expresión regular es:

$L(\alpha_i) = \{w \in V^* \mid (q_i,w) \vdash^* (q_F , \lambda) , q_F \in F\}$ (3.1)

El método que se propone para obtener una expresión regular α a partir de un AF es el siguiente:
1. Obtener las ecuaciones características del autómata;
2. Resolver el sistema de ecuaciones;
3. α ← solución para el estado inicial;

Para comprobar que este método es válido tendríamos que probar que se cumple L(αi) = {w ∈ V* | (qi,w) ⊢* (qF , λ) , qF ∈ F} (3.1) para toda solución qi = αi del sistema de ecuaciones, y en particular la solución para el estado inicial es la expresión regular correspondiente al autómata. Aunque no lo vamos a demostrar formalmente, por la forma de obtener las ecuaciones características del autómata y la validez del método de resolución de sistemas de ecuaciones de expresiones regulares. En realidad, no es necesario resolver todas las incógnitas, sólo necesitamos despejar la incógnita correspondiente al estado inicial.

Ejemplo Consideremos de nuevo el autómata finito M



Las ecuaciones características correspondientes a este autómata son:

q0 = 0q0 + 1q1
q1 = 0q0 + 1q2 + λ
q2 = 1q1 + 0q2

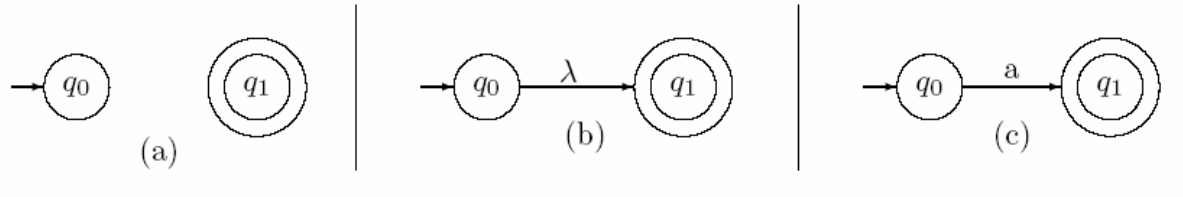
Comenzando por la última ecuación se tiene que q2 = 0□1q1 y sustituyendo en la segunda ecuación queda q1 = 0q0 + 10□1q1 + λ
de donde se obtiene que q1 = (10□1)□ (0q0 + λ) y este valor se sustituye en la primera ecuación q0 = 0q0 + 1 (10□1)□ (0q0 + λ) = 0q0 + 1 (10□1)□ 0q0 + 1 (10□1)□

Esta ecuación es fundamental y por el lema de Arden tiene como única solución
q0 = (0 + 1 (10□1)□ 0)□ 1 (10□1)□
y por tanto (0 +1 (10□1)□ 0)□1(10□1)□ es la expresión regular que describe el lenguaje L(M).

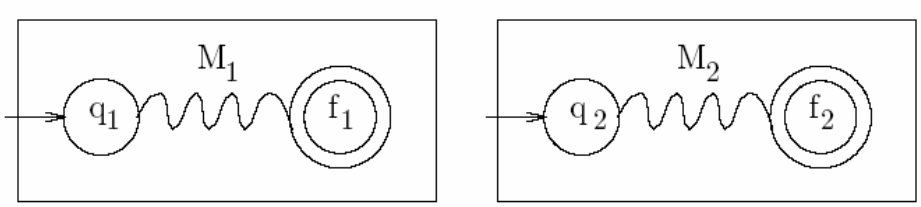
Teorema de Síntesis de Kleene

Si L es un lenguaje asociado a una expresión regular α entonces existe un autómata finito M tal que L = L(α) = L(M).
Demostrar por inducción sobre el número de operadores de α (+, □, □) que existe un AFND-λ M con un sólo estado final sin transiciones y distinto del estado inicial, de forma que L(α) = L(M).

Base.- (cero operadores) α puede ser: ∅, λ, a, donde a ∈ V. Los autómatas que aceptan el lenguaje vacío, el lenguaje {λ} y el lenguaje {a}, son, por este orden, los siguientes: (a), (b) y (c)



Inducción.- (uno o más operadores en α). Supongamos que se cumple la hipótesis para expresiones regulares de menos de n operadores. Sean las expresiones regulares α1 y α2 donde op(α1), op(α2) < n. Entonces, por hipótesis existen dos autómatas finitos M1 y M2 tal que L(M1) = L(α1) y L(M2) = L(α2),
donde M1 = (Q1, V1,Δ1, q1, {f1}) yM2 = (Q2, V2, Δ2, q2, {f2})
y podemos suponer sin pérdida de generalidad que Q1 ∩ Q2 = ∅. Estos autómatas podemos representarlos esquemáticamente como:

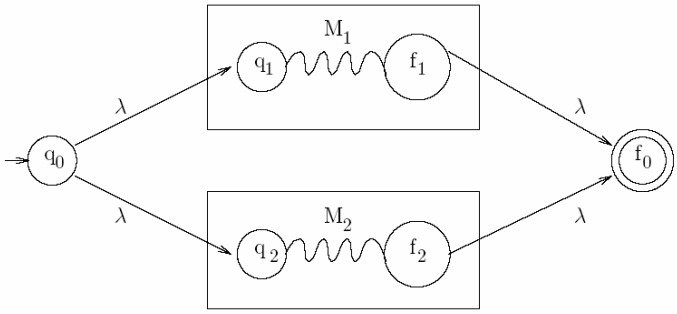


Supongamos que tenemos una expresión regular α con n operadores. Vamos a construir un autómata M tal que L(M) = L(α) y para eso distinguimos tres casos correspondientes a las tres formas posibles de expresar α en función de otras expresiones regulares con menos de n operadores.

$\alpha = \alpha 1 + \alpha 2$ tal que $op(\alpha 1), op(\alpha 2) < n$. Los autómatas correspondientes a α1 y α2 son respectivamente M1 y M2, como hemos dicho antes. A partir de M1 y M2 construimos otro autómata M = (Q1 □ Q2 □ {q0, f0} , V1 □ V2,Δ, q0, {f0}) donde Δ se define como:

a) Δ (q0, λ) = {q1, q2}
b) Δ (q, σ) = Δ1(q, σ), □ q □ Q1 − {f1}, σ □ V1 □ {λ}
c) Δ (q, σ) = Δ2(q, σ), □ q □ Q2 − {f2}, σ □ V2 □ {λ}
d) Δ (f1, λ) = Δ (f2, λ) = {f0}

M se puede representar gráficamente del siguiente modo:



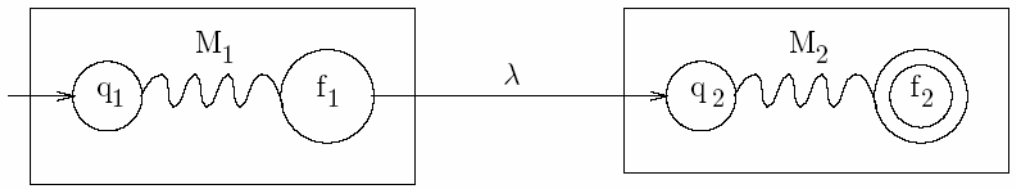
Cualquier camino de q_0 a f_0 debe pasar forzosamente a través del autómata M_1 o del autómata M_2. Si una cadena w es aceptada por M, entonces debe ser aceptada también por M_1 o por M_2.

Es decir, L(M) = L(M1) □ L(M2) =
= L(α1) □ L(α2), por hipótesis de inducción
= L(α1 + α2), por definición de lenguaje asociado a α1 + α2
= L(α), como queríamos demostrar.

2. $\alpha = \alpha 1 \square \alpha 2$ tal que $op(\alpha 1), op(\alpha 2) < n$. A partir de M1 y M2 construimos otro autómata
M = (Q1 □ Q2, V1 □ V2,Δ, q1, {f2}) donde Δ se define como:

a) Δ (q, σ) = Δ1(q, σ), □ q □ Q1 − {f1}, σ □ V1 □ {λ}
b) Δ (f1, λ) = {q2}
c) Δ (q, σ) = □2(q, σ), □ q □ Q2, σ □ V2 □ {λ}

M se puede representar esquemáticamente como:



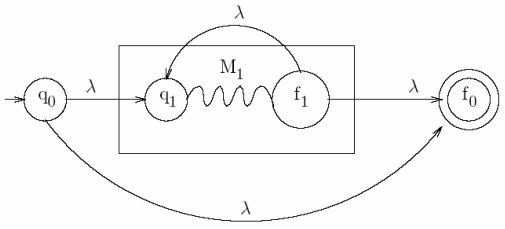
Cualquier camino de q1 a f2 debe pasar forzosamente a través del autómata M1 y del autómata M2. Si una cadena w es aceptada por M, entonces esa cadena se puede descomponer como w = w1.w2, de forma que w1 debe ser aceptada por M1 y w2 por M2. Según esto, L(M) = L(M1) □ L(M2) =

= L(α1) □ L(α2), por hipótesis de inducción
= L(α1 □α2), por definición de lenguaje asociado a α1 □α2
= L(α), como queríamos demostrar.

3. α = (α1)□ tal que op(α1) = n−1. El autómata correspondiente a α1 es M1, a partir del cual construimos otro autómata M = (Q1 □ {q0, f0} ,V1,Δ, q0, {f0}) donde Δ se define como:

a) Δ (q0, λ) = Δ (f1, λ) = {q1, f0}
b) Δ (q, σ) = Δ1(q, σ), □ q □ Q1 − {f1}, σ □ V1 □ {λ}

M se puede representar del siguiente modo:

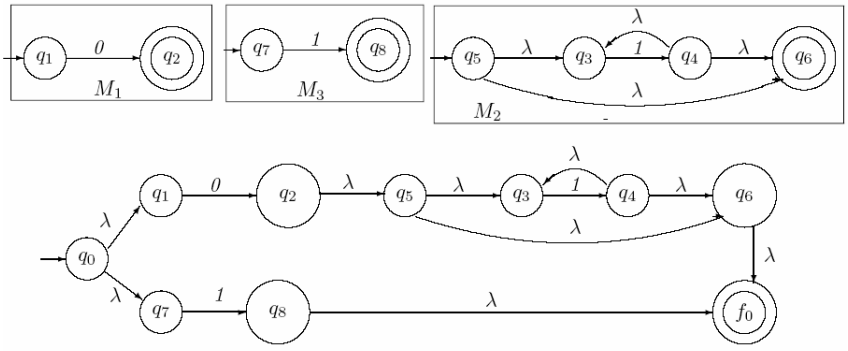


Este autómata acepta cadenas de la forma w = w1w2 . . . wj , donde j ≥ 0 y cada subcadena wi es aceptada por M1. Por tanto,

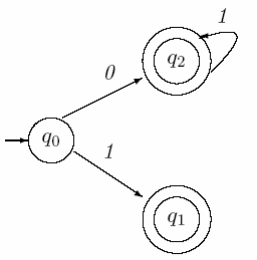
$$L(M) = \bigcup_{n=0}^{\infty} (L(M_1))^n =$$

$= \bigcup_{n=0}^{\infty} (L(\alpha_1))^n$, por hipótesis de inducción
= (L(α1))□ , por definición de clausura de un lenguaje
= L(α□₁), por definición de lenguaje asociado a α□
= L(α), como queríamos demostrar.

Ejemplo Siguiendo el método anterior, en la figura siguiente se ha construido un autómata para la expresión regular 01* +1, donde M_1 representa el autómata para la expresión regular 0, M_2 representa 1* y M_3 la expresión regular 1. En el autómata final se han integrado simultáneamente los autómatas para la concatenación (0 con 1*) y la suma de expresiones regulares 01*+1.



Este método de construcción de AFs para expresiones regulares está pensado para ser implementado de forma automática mediante un programa. Se puede haber pensado en el siguiente autómata:



LECCION 14. - PROPIEDADES DE LOS LENGUAJES REGULARES[11]

En los AFN es posible aplicar métodos modulares de diseño, que permiten manejar mejor la complejidad de los problemas. Son estos métodos modulares.

AFN para la unión de lenguajes

Si ya contamos con dos AFN, sean M_1 y M_2, es posible combinarlos para hacer un nuevo AFN que acepte la unión de los lenguajes que ambos autómatas aceptaban.

Sean $M_1 = (K_1, \Sigma_1, \Delta_1, s_1, F_1)$ y $M_2 = (K_2, \Sigma_2, \Delta_2, s_2, F_2)$ dos autómatas que aceptan los lenguajes L_1, L_2. Podemos entonces construir un AFN M_3 que acepte $L_1 \cup L_2$ de la siguiente manera: Sea q un nuevo estado que no está en K_1 ni en K_2. Entonces

[11] BRENA PIÑERO, Ramon F. Autómatas y lenguajes un enfoque de diseño (2003) ITESM, En: http://lizt.mty.itesm.mx/~rbrena/AyL.html

hacemos un autómata M_3 cuyo estado inicial es q, y que tiene transiciones vacías de q a s_1 y a s_2. Esta simple idea permite escoger en forma no determinista entre ir al autómata M_1 o a M_2, según el que convenga: si la palabra de entrada w está en L_1, entonces se escoge ir a M_1,,,,, y en contraposición a M_2 para L_2.

Formalmente $M_3 = (K_1 \cup K_2 \cup \{q\}, \Sigma_1 \cup \Sigma_2, \Delta_1 \cup \Delta_2 \cup \{(q, \varepsilon, s_1), (q, \varepsilon, s_2)\}, q, F_1 \cup F_2)$.
En la figura siguiente se representa gráficamente M_3.

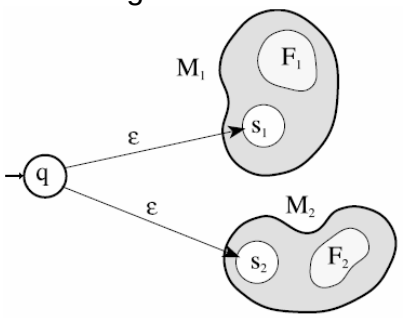


Figura 14: AFN para la unión de dos lenguajes

Ejemplo.- Diseñar un autómata no determinista que acepte las palabras sobre {a, b} que tengan un número par de a o que terminen en bb.

Solución.- En la figura 15(a) se presenta un AFN que acepta las palabras que contienen un número par de a's, y en 15(b) otro que acepta las palabras que terminan en bb. Finalmente, en 15(c) está el AFN que acepta el lenguaje dado.

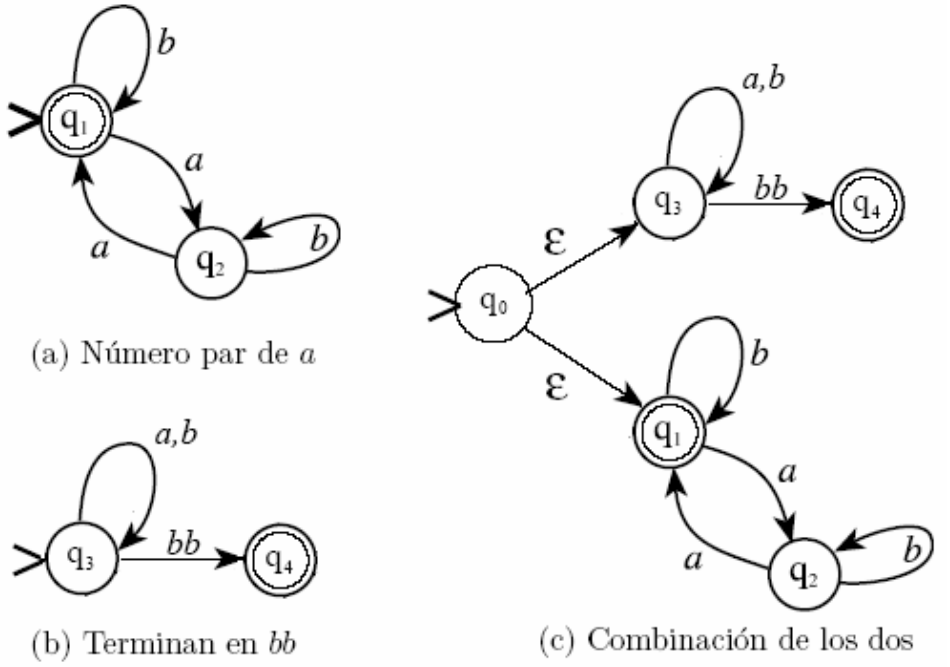


Figura 15: Unión de dos lenguajes, Combinación de AFN

AFN para la concatenación de lenguajes

Similarmente al caso anterior, sean $M_1 = (K_1,\Sigma_1,\Delta_1, s_1, F_1)$ y $M_2 = (K_2, \Sigma_2, \Delta_2, s_2, F_2)$ dos autómatas que aceptan los lenguajes L_1, L_2 respectivamente. Podemos entonces construir un AFN M_3 que acepte L_1L_2 de la siguiente manera: Añadimos unas transiciones vacías que van de cada uno de los estados finales de M1 al estado inicial de M_2; también se requiere que los estados finales de M1 dejen de serlo.
Formalmente $M_3 = (K_1 \cup K_2, \Sigma_1 \cup \Sigma_2, \Delta_1 \cup \Delta_2 \cup \{(p, \varepsilon, s_2) \mid p \in F_1\}, s1, F_2)$

El funcionamiento de M_3 es como sigue:

cuando se recibe una palabra $w = w_1w_2$, $w_1 \in L_1$, $w_2 \in L_2$, entonces se empieza procesando w_1 exactamente como lo haría M_1, hasta llegar hasta alguno de los antiguos estados finales de M_1; entonces se empieza procesando w_2 como lo haría M_2; forzosamente debe ser posible llegar a un estado final de M_2, ya que por hipótesis M_2 acepta w_2. En la figura 16 se representa M_3.

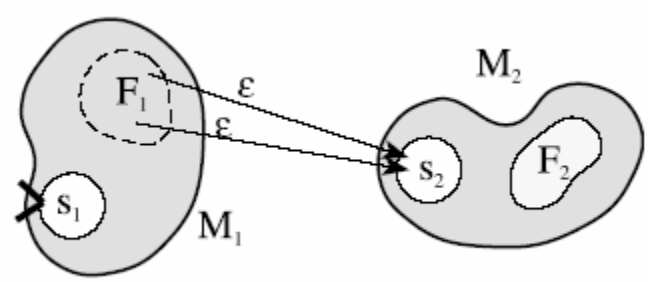


Figura 16: Concatanación de dos lenguajes,Combinación de AFNs

Ejemplo.- Construir un AFN que acepte el lenguaje en {a, b} donde las a's vienen en grupos de al menos dos seguidas, y los grupos de a's que son repeticiones de aaa están a la derecha de los que son repeticiones de aa, como en baabaaa, aaa, baab o baaaaa. Esta condición no se cumple, por ejemplo, en bbaaabaa ni en aaabaaaa.

Solución.- Un AFN, ilustrado en la figura 17(a), acepta palabras que contienen b's y grupos de aa en cualquier orden. Otro AFN –figura 17(b)– acepta un lenguaje similar, pero con grupos de aaa. La solución es su concatenación, que se presenta en la figura 17(c).

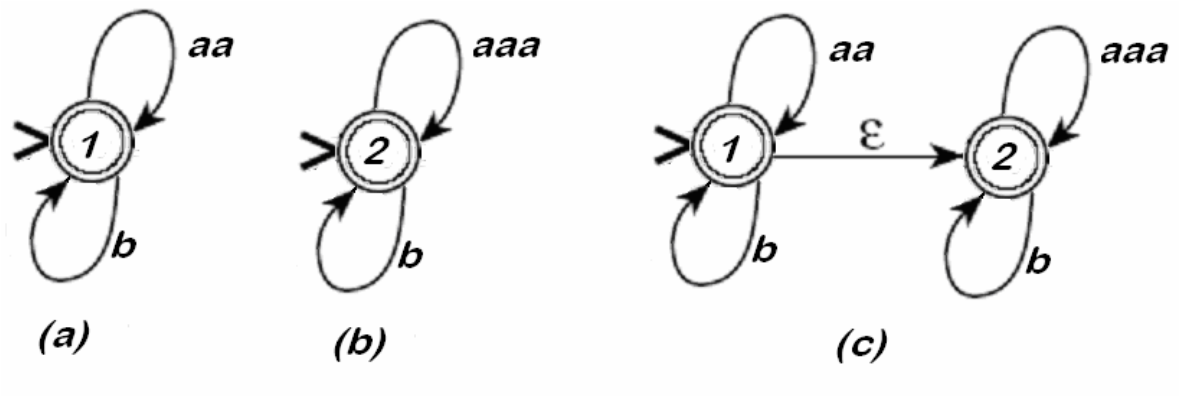


Figura 17: Concatenación de dos AFN

LECCION 15. - EQUIVALENCIA DE AUTÓMATAS FINITOS DETERMINISTICOS Y AUTÓMATAS FINITOS NO DETERMINÍSTICOS.

Los autómatas finitos determinísticos (AFD) son un subconjunto propio de los no determinísticos (AFN), lo que quiere decir que todo AFD es un AFN. Se puede pensar entonces que los AFN son “más poderosos” que los AFD, en el sentido de que habría algunos lenguajes aceptados por algún AFN para los cuales no hay ningún AFD que los acepte. Sin embargo, en realidad no sucede así.

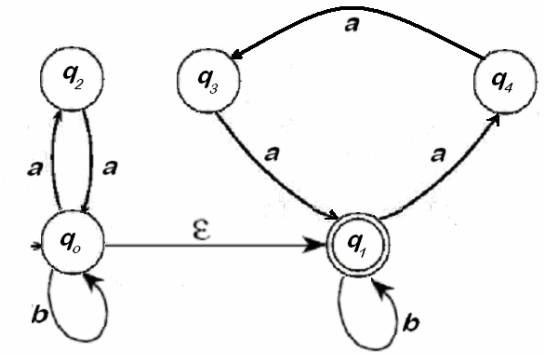


Figura 18: AFN a transformar en AFD

Para todo AFN N, existe algún AFD D tal que L(N) = L(D).

Este resultado, sorprendente, pero muy útil, puede probarse en forma constructiva, proponiendo para un AFN cómo construir un AFD que sea equivalente.

El método que se usa para pasar de un AFN a un AFD se basa en la idea de considerar el conjunto de estados en los que podrá encontrarse el AFN al haber consumido una cierta entrada.

3.15.1. El método de los conjuntos de estados

Dado un AFN M, consideremos la idea de mantener un conjunto de estados Q_i en los que sea posible estar en cada momento consumiendo las letras de una palabra de entrada.

Por ejemplo, considérese el AFN de la figura 18. Se quiere analizar qué sucede cuando este AFN recibe la palabra baaaaab. Para ello, se lleva el registro de los conjuntos de estados en los que pueda encontrarse el AFN. Inicialmente, podrá encontrarse en el estado inicial q0, pero sin "gastar" ningún caracter puede estar también en el estado q1, o sea que el proceso arranca con el conjunto de estados Q0 = {q0, q1}. Al consumirse el primer caracter, b, se puede pasar de q0 a q0 o bien a q1 (pasando por el ε), mientras que del q1 sólo se puede pasar a q1. Entonces, el conjunto de estados en que se puede estar al consumir la b es Q1 = {q0, q1}. Y así en adelante.

La tabla siguiente resume los conjuntos de estados por los que se va pasando para este ejemplo:

Entrada	Estados
	$\{q_0,q_1\}$
b	$\{q_0,q_1\}$
a	$\{q_2,q_4\}$
a	$\{q_0,q_1,q_3\}$
a	$\{q_1,q_2,q_4\}$
a	$\{q_0,q_1,q_3,q_4\}$
a	$\{q_1,q_2,q_3,q_4\}$
b	q_1

Puesto que el último conjunto de estados $\{q_1\}$ incluye a un estado final, se concluye que la palabra de entrada puede ser aceptada. Otra conclusión es que si se considera a los conjuntos de estados Q_i como una especie de "mega-estados" de cierto autómata, entonces se han estado siguiendo los pasos de ejecución de un AFD con "mega-estados".

Una vez se comprenda lo anterior, se cocluye que, si en vez de considerar una palabra en particular, como fue baaaaab, se considera cada posible caracter que puede llegar a estar en un "mega-estado", entonces se puede completar un AFD, que será equivalente al AFN dado. Para poder ser exhaustivos, se necesita organizar las entradas posibles de manera sistemática.

Ejemplo: Considérese el problema de transformar a AFD el AFN de la figura 18.

Se consideran el conjunto de estados del AFN en los que puede encontrarse éste en cada momento. El conjunto inicial de estados estará formado por los estados del AFN de la figura 18 en los que puede estar antes de consumir el primer caracter, esto es, q_0 y q_1. Dicho conjunto aparece en la figura 19(a).

A partir de ahí, tras recibir un caracter a, el AFN puede encontrarse ya sea en q_2 o en q_4, los cuales se incluye un nuevo conjunto de estados, al que se llega con una transición con a, como se ilustra en la figura 19(b); similarmente, a partir del conjunto inicial de estados $\{q_0, q_1\}$ con la letra b se llega al mismo conjunto $\{q_0, q_1\}$, lo cual se representa con un "lazo" así mismo en la figura 19(b).

Con este mismo procedimiento se siguen formando los conjuntos de estados; por ejemplo, a partir de $\{q_2, q_4\}$, con una a se pasa a $\{q_3, q_0, q_1\}$. Continuando así, al final se llega al diagrama de la figura 19(c).

Un detalle importante a observar en este procedimiento es que en ocasiones no hay estados adonde ir; por ejemplo, a partir del conjunto de estados $\{q_2, q_4\}$, con b no se llega a ningún estado. En casos como ´este, se considera que habrá una transici´on con b a un nuevo conjunto de estados vacíos, esto es {}, como se aprecia en la figura 19(c). Por supuesto, este estado vacío tendrá transiciones con a y con b igualmente.

