Universidade Vigo

Teoría de Autómatas y Lenguajes Formales

Versión 11

Dr. Arno Formella
Universidade de Vigo
Escola Superior de Enxeñaría Informática
Departamento de Informática
Área de Linguaxes e Sistemas Informáticos
E-32004 Ourense

http://www.ei.uvigo.es/~formella

formella@ei.uvigo.es

Junio 2010

Índice

1. Curso

1.1. Administración

Véase Guía Docente (página 10).

1.2. Clases

	Lunes	Martes	Viernes
10-11	TALF [22]	TALF [22]	
11-12		TALF [22]	
12-13			TALF1 [SO5]
13-14			TALF3 [SO5]
16-17	TALF2 [SO5]		

1.3. Tutorías

	Lunes	Miércoles
11-12	Teoría	
12-13	Teoría	
13-14	Teoría	
16-17		
17-18	Teoría/Prácticas	
18-19	Teoría	Prácticas
19-20	Teoría	Prácticas

1.4. Evaluación

1.4.1. Asistentes y no asistentes

Véase también Guía Docente (página 10).

Un asistente es alguien que quiere aprovechar de la evaluación continua. Eso se muestra con tres componentes: asistencia a clases de teoría, entrega de trabajos semanalmente, participación a evaluación continua en prácticas.

Un no asistente es alguien que *i)* no participa en la evaluación continua, y que *ii)* quiere realizar solamente los exámenes, tanto de teoría como de prácticas, (claro tendrá todo el apoyo tanto en clase como en tutorías).

¿Cuándo uno es *asistente* y cuando *no asistente*? Pues muy simple:

- *Asistente:* si quiere aprovechar de la evaluación continua, tanto en teoría como en prácticas.
- *no Asistente:* Si quiere realizar los exámenes, tanto en teoría como en prácticas.

Consecuencias: Quien no realiza examen de prácticas, asume evaluación continua tanto en teoría como en prácticas.

Se asume por defecto que el alumno quiere ser asistente, sino quiere ser lo, lo comunica—como muy tarde—al principio del examen de junio.

1.4.2. Entregas

- Habrá unas 13 hojas.
- Los nuevos problemas se distribuen los martes.
- Las entregas se realizan de forma personal el siguiente martes con clases presenciales.
- Una entrega consiste en **una hoja** escrito **a mano** (es decir, ni impreso ni copia) personalmente.
- Dicha hoja lleva en la parte derecha arriba los apellidos, el nombre y el DNI. (**¡Hojas que no cumplen con eso no serán ni mirados desde lejos!**)
- No hay excepciones para dicha forma de entrega.
- No se publican los resultados/correcciones individuales, sino se publica una posible solución correcta.
- El alumno debe autoevaluarse con las soluciones publicadas, la copia de la entrega guardada y mejor en grupo de compañeros.
- Se puede venir para hablar de las entregas en tutorías.
- Tres de las entregas se seleccionarán al azar para su incorporación en el proceso de evaluación final de la asignatura (mira Guía Docente).

1.5. Exámenes

Convocatoria	Fecha	Hora	Aula
Junio	15.06.10	10:00	Magna
Septiembre	06.09.10	16:00	Magna

2. Sobre este documento

Este documento es un servicio adicional del profesor para los estudiantes. Se recuerda que la asignatura es **presencial**. Observa los siguientes comentarios importantes:

Los apuntes

- **no necesariamente están completos.** El contenido de la asignatura se define por lo que se expone durante las clases presenciales en congruencia con el programa oficial de la asignatura.
- **no necesariamente son correctos.** Intento lo mejor. Siempre existe la posibilidad de que haya fallos ortográficos o de redacción. Además, como en todos los campos de la informática, lo correcto no es lo que diga el profesor, **lo correcto es, lo que es correcto**.
- **no necesariamente siguen el orden de las clases presenciales.** Es posible que el orden de los apartados no es exactamente igual que el orden presentado en clases.
- **contienen apartados que no se han dado en todos los cursos.** Siendo clases presenciales, los estudiantes ya sabrán distinguir.
- **todavía están sin gráficos**.

Los apuntes se han escrito sin el uso de una enumeración explícita de las definiciones, lemas, y teoremas, como se suele usar en tal contexto. Se anima al lector que estructure su 'novela individual' con annotaciones en los márgenes para enumerar las definiciones, lemas, y teoremas y relacionar así de mejor manera las diferentes partes.

2.1. Versiones y lista de correcciones

A parte de algunas correcciones de ortografía y de correcciones menores se han realizado las siguientes cambios de versión en versión:

- **Versión 11**
  - Se ha modificado ligeramente la sección de las relaciones de equivalencia (página 26).
  - Se ha revisado la definición de la Máquina de Turing (página 18) que tenía el error de $B \notin \Gamma$, pero la B tiene que pertenecer a Γ sino la máquina se quedaría atrapada una vez visitando tal casilla.
  - Se ha añadido un apartado sobre la evaluación (página 5).
  - Se ha añadido enlaces en red en la bibliografía.
- **Versión 10**
  - Faltaba un $\emptyset$ en la definition de una gramática (página 29).
  - Por problemas de notación de números flotantes, aparecieron comas en vez de puntos en los ejemplos de expresiones regulares (página 58).
- **Versión 9**
  - En el quinto paso de la transformación a FNC (página 82) se ha olvidado eliminar variables que puedan ser inacesibles después de haber eliminado las reglas unitarias.
  - En el primer paso de la transformación de gramática lineal por la derecha a gramática lineal por la izquierda (página 66) estaban la variables a la izquierda en vez a la derecha.
- **Versión 8**
  - Como siempre, más errores menores corregidos.
  - Está incorporado la guía docente (página 10).
  - Se actualizó la bibliografía básica.
- **Versión 7**
  - Se han corregido unos errores menores de uso de símbolos y faltas de parentesis.
  - Había un fallo en el tercer paso de la construcción de una gramática en forma normal de Greibach (página 84), dado que en el segundo paso se pueden generer reglas largas, entonces, Y puede ser una palabra sobre las variables (se usa Υ ahora).
- **Versión 6**
  - Se cambió ligeramente la introducción añadiendo sobre todo un apartado sobre el concepto de la máquina de turing y su universalidad.
- **Versión 5**

- Se ha movido la definición de equivalencia entre autómatas al comienzo de la sección 7.3.
- Había un fallo en la definición del lenguaje L_{dos} (página 40).
- Se ha aumentado la definición de δ^* para AFND con otra equación a lo mejor más sencillo de entender (página 40).

■ **Versión 4**

- Había un fallo en las propiedades de la concatenación de lenguajes: $L_1.\emptyset = \emptyset$ (y no $= L_1$) (página 24).

■ **Versión 3**

- Se ha modificada ligeramente la introducción.

■ **Versión 2**

- (Sección 5.6) mejorado la precisión de la observación
- (Sección 9) Si $\delta(q, \sigma) = p$ es una transición del AFD, con $q \in Q$, $p \in F$ y $\sigma \in \Sigma$, entonces añadimos a P la producción $q \longrightarrow \sigma$.
- (Sección 9) Entonces el sistema de producciones P de la gramática será:

$$P \quad = \quad \{q_0 \longrightarrow aq_0|a|bq_1|b|cq_2|c|\epsilon, q_1 \longrightarrow bq_1|b|cq_2|c, q_2 \longrightarrow cq_2|c\}$$

- (Sección 11.1) Si el lenguaje de partida L contiene la palabra vacía ($\epsilon \in L$) entonces se detecta en el paso 4 que el símbolo inicial pertenece a E (o incluso a E_ϵ), en este caso eliminamos con un nuevo símbolo, por ejemplo $\$'$, la aparencia de $\$ en los lados derechos y añadimos la regla $\$ \longrightarrow \epsilon$. Tal gramática sigue estando en forma normal de Chomsky y genera L.
- (Sección 11.2) Obviamente cualquier gramática en forma normal de Greibach es una gramática libre de contexto que se verifica directamente analizando la forma de producciones permitidas.