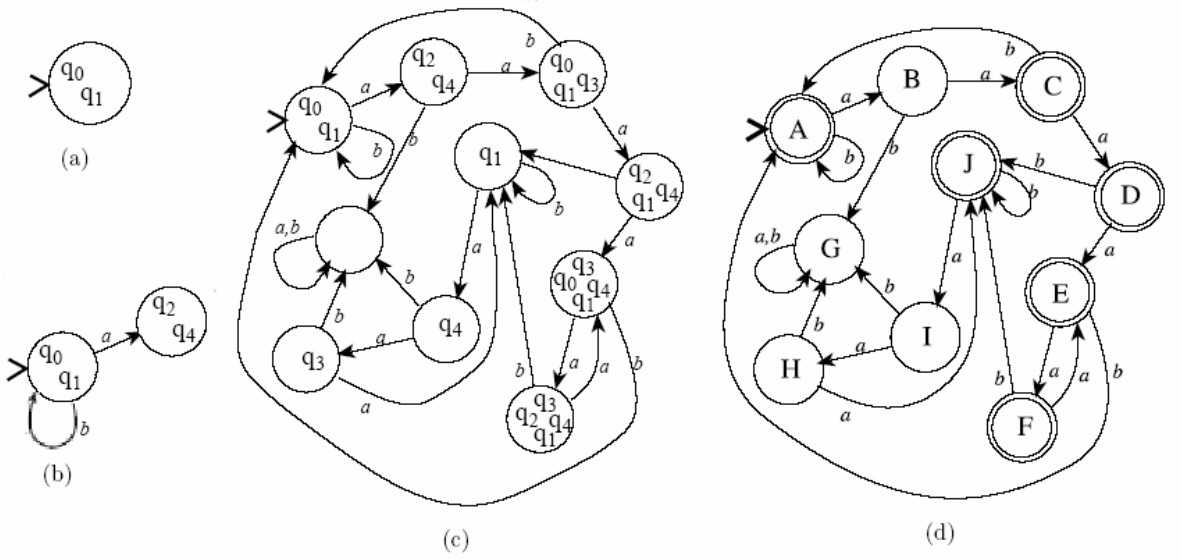


Figura 19: Transformación de un AFN a un AFD

Si se ven los círculos como estados, se ha construido un AFD. Unicamente falta determinar cuales de los nuevos estados son finales y cuales no. Obviamente, si uno de los conjuntos de estados contiene un estado final del antiguo AFN, esto muestra que es posible que en ese punto el AFN hubiera aceptado la palabra de

entrada, si ésta se terminara. Por lo tanto, los estados finales del nuevo autómata serán aquellos conjuntos de estados que contengan algún estado final. Así, en el AFD de la figura 19(d) se marcan los estados finales; además se borran los estados del antiguo AFN de cada uno de los círculos, y se bautizan cada conjunto de estados como un estado.

3.15.2. Una transformación inofensiva

Cuando se quiere aplicar el método descrito en los párrafos precedentes, una dificultad que puede presentarse es que algunas flechas del autómata tienen como etiquetas palabras de varias letras, y desde luego no podemos tomar “un pedazo” de una transición. Esta situación se aprecia en el AFN de la figura 20. En efecto, si a partir del estado inicial se intenta consumir la entrada “a”, se observa que no hay una transición que permita hacerlo, aún cuando hay una transición (q_0, aa, q_1) cuya etiqueta empieza con a.

Una solución a esta dificultad es normalizar a 1 como máximo la longitud de las palabras que aparecen en las flechas. Esto puede hacerse intercalando $|w| - 1$ estados intermedios en cada flecha con etiqueta w. Así, por ejemplo, de la transici´on (q_1, aaa, q_1) de la figura 20, se generan las transiciones siguientes: (q_1, a, q_2), (q_2, a, q_3), (q_3, a, q_1), donde los estados q_2 y q_3 son estados nuevos generados para hacer esta transformación.

Con esta transformación se puede pasar de un AFN cualquiera M a un AFN M_0 equivalente cuyas transiciones tienen como maximo un caracter. Esta transformación es “inofensiva” en el sentido de que no altera el lenguaje aceptado por el AFN.

Por ejemplo, para el AFN de la figura 20 se tiene el AFN transformado de la figura 18.

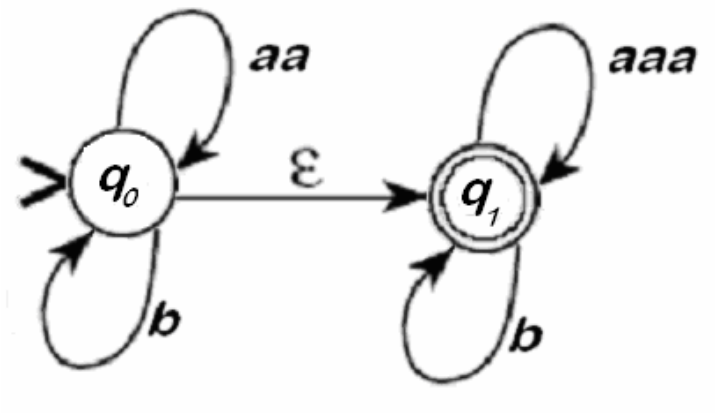


Figura 20: AFN Con transición de varias letras.

3.15.3. Formalización del algoritmo de conversión

Ahora se va a precisar el método de conversión de AFN a AFD con suficiente detalle como para que la programación en computador sea relativamente sencilla. Sin embargo, no se va a describir el algoritmo en términos de ciclos, instrucciones de asignación, condicionales, etc., que son típicos de los programas imperativos. Se va a presentar un conjunto de definiciones que capturan los resultados intermedios en el proceso de conversión de AFN a AFD. Estas definiciones permiten programar en forma casi directa el algoritmo de conversión, si se utiliza un lenguaje de programación adecuado, preferentemente de tipo funcional, como por ejemplo Scheme.

Este ejemplo se va a ir presentando con las definiciones partiendo de la más sencilla, hasta llegar a la más compleja.

Primero introducimos una función transicion(q, σ), que a partir de un estado q y un caracter dado σ obtiene el conjunto de estados a los que se puede llegar desde q directamente gastando el caracter σ. Por ejemplo, tomando el AFN de la figura 21, tenemos que transicion(q_0, b) = {q_0, q_1}. Similarmente, transicion(q_1, b) = {q_1}, y transicion(q_3, a) = {}.

Se puede definir matemáticamente de la forma siguiente:

transición(q, σ) = {p | (q, σ, p) $\in$ σ}

Sin embargo, esta definición no toma en cuenta el hecho de que a veces es posible tener transiciones que no gastan ningún caracter -aquellas marcadas con ε. Así, en la figura 18, se puede pasar de q_2 a q_0 y luego continuar de q_0 a q_1, por lo que en realidad se tiene que considerar a q_1 como uno de los estados a los que se puede llegar desde {q_1, q_2} gastando una a. Por lo tanto, hay que modificar la definición anterior.

Se define una función auxiliar cerr-ε(q) que es el conjunto de estados a los que se puede llegar desde el estado q pasando por transiciones vacías. Además, si con una transición vacía se llega a otro estado que también tiene transiciones vacías, hay que continuar añadiendo a cerr-ε (q) los estados a los que se llegue, hasta que no sea posible añadir nuevos estados. Por ejemplo, en la figura 21, cerr -ε (q_1) = {q_1, q_2, q_4}, cerr-ε(q_2) = {q_2, q_4}, y cerr-ε(q_0) = {q_0}.

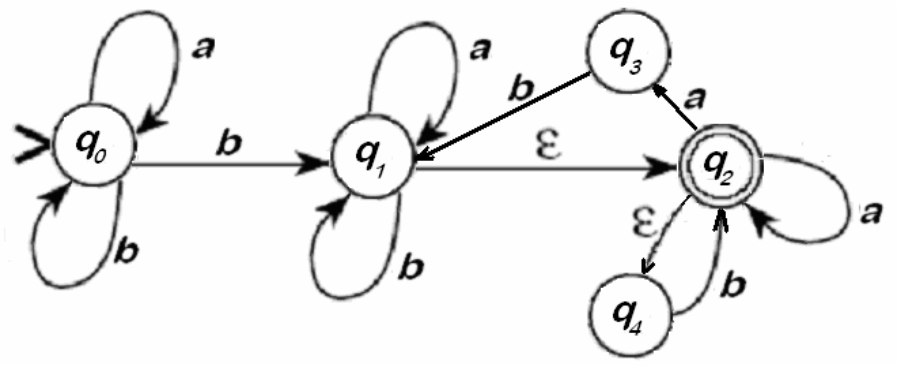


Figura 21: AFN con transiciones vacias

cerr -ε (q) se acostumbra llamar cerradura al vacío porque matemáticamente es la cerradura de q con la relación {(x, y) | (x, ε, y) ∈ Δ}.

La función cerr-ε(q) se puede definir como sigue:

La cerradura al vacío cerr-ε(q) de un estado q es el más pequeño conjunto que contiene:

1. Al estado q;
2. Todo estado r tal que existe una transici´on (p, ε, r) ∈ Δ, con p ∈ cerr -ε (q).

Es fácil extender la definición de cerradura al vacío de un estado para definir la cerradura al vacío de un conjunto de estados:

La cerradura al vacío de un conjunto de estados CERR-ε ({q_1, ... , q_n}) es igual a cerr -ε (q_1)∪, ... , ∪ cerr-ε (q_n).

Ejemplo. Sea el AFN de la figura 21. Entonces CERR-ε({q_1, q_3}) = {q_1, q_2, q_3, q_4}.

Con la función de cerradura al vacío ya estamos en condiciones de proponer una versión de la función transicion que tome en cuenta las transiciones vacías. Llamaremos a esta función “transicion-ε”, y la definimos de forma que transicion-ε (q, σ) sea el conjunto de estados a los que se puede llegar desde q gastando σ, inclusive pasando por transiciones vacías. El algoritmo es como sigue, para un estado q y un caracter σ:

1. Calcular Q0 = cerr -ε(q)
2. Para cada estado de Q_0, obtener transicion(q, σ), y unir todos los conjuntos obtenidos, dando por resultado un conjunto Q_1.
3. transicion-ε (q, σ) = CERR-ε (Q1).

Por ejemplo, tomando la figura 21, para calcular transicion-$\varepsilon(q_1, a)$, los pasos son como sigue:

1. $Q0 = \{q_1, q_2, q_4\}$
2. transicion$(q_1, a) = \{q_1\}$, transicion$(q_2, a) = \{q_2, q_3\}$, y transicion$(q_4, a) = \{\}$, por lo que uniendo estos conjuntos, $Q_1 = \{q_1, q_2, q_3\}$.
3. transicion-$\varepsilon(q_1, a)$ = CERR-$\varepsilon(\{q_1, q_2, q_3\}) = \{q_1, q_2, q_3, q_4\}$.

Como última definición, es directo extender la función transicion-$\varepsilon(q, \sigma)$, que se aplica a un estado y un caracter, a una función que se aplique a un conjunto de estados y un caracter; llamamos a esta función TRANSICION-$\varepsilon(Q, \sigma)$, para un conjunto de estados Q y un caracter σ. Simplemente aplicamos transicion-$\varepsilon(q, \sigma)$ para cada uno de los estados $q \in Q$, y juntamos los resultados en un solo conjunto.

Por ejemplo, en la figura 21

TRANSICION-ε $(\{q_0, q_2\}, a) = \{q_0, q_2, q_3, q_4\}$.

Finalmente resumimos el proceso global de transformación de un AFN a un AFD en el siguiente algoritmo.

Algoritmo de transformación AFN – AFD:

Dado un AFN $(K, \sum, \Delta, s, F)$, un AFD equivalente se obtiene por los siguientes pasos:

1. El conjunto de estados inicial es cerr -$\varepsilon(s)$.
2. El alfabeto del AFD es el mismo del AFN.
3. Para cada conjunto de estados Q ya presente, hacer:
a) Añadir el conjunto de estados TRANSICION-$\varepsilon(Q, \sigma)$ para cada caracter σ del alfabeto, si no ha sido creado aún.
b) Añadir transiciones $((Q, \sigma), Q_\sigma)$ para cada conjunto de estados Q_σ creado en el paso anterior.
4. Los conjuntos de estados que contengan un estado en F serán finales.

Más diseño de AFN: Intersección de lenguajes

Los problemas de diseño de AFN en que se combinan dos condiciones que se deben cumplir simultáneamente son particularmente difíciles de resolver. Un ejemplo de estos problemas es: “obtener un AFN que acepte las palabras que contengan la cadena abb un número impar de veces y ba un número par de veces”.

Es mejor contar con un método modular que permita combinar de una manera sistemática las soluciones parciales para cada una de las condiciones.Esto es posible, si se considera la siguiente propiedad de la intersección de conjuntos:

$$L_1 \cap L_2 = (L_1^C \cup L_2^C)^C$$

Esta fórmula sugiere un procedimiento práctico para obtener un AFN que acepte la intersección de dos lenguajes dados. Esto se ilustra en el siguiente ejemplo.

Ejemplo.- Obtener un AF para el lenguaje en el alfabeto {a, b} en que las palabras son de longitud par y además contienen un número par de a's. Este problema parece bastante difícil, pero se vuelve fácil utilizando la fórmula de intersección de lenguajes. En efecto, se inicia calculando los AFD para los lenguajes que cumplen independientemente las dos condiciones.

El AFD M_1 de la figura 22(a) acepta las palabras de longitud par, mientras que M_2 de 2.32(b) acepta las palabras con un número par de a's.

Ahora se obtienen los AFD que aceptan el complemento de los lenguajes de M_1 y M_2, cambiando los estados finales por no finales y viceversa; sean $M_1^C \cup M_2^C$

Es muy importante notar que sólo es posible complementar AFD's y no cualquier AFN.

En efecto, si en un AFN simplemente se cambian estados finales por no finales y viceversa, en general se llega a un resultado erróneo (esto es, el autómata resultante no es equivalente al original) Combinamos estos autómatas utilizando el procedimiento para la unión de lenguajes, dando un AFN M_3 (figura 22(c)), el cual es convertido a un AFD M_4. Finalmente, este AFD es simplificado y "complementado", dando M_4^c (figura 22(d)), que es el autómata buscado.

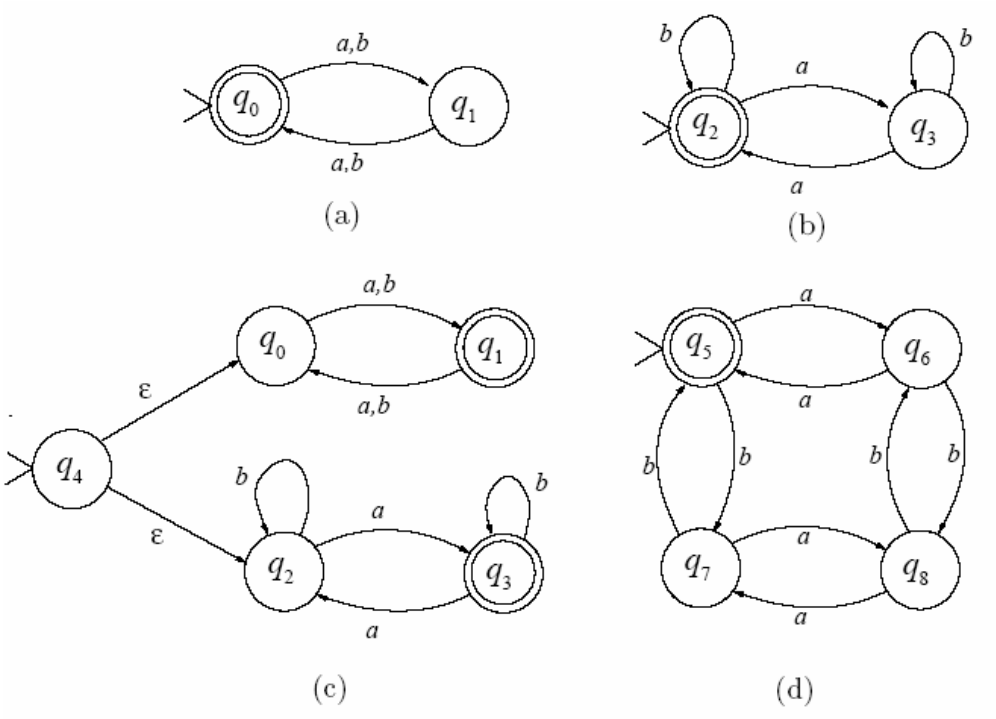


Figura 22: Intersección de dos AFN

ACTIVIDADES:

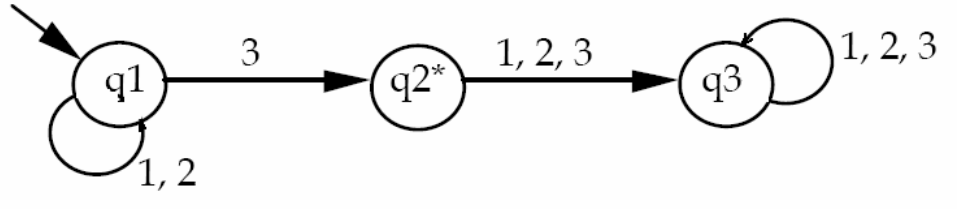
Ejercicios Propuestos:

1.- Obtener el lenguaje reconocido por el siguiente AFD:

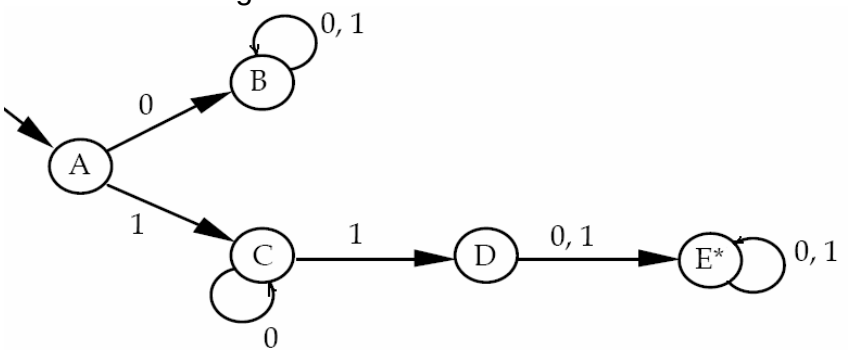
A = ({a, b, c}, {q0, q1, q2, q3, q4}, f, q0, {q2})

f(q0, a) = q1 f(q0, b) = q4 f(q0, c) = q4
f(q1, a) = q4 f(q1, b) = q1 f(q1, c) = q2
f(q2, a) = q4 f(q2, b) = q4 f(q2, c) = q2
f(q3, a) = q4 f(q3, b) = q3 f(q3, c) = q2
f(q4, a) = q4 f(q4, b) = q4 f(q4, c) = q4

2.- Determinar el lenguaje que reconoce el siguiente AFD:



3.-Dado el autómata finito siguiente:



Definir la gramática lineal izquierda que describe el mismo lenguaje reconocido por el autómata.

4.-Decir cuáles de las siguientes palabras son reconocidas por el siguiente AFND: 110, 01, 100

AFND = ({0, 1}, {q0, q1, q2}, q0, {q1})

f(q0, 0) = ∅	f(q0, 1) = {q1, q2}	f(q0, λ) = ∅
f(q1, 0) = {q0}	f(q1, 1) = {q0, q1}	f(q1, λ) = {q0}
f(q2, 0) = {q2}	f(q2, 1) = ∅	f(q2, λ) = {q1}

5.- Determinar la pertenencia de las cadenas: aab, aba y ba, al lenguaje reconocido por el siguiente AFND:

A = ({a,b}, {q1,q2,q3,q4}, f, q1, {q1}, {(q2,q4), (q3,q4), (q4,q3)})

f(q1,a) = {q2}	f(q1,b) = ∅
f(q2,a) = {q1,q3,q4}	f(q2,b) = {q1,q3}
f(q3,a) = ∅	f(q3,b) = {q1,q4}
f(q4,a) = ∅	f(q4,b) = {q3}

MINIMIZACIÓN DE AUTÓMATAS FINITOS

6.-Obtener para cada uno de los siguientes autómatas finitos su autómata mínimo Equivalente:

A1

f	0	1
→ p1	p2	p4
p2	p2	p2
p3	p2	p4
p4*	p3	p2
p5	p6	p7
p6	p7	p3
p7	p7	p6
p8*	p7	p6

A2

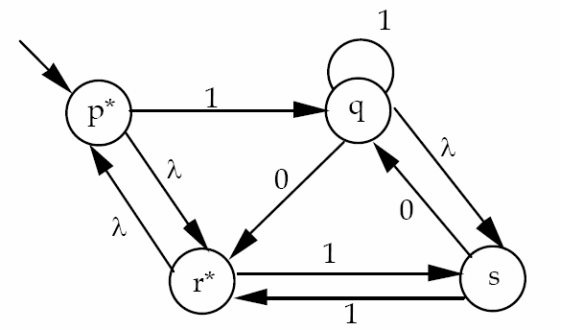
f	0	1
→ q1	q3	q2
q2	q2	q2
q3	q6	q1
q4*	q2	q3
q5	q6	q7
q6	q7	q2
q7	q7	q8
q8*	q7	q6

7.- Dada la gramática lineal derecha: G = (T, N, S, P) T = {0, 1} .

N = {S, A} P = { S::= 1A | 1S, A::= 0A | 1A | 1 }

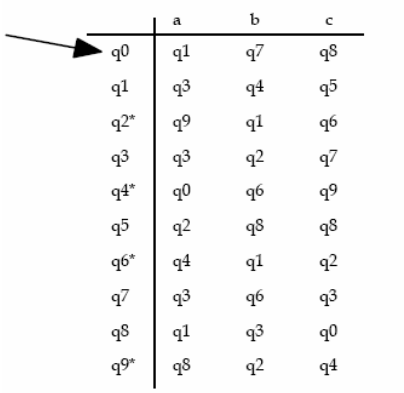
encontrar el autómata finito determinista (AFD) mínimo asociado.

8.-Dado el autómata finito:



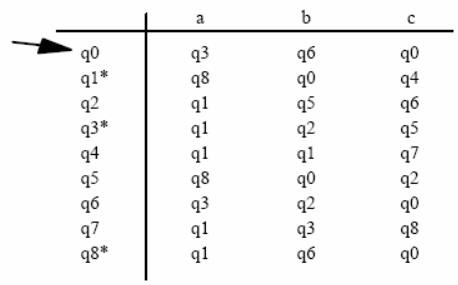
(a) Construir el AFD mínimo equivalente
(b) Deducir el lenguaje que reconoce

9.- Encontrar el autómata mínimo equivalente al siguiente AFD:



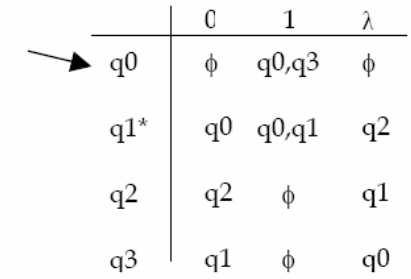
	a	b	c
→ q0	q1	q7	q8
q1	q3	q4	q5
q2*	q9	q1	q6
q3	q3	q2	q7
q4*	q0	q6	q9
q5	q2	q8	q8
q6*	q4	q1	q2
q7	q3	q6	q3
q8	q1	q3	q0
q9*	q8	q2	q4

10.- Construir el autómata finito determinista mínimo equivalente al siguiente:



	a	b	c
→ q0	q3	q6	q0
q1*	q8	q0	q4
q2	q1	q5	q6
q3*	q1	q2	q5
q4	q1	q1	q7
q5	q8	q0	q2
q6	q3	q2	q0
q7	q1	q3	q8
q8*	q1	q6	q0

11.- Encontrar el AFD mínimo equivalente al siguiente AFND:



	0	1	λ
→ q0	ϕ	q0,q3	ϕ
q1*	q0	q0,q1	q2
q2	q2	ϕ	q1
q3	q1	ϕ	q0

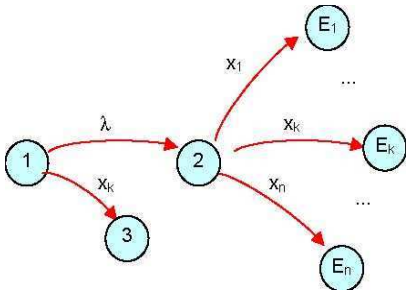
Ejercicios Resueltos[12]:

Suponga que permitimos que los autómatas finitos cambien de un estado a otro sin leer un símbolo de sus cintas de entrada. En un diagrama de transiciones, este tipo de transiciones normalmente se representa como un arco con etiqueta λ en vez de un símbolo del alfabeto; y se le conoce como transición λ. Muestre que cualquier diagrama de transiciones que tenga transiciones λ se puede modificar para que ya no contenga este tipo de transiciones pero que a la vez acepte el mismo lenguaje (el diagrama modificado puede ser no determinista).

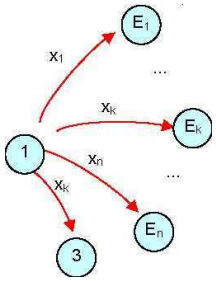
[12] Ingenieria Técnica en Informatica de Sistemas y de Gestión de la UNED España ASIGNATURA: TEORÍA DE UTÓMATAS I Tutoría del Centro Asociado de Plasencia en: URL http://dac.escet.urjc.es/lrincon/uned/ta1/ProblemasFinalCapitulo.pdf

Solución: Pueden darse estos casos:

1.- La transición λ conduce a un estado de cual salen otros arcos.
1.1 El estado al que conduce la transición λ no es de aceptación.

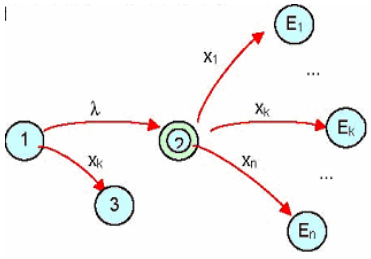


Esto significa que desde el estado 1, se puede pasar a cualquiera de los estados E_1, ..., E_j, E_k, leyendo respectivamente x_1, ..., x_j, ...x_k. Por lo cual podemos eliminar el estado 2, sacando del 1 los mismos arcos que salían del 2 (si los hubiera).



Resulta que ahora, el diagrama que aparentemente era determinista, ya no lo es. Hay dos arcos rotulados con la entrada x_k.

1.2 El estado al que conduce la transición λ es de aceptación

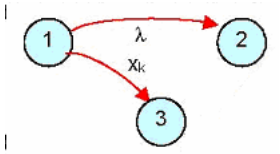


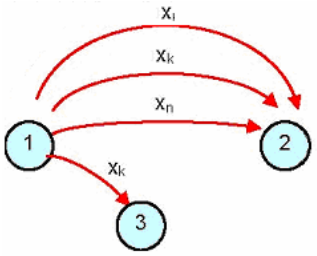
Esto significa que desde el estado 1, se puede aceptar la cadena sin leer ningún símbolo más; o pasar a los estados E_1, ..., E_j, E_k, leyendo respectivamente x_1, ..., x_j, ...x_k.. En otras palabras, que el estado 1 es de aceptación. Por lo cual podemos eliminar el estado 2, hacer de aceptación el 1; y sacar del 1 los mismos arcos que salían del 2.

2.- La transición λ conduce a un estado de cual no salen otros arcos.
2.1 El estado al que conduce la transición λ no es de aceptación.

Esto significa que sea cual sea la entrada en el estado 1, se puede pasar a un "callejón sin salida". Y esta posibilidad podemos representarla sacando un arco con cada entrada posible hacia el "callejón sin salida".

x1

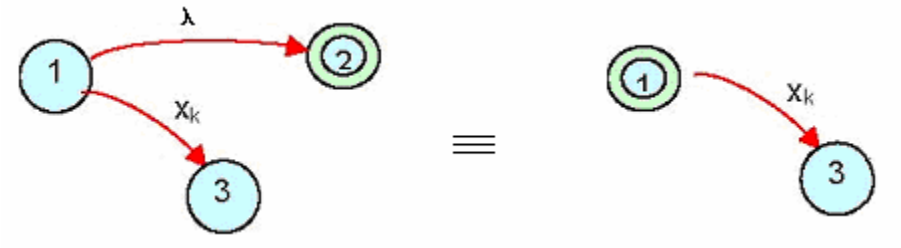




Es curioso, pero tiene su justificación. Al fin y al cabo; si vamos a caer en ese callejón sin salida (sin llegar a un estado de aceptación), para los efectos del

análisis de la cadena, da igual que leamos otra entrada más (sea cual sea) o que no la leamos.

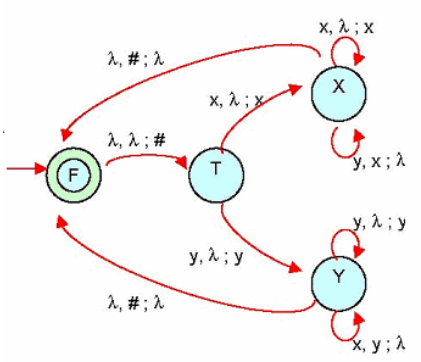
2.2 El estado al que conduce la transición λ es de aceptación. El tratamiento es igual que en 1.2



Esto significa que desde el estado 1, se puede aceptar la cadena sin leer ningún símbolo más. En otras palabras, que el estado 1 es de aceptación. Por lo cual podemos eliminar el estado 2 y hacer de aceptación el 1.

Muestre que el lenguaje del alfabeto {x, y} que consiste en aquellas cadenas con el mismo número de x e y es independiente del contexto determinista:

Solución:



Comentarios: El estado inicial es de aceptación, para permitir cadenas con cero x's y cero y's.

Durante el estado X, el número de x's es mayor que el de y's. Se introduce una x cada vez que se lee una x; y se saca una x cada vez que se lee una y. Si en algún momento se iguala este número (símbolo # en la cima), se vuelve al estado F

Durante el estado Y, el número de y's es mayor que el de x's. Se introduce una y cada vez que se lee una y; y se saca una y cada vez que se lee una x. Si en algún momento se iguala este número (símbolo # en la cima), se vuelve al estado F

Si durante el estado F se recibe FDC, la cadena queda aceptada.

Muestre que si L es un lenguaje independiente del contexto, entonces el lenguaje que consiste en las cadenas de L escritas a la inversa también es independiente del contexto.
Solución:

Un lenguaje independiente del contexto se puede caracterizar mediante una gramática independiente del contexto que "casi" tiene la forma normal de

Chomsky. Las reglas de dicha gramática pueden tener estas formas:

S' → λ (Nuevo símbolo inicial en el lado izquierdo y λ en el derecho para permitir las cadenas vacías)
S' → x (Nuevo símbolo inicial en el lado izquierdo y un terminal en el lado derecho)
S' → AB (Nuevo símbolo inicial en el lado izquierdo y un par de no terminales en el lado derecho)
S → x (Antiguo símbolo inicial en el lado izquierdo y un terminal en el lado derecho)
S → AB (Antiguo símbolo inicial en el lado izquierdo y un par de no terminales en el lado derecho)

Si en las reglas cuyo lado derecho son dos no terminales se invierte el orden de dichos no terminales:
S' → λ
S' → x
S' → BA
S → x
S → BA

Resulta que las cadenas aceptadas son exactamente las inversas.

Ejercicios Resueltos.[13]

Ejercicio 1- Define, de manera recursiva, el lenguaje L de las cadenas de paréntesis correctas.
(Es decir, (())() ∈ L, ()(() ∉ L)
SOLUCIÓN:
1. ()∈ L (aunque λ∉ L, se puede considerar alternativamente que λ ∈ L para definir con 2 y 3 las demás palabras).
2. Si $x \in$ L, entonces (x) ∈ L .
3. Si $x, y \in$ L, entonces $x\,y \in$ L.
4. No hay más palabras en L que las formadas por las reglas anteriores.

Ejercicio 2- Encuentra una expresión regular correspondiente al lenguaje de las cadenas en {*a, b*}* que no empiezan por *b* y no contienen dos *a*'s consecutivas. Construye un AFN-λ reconocedor de este lenguaje.

Una ER que describe el lenguaje es λ+*a*(*b*+*ba*)*. El AFN-λ se puede construir directamente utilizando el Teorema de Kleene.

---

[13] Ingeniería Superior en Informática, Grupos A, B y C Teoría de Autómatas y Lenguajes formales. Examen Final-19 de Junio de 2003, Solución Ejercicios

Ejercicio 3 (a) Define, mediante expresiones regulares, las clases de equivalencia respecto a la relación de distinguibilidad con respecto al lenguaje del ejercicio anterior, utilizando para ello un conjunto distinguible máximo.
(b) Define completamente la operación de concatenación por la derecha de las clases con las letras del alfabeto.
(c) Define a partir de (a) y (b) el AF mínimo.

SOLUCIÓN:
(a) Para encontrar un CDM, vamos estudiando las distintas palabras del lenguaje. Empezamos comparando las palabras de longitud 1 con la palabra vacía y entre sí. Es fácil ver que {λ, *a*} es un conjunto distinguible (*z* = *a*). También son distinguibles {λ, *b*} (*z* = *a*) y {*a, b*} (*z* = *b*). Por tanto, {λ, *a*, *b*} es un conjunto distinguible.
(b) En cuanto a las palabras de longitud 2, es fácil ver que *aa* es indistiguible de *b*, que *ba* es indistinguible de *b* y que *bb* es indistinguible de *b* (son palabras que no están en el lenguaje y que, seguidas de cualquier cola, tampoco), mientras que *ab* es distinguible de λ , *a* y *b*.

Con esto, es sencillo darse cuenta que si *w* es una palabra de longitud mayor o igual que 3, resulta que:

(c) si *w* empieza por *a*, acaba por *a* y no contiene la subcadena *aa*, entonces es indistinguible de *a*,
(d) si *w* empieza por *a*, acaba por *b* y no contiene la subcadena *aa*, entonces es indistinguible de *ab*, y,
(e) si *w* empieza por *b* o contiene la subcadena *aa*, entonces es indistinguible de *b*.
Por tanto, un CDM es {λ*, a, b, ab*}.
Las correspondientes clases de equivalencia vienen dadas por las siguientes ER:
$[\lambda] \rightarrow \lambda$
$[a] \rightarrow a(b^+a)^*$.
$[ab] \rightarrow a(b^+a)^*b^+$
$[b] \rightarrow b(b + a)^* + (b + a)^*aa\ (b + a)^*$

Es sencillo ver que la unión de estas clases es $\{a, b\}^*$ y que la intersección de cada dos de ellas es vacía.

(b) La operación de concatenación a la derecha viene dada por
$[\lambda]a = [a]$, $[\lambda]b = [b]$
$[a]a = [b]$, $[a]b = [ab]$
$[b]a = [b]$, $[b]b = [b]$
$[ab]a = [a]$, $[ab]b = [ab]$

(c) El AFD mínimo viene dado por la siguiente tabla (A-[λ], B-[*a*], C-[*ab*], D-[*b*]):

	a	b
→ *A	B	D
*B	D	C
*C	B	C
	D	D D

Ejercicio 4 Demostrar que el lenguaje L = $\{a^i b^j c^k : j > i + k\}$ no es regular, utilizando el Lema de Bombeo.

SOLUCIÓN: Sea n la constante del Lema de Bombeo. Consideramos la palabra $x = a^n b^{2n+1} b^n$ y consideramos todas las descomposiciones de $x = uvw$, con $|uv| \leq n$ y $|v| > 0$. Por tanto, v está formado sólo por a's, y contiene al menos una. De esta forma, $uv^k w \notin$ L para todo $k \geq 2$.

Ejercicio 5 Consideramos la siguiente tabla de transición de un AFD:

	a	b
	q_1	q_2 q_1
	q_2	q_2 q_1

donde q_1 es el estado inicial y q_2 es el único estado de aceptación. Calcula una expresión regular para el lenguaje reconocido por el autómata, utilizando el método de la demostración del teorema de Kleene.

SOLUCIÓN: Utilizando la demostración del Teorema de Kleene, basta calcular

$R^{(2)}_{1,2}$ = L(1, 2, 2). Aplicando $R^{(2)}_{1,2} = R^{(1)}_{1,2} + R^{(1)}_{1,2}(R^{(1)}_{2,2})^* R^{(1)}_{2,2}$, tenemos que

$R^{(0)}_{1,1} = \lambda + b$, $R^{(0)}_{1,2} = a$, $R^{(0)}_{2,1} = b$, $R^{(0)}_{2,2} = \lambda + a$

Por tanto, $R^{(1)}_{1,2} = a + (\lambda + b)(\lambda + b)^* a = b^* a$ y $R^{(1)}_{2,2} = (\lambda + a) + b(\lambda + b)^* a = \lambda + b^* a$.

Finalmente $R = R^{(2)}_{1,2} = b^* a + (b^* a)(\lambda + b^* a)^* (\lambda + b^* a) = (b^* a)^+$.

Ejercicio 6 Considera la gramática incontextual G dada por: $S \rightarrow SS \mid \lambda \mid ab$
Demuestra que es ambigua. Encuentra una expresión regular que describa L(G) y define una gramática regular equivalente no ambigua.

SOLUCIÓN: Para ver que G es ambigua, mostramos dos derivaciones más a la izquierda para la palabra $w = ab$:
$S \Rightarrow SS \Rightarrow abS \Rightarrow ab$
$S \Rightarrow ab$
Una expresión regular que describe L(G) es $(ab)^*$, y una gramática incontextual regular no ambigua que describe L(G) es $S \rightarrow aB \mid \lambda$; $B \rightarrow bS$
(Es sencillo obtener esta gramática a partir de un AFD que reconoce el lenguaje).

BIBLIOGRAFÍA DE LA UNIDAD

eBook Automatas y Lenguajes:http://lizt.mty.itesm.mx/%7Erbrena/Libro/form2AyL.php de ramon.brena@itesm.mx

IOST F. Hans, Teoría de autómatas y lenguajes formales, CAPÍTULO 2, LENGUAJES REGULARES Y AUTOMATAS FINITOS (2001) en: http://iie.ufro.cl/~hansiost/automatas/Capitulo2.doc

INTRODUCCIÓN

Los lenguajes independientes del contexto que tambien se conocen con el nombre de gramaticas de contexto libre son un método recursivo sencillo de especificación de reglas gramaticales con las que se pueden generar cadenas de un lenguaje.

Es factible producir de esta manera todos los lenguajes regulares, ademas de que existen ejemplos sencillos de gramaticas de contexto libre que generan lenguajes no regulares.

Las reglas gramaticales de este tipo permiten que la sintaxis tenga variedad y refinamientos mayores que los realizados con lenguajes regulares, en gran medida sirven para especificar la sintaxis de lenguajes de alto nivel y otros lenguajes formales.

OBJETIVO GENERAL

Conocer los modelos de computación que corresponden a los lenguajes independientes del contexto y su aplicación.

OBJETIVOS ESPECIFICOS

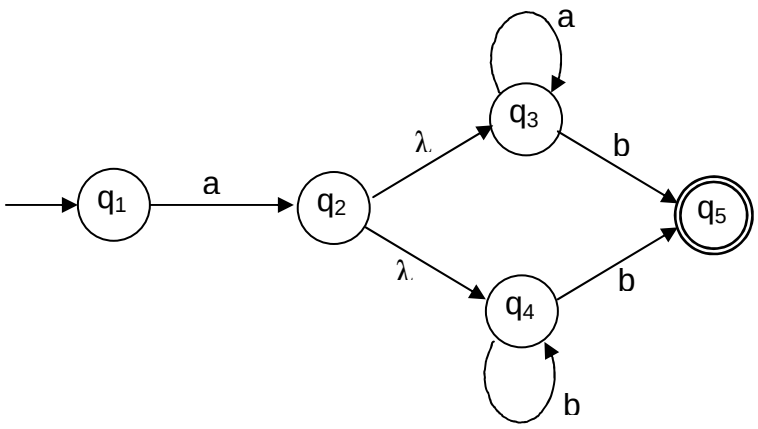
Generalizar los conceptos de autómatas finitos y gramaticas regulares.

Reconocer el potencial de procesamiento del lenguaje del automata con los autómatas de pila.

LECCION 16. – GRAMATICAS REGULARES

Hasta ahora, se han visto dos formas de definir lenguajes: a través de autómatas finitos, y por medio de expresiones regulares. Ahora veremos otra forma, usando el concepto de gramáticas.

Sea el autómata:



$a(a^{*} \cup b^{*})b$

Se puede pensar que este autómata es un generador de cadenas a partir de una cadena vacía. Así, las cadenas que puede generar el autómata son todas aquellas que perteneces al lenguaje representado por el autómata L(M). Observando el autómata podemos decir que todas las cadenas inician con una ‘a’, luego viene una serie de a’s o b’s y finalmente termina con una ‘b’.

Podemos utilizar la siguiente notación para representar esto:

$S \rightarrow a\ E$: partiendo del estado inicial S, vendrá una ‘a’ y luego otra cadena que denotamos genéricamente con una E.

$E \rightarrow A,\ E \rightarrow B$: hay dos posibilidades para E, que venga una exp. tipo A o tipo B.

Luego, para expresiones tipo A y B tenemos:

$A \rightarrow b,\ A \rightarrow aA$

$B \rightarrow b,\ B \rightarrow bB$

Finalmente, tendremos el siguiente conjunto de expresiones (utilizamos $|$ para representar ‘o’).

1. $S \rightarrow a\ E$
2. $E \rightarrow A \mid B$
3. $A \rightarrow aA \mid b$
4. $B \rightarrow bB \mid b$

Estas expresiones pueden ser consideradas como reglas de sustitución y permiten generar cualquier cadena o palabra válida dada por el autómata del cual se obtuvieron. Por ejemplo, sea la cadena a^3b:

$$S \rightarrow a\ E \underset{(2)(3)}{\Rightarrow} a\ a\ A \underset{(3)}{\Rightarrow} a\ a\ a\ A \underset{(3)}{\Rightarrow} a\ a\ a\ b = a^3b$$

Una gramática regular G es una cuádrupla G = ($\sum$, N, S, P), donde:

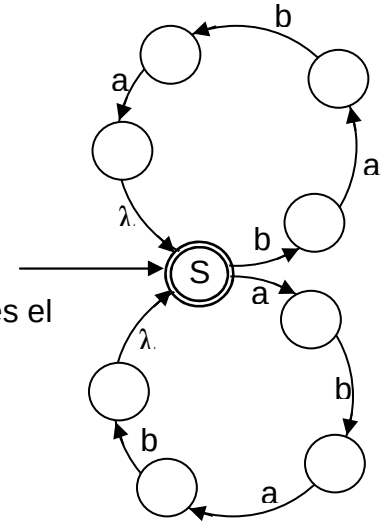
$\sum$: alfabeto (no vacío) de símbolos terminales.
N : es un conjunto (no vacío) de símbolos no terminales.
S : es denominado el símbolo inicial y $S \in N$.
P : es una colección o conjunto de reglas de sustitución llamadas producciones y que son de la forma $A \rightarrow w$, donde $A \in N$ y *w* es una cadena sobre $\sum \cup N$.

Además, *w* debe satisfacer las siguientes condiciones:

1. *w* contiene un no terminal como máximo.
2. si *w* contiene un no terminal, entonces es el símbolo que está en el extremo derecho de *w*.

Ejemplo: Dada la gramática regular dada por:

$S \rightarrow b\ A \mid a\ B \mid \lambda$
$A \rightarrow a\ b\ a\ S$
$B \rightarrow b\ a\ b\ S$



Genera un lenguaje regular. Podemos obtener entonces el autómata y la expresión regular correspondientes.

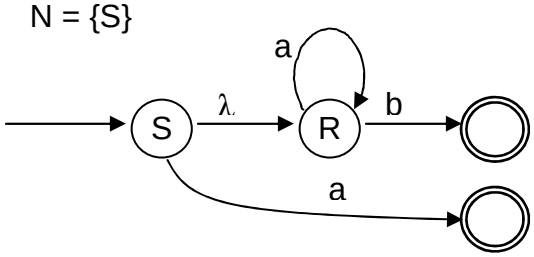
Autómata:

Exp. Regular:

$(baba \cup abab)^*$

Ejemplo: Obtener una gramática regular para el lenguaje representado por:
$a^*b \cup a$

R: $\sum = \{a, b\}$
$N = \{S\}$



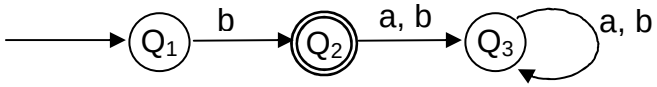
P: $S \rightarrow R \mid a$
$R \rightarrow aR \mid b$

LECCION 17. - GRAMÁTICAS REGULARES Y LENGUAJES REGULARES

Sea un lenguaje regular, se puede obtener una gramática regular que genere L a partir de un AFD $M = (Q, \Sigma, s_0, F, \delta)$ para el cual $L = L(M)$. Se define esta gramática $G = (N, \Sigma, S, P)$ de la siguiente manera:

$N = Q$
$\Sigma = \Sigma$
$S = s_0$
$P = \{q \rightarrow a\ p \mid \delta(q, a) = p\} \cup \{q \rightarrow \lambda \mid q \in F\}$

Por ejemplo, sea el AFD de la siguiente figura:



acepta el lenguaje a^*b. La correspondiente gramática regular será:

$Q_1 \rightarrow a\ Q_1 \mid b\ Q_2$
$Q_2 \rightarrow a\ Q_3 \mid b\ Q_3 \mid \lambda$
$Q_3 \rightarrow a\ Q_3 \mid b\ Q_3$

Es posible probar que cualquier palabra ω que sea aceptada por el AFD M, puede ser generada por la gramática regular G. Esto significa que $L(G) = L(M)$.

También es posible deducir un AFND a partir de una gramática regular G. Sea $G = (N, \sum, S_0, P)$ una G.R., se define $M = (Q, \sum, S, F, \delta)$ como sigue:

$Q = N \cup \{f\}$, donde f es un nuevo símbolo.
$S_0 = S$
$F = \{f\}$

y δ se construye a partir de las producciones de P, como se indica a continuación:

1. Si $A \rightarrow \sigma_1 \ldots \sigma_n \; B$ es una producción de P, con A y B como no terminales, entonces se agregan a Q los nuevos estados $q_1, q_2, \ldots, q_{n-1}$ y las transiciones siguientes:

   $$\delta(A, \sigma_1 \ldots \sigma_n) = \delta(q_1, \sigma_1 \ldots \sigma_n) = \ldots = \delta(q_{n-1}, \sigma_n) = B$$

2. Si $A \rightarrow \sigma_1 \ldots \sigma_n$ es una producción de P, entonces se añadirán a Q los nuevos estados $q_1, q_2, \ldots, q_{n-1}$ y a δ, las transiciones siguientes:

   $$\delta(A, \sigma_1 \ldots \sigma_n) = \delta(q_1, \sigma_1 \ldots \sigma_n) = \ldots = \delta(q_{n-1}, \sigma_n) = f$$

Ejemplo: Sea la gramática Regular dada por.

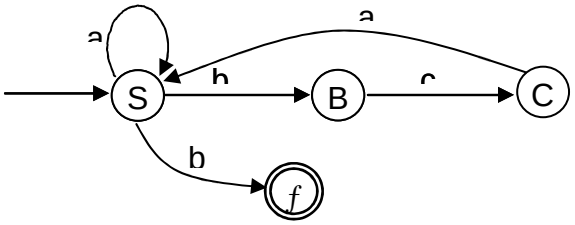
$S \rightarrow aS \mid bB \mid b$
$B \rightarrow cC$
$C \rightarrow aS$

Obtenemos el AFND correspondiente, utilizando el método anterior:

$Q = \{S, B, C, f\}$
$S_0 = S$
$F = \{f\}$

i. $S \rightarrow aS$: $\delta(S,a) = S$
ii. $S \rightarrow bB$: $\delta(S,b) = B$
iii. $S \rightarrow b$: $\delta(S,b) = f$
iv. $B \rightarrow cC$: $\delta(B,c) = C$
v. $C \rightarrow aS$: $\delta(C,a) = S$

Así, podemos construir el AFND siguiente:



Para el caso de una producción de la forma: $A \rightarrow abaS$, tendremos las siguientes transiciones y nuevos estados:

$\delta (A, a\ b\ a) = S$, se descompone en :

$\delta (A, a) = q_1$; $\delta (q_1, b) = q_2$; $\delta (q_2, a) = S$

donde q_1 y q_2 son nuevos estados que se agregan a Q.

LECCION 18. - GRAMÁTICAS INDEPENDIENTES DEL CONTEXTO.

Cuando se eliminan las restricciones impuestas al lado derecho de las producciones de una gramática, se pueden tener producciones como por ejemplo:

$S \rightarrow aB \mid bA$
$A \rightarrow a \mid aS \mid bAA$
$B \rightarrow b \mid bS \mid aBB$
o incluso:
$S \rightarrow aSb \mid \lambda$

Es decir, no hay ninguna restricción respecto de la cantidad y posición de los símbolos no terminales en el lado derecho de las producciones.

Una gramática independiente del contexto (GIC) es una cuádrupla $G=(N, \sum, S, P)$, donde:

N: es una colección finita (no vacía) de símbolos no terminales.
$\sum$: es un alfabeto.
S: es un no terminal llamado símbolo inicial.
P: un conjunto de producciones tal que $P \subseteq N \times (N \cup \sum)^*$.

Los lenguajes generados por una GIC son llamados **Lenguajes Independientes del Contexto** (LIC).

Es posible probar que la gramática independiente del contexto dada por:

$S \rightarrow aSb \mid \lambda$

genera el lenguaje independiente del contexto $\{a^n\ b^n / n \geq 0\}$ y se puede probar que este no es un lenguaje regular. Esto indica que hay LIC que no son lenguajes regulares; y por lo tanto el conjunto de los LIC contienen al conjunto de los lenguajes regulares.

4.18.1. Árboles de derivación y ambigüedad.

Al derivar una cadena a través de una GIC, el símbolo inicial se sustituye por alguna cadena. Los no terminales se van sustituyendo uno tras otro por otras cadenas hasta que ya no quedan símbolos no terminales, queda una cadena con sólo símbolos terminales.

A veces es útil realizar un gráfico de la derivación. Tales gráficos tienen forma de árbol y se llaman “arbol de derivación” o “árbol de análisis”.

Para una derivación dada, el símbolo inical “S” etiqueta la raíz del árbol. El nodo raíz tienen unos nodos hijos para cada símbolo que aparezca en el lado dereho de la producción, usada para reemplazar el símbolo inicial. De igual forma, cada símbolo no terminal tienen unos nodos hijos etiquetados con símbolos del lado derecho de la producción usada para sustituir ese no terminal.

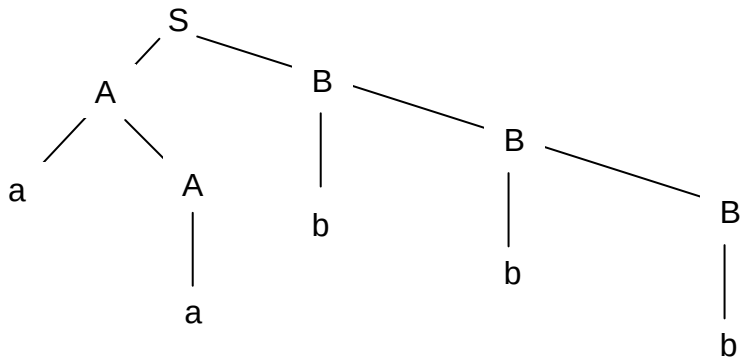
Ejemplo**:** Sea la GIC dada por

$S \rightarrow AB$
$A \rightarrow aA \mid a$
$B \rightarrow bB \mid b$

La cadena aabbb tienen la siguiente derivación:

S => AB => AbB => AbbB => Abbb => aAbbb => aabbb

Con esto, el árbol de derivación será dado por:



Existen muchas derivaciones posibles para la cadena aabbb, por ejemplo:

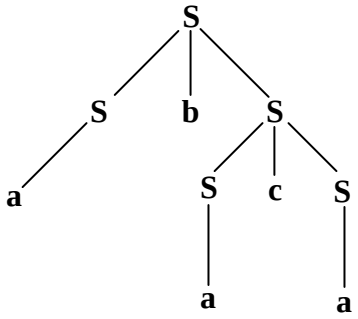
- S => AB => aAB => aaB => aabB => aabbB => aabbb
- S => AB => aAB => aAbB => aAbbB => aAbbb => aabbb

Pero el árbol de derivación será siempre el mismo.Sin embargo, esto no siempre se cumple.

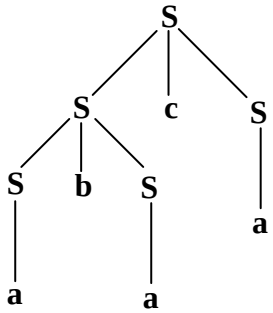
Considérese la gramática: $S \rightarrow SbS \mid ScS \mid a$
la cadena ‘abaca’ se puede derivar:

1. S => SbS => SbScS => SbSca => Sbaca => abaca.

2. $S \Rightarrow ScS \Rightarrow SbScS \Rightarrow abScS \Rightarrow abacS \Rightarrow abaca$



Derivación 1



Derivación 2

En este caso, para una misma cadena, se obtuvieron árboles de derivación distintos.

Entonces, diremos que una gramática es “**ambigüa**” cuando hay dos o más árboles de derivación distintos para una misma cadena.