3. Guía docente

3.1. Contextualización

3.1.1. Perfil de los créditos de la asignatura

Los créditos de la materia son teóricos, matemáticos, y fundamentales con una fuerte aplicabilidad en todos los ámbitos de la informática.

3.1.2. Ubicación y relación con el Plan de Estudios

TALF se situa como asignatura obligatoria en el segundo cuatrimestre del segundo año de la titulación de Ingeniero Técnico en Informática de Gestión. Se introducen con cierto nivel de abstracción matemática los fundamentos de la teoría de la Informática que engloban el concepto de lenguaje formal con su descripción en la jerarquia de Chomsky y el concepto de autómata como entidad que implementa algoritmos con su descripción más general, la Máquina de Turing. El contenido de TALF forma la base necesaria para comprender el funcionamiento de compiladores, la evaluación de algoritmos, la descripción de datos y de información, la especificación de interfaces, el funcionamiento de procesadores, y las capacidades del proceso de cálculo.

3.2. Objetivos

3.2.1. Objetivos generales

- Comprender los fundamentos básicos de los lenguajes formales, sus propiedades y mecanismos de representación.
- Entender el funcionamiento de las gramáticas como generadoras de lenguajes y diferenciar sus tipos.
- Destacar el papel de los autómatas finitos en el reconocimiento de lenguajes y distinguir entre los diferentes tipos de autómatas.
- Relacionar tipos de lenguajes con autómatas y gramáticas, sobre todo para lenguajes regulares y lenguajes libres de contexto.
- Comprender y analizar algoritmos básicos en el contexto de lenguajes formales.
- Conocer la conexión entre redes neuronales y autómatas.

3.2.2. Competencias

Competencias específicas:

1. Conocer, comprender y aplicar los métodos, teorías y tecnologías propias de los fundamentos de las estructuras de datos, programación y sistemas para analizar modelar, manipular, diseñar y desarrollar sistemas informáticos a nivel básico.
2. Aplicar metodologías de desarrollo en el análisis y diseño detallado de los sistemas informáticos.
3. Asesorar a los programadores en los problemas que se les plantean con la programación de los sistemas.
4. Conocer, comprender y aplicar teorías, métodos, técnicas y herramientas de la matemática discreta, la lógica, el álgebra y el análisis matemático para analizar, modelar, manipular y diseñar elementos y sistemas informáticos.
5. Encontrar soluciones algorítmicas a problemas, comprendiendo la idoneidad y complejidad de las soluciones propuestas.
6. Definir, describir y especificar interfaces de usuario y relacionarlas con las características específicas de los procesos y los sistemas informáticos

Competencias transversales:

instrumentales:

1. Capacidad de análisis, síntesis y evaluación
2. Capacidad de organización y planificación
3. Comunicación oral y escrita en la lengua nativa
4. Capacidad de abstracción: capacidad de crear y utilizar modelos que reflejen situaciones reales
5. Capacidad de diseñar y realizar experimentos sencillos y analizar e interpretar sus resultados
6. Capacidad de buscar, relacionar y estructurar información proveniente de diversas fuentes y de integrar ideas y conocimientos
7. Resolución de problemas
8. Capacidad de tomar decisiones
9. Capacidad para argumentar y justificar lógicamente las decisiones tomadas y las opiniones

personales:

1. Capacidad de actuar autónomamente
2. Capacidad de relación interpersonal

sistémicas:

1. Razonamiento crítico
2. Aprendizaje autónomo
3. Creatividad
4. Tener iniciativa y ser resoluto
5. Tener motivación por la calidad y la mejorar continua

3.3. Prerrequisitos

Conocimientos matemáticos básicos: matemática discreta, álgebra.