En algunos casos, si la gramática es ambigua, se puede encontrar otra gramática equivalente pero que no sea ambigüa. Por ejemplo:

$S \rightarrow A \mid B$
$A \rightarrow a$
$B \rightarrow a$

... es ambigua porque tiene árboles de derivación distintos para la cadena ‘a’. Una gramática equivalente que no es ambigüa es:

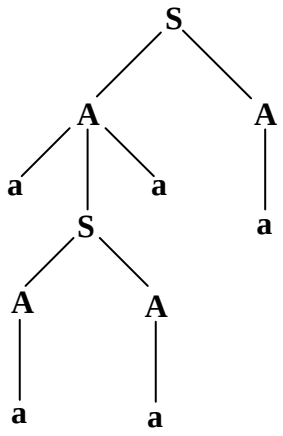
$S \rightarrow a$

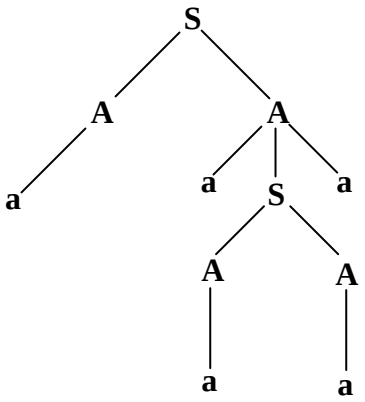
Si todas las GIC para un lenguaje son ambigüas, se dice que el lenguaje es un lenguaje independiente del contexto (LIC) inherentemente ambigüo.

Ejemplo: Considere la gramática G_1 dada por:

$N = \{S,A\}$; $\sum = \{a\}$; S ; $P: \{S \rightarrow AA ; A \rightarrow aSa ; A \rightarrow a\}$

Se pueden obtener 2 árboles de derivación para la cadena ‘aaaaa’:





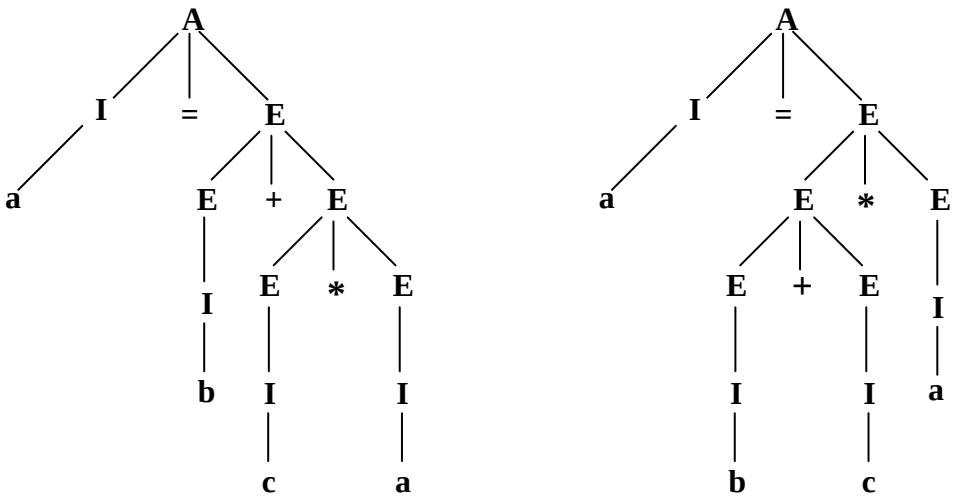
La gramática G_1 no es una gramática regular, pero es relativamente fácil mostrar que $L(G_1)$ es el conjunto regular $\{a^2,a^5,a^8,a^{11},a^{14}, ...\}$. La ambigüedad no es inherente al lenguaje y puede definirse una gramática libre de contexto mucho más simple:

$$G_2 = (\{T\}, \{a\}, T, \{T \rightarrow aaaT, T \rightarrow aa\})$$

Esta gramática es claramente no ambigüa. La ambigüedad no es una característica deseable en una gramática que describe un lenguaje, especialmente en un lenguaje de programación. No sería claro cuál es el árbol de derivación que tendría que usarse para inferir el significado de una cadena. Para ilustrar lo anterior, considere el siguiente ejemplo:

G: $A \rightarrow I=E$
$I \rightarrow a \mid b \mid c$
$E \rightarrow E+E \mid E*E \mid (E) \mid I$

La cadena: a=b+c*a , es una cadena de este lenguaje y se pueden obtener 2 árboles de derivación para ella:



Si se pretende obtener el valor del resultado de la expresión a la derecha del operador de asignación (=), se obtienen 2 resultados posibles, b+(c*a) o (b+c)*a .En general, estos resultados no son iguales.

LECCION 19. - FORMAS CANÓNICAS PARA LAS GIC

La definición de una gramática independiente del contexto es demasiado amplia, y por lo tanto, es deseable establecer una forma canónica que restrinja los tipos de producciones que pueden utilizarse.

Una producción $A \rightarrow B$ en una GIC $G = (N, \sum, S, P)$ es útil si esta es parte de una derivación comenzando con el símbolo incial y terminando con una cadena terminal. Esto es, $A \rightarrow B$ es útil si hay una derivación:

$$S \overset{*}{\Rightarrow} w_1Aw_2 \overset{*}{\Rightarrow} w_1Bw_2 \overset{*}{\Rightarrow} x, \text{ donde } x \in \sum^*$$

- Una producción que no es útil es llamada “no usada”.
- Un noterminal que no aparece en ninguna producción útil es llamado “no usado”.
- Un no terminal que no está entre los terminales “no usados”, es llamdo “útil”.

Ejemplo:

(1) S → gAe | aYB | CY
(2) A → bBY | ooC
(3) B → dd | D
(4) C → jVB | gi
(5) D → n
(6) U → kW

1. Para el no terminal **W**, es imposible encontrar una derivación que iniciando en S produzca una sentencia que contenga W. Lo mismo ocurre con **U**.

2. No hay derivaciones que contengan el no terminal **Y** y que puedan producir

(7) $V \rightarrow baXXX \mid oV$
(8) $W \rightarrow c$
(9) $X \rightarrow fV$
(10) $Y \rightarrow Yhm$

una cadena terminal. Lo mismo ocurre con **X** y **V**.

3. **B** se utiliza en conjunto con no terminales “no usados” y por lo tanto es “no usado” también. Con **D** ocurre lo mismo.

Todo lenguaje independiente del contexto L (no vacío) puede ser generado por una gramática libre de contexto que contenga sólo producciones útiles y no terminales útiles.

Una producción de la forma $A \rightarrow B$, donde $A, B \in N$ (no terminales), es llamada una producción unitaria o producción no generativa.

Ejemplo:

$A \rightarrow B$
$B \rightarrow w_1 \mid C$

Se puede eliminar $A \rightarrow B$ e incluir la producción $A \rightarrow w_1 \mid C$.

Utilizando todo lo anterior al ejemplo de la página anterior, tendremos como resultado la gramática:

$S \rightarrow gAe$
$A \rightarrow ooC$
$C \rightarrow gi$

Una gramática independiente del contexto esta en la forma normal de Chomsky si no contiene producciones λ ($A \rightarrow \lambda$) y si todas las producciones son de la forma $A \rightarrow a$, para $a \in \sum$, o de la forma $A \rightarrow BC$, donde B, C son no terminales. Esto es, en la forma normal de Chomsky, el lado derecho de cada producción contiene un único símbolo terminal o un par de no terminales.

Cualquier lenguaje libre de contexto L puede ser generado por una gramática libre de contexto en la forma normal de Chomsky.

A partir de lo anterior se infiere que cualquier gramática libre de contexto puede ser transformada a la forma normal de Chomsky. La estrategia básica será agregar nuevos símbolos no terminales a aquellas producciones no normalizadas tales como por ejemplo $A \rightarrow JKcb$ y reemplazarlas por un conjunto de producciones equivalentes tales como:

$$A \rightarrow JY_1 \quad , \quad Y_1 \rightarrow KY_2 \quad , \quad Y_2 \rightarrow X_cX_b \quad , \quad X_c \rightarrow c \quad , \quad X_b \rightarrow b$$

Donde Y_1, Y_2, X_c y X_b son nuevos símbolos no terminales.

Ejemplo:

G_1:	**G_2:**
(1) $S \rightarrow SABC$	$S \rightarrow SY_{11}$, $Y_{11} \rightarrow AY_{12}$, $Y_{12} \rightarrow BC$
(2) $S \rightarrow be$	$S \rightarrow X_bX_e$
(3) $S \rightarrow CBh$	$S \rightarrow CY_{31}$, $Y_{31} \rightarrow BX_h$
(4) $A \rightarrow aaC$	$A \rightarrow X_aY_{41}$, $Y_{41} \rightarrow X_aC$
(5) $B \rightarrow Sf$	$B \rightarrow SX_f$
(6) $B \rightarrow ggg$	$B \rightarrow X_gY_{61}$, $Y_{61} \rightarrow X_gX_g$
(7) $C \rightarrow cA$	$C \rightarrow X_cA$
(8) $C \rightarrow d$	$C \rightarrow d$
	$X_b \rightarrow b$, $X_e \rightarrow e$, $X_h \rightarrow h$, $X_a \rightarrow a$, $X_f \rightarrow f$, $X_g \rightarrow g$

Gramáticas Libres de Contexto

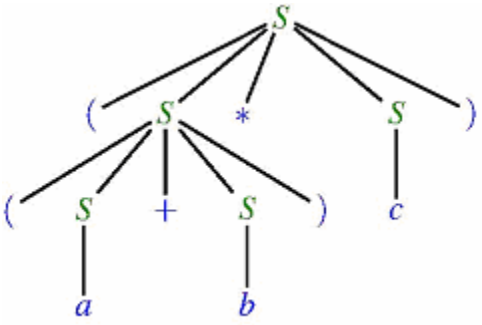


Figura 22: Arbol de Derivación

4.19.1. Simplificación De Las Gramáticas Libres De Contexto

Para un mismo lenguaje de tipo 2 existen muchas gramáticas libres de contexto que lo generan. Estas será n de formas muy diversas por lo que, en general se hace muy difícil trabajar con ellas. Por este motivo interesa simplificarlas lo mas posible y definir unas formas normales para las gramáticas que las hagan mas homogéneas.

4.19.2. Eliminación de Símbolos y Producciones Inútiles

Un símbolo $X \in (V \cup T)$ se dice útil si y solo si existe una cadena de derivaciones en G tal que $S \Rightarrow \propto X \beta \Rightarrow w \in T$

es decir si interviene en la derivación de alguna palabra del lenguaje generado por la gramática. Una producción se dice útil si y solo si todos sus símbolos son útiles. Esto es equivalente a que pueda usarse en la derivación de alguna palabra del lenguaje asociado a la gramática. Esta claro que eliminando todos los símbolos y producciones inútiles el lenguaje generado por la gramática no cambia.

El algoritmo para eliminar los símbolos y producciones inútiles consta de dos pasos fundamentales:

1. Eliminar las variables desde las que no se puede llegar a una palabra de T y las producciones en las que aparezcan.
2. Eliminar aquellos símbolos que no sean alcanzables desde el estado inicial, S, y las producciones en las que estos aparezcan.

El primer paso se realiza con el siguiente algoritmo (V' es un conjunto de variables):
1. $V' = 0$
2. Para cada producción de la forma $A \rightarrow w$, A se introduce en V'.
3. Mientras V'cambie
4. Para cada producción $B \rightarrow \propto$
5. Si todas las variables de $\propto$ pertenecen a V' ,B se introduce en V'
6. Eliminar las variables que esten en V y no en V'
7. Eliminar todas las producciones donde aparezca una variable de las eliminadas en el paso anterior

El segundo paso se realiza con el siguiente algoritmo:

– V''y J son conjuntos de variables. J son las variables por analizar.
– T' es un conjunto de símbolos terminales

1. $J = \{S\}$ $V'' = \{S\}$ $T' = 0$
2. Mientras $J \neq 0$
3. Extraer un elemento de J : A, ($J = J - \{A\}$).
4. Para cada produccion de la forma $A \rightarrow \propto$
5.Para cada variable B en $\propto$
6. Si B no está en V'' añadir B a J y a V''
7. Poner todos los simbolos terminales de $\propto$ en T'
8. Eliminar todas las variables que no esten en V'' y todos los simbolos terminales que no esten en T' .

9. Eliminar todas las producciones donde aparezca un simbolo o variable de los eliminados

Ejemplo: Es importante aplicar los algoritmos anteriores en el orden especificado para que se garantice que se eliminan todos los símbolos y variables inútiles.

Como ejemplo de lo anterior, supongamos que tenemos la gramática dada por

$$S \to AB,\ S \to a,\ A \to a$$

En el primer algoritmo se elimina B y la producción S $\to$ AB.
Entonces en el segundo se elimina la variable A y la producción A $\to$ a.
Sin embargo, si aplicamos primero el segundo algoritmo, entonces no se elimina nada. Al aplicar despues el primero de los algoritmos se elimina B y la producción S $\to$AB. En definitiva, nos queda la gramática S $\to$ a, A $\to$ a donde todavía nos queda la variable inútil A.

Ejemplo: Eliminar símbolos y producciones inútiles de la gramática
S $\to$ gAe, S $\to$ aY B, S $\to$ cY,
A $\to$ bBY, A $\to$ ooC, B $\to$ dd,
B $\to$ D, C $\to$ jV B, C $\to$ gi,
D $\to$ n, U $\to$ kW, V $\to$ baX X X ,
V $\to$ oV, W $\to$ c, X $\to$ f V, Y $\to$ Y hm

se aplica el primer algoritmo.

Inicialmente V´ tiene las variables V = {B, D, C, W }.
En la siguiente iteración añadimos a V´ las variables que se alcanzan desde estas: A y W . V´ queda igual a {B, D, C, W, A, U }.
En la siguiente V´ = {B, D, C, W, A, U, S}.
En la siguiente V´ = {B, D, C, W, A, U, S}.

Como V´ no cambia ya se ha terminado el ciclo. Estas son las variables desde las que se alcanza una cadena de símbolos terminales. El restos de las variables: X , Y, V, son inútiles y se pueden eliminar. También se eliminan las producciones asociadas. La gramática resultante es:

S $\to$ gAe, A $\to$ ooC,
B $\to$ dd, B $\to$ D, C $\to$ gi,
D $\to$ n, U $\to$ kW, W $\to$ c
A esta gramática le aplicamos el segundo algoritmo

J = {S}, V´´ = {S}, T = 0

Tomando S de J y ejecutando las instrucciones del ciclo principal nos queda
J = {A}, V´´ = {S, A}, T = {g, e}

Tomando A de J, tenemos
J = {C}, V´´ = {S, A, C}, T = {g, e, o}

Sacando C de J, obtenemos
J = 0, V = {S, A, C}, T´ = {g, e, o, i}

Como J = 0 se acaba el ciclo. Solo nos queda eliminar las variables inútiles: B, D, U, W, los símbolos terminales inútiles: n, k, c, d, y las producciones donde estos aparecen. La gramática queda

S → gAe, A → ooC, C → gi

En esta gramática solo se puede generar una palabra: googige.

Si el lenguaje generado por una gramática es vacío, esto se detecta en que la variable S resulta inútil en el primer algoritmo. En ese caso se pueden eliminar directamente todas las producciones, pero no el símbolo *S*.

Ejemplo eliminar símbolos y producciones inútiles de la gramática

S → aSb,S → ab,S →bcD,

S →cSE,E →aDb,F → abc,E → abF

4.19.3. Producciones Nulas

Las producciones nulas son las de la forma $A \rightarrow \varepsilon$. Vamos a tratar de eliminarlas sin que cambie el lenguaje generado por la gramática ni la estructura de los árboles de derivación. Evidentemente si $L(G)$ $\varepsilon\in$ $L(G)$ no vamos a poder eliminarlas todas sin que la palabra nula deje de generarse. Así vamos a dar un algoritmo que dada una gramática G, construye una gramática $G´$ equivalente a la anterior sin producciones nulas y tal que $L(G´)= L(G) -\{\varepsilon\}$. Es decir, si la palabra nula era generada en la gramática original entonces no puede generarse en la nueva gramática.

Primero se calcula el conjunto de las variables anulables:
H es el conjunto de las variables anulables
1. $H = 0$
2. Para cada produccion $A \rightarrow \varepsilon$, se hace $H = H \cup \{A\}$
3. Mientras *H* cambie
4. Para cada produccion $B \rightarrow A_1A_2 ...A_n$,
donde $A_i \in H$ para todo $i = 1,...,n$, se hace $H = H \cup \{B\}$

Una vez calculado el conjunto, H, de las variables anulables, se hace lo siguiente:

1. Se eliminan todas las producciones nulas de la gramatica
2. Para cada produccion de la gramatica de la forma $A \rightarrow \propto_1 ... \propto_n$, donde $\alpha_i \in V \cup T$
3. Se elimina la produccion $A \rightarrow \propto_1 ... \propto_n$
4. Se añaden todas las producciones de la forma $A \rightarrow \beta_1 ... :\beta_n$
5. donde $\beta_i = \propto_i$ si $\propto_i \notin H$

$$(\beta_i = \propto_i) \rightarrow (\beta_i = \varepsilon) \text{ si } \propto \in H$$
y no todos los β_i puedan ser nulos al mismo tiempo

G´ es la gramática resultante después de aplicar estos dos algoritmos.

Si G generaba inicialmente la palabra nula, entonces, a partir de *G´*, podemos construir una gramática *G´´* con una sola producción nula y que genera el mismo lenguaje que *G*. para ello se añade una nueva variable, *S´*, que pasa a ser el símbolo inicial de la nueva gramática, *G´´*. También se añaden dos producciones:

$$S´ \rightarrow S,\ S´ \rightarrow \varepsilon$$

Ejemplo: Eliminar las producciones nulas de la siguiente gramática:

$$S \rightarrow ABb,\ S \rightarrow ABC,\ C \rightarrow abC,$$
$$B \rightarrow bB,\ B \rightarrow \varepsilon,\ A \rightarrow aA,$$
$$A \rightarrow \varepsilon,\ C \rightarrow AB$$

Las variables anulables despues de ejecutar el paso 2 del primer algoritmo son B y A. Al ejecutar el paso 3, resulta que C y S son también anulables, es decir H = {A, B, C, S}. Al ser S anulable la palabra vacía puede generarse mediante esta gramática. En la gramática que se construye con el segundo algoritmo esta palabra ya no se podrá generar. Al ejecutar los pasos 3 y 4 del segundo algoritmo para las distintas producciones no nulas de la gramática resulta

3. Se elimina la producción $S \rightarrow ABb$
4. Se añade $S \rightarrow ABb,\ S \rightarrow Ab,\ S \rightarrow Bb$

3. Se elimina $S \rightarrow ABC$
4. Se añade $S \rightarrow ABC,\ S \rightarrow AB,\ S \rightarrow AC,$
$S \rightarrow BC,\ S \rightarrow A,\ S \rightarrow B,\ S \rightarrow C$

3. Se elimina $C \rightarrow abC$
4. Se añade $C \rightarrow abC,\ C \rightarrow ab$

3. Se elimina $B \rightarrow bB$

4. Se añade $B \rightarrow bB$, $B \rightarrow b$

3. Se elimina $A \rightarrow aA$
4. Se añade $A \rightarrow aA$, $A \rightarrow a$

3. Se elimina $C \rightarrow AB$
4. Se añade $C \rightarrow AB$, $C \rightarrow A$, $C \rightarrow B$

En definitiva, la gramática resultante tiene las siguientes producciones:

$$S \rightarrow ABb,\ S \rightarrow Ab,\ S \rightarrow Bb,\ S \rightarrow ABC,\ S \rightarrow AB,\ S \rightarrow AC,\ S \rightarrow BC,\ S \rightarrow A,\ S \rightarrow B,$$
$$S \rightarrow C,\ C \rightarrow abC,\ C \rightarrow ab,\ B \rightarrow bB,\ B \rightarrow b,\ A \rightarrow aA,$$
$$A \rightarrow a,\ C \rightarrow AB,\ C \rightarrow A,\ C \rightarrow B$$

Ejemplo: Sea la gramática
$S \rightarrow aHb$, $H \rightarrow aHb$, $H \rightarrow \varepsilon$

La única variable anulable es H. Y la gramática equivalente, que resulta de aplicar el algoritmo 2 es:

$S \rightarrow aHb$, $H \rightarrow aHb$, $S \rightarrow ab$, $H \rightarrow ab$

4.19.4. Producciones Unitarias

Las producciones unitarias son las que tienen la forma
$A \rightarrow B$
donde $A, B \in V$. Veamos como se puede transformar una gramática G, en la que $\varepsilon \notin L(G)$, en otra gramática equivalente en la que no existen producciones unitarias. Para ello, hay que partir de una gramática sin producciones nulas. Entonces, se calcula el conjunto H de parejas (A, B) tales que B se puede derivar a partir de
A: $A \Rightarrow B$. Eso se hace con el siguiente algoritmo

1. $H = 0$
2. Para toda produccion de la forma $A \rightarrow B$, la pareja (A, B) se introduce en H.
3. Mientras H cambie
4. Para cada dos parejas (A, B), (B, C)
5. Si la pareja (A, C) no est en H
(A, C) se introduce en H

6. Se eliman las producciones unitarias
7. Para cada produccion $A \rightarrow \propto$
8. Para cada pareja $(B, A) \in H$
9. Se añade una produccion $B \rightarrow \propto$

Ejemplo: Consideremos la gramática resultante de un ejemplo anterior

$$S \to ABb,\ S \to Ab,\ S \to Bb,\ S \to ABC,\ S \to AB,\ S \to AC,\ S \to BC,\ S \to A,\ S \to B,$$
$$S \to C,\ C \to abC,\ C \to ab,\ B \to bB,\ B \to b,\ A \to aA,$$
$$A \to a,\ C \to AB,\ C \to A,\ C \to B$$

El conjunto H est formado por las parejas {(S, A), (S, B), (S, C), (C, A), (C, B)}.

Entonces se eliminan las producciones $S \to A$, $S \to B$, $S \to C$, $C \to A$, $C \to B$

Se añaden a continuacion las producciones (no se consideran las repetidas*)*

$S \to a$, $S \to aA$, $S \to bB$, $S \to b$, $S \to abC$, $S \to ab$, $C \to aA$, $C \to a$, $C \to bB$, $C \to b$

Ejemplo: El lenguaje $L = \{a^n b^n : n \geq 1\} \cup \{a^n ba^n : n \geq 1\}$ viene generado por la siguiente gramática de tipo 2:

S →A, S →B, A →ab, A →aHb
H →ab, H →aHb, B →aBa, B →b

Las parejas resultantes en el conjunto H son {(S, A), (S, B)}. La gramática sin producciones unitarias equivalente es:

A→ab, A→aHb, H→ab, H→aHb, B→aBa, B→b, S→ab, S→aHb, S→aBa, S→b.

LECCION 20. - FORMAS NORMALES

4.20.1. Forma Normal de Chomsky

Una gramática de tipo 2 se dice que está en forma normal de Chomsky si y solo si todas las producciones tienen la forma
$A \to BC$, $A \to a$,
donde $A,B,C \in V$, $a \in T$.

Toda gramática de tipo 2 que no acepte la palabra vacía se puede poner en forma normal de Chomsky. Para ello lo primero que hay que hacer es suprimir las producciones nulas y unitarias. A continuación se puede ejecutar el siguiente algoritmo:

1. Para cada produccion de la forma $A \to \alpha_1 \ldots \alpha_n$, $\alpha_i \in (V \cup T)$,$n \geq 2$
2. Para cada α_i, si α_i es terminal: $\alpha_i = a \in T$
3. Se añade la produccion $Ca \to a$
4. Se cambia α_i por Ca en $A \to \alpha_1 \ldots \alpha_n$
5. Para cada produccion de la forma $A \to B_1,\ldots B_m$, $m \geq 3$

6. Se añaden (m −2)variables $D_1,D_2,...,D_{m-2}$ (distintas para cada produccion)
7. La produccion $A \rightarrow B_1 ...B_m$ se reemplaza por
$A \rightarrow B_1D_1$, $D_1 \rightarrow B_2D_2$, ..., $D_{m-2} \rightarrow B_{m-1}B_m$

Ejemplo *Sea la Gramática G =({S,A,B},{a,b},P,S) dada por las producciones*
S → bA | aB, A → bAA | AS | a, B → aBB | bS | b
Para pasarla a forma normal de Chomsky, en el ciclo asociado al paso 1 se añaden las producciones
Ca → a, Cb → b
y las anteriores se transforman en

S → CbA | CaB, A → CbAA | AS | Ca, B → CaBB | CbS | Cbb
Al aplicar el paso asociado al paso 5, la gramática queda

S → CbA | CbB, A → CbD1 | AS | a, D1 → AA,
B → CaE1 | CbS | b, E1 → B, Ca → a, Cb → b
Con esto la gramática ya está en forma normal de Chomsky.

4.20.2. Forma Normal de Greibach

Una gramática se dice que está en forma normal de Greibach si y solo si todas las producciones tienen la forma $A \rightarrow a\propto$ donde $a \in T$, $\propto \in V^*$. Toda gramática de tipo 2 que no acepte la palabra vacía se puede poner en forma normal de Greibach. Para ello hay que partir de una gramática en forma normal de Chomsky y aplicarle el siguiente algoritmo. En realidad no es necesario que la gramática esté en forma normal de Chomsky. Basta que todas las producciones sean de uno de los tipos siguientes:
– $A \rightarrow a\propto$, $a \in T, \alpha \in V^*$.
– $A \rightarrow \propto$, $\propto \in V^*$.

Claro está, en una gramática en forma normal de Chomsky, todas las producciones son de alguno de estos dos tipos. En este algoritmo se supone que el conjunto de variables inicial de la gramática está numerado

$V = \{A_1,...,A_m\}$.

El algoritmo se basa en dos operaciones básicas. La primera es eliminar una producción, $A \rightarrow B\propto$ de la gramática G, donde $A \neq B$. Esto se hace con los siguientes pasos:

1. Eliminar $A \rightarrow B\propto$
2. Para cada produccion $B \rightarrow \beta$
3. Añadir $A \rightarrow \beta\propto$

La otra operación básica consiste en eliminar todas las producciones del tipo $A \rightarrow A\propto$ donde $\propto \in V^*$. Esto se hace siguiendo los siguiente pasos:

1. Añadir una nueva variable B_A
2. Para cada produccion $A \rightarrow A\propto$
3. Añadir $B_A \rightarrow \propto$ y $B_A \rightarrow \propto B_A$
4. Eliminar $A \rightarrow A\propto$
5. Para cada produccion $A \rightarrow \beta$ β no empieza por A
6. Añadir $A \rightarrow \beta B_A$

Llamemos $ELIMINA_1(A \rightarrow B\propto)$ a la función que realiza el primer paso y $ELIMINA_2(A)$ a la función que reliza el segundo paso. Si si llama a $ELIMINA_2(A_j)$, la variable que añadimos la notaremos como B_j.