3.4. Resumen del contenido

3.4.1. Descriptor de la asignatura (BOE)

Máquinas secuenciales y autómatas finitos. Máquina de Turing. Gramáticas y lenguajes formales. Redes neuronales.

3.4.2. Teoría

1. Introducción y conceptos básicos
2. Gramáticas generativas
3. Máquinas secuenciales y autómatas
4. Autómatas finitos, gramáticas regulares, expresiones regulares
5. Autómatas de pila, gramáticas libres de contexto
6. Máquinas de Turing, gramáticas sensitivos al contexto y generales
7. Redes neuronales

3.4.3. Práctica

En las prácticas se verá el uso de lenguajes formales, autómatas y expresiones regulares en herramientas didácticas (JFLAP) que permitan explorar y ejercitar los conceptos y procedimientos recogidos en el temario de la teoría. Se verá herramientas en el ámbito de la aplicación de la informática (por ejemplo: búsquedas en el sistema operativo, búsquedas y sustituciones en editores de texto, especificación formal de contenidos de ficheros, etc.). Además se implementará algunos de los algoritmos básicos que se estudian en la teoría.

3.5. Metodologías y estrategias de aprendizaje

El aprendizaje del estudiante de base en una combinación de diferentes metodologías y estrategias que se detalla como sigue:

1. Clases magistrales donde se desarrollan los conceptos teóricos en pizarra
2. Clases magistrales donde se elabora sobre ejercicios prácticos
3. Clases en el laboratorio con herramientas y aplicaciones para ejercer los conocimientos adqueridos y usar los en entornos reales
4. Lectura asignada para repetir concocimiento y adquerir por medios propios nuevos aspectos
5. Realización frecuente de ejercicios con entrega individual y autocontrol y/o control cruzado entre estudiantes

3.6. Plan de trabajo del alumnado presencial

Este plan varia en cierto modo del plan de trabajo del programa porque considera las semanas reales con sus festivos del calendario.

Actividades	**Horas Pres.**	**Factor**	**Horas Non Pres.**	**TOTAL**
Clase magistral con avance teórico	28	0.5	14	42
Clase magistral con enfoque práctico	12	0.5	6	18
Clase práctica en laboratorio	12	0.5	6	18
Coordinación en grupo, intercambio de soluciones	2	4	8	10
Elaboración de entregables	0	?	26	26
Preparación examen con ejercicios en grupo	0	?	9	9
Examen final	2	0	0	2
TOTAL	56		69	125

3.7. Evaluación de los procesos y resultados de aprendizaje. Criterios de evaluación

3.7.1. Criterios de evaluación para asistentes

1. La evaluación de la Teoría constará de un examen final (50 % de la nota final) y una evaluación con mostreo aleatorio de 3 de 13 entregas durante el curso (15 % de la nota final).

2. La evaluación de las Prácticas (30 % de la nota final) constará de una evaluación continua durante las clases en el laboratorio para la convocatoria de junio.

3. Para las convocatorias de septiembre y diciembre el alumno o bien se puede aprovechar de los resultados guardados de la evaluación continua o bien se puede presentar a un examen de Prácticas.

4. 5 % de la nota final se obtiene por por lo menos 90 % de asistencia a clases de teoría.

5. Para aprobar la asignatura hay que aprobar la parte de Teoría (es decir, llegar a un 50 % de los puntos del examen de Teoría) y llegar en la nota ponderada a un mínimo de un 5.

3.7.2. Criterios de evaluación para no asistentes

1. La evaluación de la Teoría constará de un examen final.

2. La evaluación de las Prácticas constará de un examen final.

3. La nota final se calcula ponderando un 70 % la Teoría y un 30 % las Prácticas.

4. Para aprobar la asignatura hay que aprobar la parte de Teoría (es decir, llegar a un 50 % de los puntos del examen de Teoría) y llegar en la nota ponderada a un mínimo de un 5.