En estas condiciones vamos a realizar un algoritmo, al final del cual todas las producciones tengan una forma que corresponda a alguno de los patrones siguientes:

– $A \rightarrow a\propto$, $a \in T, \alpha \in V^*$.
– $A_i \rightarrow A_j\propto$, $j > i$, $\propto \in V^*$.
– $B_j \rightarrow A_i\propto$, $\propto \in V^*$

El algoritmo es como sigue:

1. Para cada $k = 1,...,m$
2. Para cada $j = 1,...,k-1$
3. Para cada produccion $A_k \rightarrow A_j\alpha$
4. $ELIMINA_1(A_k \rightarrow A_j\alpha)$
5. Si existe alguna produccion de la forma $A_k \rightarrow A_k\alpha$
6. $ELIMINA_2(A_k)$

A continuación se puede eliminar definitivamente la recursividad por la izquierda con el siguiente algoritmo pasando a forma normal de Greibach

1. Para cada $i = m-1,...,1$
2. Para cada produccion de la forma $A_i \rightarrow A_j\alpha$, $j > i$
3. $ELIMINA_1(A_i \rightarrow A_j\alpha)$
4. Para cada $i = 1,2,...,m$
5. Para cada produccion de la forma $B_j \rightarrow A_i\alpha$.
6. $ELIMINA_1(B_j \rightarrow A_i\alpha)$

El resultado del segundo algoritmo es ya una gramática en forma normal de Greibach.

Ejemplo: Pasar a forma normal de Greibach la gramática dada por las producciones

$A_1 \to A_2A_3$, $A_2 \to A_3A_1$, $A_2 \to b$, $A_3 \to A_1A_2$, $A_3 \to a$
Aplicamos ELIMINA$_1$ a $A_3 \to A_1A_2$. Se elimina esta producción y se añade: $A_3 \to A_2A_3A_2$

Queda:
$A_1 \to A_2A_3$, $A_2 \to A_3A_1$, $A_2 \to b$, $A_3 \to a$, $A_3 \to A_2A_3A_2$
Aplicamos ELIMINA$_1$ a $A_3 \to A_2A_3A_2$

Se elimina esta producción y se añaden: $A_3 \to A_3A_1A_3A_2$, $A_3 \to bA_3A_2$

Queda:

$A_1 \to A_2A_3$, $A_2 \to A_3A_1$, $A_2 \to b$, $A_3 \to a$, $A_3 \to A_3A_1A_3A_2$, $A_3 \to bA_3A_2$

Aplicamos ELIMINA$_2$ a A_3 Se añade B_3 y las producciones $B_3 \to A_1A_3A_2$, $B_3 \to A_1A_3A_2B_3$ Se elimina $A_3 \to A_3A_1A_3A_2$. Se añaden las producciones: $A_3 \to aB_3$, $A_3 \to bA_3A_2B_3$

Queda:
$A_1 \to A_2A_3$, $A_2 \to A_3A_1$, $A_2 \to b$, $A_3 \to a$, $A_3 \to bA_3A_2$ $B_3 \to A_1A_3A_2$, $B_3 \to A_1A_3A_2B_3$ $A_3 \to aB_3$, $A_3 \to bA_3A_2B_3$

Se aplica ELIMINA$_1$ a $A_2 \to A_3A_1$.

Se elimina esta producción y se añaden:

$A_2 \to aA_1$, $A_2 \to aB_3A_1$, $A_2 \to bA_3A_2B_3A_1$, $A_2 \to bA_3A_2A_1$

Queda:
$A_1 \to A_2A_3$, $A_2 \to b$, $A_2 \to aA_1$, $A_2 \to aB_3A_1$, $A_2 \to bA_3A_2B_3A_1$, $A_2 \to bA_3A_2A_1$, $A_3 \to a$, $A_3 \to bA_3A_2$, $B_3 \to A_1A_3A_2$, $B_3 \to A_1A_3A_2B_3$ $A_3 \to aB_3$, $A_3 \to bA_3A_2B_3$

Se aplica ELIMINA$_1$ a $A_1 \to A_2A_3$.

Se elimina esta producción y se añaden:

$A_1 \to bA_3$, $A_1 \to aA_1A_3$, $A_1 \to aB_3A_1A_3$, $A_1 \to bA_3A_2B_3A_1A_3$, $A_1 \to bA_3A_2A_1A_3$

Queda:

$A_2 \rightarrow b$, $A_2 \rightarrow aA_1$, $A_2 \rightarrow aB_3A_1$,
$A_2 \rightarrow bA_3A_2B_3A_1$, $A_2 \rightarrow bA_3A_2A_1$, $A_3 \rightarrow a$,
$A_3 \rightarrow bA_3A_2$, $B_3 \rightarrow A_1A_3A_2$, $B_3 \rightarrow A_1A_3A_2B_3$,
$A_3 \rightarrow aB_3$, $A_3 \rightarrow bA_3A_2B_3$, $A_1 \rightarrow bA_3$,
$A_1 \rightarrow aA_1A_3$, $A_1 \rightarrow aB3A1A3$, $A_1 \rightarrow bA_3A_2B_3A_1A_3$,
$A_1 \rightarrow bA_3A_2A_1A_3$

Se aplica ELIMINA$_1$ a $B_3 \rightarrow A_1A_3A_2$ Se elimina esta producción y se añaden:
$B_3 \rightarrow bA_3A_3A_2$, $B_3 \rightarrow aA_1A_3A_3A_2$, $B_3 \rightarrow aB_3A_1A_3A_3A_2$,
$B_3 \rightarrow bA_3A_2B_3A_1A_3A_3A_2$, $B_3 \rightarrow aB_3A_1A_3A_3A_2$,

Queda:
$A_2 \rightarrow b$, $A_2 \rightarrow aA_1$ $A_2 \rightarrow aB_3A_1$,
$A_2 \rightarrow bA_3A_2B_3A_1$, $A_2 \rightarrow bA_3A_2A_1$ $A_3 \rightarrow a$,
$A_3 \rightarrow bA_3A_2$, $B_3 \rightarrow A_1A_3A_2B_3$, $A_3 \rightarrow aB_3$,
$A_3 \rightarrow bA_3A_2B_3$, $A_1 \rightarrow bA_3$, $A_1 \rightarrow aA_1A_3$,
$A_1 \rightarrow aB_3A_1A_3$, $A_1 \rightarrow bA_3A_2B_3A_1A_3$, $A_1 \rightarrow bA_3A_2A_1A_3$
$B_3 \rightarrow bA_3A_3A_2$, $B_3 \rightarrow aA_1A_3A_3A_2$, $B_3 \rightarrow aB_3A_1A_3A_3A_2$,
$B_3 \rightarrow bA_3A_2B_3A_1A_3A_3A_2$, $B_3 \rightarrow aB_3A_1A_3A_3A_2$,

Se aplica ELIMINA$_1$ a $B_3 \rightarrow A_1A_3A_2B_3$. Se elimina esta producción y se añaden:
$B_3 \rightarrow bA_3A_3A_2B_3$, $B_3 \rightarrow aA_1A_3A_3A_2B_3$, $B_3 \rightarrow aB_3A_1A_3A_3A_2B_3$, $B_3 \rightarrow bA_3A_2B_3A_1A_3A_3A_2B_3$,
$B_3 \rightarrow aB_3A_1A_3A_3A_2B_3$,

Resultado:

$A_2 \rightarrow b$, $A_2 \rightarrow aA_1$, $A_2 \rightarrow aB_3A_1$,
$A_2 \rightarrow bA_3A_2B_3A_1$, $A_2 \rightarrow bA_3A_2A_1$ $A_3 \rightarrow a$,
$A_3 \rightarrow bA_3A_2$, $A_3 \rightarrow aB_3$, $A_3 \rightarrow bA_3A_2B_3$,
$A_1 \rightarrow bA_3$, $A_1 \rightarrow aA_1A_3$, $A_1 \rightarrow aB_3A_1A_3$,
$A_1 \rightarrow bA_3A_2B_3A_1A_3$, $A_1 \rightarrow bA_3A_2A_1A_3$, $B_3 \rightarrow bA_3A_3A_2$,
$B_3 \rightarrow aA_1A_3A_3A_2$, $B_3 \rightarrow aB_3A_1A_3A_3A_2$, $B_3 \rightarrow bA_3A_2B_3A_1A_3A_3A_2$,
$B_3 \rightarrow aB_3A_1A_3A_3A_2B_3$, $B_3 \rightarrow bA_3A_2B_3A_1A_3A_3A_2B_3$, $B_3 \rightarrow aA_1A_3A_3A_2B_3$,
$B_3 \rightarrow aB_3A_1A_3A_3A_2B_3$ $B_3 \rightarrow aB_3A_1A_3A_3A_2B_3$

LECCION 21. - DEFINICIÓN DE AUTÓMATA CON PILA[14]

Por ejemplo el lenguaje de los paréntesis bien balanceados, que se sabe es propiamente un LLC (lenguaje Libre de contexto,Qué máquina se requiere para distinguir las palabras de paréntesis bien balanceados de las que tienen los paréntesis desbalanceados?

Una primera idea podría ser la de una máquina que tuviera un registro aritmético que le permitiera contar los paréntesis; dicho registro sería controlado por el control finito, quien le mandaría símbolos I para incrementar en uno el contador y D para decrementarlo en uno. A su vez, el registro mandaría un símbolo Z para indicar que está en cero, o bien N para indicar que no está en cero. Entonces para analizar una palabra con paréntesis lo que haríamos sería llevar la cuenta de cúantos paréntesis han sido abiertos pero no cerrados; en todo momento dicha cuenta debe ser positiva o cero, y al final del cálculo debe ser exactamente cero. Por ejemplo, para la palabra (())() el registro tomaría sucesivamente los valores 1, 2, 1, 0, 1, 0.

Como segundo ejemplo, considérese el lenguaje de los palíndromos (palabras que se leen igual al derecho y al revés, como ANITALAVALATINA). Aquí la máquina contadora no va a funcionar, porque se necesita recordar toda la primera mitad de la palabra para poder compararla con la segunda mitad. Más bien pensaríamos en una máquina que tuviera la capacidad de recordar cadenas de caracteres arbitrarias, no números. Siguiendo esta idea, podríamos pensar en añadir al AF un almacenamiento auxiliar, que llamaremos pila, donde se podrían ir depositando caracter por caracter cadenas arbitrariamente grandes, como se aprecia en la Figura 23. A estos nuevos autómatas con una pila auxiliar los llamaremos autómatas de pila (AP)

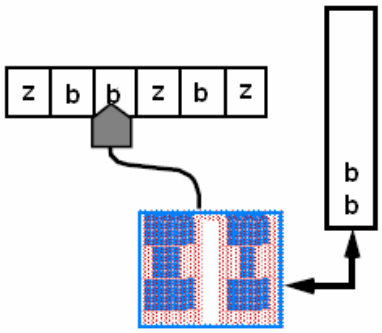


Figura 23. Automata con Pila Auxiliar

[14] MORAL CALLEJÓN Serafín Teoría de autómatas y lenguajes formales, En http://decsai.ugr.es/~smc/docencia/mci/automata.pdf

5.21.1. Funcionamiento de los Autómatas de Pila[15]

La pila funciona de manera que el ultimo carácter que se almacena en ella es el primero en salir ("LIFO" por las siglas en inglés), como si apiláramos platos uno encima de otro, y naturalmente el primero que quitaremos es el último que hemos colocado. Un aspecto crucial de la pila es que sólo podemos modificar su "tope", que es el extremo por donde entran o salen los caracteres. Los caracteres a la mitad de la pila no son accesibles sin quitar antes los que están encima de ellos.
La pila tendrá un alfabeto propio, que puede o no coincidir con el alfabeto de la palabra de entrada. Esto se justifica porque puede ser necesario introducir en la pila caracteres especiales usados como separadores, según las necesidades de diseño del autómata.

Al iniciar la operación de un AP, la pila se encuentra vacía. Durante la operación del AP, la pila puede ir recibiendo (y almacenando) caracteres, según lo indiquen las transiciones ejecutadas. Al final de su operación, para aceptar una palabra, la pila debe estar nuevamente vacía.

En los AP las transiciones de un estado a otro indican, además de los caracteres que se consumen de la entrada, también lo que se saca del tope de la pila, así como también lo que se mete a la pila.

Al igual que los AF, los AP tienen estados finales, que permiten distinguir cuando una palabra de entrada es aceptada.

De hecho, para que una palabra de entrada sea aceptada en un AP se deben cumplir todas las condiciones siguientes:

1. La palabra de entrada se debe haber agotado (consumido totalmente).
2. El AP se debe encontrar en un estado final.
3. La pila debe estar vacía.

los lenguajes generados por las gramáticas libres de contexto también tienen un autómata asociado que es capaz de reconocerlos. Estos autómatas son parecidos a los autómatas finitos determinísticos, solo que ahora tendrán un dispositivo de memoria de capacidad ilimitada: una pila. A continuación daremos la definición formal de autómata con pila no determinístico (APND). Al contrario que en los autómatas finitos, los autómatas con pila no determinísticos y determinísticos no aceptan las mismas familias de lenguajes. Precisamente son los no determísticos

[15] MARTÍN C. John (2004). Lenguajes Formales y teoría de la computación Tercera edición Mcgraw – Hill Interamericana
BRENA PIÑERO, Ramon F. Autómatas y lenguajes un enfoque de diseño (2003) ITESM, En: http://lizt.mty.itesm.mx/~rbrena/AyL.html

los asociados con los lenguajes libres de contexto. Los determinísticos aceptan una familia mas restringida de lenguajes.

Un autómata con pila no determinístico(APND) es una septupla $(Q,A,B,\delta,q_0,Z_0\ F)$ en la que
– Q es un conjunto finito de estados
– A es un alfabeto de entrada
– B es un alfabeto para la pila
– δ es la función de transición

$\delta : Q \times (A \cup \{ \varepsilon\}) \times B \rightarrow \vartheta Q \times B^*)$
– q_0 es el estado inicial
– Z_0 es el símbolo inicial de la pila
– F es el conjunto de estados finales

La función de transición aplica cada estado, cada símbolo de entrada (incluyendo la cadena vacía) y cada símbolo tope de la pila en un conjunto de posibles movimientos. Cada movimiento parte de un estado, un símbolo de la cinta de entrada y un símbolo tope de la pila. El movimento en sí consiste en un cambio de estado, en la lectura del símbolo de entrada y en la subtitución del símbolo tope de la pila por una cadena de símbolos.

Ejemplo Sea el autómata M =({ q1, q2} , { 0, 1, c} , { R, B, G} , δ, q_1, R, 0)donde

δ (q1, 0, R)={ (q1, BR)}	δ (q1, 1, R)={ (q1, GR)}
δ (q1, 0, B)={ (q1, BB)}	δ (q1, 1, B)={ (q1, GB)}
δ (q1, 0, G)={ (q1, BG)}	δ (q1, 1, G)={ (q1, GG)}
δ (q1, c, R)={ (q2, R)}	δ (q1, c, B)={ (q2, B)}
δ (q1, c, G)={ (q2, G)}	δ (q2, 0, B)={ (q2, ε)}
δ (q2, 1, G)={ (q2, ε)}	δ (q2, ε, R)={ (q2, ε)}

La interpretación es que si el autómata está en el estado q_1 y lee un 0 entonces permanece en el mismo estado y añade una B a la pila; si lo que lee es un 1, entonces añade una G; si lee una c pasa a q_2. En q_2 se saca una B por cada 0, y una G por cada 1.

Se llama descripción instantánea o configuración de un autómata con pila a una tripleta $(q,u,\alpha) \in Q \times A^* \times B^*$ en la que q es el estado en el se encuentra el autómata, u es la parte de la cadena de entrada que queda por leer y α el contenido de la pila (el primer símbolo es el tope de la pila).

Definición:Se dice que de la configuración $(q,au,Z\alpha)$ se puede llegar a la configuración $(p,u,\beta\alpha)$y se escribe $(q,au,Z\alpha) \vdash (p,u,\beta\alpha)$ si y solo si $(p,\beta) \in \delta(q,a,Z)$

donde a puede ser cualquier símbolo de entrada o la cadena vacía.

Definición:Si C_1 y C_2 son dos configuraciones, se dice que se puede llegar de C_1 a C_2 mediante una sucesión de pasos de cálculo y se escribe $C_1 \vdash C_2$ si y solo si existe una sucesión de configuraciones $T_1,...,T_n$ tales que C1 =T1 ├ T ├ Tn−1 ├ Tn =C2

Teorema: a) Si M es un APND entonces existe otro autómata M´, tal que N(M)=L(M´)
b) Si M es un APND entonces existe otro autómata M´, tal que L(M)=N(M´).

Demostración:

-a) Si M =(Q,A,B,δ,q_0,Z_0,F), entonces el autómata M´ se construye a partir de M siguiendo los siguientes pasos:

–Se añaden dos estados nuevos, q'_0 y q_f . El estado inicial de *M´* será q'_0 y q_f será estado final de *M´* .

–Se añade un nuevo símbolo a *B*: Z'_o . Este será el nuevo símbolo inicial de la pila.

–Se mantienen todas las transiciones de *M*, añadiéndose las siguientes:

$\delta(q'_0,\varepsilon,Z'_0)=\{(q_0,Z_0Z'_0)\}$
$\delta(q,\varepsilon,Z'_0)=\{(q_f,Z'_0)\}, \forall q \in Q$

b) Si *M* =(*Q*, *A*, *B*, δ, q_0, Z_0, *F*), entonces el autómata *M´* se construye a partir de *M* siguiendo los siguientes pasos:
– Se añaden dos estados nuevos, q_0 y q_s. El estado incial de *M´* será q_0.
– Se añade un nuevo símbolo a *B*: Z'_0 . Este será el nuevo símbolo inicial de la pila.
– Se mantienen todas las transiciones de *M*, añadiéndose las siguientes:

$\delta(q_0,\varepsilon,Z'_0)=\{(q_0,Z_0Z'_0)\}$
$\delta(q,\varepsilon,H)=\{(q_s,H)\}, \forall q \in F, H \in B \cup \{Z'_0\}$
$\delta(qs,\varepsilon,H)=\{(qs,\varepsilon)\}, \forall H \in B \cup \{Z'_0\}$

LECCION 22. - DISEÑO DE AUTÓMATAS DE PILA

El problema de diseño de los AP consiste en obtener un AP M que acepte exactamente un lenguaje L dado. Por exactamente queremos decir, como en el caso de los autómatas finitos, que, por una parte, todas las palabras que acepta

efectivamente pertenecen a L, y por otra parte, que M es capaz de aceptar todas las palabras de L.

Aunque en el caso de los AP no hay metodologías tan generalmente aplicables como era el caso de los autómatas finitos, siguen siendo válidas las ideas básicas del diseño sistemático, en particular establecer claramente qué es lo que "recuerda" cada estado del AP antes de ponerse a trazar transiciones a diestra y siniestra. Para los AP, adicionalmente tenemos que establecer una estrategia clara para el manejo de la pila.

En resumen, a la hora de diseñar un AP tenemos que repartir lo que requiere ser "recordado" entre los estados y la pila. Distintos diseños para un mismo problema pueden tomar decisiones diferentes en cuanto a que recuerda cada cual.

Ejemplo:.- Diseñar un AP que acepte exactamente el lenguaje con palabras de la forma $a^n b^n$, para cualquier número natural n.

Una idea que surge inmediatamente es la de utilizar la pila como "contador" para recordar la cantidad de a's que se consumen, y luego confrontar con la cantidad de b's. Una primera versión de este diseño utiliza un sólo estado q, con transiciones a/ε/a y b/a/ε de q a sí mismo, como en la figura 24(a).

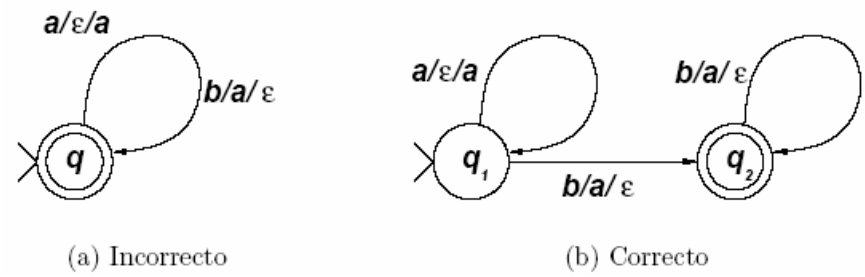


Figura 24:. Automata de pila para el lenguaje $a^n b^n$

Para verificar el funcionamiento del autómata, podemos simular su ejecución, listando las situaciones sucesivas en que se encuentra, mediante una tabla que llamaremos "traza de ejecución". Las columnas de una traza de ejecución para un AP son: el estado en que se encuentra el autómata, lo que falta por leer de la palabra de entrada, y el contenido de la pila.

Por ejemplo, la traza de ejecución del AP del ejemplo, para la palabra aabb, se muestra a continuación:

Estado	Por leer	Pila
q	aabb	ε
q	abb	a
q	bb	aa
q	b	a
q	ε	ε

Concluímos que el AP efectivamente puede aceptar palabras como $a^n b^n$. Sin embargo, hay un problema: el AP también acepta palabras como abab, que no tienen la forma deseada (es fácil construir la traza de ejecución correspondiente para convencerse de ello). El problema viene porque no hemos recordado cuando se terminan las a y comienzan las b, por eso ha sido posible mezclarlas en abab.

Una solución es utilizar los estados para memorizar las situaciones de estar consumiendo a o estar consumiendo b. El diagrama de estados correspondiente se muestra en la figura 24(b).

Ejemplo:.- Proponer un AP que acepte el lenguaje de los palíndromos con un número par de símbolos, esto es, palabras que se leen igual de izquierda a derecha y de derecha a izquierda, y que tienen por tanto la forma ww^R, donde w^R es el reverso de w (esto es, invertir el orden), en el alfabeto {a, b}. Por ejemplo, las palabras abba, aa y bbbbbb pertenecen a este lenguaje, mientras que aab y aabaa no.

Una estrategia de solución para diseñar este AP sería almacenar en la pila la primera mitad de la palabra, y luego irla comparando letra por letra contra la segunda mitad. Tendríamos dos estados s y f, para recordar que estamos en la primera o segunda mitad de la palabra. En la figura 25 se detalla este AP.

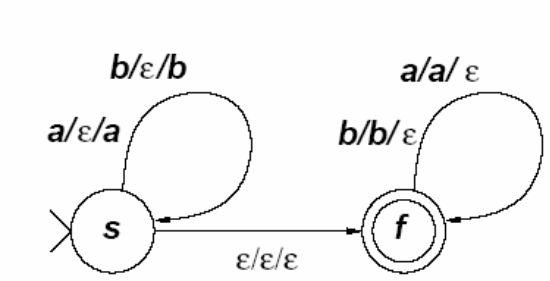


Figura 25: Autómata de pila para el lenguaje { ww^R }

Se puede apreciar en el AP de dicha figura la presencia de una transición de *s a f*, en que ni se consumen caracteres de la entrada, ni se manipula la pila. Esta transición parece muy peligrosa, porque se puede "disparar" en cualquier

momento, y si no lo hace exactamente cuando hemos recorrido ya la mitad de la palabra, el AP podrá llegar al final a un estado que no sea final, rechazando en consecuencia la palabra de entrada. Entonces, ¿cómo saber que estamos exactamente a la mitad de la palabra?

Conviene en este punto recordar que en un autómata no determinista una palabra es aceptada cuando existe un cálculo que permite aceptarla, independientemente de que un cálculo en particular se vaya por un camino erróneo. Lo importante es, pues, que exista un cálculo que acepte la palabra en cuestión. Por ejemplo, la siguiente tabla muestra un cálculo que permite aceptar la palabra w = abba:

Estado	Falta leer	Pila	Transición
s	abba	ε	
s	bba	A	1
s	ba	ba	2
f	ba	ba	3
f	a	A	5
f	ε	ε	4

LECCION 23. - COMBINACIÓN MODULAR DE AUTÓMATAS DE PILA

En los AP también es posible aplicar métodos de combinación modular de autómatas, como se hizo con los autómatas finitos. En particular, es posible obtener AP que acepten la unión y concatenación de los lenguajes aceptados por dos AP dados.

En el caso de la unión, dados dos AP M_1 y M_2 que aceptan respectivamente los lenguajes L_1 y L_2, podemos obtener un AP que acepte la unión $L_1 \cup L_2$, itroduciendo un nuevo estado inicial s_0 con transiciones ε/ε/ε a los dos antiguos estados iniciales s_1 y s_2, como se ilustra en la figura 26.

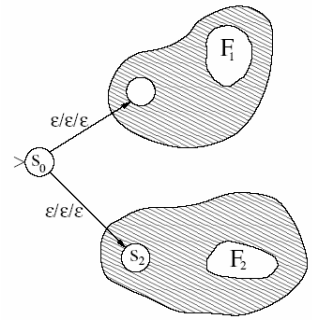


Figura 26: Unión de Autómatas de Pila

Ejemplo:- Obtener un AP que acepte el lenguaje $\{a^n b^m \mid n \neq m\}$. Claramente este lenguaje es la unión de $a^n b^m \mid n > m\}$ con $\{ a^n b^m \mid n < m\}$, por lo que basta obtener

los AP de cada uno de ellos, y combinarlos con el método descrito.

Ejemplo.- Diseñar un AP que acepte el lenguaje $L = \{a^i b^j c^k | \neg(i = j = k)\}$. Nos damos cuenta de que L es la unión de dos lenguajes, que son:

$$L = \{a^i b^j c^k \mid i \neq j\} \; [\; \{a_i b_j c_k \mid j \neq k\}$$

Para cada uno de estos dos lenguajes es fácil obtener su AP. Para el primero de ellos, el AP almacenaría primero las a's en la pila, para luego ir descontando una b por cada a de la pila; las a's deben acabarse antes de terminar con las b's o bien deben sobrar a's al terminar con las b's; las c's no modifican la pila y simplemente se verifica que no haya a o b después de la primera c.

También es posible obtener modularmente un AP que acepte la concatenación de los lenguajes aceptados por dos AP dados.

Sin embargo, la construcción de un AP que acepte la concatenación de dos lenguajes a partir de sus respectivos AP M_1 y M_2, es ligeramente más complicada que para el caso de la unión. La idea básica sería poner transiciones vacías que vayan de los estados finales de M_1 al estado inicial de M_2. Sin embargo, existe el problema que hay que garantizar que la pila se encuentre vacía al pasar de M_1 a M_2, pues de otro modo podría resultar un AP incorrecto.

Para esto, es posible utilizar un caracter especial, por ejemplo "@", que se mete a la pila antes de iniciar la operación de M_1, el cual se saca de la pila antes de iniciar la operación M_2.

LECCCION 24. - AUTÓMATAS CON PILA Y LENGUAJES LIBRES DE CONTEXTO.