3.8. Observaciones

Se ha observado en los años anteriores que la falta de constancia durante el curso es la causa principal de muchos malos resultados. TALF es una asignatura cual, por la naturaleza de su contenido, no se puede *aprender* en 15 días de estudios aislados antes de un examen. Dado que en el segundo cuatrimestre no se puede esperar de empezar las actividades académicas de la asignatura antes del 22 de febrero y se tiene que terminar en junio, se dispone solamente de unos tres meses y medio para la distribución de los créditos ETCS que conlleva que como mucho se puede planificar unas 100 horas para no sobrecargar el alumno en el conjunto de actividades durante el período lectivo. El alumno debe distribuir las horas restantes en los periodos no lectivos.

4. Introducción

¿Por qué es importante la teoría de lenguajes formales y autómatas?

Bueno, aclaramos primero un poco las palabras usadas.

¿Qué es un lenguaje formal?

Conocemos lenguajes naturales.

- español, alemán, inglés, chino, árabe...
- cuando nacemos no sabemos ningún lenguaje
- se puede aprender cualquier lenguaje natural (por lo menos si se ha nacido en un entorno adecuado, y no hay obstáculos orgánicos)
- el lenguaje es una secuencia de fonemas o símbolos
  - que forman sílabas, palabras, frases, párrafos, capítulos, novelas, libros, bibliotecas etc.
  - que tiene una sintaxis (fonética o ortografía)
  - que tiene una gramática (reglas de concatenación y construcción de palabras para formar frases)
  - (que tiene un estilo (forma de unir frases para generar textos))

Hoy en día aparecen cada vez más símbolos (llamados iconos) con el propósito de *simplificar* sobre todo las *interfaces al usuario*, es decir, se quiere transmitir una semántica a partir de un símbolo. Como ejercicio para reflexionar y criticar el uso excesivo de tal moda, se pide: intenta averiguar el significado de los siguientes símbolos (relacionados semánticamente entre si) que están en uso desde hace cientos de años por una gran parte de la humanidad:

En el contexto de esta asignatura lenguajes formales serán meramente conjuntos de secuencias de símbolos (cuya construcción se consigue con una gramática formal).

¿Qué es un autómata?

- dispositivo mecánico o electrónico o biológico
  - que en un punto de tiempo está en un estado
  - que dado una razón (por ejemplo una señal de entrada) cambia de estado
- ejemplos son: reloj mecánico o electrónico, máquina para lavar, todo un ordenador, ¿el cerebro?
- ya se han construido relojes biológicos con trozos de DNA artificial y síntesis de proteínas que visualizan su cambio de estado con luz fluorescente

En el contexto de esta asignatura autómatas serán máquinas matemáticas con estados y funciones de transición (donde se puede añadir entrada, salida, memoria interna modificable, etc.).

- Los conceptos de gramáticas (formales) y de los autómatas describen el mismo fenómeno y están muy relacionados con los algoritmos.
- Se distingue la Teoría de Computabilidad y la Teoría de Complejidad, es decir, la búsqueda de respuestas a preguntas como: ¿Qué es computable? y ¿Cuántos recursos (memoria, espacio, tiempo, transiciones) se necesitan?

Es decir, la Teoría de los Lenguajes Formales (y de los Autómatas) permite responder a preguntas escenciales de la Informática. Dos observaciones importantes:

- Tesis de Church: Todo lo que es computable se puede calcular con una Máquina de Turing.
- Existen problemas que no son computables.

Resumen: Sin TALF no hay lenguajes, no hay compiladores, no hay programas, no hay ordenadores, no hay red, no hay nada.