Teorema Si un lenguaje es generado por una gramática libre del contexto, entonces es aceptado por un Autómata con Pila No-Determinístico.

Demostración.-Supongamos que la gramática no acepta la palabra vacía. En caso de que acepte la palabra vacía se le eliminaría y despues se podría transformar el autómata para añadir la palabra vacía al lenguaje aceptado por el autómata.

Transformemos entonces la gramática a forma normal de Greibach. El autómata con pila correspondiente es $M = (\{q\}, T, V, \delta, q, S, 0)$ donde la función de transición viene dada por

$$(q, \gamma) \in \delta(q, a, A) \Leftrightarrow A \rightarrow a\gamma \in P$$

Este autómata acepta por pila vacía el mismo lenguaje que genera la gramática.

Ejemplo: Para la gramática en forma normal de Greibach:

$$S \rightarrow aAA$$
$$A \rightarrow aS \mid bS \mid a$$

el autómata es $M = (\{q\}, \{a, b\}, \{A, S\}, \delta, q, S, 0)$

donde $\delta(q, a, S) = \{(q, AA)\}$

$\delta(q,a,A) = \{(q,S),(q,\varepsilon)\}$

$\delta(q,b,A) = \{(q,S)\}$

Teorema Si $L = N(M)$donde M es un APND, existe una gramática libre del contexto G, tal que $L(G) = L$.

Demostración.-

Sea $M = (Q,A,B, \delta, q_0, Z_0, 0)$, tal que $L = N(M)$. La gramática $G = (V,A,P,S)$se construye de la siguiente forma:

–V será el conjunto de los objetos de la forma [q,C, p], donde $p,q \in Q$ y $C \in B$, además de la variable S que será la variable inicial.

–P será el conjunto de las producciones de la forma

1. $S \rightarrow [q_0,Z,q]$para cada $q \in Q$.
2. $[q,C,q_m] \rightarrow a[p,D_1,q_1][q_1,D_2,q_2]...[q_{m-1},D_{m,qm}]$

donde $a \in A \cup \varepsilon$, y $C,D_1,...,D_m \in B$ tales que

$$(p,D_1D_2...D_m) \in \delta(q,a,C)$$

(si m =0, entonces la producción es $[q,A, p] \rightarrow a$).

Esta gramática genera precisamente el lenguaje *N*(*M*). La idea de la demostración es que la generación de una palabra en esta gramática simula el funcionamiento del autómata no determinístico. En particular, se verifica que [*q*,*C*, *p*] genera la palabra *x* si y solo si el autómata partiendo del estado *q* y llegando al estado *p*, puede leer la palabra *x* eliminando el símbolo *C* de la pila.

Ejemplo Si partimos del autómata $M = (\{q_0,q_1\},\{0,1\},\{X ,Z\},\delta,q_0,Z_0,0)$, donde

$$\delta(q_0,0,Z_0) = \{(q_0,X\ Z_0)\},\ \delta(q_1,1,X) = \{(q_1,\varepsilon)\}$$
$$\delta(q_0,0,X) = \{(q_0,X\ X)\},\ \delta(q_1,\varepsilon,X) = \{(q_1,\varepsilon)\}$$
$$\delta(q_0,1,X) = \{(q_1,\varepsilon)\},\ \delta(q_1,\varepsilon,Z_0) = \{(q_1,\varepsilon)\}$$

las producciones de la gramática asociada son*:*

$$S \rightarrow [q_0, Z_0, q_0]$$

$$S \rightarrow [q0, Z_0, q_1]$$

$[q_0, Z_0, q_0] \rightarrow 0[q_0, X , q_0][q_0, Z_0, q_0]$
$[q_0, Z_0, q_1] \rightarrow 0[q_0, X , q_0][q_0, Z_0, q_1]$
$[q_0, Z_0, q_0] \rightarrow 0[q_0, X , q_1][q_1, Z_0, q_0]$
$[q_0, Z_0, q_1] \rightarrow 0[q_0, X , q_1][q_1, Z_0, q_1]$

$[q_0, X , q_0] \rightarrow 0[q_0, X , q_0][q_0, X , q_0]$
$[q_0, X , q_1] \rightarrow 0[q_0, X , q_0][q_0, X , q_1]$
$[q_0, X , q_0] \rightarrow 0[q_0, X , q_1][q_1, X , q_0]$
$[q_0, X , q_1] \rightarrow 0[q_0, X , q1][q_1, X , q_1]$

$[q_0, X , q_1] \rightarrow 1$
$[q_1, X , q_1] \rightarrow 1$
$[q_1, X , q1] \rightarrow \varepsilon$
$[q_1, Z0, q_1] \rightarrow \varepsilon$

Elimando símbolos y producciones inútiles queda

$S \rightarrow [q_0, Z_0, q_1]$
$[q_0, Z_0, q_1] \rightarrow 0[q_0, X , q_1][q_1, Z_0, q_1]$
$[q_0, X , q_1] \rightarrow 0[q_0, X , q_1][q_1, X , q_1]$
$[q_1, X , q_1] \rightarrow 1$
$[q_1, X , q_1] \rightarrow \varepsilon$
$[q_1, Z0, q_1] \rightarrow \varepsilon$

LECCION 25. - RELACIÓN ENTRE LOS AUTÓMATAS DE PILA Y LENGUAJES LIBRES DE CONTEXTO

Ahora vamos a establecer el resultado por el que iniciamos el estudio de los AP, es decir, verificar si son efectivamente capaces de aceptar los LLC.

Teorema.- Los autómatas de pila aceptan exactamente los LLC.

Vamos a examinar la prueba de esta afirmación, no solamente por el interés por la rigurosidad matemática, sino sobre todo porque provee un método de utilidad práctica para transformar una GLC en un AP. La prueba de este teorema se puede dividir en dos partes:

1. Si M es un AP, entonces L(M) es un LLC
2. Si L es un LLC, entonces hay un AP M tal que L(M) = L

Vamos a presentar únicamente la prueba con la parte 2, que consideramos de mayor relevancia práctica.

Sea una gramática G = (V,$\sum$,R, S). Entonces un Automata de Pila M que acepta exactamente el lenguaje generado por G se define como sigue:

$$M = (\{p, q\}, \sum, V \cup \sum, \Delta, p, \{q\})$$

donde Δ contiene las siguientes transiciones:

1. Una transición ((p, ε, ε), (q, S))
2. Una transición ((q, ε,A), (q, x)) para cada $A \rightarrow x \in R$
3. Una transición ((q, σ, σ), (q, ε)) para cada σ 2 $\sum$

Ejemplo.- Obtener un AP que acepte el LLC generado por la gramática con reglas:

1. $S \rightarrow aSa$
2. $S \rightarrow bSb$
3. $S \rightarrow c$

Las transiciones del AP correspondiente están dadas en la tabla siguiente:

1	(p, ε,ε)	(q,S)
2	(q, ε,S)	(q,aSa)
3	(q, ε,S)	(q,bSb)
4	(q, ε,S)	(q,c)
5	(q,a, a)	(q, ε)
6	(q,b, b)	(q, ε)
7	(q,c, c)	(q, ε)

El funcionamiento de este AP ante la palabra abcba aparece en la siguiente tabla:

Estado	Falta leer	Pila
P	abcba	ε
Q	abcba	S
Q	abcba	aSa
q	bcba	Sa
q	bcba	bSba
q	cba	Sba
q	cba	cba
q	ba	ba
q	a	a
q	ε	ε

Se justifica intuitivamente el método que se introdujo para obtener un AP equivalente a una gramática dada. Si se observa las transiciones del AP, veremos que solamente tiene dos estados, p y q, y que el primero de ellos desaparece del cálculo en el primer paso; de esto concluímos que el AP no utiliza los estados para "recordar" características de la entrada, y por lo tanto reposa exclusivamente en el almacenamiento de caracteres en la pila. En efecto, podemos ver que las transiciones del tipo 2, lo que hacen es reemplazar en la pila una variable por la

cadena que aparece en el lado derecho de la regla correspondiente. Dado que la (única) transición de tipo 1 (transición 1 del ejemplo) coloca el símbolo inicial en la pila, a continuación lo que hacen las reglas de tipo 2 es realmente efectuar toda la derivación dentro de la pila de la palabra de entrada, reemplazando un lado izquierdo de una regla por su lado derecho. Una vez hecha la derivación de la palabra de entrada, –la cual estaría dentro de la pila, sin haber aún gastado un solo caracter de la entrada– podemos compararla caracter por caracter con la entrada, por medio de las transiciones de tipo 3.

Existe sin embargo un problema técnico: si observamos la “corrida” para la palabra abcba, nos daremos cuenta de que no estamos aplicando las reglas en el orden descrito en el párrafo anterior, esto es, primero la transición del grupo 1, luego las del grupo 2 y finalmente las del grupo 3, sino que más bien en la cuarta línea de la tabla se consume un caracter a (aplicación de una transición del grupo 3) seguida de la aplicación de una transición del grupo 2. Esto no es casualidad; lo que ocurre es que las variables no pueden ser reemplazadas por el lado derecho de una regla si dichas variables no se encuentran en el tope de la pila. En efecto, recuérdese que los AP solo pueden accesar el caracter que se encuentra en el tope de la pila.

Por esto, se hace necesario, antes de reemplazar una variable por la cadena del lado derecho de una regla, “desenterrar” dicha variable hasta que aparezca en el tope de la pila, lo cual puede hacerse consumiendo caracteres de la pila (y de la entrada, desde luego) mediante la aplicación de transiciones del tipo 3.

Cap 6

PROPIEDADES DEL LENGUAJES INDEPENDIENTES DE CONTEXTO

LECCION 26. - LEMA DE BOMBEO

Comenzamos esta sección con un lema que nos da una condición necesaria que deben de cumplir todos los lenguajes libres de contexto. Nos sirve para demostrar que un lenguaje dado no es libre de contexto, comprobando que no cumple esta condición necesaria.

Lema 2 (Lema de Bombeo para lenguajes libres de contexto) Sea L un lenguaje libre de contexto. Entonces, existe una constante n, que depende solo de L, tal que si $z \in L$ y $|z| \geq n$, z se puede escribir de la forma z = uvwxy de manera que

1. $|vx| \geq 1$
2. $|vwx| \leq n$, y
3. $\forall\, i \geq 0$, $uv^i wx^i y \in L$

Demostración.- Supongamos que la gramática no tiene producciones nulas ni unitarias (si existiesen siempre se podrían eliminar). Supongamos un árbol de derivación de una palabra *u* generada por la gramática. Es fácil ver que si la longitud de *u* es suficientemente grande, en su árbol de derivación debe existir un camino de longitud mayor que el número de variables. Sea *N* un número que garantice que se verifica esta propiedad. En dicho camino, al menos debe de haber una variable repetida. Supongamos que esta variable es *A*, y que la figura 6.1 representa el árbol de derivación y dos apariciones consecutivas de *A*.

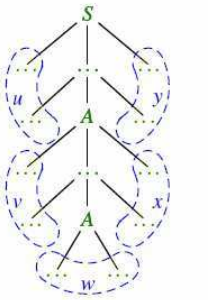


Figura 27: Arbol de Derivación en el lema del Bombeo

Ejemplo Vamos a utilizar el lema de bombeo para probar que el lenguaje $L = \{a^i b^i c | i \geq 1\}$ no es libre de contexto.

Supongamos que L fuese libre de contexto y sea n la constante especificada en el Lema de Bombeo. Consideremos la palabra $z = a^n b^n c^n \in L$, que tiene una longitud mayor que n. Consideremos que z se puede descomponer de la forma z = uvxy, verificando las condiciones del lema de bombeo.

Como |vwx|≤ n, no es posible para vx tener símbolos a y c al mismo tiempo: entre la última a y la primera c hay n símbolos. En estas condiciones se pueden dar los siguientes casos:

–|vx| contiene solamente símbolos a. En este caso para i = 0, uv^0wx^0y = uwy debería pertenecer a L por el lema de bombeo. Pero uwy contiene n símbolos b, n símbolos c, menos de n símbolos a, con lo que no podría pertenecer a L y se obtiene una contradicción.

–|vx| contiene solamente símbolos b. Se llega a una contradicción por un procedimiento similar al anterior.

–|vx| contiene solamente símbolos c. Se llega a una contradicción por un procedimiento similar.

–|vx| contiene símbolos a y b. En este caso, uwy tendría más símbolos c que a o b, con lo que se llegaría de nuevo a una contradicción.

–|vx| contiene símbolos b y c. En este caso, uwy tendría más símbolos a que b o c, con lo que se llegaría también a una contradicción.

En todo caso se llega a una contradicción y el lema de bombeo no puede cumplirse, con lo que L no puede ser libre de contexto.

Es importante señalar que el lema de bombeo no es una condición suficiente. Es solo necesaria. Así si un lenguaje verifica la condición del lema de bombeo no podemos garantizar que sea libre de contexto.

Un ejemplo de uno de estos lenguajes es

$$L = \{a^i b^j c^k d^l \mid (i = 0) \vee (j = k = l)\}$$

Ejemplo Demostrar que el lenguaje $L = \{a^i b^j c^i d^j : i, j \geq 0\}$ no es libre de contexto.

Ejemplo Demostrar que el lenguaje $L = \{a^i bc^k : i \geq j \geq k \geq 0\}$ no es libre de contexto.

LECCION 27. - PROPIEDADES DE CLAUSURA DE LOS LENGUAJES LIBRES DE CONTEXTO

Teorema Los lenguajes libres de contexto son cerrados para las operaciones:

– Unión
– Concatenación
– Clausura

Demostración
Sean $G_1 =(V_1, T_1, P_1, S_1)$ y $G_2 =(V_2, T_2, P_2, S_2)$ dos gramáticas libres de contexto y L_1 y L_2 los lenguajes que generan. Supongamos que los conjuntos de variables son disjuntos. Demostraremos que los lenguajes $L_1 \cup L_2$, L_1L_2 y L_1^* de tipo 2 que los generen. son libres de contexto, encontrando gramáticas [1]

$L_1 \cup L_2$. Una gramática que genera este lenguaje es $G_3 =(V_1 \cup V_2 \cup \{S_3\}, T_1 \cup T_2, P_3, S_3)$, donde S_3 es una nueva variable, y $P_3 =P_1 \cup P_2$ más las producciones $S_3 \rightarrow S_1$ y $S_3 \rightarrow S_2$.

L_1L_2. Una gramática que genera este lenguaje es $G_4 =(V_1 \cup V_2 \cup \{S_4\}, T_1 \cup T_2, P_4, S_4)$, donde S_4 es una nueva variable, y $P_4 =P_1 \cup P_2$ más la producción $S_4 \rightarrow S_1S_2$.

L_1^* Una gramática que genera este lenguaje es $G_5 =(V_1 \cup \{S_5\}, T_1 \cup P5, S_5)$ donde P_5 es P_1 más las producciones $S_5 \rightarrow S_1S_5$ y $S_5 \rightarrow \varepsilon$.

Algunas propiedades de clausura de los lenguajes regulares no se verifican en la clase de los lenguajes libres de contexto, como las que expresan el siguiente teorema y corolario.

Teorema La clase de los lenguajes libres de contexto no es cerrada para la intersección.
Demostración.-Sabemos que el lenguaje $L =\{ a^ib^ic^j \mid i \geq 1\}$ no es libre de contexto.
Por otra parte los lenguajes $L_2 =\{a^ib^ic^j \mid i \geq 1 \text{ y } j \geq 1\}$ y $L_3 =\{ a^ib^jc^j \mid i \geq 1 \text{ y } j \geq 1\}$ si lo son.
El primero de ellos es generado por la gramática:
$S \rightarrow AB$
$A \rightarrow aAb|ab$
$B \rightarrow cB|c$

y el segundo, por la gramática:

$S \rightarrow CD$

$C \rightarrow aC|a$
$D \rightarrow bDc|bc$

Como $L_2 \cap L_3 = L_1$, se deduce que la clase de lenguajes libres de contexto no es cerrada para la intersección

Corolario La clase de lenguajes libres de contexto no es cerrada para el complementario.

Demostración.-

Es inmediato, ya que como la clase es cerrada para la unión, si lo fuese para el complementario, se podría demostrar, usando las leyes De Morgan que lo es también para la intersección.

LECCION 28. - ALGORITMOS DE DECISIÓN PARA LOS LENGUAJES LIBRES DE CONTEXTO

Existen una serie de problemas interesantes que se pueden resolver en la clase de los lenguajes libres de contexto. Por ejemplo, existen algoritmos que nos dicen si un Lenguaje Libre de Contexto (dado por una gramática de tipo 2 o un autómata con pila no determinístico) es vacío, finito o infinito. Sin embargo, en la clase de lenguajes libres de contexto comienzan a aparecer algunas propiedades indecidibles. A continuación, veremos algoritmos para las propiedades decidibles y mencionaremos algunas propiedades indecidibles importantes.

Teorema Existen algoritmos para determinar si un lenguaje libre de contexto es
a) vacío
b) finito
c) infinito

Demostración.

a) En la primera parte del algoritmo para eliminar símbolos y producciones inútiles de una gramática, se determinaban las variables que podían generar una cadena formada exclusivamente por símbolos terminales. El lenguaje generado es vacío si y solo si la variable inicial S es eliminada: no puede generar una palabra de símbolos terminales.

b) y c) Para determinar si el lenguaje generado por una gramática de tipo 2 es finito o infinito pasamos la gramática a forma normal de Chomsky, sin símbolos ni producciones inútiles. En estas condiciones todas las producciones son de la forma:

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

Se construye entonces un grafo dirigido en el que los vértices son las variables y en el que para cada producción de la forma $A \to BC$ se consideran dos arcos: uno de A a B y otro de A a C. Se puede comprobar que el lenguaje generado es finito si y solo si el grafo construido de esta forma no tiene ciclos dirigidos.

Ejemplo Consideremos la gramática con producciones,
S $\to$ AB
A $\to$ BC|a
B $\to$ CC|b
C $\to$ a
El grafo asociado es el de la figura 28(a). No tiene ciclos y el lenguaje es finito. Si añadimos la producción C $\to$ AB, el grafo tiene ciclos (figura 28(b)) y el lenguaje generado es infinito.

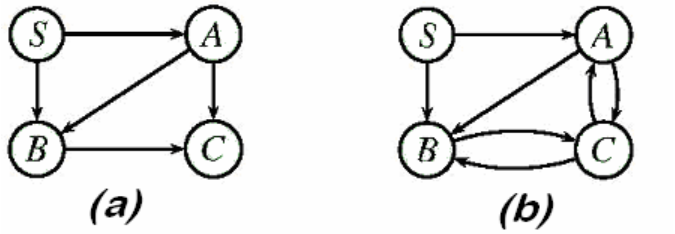


Figura 28: Grafo asociado a una gramatica de tipo 2 con lenguaje finito e Infinito

LECCION 29. - ALGORITMOS DE PERTENENCIA

Estos algoritmos tratan de resolver el siguiente problema: dada una gramática de tipo 2, $G = (V, T, P, S)$y una palabra $u \in T^*$, determinar si la palabra puede ser generada por la gramática. Esta propiedad es indecidible en la clase de lenguajes recursivamente enumerables, pero es posible encontrar algoritmos para la clase de lenguajes libres de contexto.

Un algoritmo simple, pero ineficiente se aplica a gramáticas en forma normal de Greibach (si una gramática no está en esta forma se pasa, teniendo en cuenta que hemos podido dejar de aceptar la palabra vacía). La pertencencia de una palabra no vacía se puede comprobar en esta forma normal de Greibach de la siguiente forma: Como cada producción añade un símbolo terminal a la palabra generada, sabemos que una palabra, u, de longitud $|u|$ ha de generarse en $|u|$ pasos. El algoritmo consistiría en enumerar todas las generaciones por la izquierda de longitud $|u|$, tales que los símbolos que se vayan generando coincidan con los de la palabra u, y comprobar si alguna de ellas llega a generar la palabra u en su totalidad. Este algoritmo para, ya que el número de derivaciones por la izquierda de una longitud dada es finito. Sin embargo puede ser muy ineficiente

(exponencial en la longitud de la palabra). Para comprobar la pertenencia de la palabra vacía se puede seguir el siguiente procedimiento:

–Si no hay producciones nulas, la palabra vacía no pertenece.
–Si hay producciones nulas, la palabra vacía pertenece si y solo si al aplicar el algoritmo que elimina las producciones nulas, en algún momento hay qhe eliminar la producción $S \rightarrow \varepsilon$.

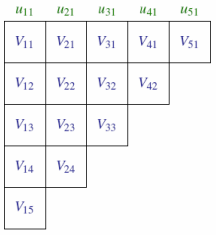
6.29.1. - El Algoritmo De Cocke-Younger-Kasami

Existen algoritmos de pertenencia con una complejidad $O(n^3)$, donde n es la longitud de la palabra de la que se quiere comprobar la pertenencia. Nosotros vamos a ver el algoritmo de Cocke-Younger-Kasami (CYK). Este algoritmo se aplica a palabras en forma normal de Chomsky. Este consta de los siguientes pasos (*n* es la longitud de la palabra).

1. Para i = 1 hasta n
2. Calcular $V_{i1} = \{A \mid A \rightarrow a$ es una produccion y el simbolo i-esimo de u es $a\}$
3. Para j = 2 hasta n
4. Para i = 1 hasta $n - j + 1$
5. $V_{ij} = 0$
6. Para k = 1 hasta $j - 1$

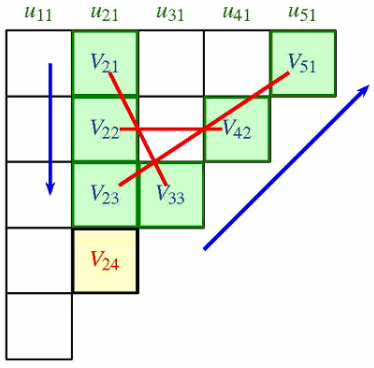
$V_{ij} = V_{ij} \cup \{A \mid A \rightarrow BC$ es una produccion, $B \in V_{ik}$ y $C \in V_{i+k,\ j-k}\}$
Este algoritmo calcula para todo i, $j (i \in \{1,...,n\}, j \leq n - j + 1)$, el conjunto de variables V_{ij} que generan u_{ij}, donde u_{ij} es la subcadena de u que comienza en el símbolo que ocupa la posición i y que contiene j símbolos. La palabra u será generada por la gramática si la variable inicial S pertenece al conjunto V_{1n}.

Los cálculos se pueden organizar en una tabla como la de la siguiente figura (para una palabra de longitud 5):



En ella, cada $V_{i,\ j}$ se coloca en una casilla de la tabla. En la parte superior se ponen los símbolos de la palabra para la que queremos comprobar la pertenencia. La ventaja es que es fácil localizar los emparejamientos de variables que hay que comprobar para calcular cada conjunto V_{ij}. Se comienza en la casilla que ocupa la misma columna y está en la parte superior de la tabla, y la casilla que está en la esquina superior derecha, emparejando todas las variables de estas dos casillas.

A continuación elegimos como primera casilla la que está justo debajo de la primera anterior, y como segunda casilla la que ocupa la esquina superior derecha de la segunda anterior. Este proceso se continúa hasta que se eligen como primera casilla todas las que están encima de la que se está calculando. La siguiente figura ilustra el proceso que se sigue en las emparejamientos (para un elemento de la cuarta fila, en una palabra de longitud 5).



Ejemplo Consideremos la gramática libre de contexto dada por las producciones
$S \rightarrow AB|BC$
$A \rightarrow BA|a$
$B \rightarrow CC|b$
$C \rightarrow AB|a$

Comprobar la pertenencia de las palabras baaba, aaaaa, aaaaaa al lenguaje generado por la gramática.

El algoritmo de Early es también de complejidad $O(n^3)$ en el caso general, pero es lineal para gramáticas $LR(1)$, que son las que habitualmente se emplean para especificar la sintaxis de las lenguajes de programación.

Al contrario que el algoritmo de Cocke-Younger-Kasami que trata de obtener las palabras de abajo hacia arriba (desde los símbolos terminales al símbolo inicial, el algoritmo de Early comenzará en el símbolo inicial (funciona de arriba hacia abajo).

Sea G una gramática con símbolo inicial S y que no tenga producciones nulas ni unitarias.

Supondremos que u[i.. j] es la subcadena de u que va de la posición i a la posición j. El algoritmo producirá registros de la forma (i, j, A, α, β), donde i y j son enteros y $A \rightarrow \alpha\beta$ es una producción de la gramática. La existencia de un registro indicará un hecho y un objetivo. El hecho es que u[i +1.. j]es derivable a partir de α y el objetivo es encontrar todos los k tales que β deriva a u[j +1..k]. Si encontramos

uno de estos k sabemos que A deriva u[i +1..k].

Para cada j, REGISTROS[j]contendrá todos los registros existentes de la forma (i, j, A, α, β).

El algoritmo consta de los siguiente pasos:

P1 *Inicialización.*-Sea

• REGISTROS[0]={(0, 0, S, ε, β): S $\rightarrow$ β es una producción}
• REGISTROS[j]= 0 para j =1, . . . , n.
• j =0

P2 Clausura.- Para cada registro (i, j, A, α, Bγ)en REGISTROS[j] y cada producción B $\rightarrow$ δ, crear el registro (j, j, B, ε, δ)e insertarlo en REGISTROS[j]. Repetir la operación recursivamente para los nuevos registros insertados.

P3 Avance.-Para cada registro (i, j, A, α, cγ) en REGISTROS[j], donde c es un símbolo terminal que aparece en la posición j +1 de u, crear (i, j +1, A, α, cγ) e insertarlo en REGISTROS[j + 1].
Hacer j = j +1.

P4 *Terminación.*-Para cada par de registros de la forma (*i*, *j*, *A*, α, ε) en R*EGISTROS*[*j*] y (*h*, *i*, *B*, γ*A*, δ) en *REGISTROS*[*i*], crear el nuevo registro (*h*, *j*, *B*, γ*A*, δ) e insertarlo en *REGISTROS*[*j*].

P5 Si $j < n$ ir a P2.

P6 Si en *REGISTROS*[*n*] hay un registro de la forma (0, *n*, *S*, α, ε), entonces *u* es generada. En caso contrario no es generada.

Ejemplo Comprobar mediante el algoritmo de Early si la palabra baa es generada por la gramática con producciones:

$$S \rightarrow AB, \quad S \rightarrow BC, \quad A \rightarrow BA, \quad A \rightarrow a,$$
$$B \rightarrow CC, \quad B \rightarrow b, \quad C \rightarrow AB, \quad C \rightarrow a$$

Después de aplicar el paso de inicialización, el contenido de REGISTROS[0] es:

REGISTROS[0]:(0,0,S,ε,AB),(0,0,S,ε,BC),(0,0,A,ε,BA),(0,0,A,ε,a),(0,0,B,ε,CC),(0,0,B,ε,b),(0,0,C,ε,AB),(0,0,C,ε,a)

REGISTROS[1] :(0, 1, B, b, ε), (0, 1, S, B, C), (0, 1, A, B, A), (1, 1, C, ε, AB), (1, 1,

C, ε, a), (1, 1, A, ε, BA), (1, 1, A, ε, a), (1, 1, B, ε, CC), (1, 1, B, ε, b)

REGISTROS[2] :(1, 2, C, a, ε), (1, 2, A, a, ε), (0, 2, S, BC, ε), (0, 2, A, BA, ε), (1, 2, C, A, B), (1, 2, B, C, C), (0, 2, S, A, B), (0, 2, C, A, B), (2, 2, B, ε, CC), (2, 2, B, ε, b), (2, 2, C, ε, AB), (2, 2, C, ε, a), (2, 2, A, ε, BA), (2, 2, A, ε, a)

REGISTROS[3] :(2, 3, C, a, ε), (2, 3, A, a, ε), (1, 3, B, CC, ε), (2, 3, B, C, C) (2, 3, C, A, B), (1, 3, A, B, A)

Como (0, 3, S, α, ε) no está en REGISTROS[3], la palabra baa no es generada

$$S \rightarrow T,\ S \rightarrow S + T,\ T \rightarrow F,\ T \rightarrow T * F,\ F \rightarrow a,\ F \rightarrow b,\ F \rightarrow (S)$$

Palabra: (a + b) * a

REGISTROS[0] :(0,0,S,ε,T), (0,0,S,ε,S +T), (0,0,T,ε,F), (0,0,T,ε,T*F), (0,0,F,ε,a), (0,0,F,ε,b),(
REGISTROS[1] :(0, 1, F, (, S)), (1, 1, S, ε, T), (1, 1, S, ε, S + T), (1, 1, T, ε, F), (1, 1, T, ε, T * F), (1, 1, F, ε, a), (1, 1, F, ε, b), (1, 1, F, ε, (S)),

REGISTROS[2] :(1, 2, F, a, ε), (1, 2, T, F, ε), (1, 2, S, T, ε), (0, 2, F, (S,)), (1, 2, S, S, +T)

REGISTROS[3] :(1, 3, S, S+, T), (3, 3, T, ε, F), (3, 3, T, ε, T *F), (3, 3, F, ε, a), (3, 3, F, ε, b), (3, 3, F, ε, (S))

REGISTROS[4] :(3, 4, F, b, ε), (3, 4, T, F, ε), (1, 4, S, S + T, ε), (3, 4, T, T, * F), (0, 4, F, (S,)), (1, 4, S, S, +T)

REGISTROS[5] :(0, 5, F, (S), ε), (0, 5, T, F, ε), (0, 5, S, T, ε), (0, 5, T, T, * F), (0, 5, S, S, +T),

REGISTROS[6] :(0, 6, T, T *, F), (6, 6, F, ε, a), (6, 6, F, ε, b), (6, 6, F, ε, (S)),

REGISTROS[7] :(6, 7, F, a, ε), (0, 7, T, T * F, ε), (0, 7, S, T, ε), (0, 7, T, T, *F), (0, 7, S, S, +T) Como tenemos (0, 7, S, T, ε), entonces la palabra (a + b) * c es generada.

LECCION 30. - PROBLEMAS INDECIDIBLES PARA LENGUAJES LIBRES DE CONTEXTO

Para terminar el apartado de algoritmos de decisión para gramáticas libres de contexto daremos algunos problemas que son indecidibles, es decir, no hay ningún algoritmo que los resuelva. En ellos se supone que G, $G1$ y $G2$ son gramáticas libres de contexto dadas y R es un lenguaje regular.

– Saber si $L(G_1) \cap L(G_2) = 0$.
– Determinar si $L(G) = T^*$, donde T es el conjunto de símbolos terminales.
– Comprobar si $L(G1) = L(G2)$.
– Determinar si $L(G1)$ □ $L(G2)$.
– Determinar si $L(G1) = R$.
– Comprobar si $L(G)$ es regular.
– Determinar si G es ambigua.
– Conocer si $L(G)$ es inherentemente ambiguo.
– Comprobar si $L(G)$ puede ser aceptado por una autómata determinístico con pila.

ACTIVIDADES:

Ejercicios Propuestos:

EJERCICIOS DE AUTOMATAS CON PILA

CONCEPTOS BÁSICOS

1.- Dado el siguiente autómata con pila, indicar:
(a) Qué lenguaje reconoce
(b) Cuáles de las siguientes palabras son aceptadas por el autómata:
aabbc, abbc, bbcc, aabbbcc (mostrando la sucesión de descripciones instantaneas)

AP = ({a, b, c}, {A, B, S}, {q, r, s, t}, q, S, f, Ø)

f(q, a, S) = {(r, S)}	f(s, b, S) = {(s, BS)}
f(q, a, A) = {(r, A)}	f(s, b, A) = {(s, λ)}
f(q, b, S) = {(s, BS)}	f(s, b, B) = {(s, BB)}
f(q, b, A) = {(s, λ)}	f(s, c, B) = {(t, λ)}
f(r, a, S) = {(q, AS)}	f(t, c, B) = {(t, λ)}
f(r, a, A) = {(q, AA)}	f(t, λ, S) = {(t, λ)}

2.- Definir formalmente el lenguaje reconocido por el siguiente autómata con pila:

A = ({a,b,c}, {S,A,B}, {q,r,s,p}, q, S, f, ∅)

f(q, a, S) = {(r, S)}	f(s, b, S) = {(s, BS)}
f(q, b, S) = {(s, BS)}	f(s, b, A) = {(s, λ)}
f(q, a, A) = {(r, A)}	f(s, b, B) = {(s, BB)}
f(q, b, A) = {(s, λ)}	f(s, c, B) = {(p, λ)}
f(r, a, S) = {(q, AS)}	f(p, c, B) = {(p, λ)}
f(r, a, A) = {(q, AA)}	f(p, λ, S) = {(p, λ)}

3.- Dado el siguiente autómata con pila indicar:

(a) Qué lenguaje reconoce por vaciado de pila.
(b) Cuáles de las siguientes palabras son aceptadas por el AP: abba, abaaba.

AP = ({a,b}, {Z}, {q0, q1, q2, q3}, q0, Z, f, ∅)

f(q0, a, Z) = {(q1, aZ)}	f(q2, b, b) = {(q2, λ)}
f(q0, b, Z) = {(q1, bZ)}	f(q2, a, a) = {(q2, λ)}
f(q1, a, a) = {(q1, aa), (q2, λ)}	f(q2, λ, Z) = {(q2, λ)}
f(q1, a, b) = {(q1, ab)}	
f(q1, b, a) = {(q1, ba)}	
f(q1, b, b) = {(q1, bb), (q2, λ)}	

CONSTRUCCIÓN DE AUTÓMATAS

4.- Construir un autómata con pila que reconozca cada uno de los siguientes lenguajes:
(a) L = { $a^n b^{2n}$/ n > 0 }
(b) L = { $a^n b^m c^n$/ n, m > 0 }

Ejercicios de Autómatas con Pila 2

(c) L = { x / x ∈ {0,1}+ & nº 0´s = nº 1´s }
(d) L = { $a^i b^j c^{i+j}$ / i, j > 0 }
(e) L = { $a^{2i} b^{3i}$ / i >= 0 }
(f) L = { $a^n b^m c^{2m} a^{n+2}$, m > 0, n >= 0 }
(g) L = { xcy / x,y ∈ {a, b}+, nº de subcadenas “ab” en x = nº de subcadenas “ba” en y }
(h) L = { $0^n 1^n$ / n > 0 } ∪ { $0^n 1^{2n}$ / n > 0 }
(i) L = {$a^n b^m c^r a^s b^n$/ s = m + r, m,n > 0, r ≥ 0}
(j) L = {$a^{2n} b^m 0^i b^{2m} a^n$, i = 0 ó 1, m, n > 0}
(l) L = {$x^{1n} 2^m$x-1 / n, m = 0 ó 1 y x ∈ {a, b}+}
(m) L = {$a^n b^m c^p$ / n,m >= 0, p > n + m}

5.- Construir un autómata con pila que reconozca por vaciado de pila el lenguaje que contiene las palabras formadas por los símbolos “0”, “1” y “2” que tienen tantas apariciones de las secuencia “01” como del símbolo “2”.

6.- Construir un autómata con pila que reconozca por vaciado de pila las palabras formadas por los símbolos “a” y “b” que tengan doble número de símbolos “a” que de símbolos “b” (incluyendo λ). Es decir, el autómata deberá reconocer por ejemplo las palabras aabbaa, abbaaa, bababaaaa, bbbaaaaaa, y no debe reconocer palabras como abab, abbabab, a, b, aaa, bbb.

7.- Construir un autómata con pila que reconozca, por vaciado de pila, el lenguaje formado por las cadenas que contienen símbolos de abrir y cerrar paréntesis, es decir “(“ y “)”, que cumple las siguientes condiciones:
- todo paréntesis que se abre debe ser cerrado posteriormente
- todo paréntesis que se cierra debe haber sido abierto anteriormente
- se permiten los paréntesis anidados.

Ténganse en cuenta los siguientes ejemplos de cadenas reconocidas y no reconocidas
por el autómata:

Cadenas reconocidas	Cadenas no reconocidas
()(((()))())	()())
((()))()	)()(
((())(()))()()(()())	((((

8.- Dado el siguiente lenguaje: L = { anbcmdmen / n, m > 0 }

(a)Construir un autómata con pila que reconozca dicho lenguaje por vaciado de pila.
(b)Comprobar mediante el uso de descripciones instantáneas que la cadena “aabccddee” es aceptada por dicho autómata.

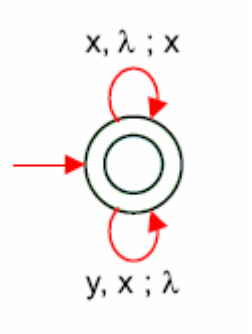
9.- Construir un autómata con pila que reconozca por vaciado de pila el lenguaje siguiente:

$$L = \{ 0^n1^n / n > 0 \} \cup \{ 0^n1^{2n} / n > 0 \}$$

10.- Dado el siguiente lenguaje:
$L = \{a^nb^md^m / n, m > 0\} \cup \{a^nb^mc^md^n / n, m > 0\}$
(a) Encontrar un autómata con pila que reconozca L por vaciado de pila.
(b) Comprobar mediante el uso de descripciones instantáneas que la cadena “aacd” es aceptada por dicho autómata.

Ejercicios Resueltos:

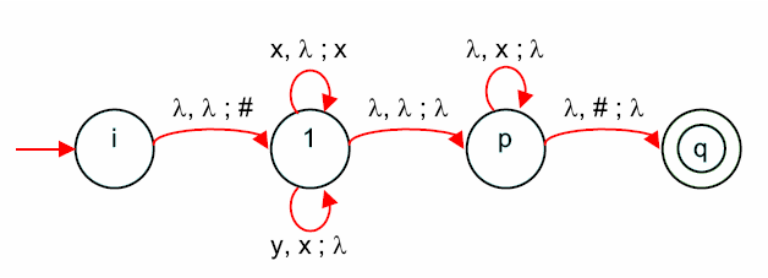
1. ¿Cuál es el lenguaje que acepta el autómata de pila cuyo diagrama de transiciones es el siguiente?



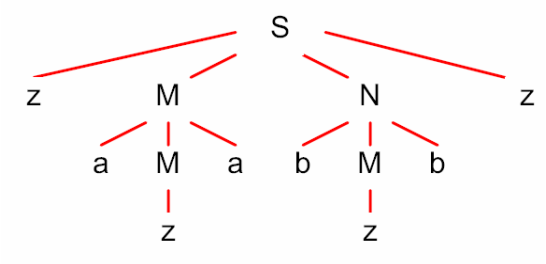
Solución:
Acepta la cadena vacía.
Según la rama superior, toda x que se reciba, se registrará en la pila; y el estado siguiente será de aceptación.
Según la rama inferior, sólo se aceptará la entrada y si se puede sacar una x de la pila.
Conclusión:
Acepta cualquier cadena que (empezando a contar por la izquierda) contenga un número de ‘y’ menor o igual a que de ‘x’. Pueden estar intercaladas
Ejemplos: xyx, xxy, xxyxyx, etc.

2.- Modifique el diagrama de transiciones del ejercicio anterior para que el autómata de pila acepte el mismo conjunto de cadenas, pero con su pila vacía.

Solución:



3.- Muestre que toda cadena derivada por la izquierda de una gramática independiente del contexto puede derivarse también por la derecha.
Solución:



El lado por donde se empieza a construir (por la derecha o por la izquierda) sólo tiene relevancia en las etapas intermedias de la construcción del árbol. Pero con la construcción terminada, no es relevante el lado por donde se empezó. La cadena es la misma en ambos casos.

INTRODUCCIÓN

En las Unidades precendentes se han estudiado lo que se puede considerar las máquinas abstractas que permiten solucionar ciertos tipos de algoritmos, los algoritmos en los que no puede recordarse más que una cantidad fija de información y otros en los que la información desarrollada durante la ejecución del algoritmo puede recuperararse solo en concordancia con la regla “lifo” ultimos en entrar primeros en salir, en esta unidad se describe una maquina abstracta, llamada Máquina de turing , que es aceptada de manera amplia como modelo general de computación, aunque las operaciones básicas de esta máquina son comparables en su sencillez a las de las máquinas estudiadas en las unidades anteriores, las nuevas maquinas pueden realizar una ámplia varidad de operaciones de computo. Ademas de aceptar lenguajes les es posible computar funciones y de conformidad con la tesis de Church-Turing, ejecutar casi cualquier procedemiento algoritmico concebible.

OBJETIVO GENERAL

Reconocer la importancia y el poder computacional de las maquinas de turing en el contexto de la solución de problemas computacionales de reconocimiento de lenguajes.

OBJETIVOS ESPECIFICOS

Estudiar las Máquinas de Turing y sus propiedades básicas

LECCION 31. - DEFINICIÓN[16]

Así como en secciones anteriores vimos cómo al añadir al autómata finito básico una pila de almacenamiento auxiliar, aumentando con ello su poder de cálculo, cabría ahora preguntarnos que es lo que habría que añadir a un autómata de pila para que pudiera analizar lenguajes como $\{a^n b^n c^n\}$. Partiendo del AP básico de la figura siguiente.

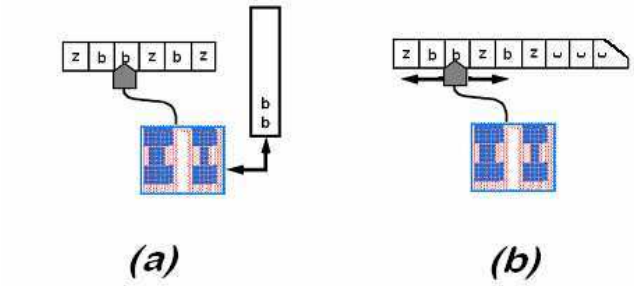


Figura (a)Automata de Pila (b) Maquina de Turing

Algunas ideas podrían ser:

1. Añadir otra pila;
2. Poner varias cabezas lectoras de la entrada;
3. Permitir la escritura en la cinta, además de la lectura de caracteres.

Vamos a enfocar nuestra atención a una propuesta en particular que ha tenido un gran impacto en el desarrollo teórico de la computación: la Máquina de Turing.

A. Turing propuso en los años 30 un modelo de máquina abstracta, como una extensión de los autómatas finitos, que resultó ser de una gran simplicidad y poderío a la vez. La máquina de Turing es particularmente importante porque es la más poderosa de todas las máquinas abstractas conocidas (A. Turing.- On computable numbers with an application to the Entscheidungs-problem, Proc. London Math. Soc., v.2, n.42, pp230-265.)

---

[16] VAQUERO SÁNCHEZ, Antonio Ramon, Calculabilidad y maquinas de turing (2004) En: http://www.fdi.ucm.es/profesor/vaquero/TALF-5-Turing.doc
BRENA PIÑERO, Ramon F. Autómatas y lenguajes un enfoque de diseño (2003) ITESM, En: http://lizt.mty.itesm.mx/~rbrena/AyL.html.

LECCION 32. - FUNCIONAMIENTO DE LA MÁQUINA DE TURING

La máquina de Turing (abreviado MT) tiene, como los autómatas que hemos visto antes, un control finito, una cabeza lectora y una cinta donde puede haber caracteres, y donde eventualmente viene la palabra de entrada. La cinta es de longitud infinita hacia la derecha, hacia donde se extiende indefinidamente, llenándose los espacios con el caracter blanco (que representaremos con “t”). La cinta no es infinita hacia la izquierda, por lo que hay un cuadro de la cinta que es el extremo izquierdo, como en la figura En la MT la cabeza lectora es de lectura y escritura, por lo que la cinta puede ser modificada en curso de ejecución. Además, en la MT la cabeza se mueve bidireccionalmente (izquierda y derecha), por lo que puede pasar repetidas veces sobre un mismo segmento de la cinta.

La operación de la MT consta de los siguientes pasos:

1. Lee un caracter en la cinta
2. Efectúua una transición de estado
3. Realiza una acción en la cinta

Las acciones que puede ejecutar en la cinta la MT pueden ser:

Escribe un símbolo en la cinta, o Mueve la cabeza a la izquierda o a la derecha Estas dos acciones son excluyentes, es decir, se hace una o la otra, pero no ambas a la vez.

La palabra de entrada en la MT está escrita inicialmente en la cinta, como es habitual en nuestros autómatas, pero iniciando a partir de la segunda posición de la cinta, siendo el primer cuadro un caracter blanco. Como la cinta es infinita, inicialmente toda la parte de la cinta a la derecha de la palabra de entrada está llena del caracter blanco (t).

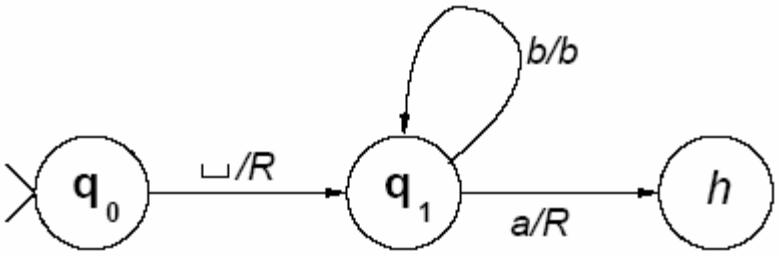


Figura.2: MT que acepta palabras que empiezan con a

Por definición, al iniciar la operación de la MT, la cabeza lectora está posicionada en el caracter blanco a la izquierda de la palabra de entrada, el cual es el cuadro más a la izquierda de la cinta.

Decimos que en la MT se llega al "final de un cálculo" cuando se alcanza un estado especial llamado halt en el control finito, como resultado de una transición. Representaremos al **halt** por "h". Al llegar al halt, se detiene la operación de la MT, y se acepta la palabra de entrada. Así, en la MT no hay estados finales. En cierto sentido el halt sería entonces el único estado final, sólo que además detiene la ejecución.

Cuando queremos que una palabra no sea aceptada, desde luego debemos evitar que la MT llegue al halt. Podemos asegurarnos de ello haciendo que la MT caiga en un ciclo infinito El lenguaje aceptado por una MT es simplemente el conjunto de palabras aceptadas por ella

Al diseñar una MT que acepte un cierto lenguaje, en realidad diseñamos el autómata finito que controla la cabeza y la cinta, el cual es un autómata con salida Así, podemos usar la notación gráfica utilizada para aquellos autómatas para indicar su funcionamiento. En particular, cuando trazamos una flecha que va de un estado p a un estado q con etiqueta σ/L, quiere decir que cuando la entrada al control finito (esto es, el caracter leído por la cabeza de la MT) es σ, la cabeza lectora hace un movimiento a la izquierda, indicada por el caracter L (left, en inglés); similarmente cuando se tiene una flecha con σ/R el movimiento es a la derecha. Cuando la flecha tiene la etiqueta σ/ξ, donde ξ es un caracter, entonces la acción al recibir el caracter σ consiste en escribir el caracter ξ en la cinta. Con estos recursos es suficiente para diseñar algunas MT, como en el siguiente ejemplo.

Ejemplo.- Diseñar (el control finito de) una MT que acepte las palabras en {a, b} que comiencen con a. La solución se muestra en la figura 29. Si la primera letra es una "a", la palabra se acepta, y en caso contrario se hace que la MT caiga en un ciclo infinito, leyendo y escribiendo "b". Nótese que la acción inmediatamente antes de caer en el "halt" es irrelevante; igual se podía haber puesto "a/a" o "a/R" como etiqueta de la flecha.

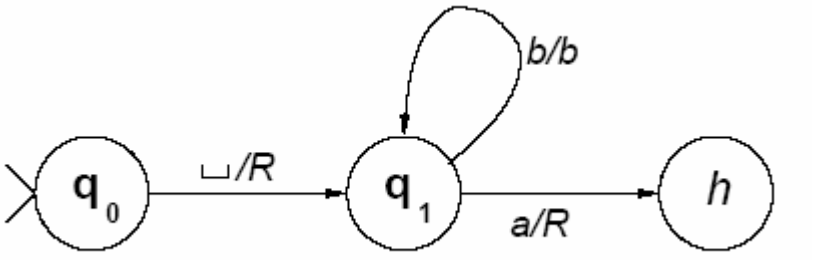


Figura.29: MT que acepta palabras que empiezan con a

Ejemplo.- Diseñar una MT que acepte las palabras en {a, b} que terminen con a. Aunque este ejemplo parece bastante similar al precedente, en realidad es más complicado, pues para ver cual es la ´ultima letra, hay que ir hasta el blanco a la derecha de la palabra, luego regresar a la ´ultima letra y verificar si es una “a”. Una solución se muestra en la figura 3.

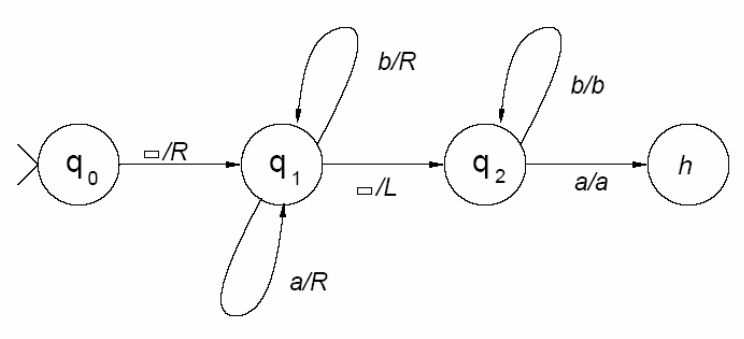
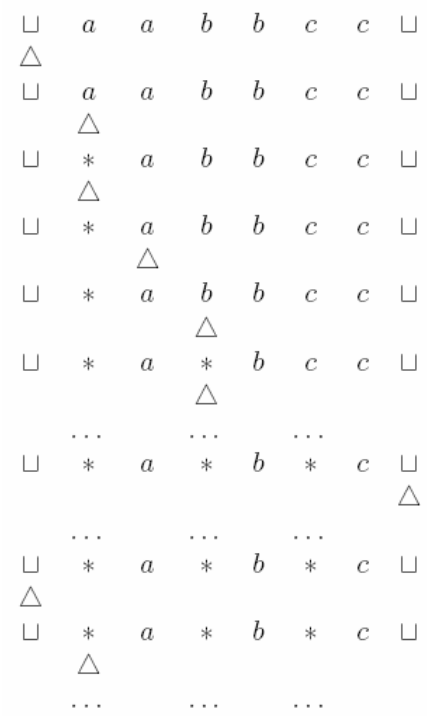


Figura 30: MT que acepta palabras que terminan con a

Ejemplo.- Probar que hay lenguajes que no son libres de contexto, pero que pueden ser aceptados por una m´aquina de Turing. Proponemos el lenguaje anbncn, que se sabe que no es LLC. Ahora construiremos una MT que lo acepte. La estrategia para el funcionamiento de dicha MT consistir´a en ir haciendo “pasadas” por la palabra, descontando en cada una de ellas una a, una b y una c; para descontar esos caracteres simplemente los reemplazaremos por un caracter “*”. Cuando ya no encontremos ninguna a, b o c en alguna pasada, si queda alguna de las otras dos letras la palabra no es aceptada; en caso contrario se llega a halt. Es útil, antes de emprender el diseño de una MT, tener una idea muy clara de cómo se quiere que funcione. Para eso se puede detallar el funcionamiento con algún ejemplo representativo, como en la tabla siguiente, para la palabra aabbcc. La posición de la cabeza se indica por el símbolo “Δ”.



Operatividad Formalmente: La máquina de Turing opera cíclicamente. Al comienzo de un ciclo se parte de una determinada configuración. El símbolo contenido en la celda explorada por la cabeza L/E más el estado de la MT determinan las acciones a realizar en un ciclo. Estas acciones son:

1) Sustituir un símbolo por otro (puede ser el mismo)
2) Desplazar la cabeza L/E: a la derecha (D), a la izquierda (I) o dejarla en la misma casilla (N).
3) Pasar la UC a otro estado (el mismo como caso particular).

Una vez realizadas esas tres acciones, el ciclo ha terminado y puede comenzar otro ciclo nuevo. En cada ciclo se realiza una instrucción. Así pues, una instrucción viene representada por una quíntupla: $a_i\ s_k\ a_j\ m\ s_l$;

$$a_i,\ a_j \in \Sigma\ ;\ s_k, s_l \in S\ \text{ y }\ m \in \{D, I, N\}$$

Los dos primeros símbolos de la quíntupla son el carácter al que apunta la cabeza L/E al comienzo del ciclo y el estado de la máquina en ese instante; los tres símbolos finales son el carácter que sustituye al que había en la casilla, el movimiento de la cabeza L/E y el estado de la máquina al final del ciclo.

Consiguientemente una MT queda completamente definida por

$\langle \Sigma , S , I , Q \rangle$

Σ: alfabeto, S: conjunto de estados, I: configuración inicial, Q: conjunto de quíntuplas (repertorio de instrucciones).

Una instrucción importante es la de Parada, equivalente a una quíntupla

$$a_i \; s_j \; a_i \; N \; s_j.$$

La máquina ha de ser no ambigua. Es decir, no debe haber, ante las mismas condiciones iniciales, reacciones diferentes de la máquina. Ello implica que no debe haber dos quíntuplas distintas con los dos primeros símbolos iguales. Sí puede haber, obviamente, dos quíntuplas diferentes con los tres últimos símbolos iguales.

LECCION 33. - DIFERENCIAS ENTRE LAS COMPUTADORAS Y LAS MÁQUINAS DE TURING

Aparte de la capacidad de memoria, las diferencias entre una MT y una computadora de estructura Von Neuman son:

En cuanto a estados: En una MT el nº de estados depende del algoritmo. En una computadora, un estado viene representado por el contenido de la memoria, y una situación por un estado y un puntero a una dirección (la que contiene a la instrucción que va a ejecutarse). La ejecución de la instrucción pasa a la máquina a otro estado y la variable de enlace adopta otro argumento (la dirección siguiente si la instrucción no era de bifurcación y la máquina es secuencial natural).

En cuanto a memoria: En una MT la memoria es la cinta de E/S más el "soporte físico" de los estados de la UC. Y ya sabemos cómo se pasa de una situación a otra: $a_i \; s_r \rightarrow a_j \; m \; s_l$. Programa en una MT es la secuencia de quíntuplas que hay que seguir para hacer el cálculo. El mismo algoritmo, expresado en lenguaje-máquina "tipo" para una computadora "tipo", o sea, programa en lenguaje máquina, es algo distinto que la secuencia de quíntuplas. Y se debe a la marcada diferencia entre n-upla e instrucción. Para formalizar el concepto de programa en lenguaje máquina no es, pues, buen modelo una MT.

En cuanto al orden de ejecución de las instrucciones: En la estructura Von Neumann el secuenciamiento lo marca el orden de colocación de las instrucciones en la memoria interna y viene asegurado por el contador de programa. En una MT el orden de ejecución lo marca en todo instante el estado de la máquina y el carácter de la cinta apuntado, que son los dos datos que determinan la quíntupla que ha de ser ejecutada.

Estrategia de trabajo:

Desplazar el puntero hacia la derecha hasta que se encuentre un blanco, haciendo los retoques necesarios para que queden tantos ‘1’s como requiere el resultado correcto.

En esta situación son suficientes dos estados. En el estado inicial la máquina puede permanecer desplazando la cabeza hacia la derecha hasta que encuentra un b. Esa es la idea central para la construcción de la máquina. Paso a paso se ha de ir definiendo cada instrucción necesaria para seguir la estrategia de trabajo. Y sólo se introducirá un nuevo estado cuando se necesite. En este caso se llega a:

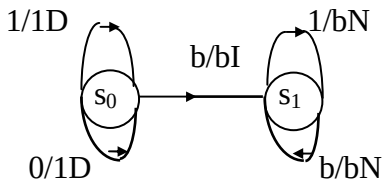


Diagrama de estados

M	s_o	s_1
0	1 D s_0	
1	1 D s_0	b N s_1
b	b I s_1	b N s_1 = Parada

Matriz de Turing
o Esquema funcional

Comprobación: Se debe tener seguridad de la corrección de la máquina construida. Para ello se debe probar el procesamiento “a mano” de ejemplos significativos. Así:

Secuencia de descripciones instantáneas:
s_0 ... b <u>1</u> 1 1 0 1 1 b b . . . configuración inicial
s_0 ... b 1 <u>1</u> 1 0 1 1 b b . . .
---
s_0 ... b 1 1 1 <u>0</u> 1 1 b . . .
s_0 ... b 1 1 1 1 <u>1</u> 1 b . . .
---
s_0 ... b 1 1 1 1 1 1 <u>b</u> . . .
s_1 b 1 1 1 1 1 <u>1</u> b ← pasa de s_0 a s_1
s_1 b 1 1 1 1 1 <u>b</u> b . . . Configuración final

También hay que comprobar para el 1er sumando = 0 y/o 2º sumando = 0. Haciéndolo, se ve que se cumple en este ejemplo. Podemos confiar en la máquina construida sólo si ha salido airosa de las comprobaciones.

Es conveniente aplicar sistemáticamente el método de construcción seguido, que consta de las siguientes etapas:

1.- Representación de los objetos: Datos y resultados.
2.- Alfabeto de la cinta y codificación de los objetos en el alfabeto. Y de ahí la configuración inicial.
3.-Definición del resultado para todos los ejemplares de entrada. Y de ahí la configuración final, excepto el estado final y la celda apuntada en esa situación.
4.-Estrategia de trabajo que ha de seguir la máquina para pasar de la configuración inicial a la final.

5.- Estados e instrucciones de la máquina.
6.- Comprobación.

Siempre que la máquina no sea muy simple se debe seguir este método.

Máquinas equivalentes
Veamos que puede haber diversas máquinas de Turing para resolver el mismo problema.

Otra MT para el mismo problema del ejemplo 1: El retoque (borrar un '1') se hace al principio. La máquina equivalente es:

$$Q = \{0\ s_0\ 0\ D\ s_1,\ 0\ s_1\ 1\ D\ s_1\ ,1\ s_0\ b\ D\ s_1,\ 1\ s_1\ 1\ D\ s_1,\ b\ s_1\ b\ N\ s_1\}$$

Explicación de la quíntupla <0 s_0 0 D s_1>: Es el caso de 1er sumando nulo. Como el resultado ha de ser el 2º sumando, éste ha de quedar intacto, así como el separador.

Comprobación:

Configuración inicial:	s_0	b b <u>1</u> 1 . . 1 0 1 1 . . 1 b
	s_1	b b b <u>1</u> . . 1 0 1 1 . . 1 b
	s_1	. . . . .
	s_1	b b b 1 . . 1 <u>0</u> 1 1 . . 1 b
	s_1	b b b 1 . . 1 1 <u>1</u> 1 . . 1 b
	s_1	. . . .
Configuración final:	s_1	b b b 1 . . 1 1 1 1 . . 1 <u>b</u>

También hay que comprobar para 1er sumando = 0 y/o 2º sumando = 0.

Para una descripción instantánea inicial distinta, la MT sería algo distinta. Por ejemplo, si se partiese de la configuración inicial s_0 ... b <u>b</u> b ... b 1 1 ... 1 0 1 1 ... 1 b b ...,
la MT correspondiente sería (borrando el 1er "1" del 1er nº):

M_2	b	1	0
s_0	b D s_0	b D s_1	0 D s_1
s_1	b N s_1	1 D s_1	1 D s_1

Hay que hacer notar que si se partiese de la 1ª MT (borrando el último "1" del 2º nº) habría que introducir un estado más, porque habría que distinguir 3 tipos de <u>b</u>: Los que se recorren hacia la derecha al principio, el primero que se encuentra cuando ha terminado de barrerse el 2º nº y el <u>b</u> que se encuentra al volver a la izquierda (por transformación del último "1" del 2º nº).

Observación sobre la Parada**:**

En el ejemplo 1 hay una casilla (0, s_1) vacía. Si la máquina se encontrara en esa situación, quedaría parada (bloqueada) por no existir instrucción para operar en ese caso. Eso no debe suceder si la máquina está bien construida. No debe haber otras instrucciones que lleven a la máquina a una tal situación (situación equivalente a una máquina de Von Neuman en un estado tal que el contenido del "contador del programa" apunta a una celda de memoria que no contiene ninguna instrucción).

Datos incorrectos:

Ahora bien en el caso de la MT puede llegarse a una situación, estando bien construida la máquina, de buena Parada aun cuando los datos estén incorrectamente representados (con errores). P.ej. supongamos en el Ejemplo 1 que queremos sumar 3 + 3 y nos hemos equivocado al escribir el 2º sumando:

Configuración inicial: ... b b 1 1 1 0 1 0 1 b b ...

La máquina trabajaría hasta llegar a la configuración final ...b b 1 1 1 1 1 1 b ..., que es la solución. Sin embargo esta situación no es deseable. Una elemental reflexión nos lleva a la conclusión de que esta máquina no distingue un símbolo separador de un error en un dato. Lo que podría hacerse en este caso es que la máquina distinguiera entre un primer símbolo y otros más, pues se supone que sólo debe haber uno. En la situación de encontrarse con un segundo “0” debe llegarse a una situación de Parada, pero esta Parada es debida a la detección de un error por la propia máquina. Es decir, todo programa debería contener una parte de reconocimiento y validación de la configuración inicial. En el caso que estamos tratando habrían de crearse un estado más y quíntuplas nuevas, al mero efecto de reconocimiento de un error en algún dato y consiguiente parada de aviso.

En s_0, cuando se encuentra el primer ‘0’, lo cambia a "1" y hay una transición a s_1, de forma que, si se encuentra otro "0" en estado s_1, pueda parar: s_1 0 0 N s_1. Ver M_e. Entonces se necesita un nuevo estado s_2, para encargarse de lo que antes se encargaba a s_1.

¿Por qué no hace bien el resultado M? ¿Qué es lo que hace en realidad M?. M (igual que M_1 y M_2) calcula la función:

$$f(x_1, x_2,...,x_n) = x_1 + 1 + x_2 + 1 + ... + x_n - 1 = \sum_{i}^{n} x_i + n - 2$$

$\sum_{i=1}$

Para n = 2, $f(x_1,x_2) = x_1 + x_2$

M_e	1	0	b
s_0	1 D s_0	1 D s_1	
s_1	1 D s_1	0 N s_1	b I s_2
s_2	b N s_2		b N s_2
		parada error	parada buena

Consideraciones similares pueden hacerse sobre todas las posibles clases de errores (b en los datos, p.e.). Las cosas pueden adquirir una complejidad grande en función de las clases de errores que admitamos como posibles.

Mientras no digamos lo contrario supondremos que la configuración inicial es correcta. Pero debemos tener en cuenta que "un algoritmo solo debe hacer aquella función para la que ha sido construido", **y nada más**. En el caso del Ej. 1, la máquina suma dos números y, además, hace otras cosas (considera que los '0', a partir del 1º, son "1").

Relación entre la cardinalidad del alfabeto y el nº de estados:

Los datos se pueden codificar con uno u otro alfabeto, con más o menos símbolos. Las MT correspondientes son diferentes en uno u otro caso. Veámoslo con un ejemplo:

Ejemplo : Sumar dos números, con la misma representación del ej. 1. $\sum = \{b, 1\}$
Configuración inicial: . . . b 1 1 1 b 1 1 b . . .

Necesitamos distinguir entre el b separador de los 2 números y el b que sigue al 2º número. Las acciones a realizar, en cada uno de esos dos casos, son diferentes. Con el mismo símbolo de entrada, la primera vez hay que sustituirlo por "1" y continuar moviendo el cursor a la derecha, pero en el 2º caso hay que dejarlo todo tal cual y desplazar el cursor a la izquierda. Teniendo en cuenta que no puede haber dos quíntuplas distintas con los mismos dos primeros símbolos y, además, que el símbolo de entrada (b) es el mismo en los dos casos, es preciso introducir otro estado más.

Tendremos entonces tres estados, caracterizados por:

s_0 : el carácter (1) que se lee es del 1er sumando.
s_1 : el carácter que se lee es del 2º sumando.
Cuando llega un b se ha terminado de explorar el 2º sumando.
s_2 : Se han terminado los sumandos.

Esquema funcional

	s_0	s_1	s_2
b	1 D s_1	b I s_2	b N s_2
1	1 D s_0	1 D s_1	b N s_2

Vemos que, para el mismo problema, se necesita un estado más. Y no es posible hacerlo con menos estados. Es ésta un **regla general**: "Cuanto menor es el nº de símbolos para representar la misma información, mayor es el nº de estados de la Máquina de Turing necesarios para resolver un problema dado".

LECCION 34. - LA MAQUINA UNIVERSAL DE TURING

Hasta aquí hemos considerado cada esquema funcional asociado a su MT propia. Pero es posible concebir una M T capaz de ejecutar cualquier algoritmo; es decir capaz de realizar los cálculos que realizaría cualquier otra MT, o sea, capaz de simular (tener el mismo comportamiento) cualquier MT particular.

Esta máquina Universal no debe ser diseñada para realizar un cálculo específico, sino para procesar cualquier información (realizar cualquier cálculo específico -MT particular- sobre cualquier configuración inicial de entrada correcta para esa MT particular). La MUT ha de tomar como información de entrada la MT particular más la C.I. . El **algoritmo universal** ha de ser un intérprete de esa información de entrada. La cinta de entrada ha de tener una C.I. tal como: ... bb quíntupla b quíntupla b... C I (MT) b b ...

El trabajo de interpretación del **esquema universal** ha de consistir en:
1º) $s_k = s_0$ (Inicialización: Suponer que la MT particular empieza estando en s_0).
2º) Acceder al carácter de la Configuración apuntado por la Cabeza L/E de la MT (a_i).
3º) Encontrar en la cinta una quíntupla que empiece por $a_i\ s_k$ ($a_i\ s_k\ a_j\ m\ s_1$).
4º) Reemplazar a_i por a_j en la casilla donde se accedió a a_i.
5º) Realizar el desplazamiento m desde la casilla donde estaba a_i. O sea, marcar la nueva casilla a la que apunta la cabeza L/E de la MT particular.
6º) $s_k = s_1$ (actualizar el estado de la MT particular).
7º) Volver a 2).

Observaciones sobre el esquema universal:

- La M U T parará cuando el cálculo esté terminado, o sea, cuando la quíntupla a aplicar sea del tipo $a_i\ s_k\ a_i\ s_k\ N\ s_k$. Así pues, en el ciclo que es el algoritmo universal,

hay que incluir dentro de la fase 3ª una exploración de la quíntupla, una vez encontrada, para decidir si hay que parar o continuar.

-Es complicado explicitar detalladamente las instrucciones que componen cada fase del Algoritmo Universal, debido al carácter unidimensional de la cinta y a la exploración casilla a casilla.

- Es obligada una codificación de la información. El nº de alfabetos y de caracteres distintos de todas las posibles M T particulares que puedan pensarse es arbitrariamente grande y, sin embargo, la M U T sólo puede disponer de un alfabeto finito determinado. Esto obliga a definir procedimientos para codificar la información de cualquier alfabeto finito al alfabeto que definamos para la M U T. Por el mismo motivo ha de haber una normalización en la notación de los estados (el estado inicial de cualquier M T debe ser designado con el mismo símbolo, y así para los sucesivos).

Una metodología para la aplicación de este concepto de normalización de la información es la numeración de Gödel, que se ve en Funciones Recursivas.

LECCION 35. - CODIFICACIÓN DE MT'S:

La codificación debe permitir a la MUT un reconocimiento preciso de cada componente sintáctico registrado en la cinta.

Para las letras del alfabeto:
c (b) = b (código del carácter blanco)
c (a_i) = 1^{i+1}; $a_i \in \sum \subseteq A$ (indeterminadamente grande); $i = 0,1,2,\ldots$

Para los estados:
c (h) = 1; h = estado de parada.
c(s_i) = 1^{i+1}; $S_i \in Q \subseteq S$ (indeterminadamente grande).
Para los desplazamientos del cursor, $m \in \{ N, I, D\}$:
c (N) = 1; c (I) = 1 1; c (D) = 1 1 1.

El código de una quíntupla 'p a q d m' está representado por una palabra $\in \{0, 1\}^*$, definida así: c(p) b c(a) b c (q) b c(d) b c(m) b.

Una MT que tiene n quíntuplas (el orden no importa): u_1, u_2, ……, u_n tiene un código: c(q_j) b c(u_1) b c(u_2) b …….. b c(u_n) b, siendo q_j: estado inicial de T.

Codificación de palabras

Sean las letras del alfabeto $Z_i \in A$ y Z una palabra $Z = Z_1 Z_2 \ldots\ldots Z_n$
Z tiene un código $c(Z) = b\ b\ c\ (Z_1)\ b\ c\ (Z_2)\ b \ldots\ldots\ b\ c\ (Z_n)\ b$

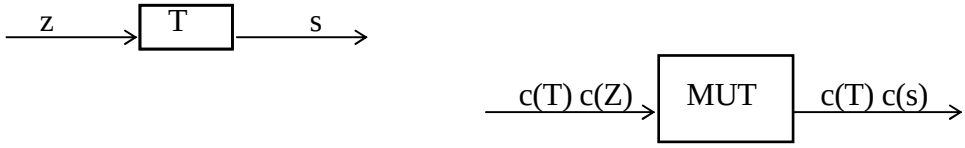
Sobre el trabajo de la MUT (continuación)

La entrada a la MUT está dada por una MT y una palabra Z. La entrada codificada es: c(T) c(Z).

Obsérvese que entre el final del código de la última quíntupla y el comienzo del código de Z hay tres blancos.

Es fácil construir la MT que transforma T en c(T) y $Z \in \sum^*$ en c(Z).

Para simular el trabajo de T con una entrada Z, la entrada a la MUT es c(T) c(Z). El trabajo ha de ser:
a) Cuando la función parcial que calcula T está definida para Z, T se detiene y da s como resultado. En este caso la MUT debe parar y dar c(s) como resultado.



b) Cuando la función que calcula T no está definida para Z, T no se detiene y la MUT tampoco debe parar. En este caso la MUT se mete en un ciclo infinito.

BIBLIOGRAFIA

M. Alfonseca, J. Sancho, M. Martínez Orga, (1990). Teoría de Lenguajes, Gramáticas y Autómatas,España,Ediciones Universidad y Cultura.

Brokshear J, Blend (1993), Teoria de la computación, Lenguajes formales, Autómatas y complejidad México edición, Addison-Wesley iberoamericana.

Dean Kelley, (1995), Teoría de autómatas y lenguajes formales, España Prentice - Hall.

Pedro Isasi, Paola Martínez, Daniel. Borrajo (1997). Lenguajes, gramáticas y autómatas Un enfoque práctico España, 2ª edición, Addison-Wesley

J.E. Hopcroft, R. Motwani, J.D. Ullman, (2002) Introducción a la Teoría de Autómatas, Lenguajes y Computación, España 2ª edición, Addison-Wesley iberoamericana.

Martín John, (2004), Lenguajes formales y teoría de la computación, México, 3ª Edición, Mc Graw Hill.

SITIOS WEB

- **Página web de la asignatura universidad rey Juan carlos**: http://www.ia.escet.urjc.es/grupo/docencia/automatas

- **Foro:** http://www.ia.escet.urjc.es/foros/automatas

- http://luisguillermo.com/castellano/default.htm#supra

- http://www.ing.puc.cl/˜jabaier/iic2222

- **aula virtual** http://www.upseros.com/

Software

Software para el aprendizaje de lenguajes y autómatas: JFLAP http://www.cs.duke.edu/%7Erodger/tools/jflap/ Excelente paquete en Java, por lo que es multiplataforma. Incluye edición gráfica de autómatas, conversiones, minimización, gramáticas, máquinas de Turing, y mucho más.

Chalchalero http://www.ucse.edu.ar/fma/sepa/chalchalero.htm Extenso paquete hecho en Argentina para PC con Windows, incluye editor de autómatas, conversiones, minimización, asi como numerosas herramientas para aprendizaje de compiladores.

REFERENCIAS

MORAL CALLEJÓN Serafín Teoría de autómatas y lenguajes formales, En http://decsai.ugr.es/~smc/docencia/mci/automata.pdf

NAVARRETE SÁNCHEZ, Isabel y otros Teoría de autómatas y lenguajes formales En: http://perseo.dif.um.es/%7Eroque/talf/Material/apuntes.pdf

MORAL CALLEJÓN, Serafín Modelos de Computación I, Departamento de ciencias de la computación Universidad de Granada En: http://decsai.ugr.es/%7Esmc/docencia/mci/automata.pdf.

NAVARRETE SÁNCHEZ, Isabel y otros Teoría de autómatas y lenguajes formales En: http://perseo.dif.um.es/%7Eroque/talf/Material/apuntes.pdf.

Wikipedia, En http://es.wikipedia.org/wiki/Lenguaje_formal

BRENA PIÑERO, Ramon F. Autómatas y lenguajes un enfoque de diseño (2003) ITESM, En: http://lizt.mty.itesm.mx/~rbrena/AyL.html.

MARIN MORALES, Roque y otros, Teoría de autómatas y lenguajes formales En: Univ. de Murcia http://perseo.dif.um.es/%7Eroque/talf/Material/apuntes.pdf

IOST F. Hans, Teoría de autómatas y lenguajes formales, CAPÍTULO 2, LENGUAJES REGULARES Y AUTOMATAS FINITOS (2001) en: http://iie.ufro.cl/~hansiost/automatas/Capitulo2.doc

Ingeniería Técnica en Informática de Sistemas y de Gestión de la UNED España TEORÍA DE AUTÓMATAS I Tutoría del Centro Asociado de Plasencia en: URL http://dac.escet.urjc.es/lrincon/uned/ta1/ProblemasFinalCapitulo.pdf.

Departamento de Ingeniería de Sistemas Facultad de Ingeniería Ciencias y Administración Universidad de La Frontera IIS340 TEORÍA DE AUTOMATAS Y LENGUAJES FORMALES APUNTES DEL CURSO Temuco, 1999 – 2001

CUEVAS LOVALLE, Juan Manuel LENGUAJES, GRAMÁTICAS Y AUTÓMATAS. Segunda Edición, (España), 2001.

Ingeniería Superior en Informática, Grupos A, B y C Teoría de Autómatas y Lenguajes formales. Examen Final-19 de Junio de 2003, Solución Ejercicios

AUTOEVALUACION

FORMATO EVALUACION ACTIVIDAD INDIVIDUAL

Cada tarea individual será revisada y evaluada según esta pauta de carácter formativo, se considerarán los siguientes aspectos:

Actividad Nº _____ Nota : Formativa Nombre: _____ Fecha: Aspectos considerados en la revisión
Fecha recepción
Envío archivo
Presentación de los contenidos
Respuesta a lo solicitado
Nivel de profundidad y/o análisis
Observaciones

PAUTA EVALUACION PRESENTACIÓN FINAL PROYECTO

Fecha: **Nombre de los integrantes en orden de presentación:**

1._____
2._____
3._____
4._____
5.

ASPECTOS A EVALUAR	Puntaje por aspecto para cada integrante (1 a 3 puntos y 1 a 2 puntos)				
	1	2	3	4	5
Dominio del contenido de persona que expone en representación del grupo	1/ 2 / 3	1/ 2 / 3	1/ 2 / 3	1/ 2 / 3	1/ 2 / 3
Exposición complementada con el uso del recurso tecnológico (presentación en PPT)	1 / 2 / 3	1 / 2 / 3	1 / 2 / 3	1 / 2 / 3	1 / 2 / 3
Defensa del proyecto con calidad y precisión de las respuestas del expositor y/o grupo	1 / 2	1 / 2	1 / 2	1 / 2	1 / 2
Habilidad para motivar y mantener la atención de la audiencia por parte del expositor.	1 / 2	1 / 2	1 / 2	1 / 2	1 / 2
Coordinación grupal para antes, durante y después de la presentación.	1 / 2	1 / 2	1 / 2	1 / 2	1 / 2
Exposición dentro del tiempo estipulado (15 minutos por grupo).	1 / 2	1 / 2	1 / 2	1 / 2	1 / 2
Puntaje máximo 14 puntos					
Nota 10%					

Nombre del Tutor:

_____

AUTOEVALUACION

Evaluación individual de cada integrante del grupo

Nombre: _____**Fecha:** _____

A continuación se presenta una serie de aspectos relacionadas con la participación en el desarrollo del curso académico

Para autoevaluar la participación individual y grupal, asignar para cada aspecto un puntaje de 1 a 5. Aquellos aspectos o afirmaciones aseveraciones que no pueda calificar, asignar 0 (cero) y justificar por escrito.

Puntaje y su descripción:

1	**2**	**3**	**4**	**5**
Insuficiente	Menos que regular	Regular	Bueno	Muy Bueno

Aspectos o afirmaciones	Puntaje
Estuve en permanente contacto con mi grupo colaborativo.	
Me comuniqué con mi tutor frente a alguna emergencia, duda, etc.	
Acusé recibo de las comunicaciones realizadas por e-mail.	
Realicé las tareas que me comprometí en forma oportuna	
Colaboré oportunamente en la fases del desarrollo de las guías didácticas	
Tomé en cuenta las observaciones y sugerencias hechas por el tutor en la planificación de las actividades.	
Busqué otras fuentes de información (textos, URL, etc.) para el desarrollo del sitio web.	
Participé activamente en el desarrollo de los contenidos de la guía didáctica	
Participé activamente en las reuniones grupales para la realización de las actividades.	
Los aportes realizados a mi grupo de trabajo fueron oportunos.	
Mantuve una interacción respetuosa con mis compañeros de grupo y tutor.	
Asistí puntualmente a todas las reuniones fijadas por el grupo	

¿Qué aspectos ha mejorado en sus procesos de aprendizaje

Justificación para afirmaciones o aspectos no calificadas con puntaje de 1 a 5:

COEVALUACION

Evaluación del coordinador a su grupo o de cada integrante del grupo a un compañero

Nombre del Evaluador:_____Fecha: _____

El trabajo grupal colaborativo relacionado con la participación y aporte de cada integrante en las actividades grupales solicitadas, en aspectos como son la discusión, análisis y generación conjunta de los contenidos requeridos en los mismos, al igual que la participación activa y colaborativa en el proceso de aprendizaje.

Como Integrante del grupo de trabajo y coordinador, se le solicita evaluar el grado de participación de cada uno de los participantes, asignando un puntaje de 1 a 5 para cada una de los siguientes aspectos. Aunque esta evaluación implica apreciación personal, la solicitud es que asigne el puntaje de manera objetiva.

Puntaje y su descripción:

1	2	3	4	5
Insuficiente	Menos que regular	Regular	Bueno	Muy Bueno

AFIRMACIONES – aspectos	NOMBRE DEL PARTICIPANTE			
Asistió a las reuniones de trabajo de grupo.				
Mantuvo permanente comunicación con el resto de los participantes del grupo.				
Ayudó a aclarar dudas o confusiones en torno a los temas tratados en las actividades.				
Intercambió información, datos, ideas, etc. con sus compañeros de grupo durante el desarrollo de las actividades.				
Hizo algún aporte en el desarrollo de las actividades.				
Participó en la solución de problemas durante el trabajo grupal.				
Ayudó a organizar el trabajo grupal.				
Fue responsable con las acciones asumidas dentro del grupo de trabajo.				
Fue puntual en las reuniones convocadas para el trabajo grupal.				
Mantuvo una interacción respetuosa con todos los integrantes del grupo.				
Consideró las ideas y aportes del resto del grupo de trabajo.				
Colaboró en la búsqueda de información y recursos para la fase del desarrollo de las unidades didácticas.				

En su rol de Coordinador del grupo de trabajo o de integrante se le solicita señalar cómo se sintió respecto al trabajo en equipo, además de las debilidades y fortalezas percibidas en su grupo de trabajo.

_____

_____