4.1. Reglas de sustitución para formar secuencias

favoritas

Con este "diagrama" podemos formar unas reglas para sustituir símbolos:

$$\begin{array}{llll} \$ \longrightarrow AB & A \longrightarrow \text{esas} & A \longrightarrow \varepsilon & B \longrightarrow CD \\ C \longrightarrow \text{son} & D \longrightarrow EF & E \longrightarrow GH & G \longrightarrow \text{mis} \\ G \longrightarrow \varepsilon & H \longrightarrow IJ & I \longrightarrow \text{clases} & J \longrightarrow \text{favoritas} \\ J \longrightarrow \varepsilon & F \longrightarrow \text{en informatica} & F \longrightarrow \varepsilon & \end{array}$$

donde usamos ε para decir que no escribimos nada.

Con dichas reglas podemos 'derivar' diferentes frases, por ejemplo:

$$\begin{aligned}
\$ &\longrightarrow AB \\
&\longrightarrow \text{esas}B \\
&\longrightarrow \text{esas}CD \\
&\longrightarrow \text{esas son}D \\
&\longrightarrow \text{esas son}EF \\
&\longrightarrow \text{esas son}GHF \\
&\longrightarrow \text{esas son}HF \\
&\longrightarrow \text{esas son}H \\
&\longrightarrow \text{esas son}IJ \\
&\longrightarrow \text{esas son clases}J \\
&\longrightarrow \text{esas son clases}
\end{aligned}$$

donde siempre hemos usado una regla adecuada para sustituir símbolos hasta llegar a tal punto que ya no se puede aplicar ninguna regla más.

Y con pequeños arreglos podemos traducirlo al alemán:

$\$ \longrightarrow AB$	$A \longrightarrow \text{dies}$		$B \longrightarrow CD$
$C \longrightarrow \text{sind}$	$D \longrightarrow EF$	$E \longrightarrow GH$	$G \longrightarrow \text{meine}$
$G \longrightarrow \varepsilon$	$H \longrightarrow JI$	$I \longrightarrow \text{Vorlesungen}$	$J \longrightarrow \text{liebsten}$
$J \longrightarrow \varepsilon$	$F \longrightarrow \text{in Informatik}$	$F \longrightarrow \varepsilon$	

es decir, hemos quitado la regla $A \longrightarrow \varepsilon$ y hemos cambiado la regla de $H \longrightarrow IJ$ a $H \longrightarrow JI$.

Otro ejemplo más sencillo.

Usamos las reglas $\$ \longrightarrow ab\$$ y $\$ \longrightarrow \varepsilon$ para generar palabras del tipo ab, $abab$, $ababab$ etc. Podemos derivar una palabra:

$$\$ \longrightarrow ab\$ \longrightarrow abab\$ \longrightarrow ababab\$ \longrightarrow ababab$$

siempre aplicando alguna de las reglas hasta tal punto que ya no se puede aplicar ninguna regla. Hemos usado el símbolo ε para decir que no sustituimos por nada (juega el mismo papel que el 0 para números).

4.2. Autómatas que aceptan secuencias

Construimos un autómata que acepta una palabra del tipo mencionado anteriormente. Entendemos por aceptar que el autómata llega a un estado final. Consumimos para cada transición de estado una letra de la palabra. Podemos dibujar un autómata:

automata

donde el estado inicial (o de comienzo) está marcado con una flecha, el estado final está marcado con un doble círculo. Las transiciones están visualizadas con flechas entre los estados que a su vez están marcados con sus símbolos correspondientes. Si empezamos en el estado inicial, y si leemos la palabra por aceptar desde la izquierda hacia la derecha, podemos saltar de estado a estado siguiendo los arcos adecuados.

Observamos que llegamos solamente al estado final si la palabra por aceptar es una palabra válida del lenguaje.

4.3. Lenguajes y autómatas

Vemos y veremos

- que las gramáticas sirven para generar palabras (y con eso lenguajes) y
- que los autómatas sirven para aceptar palabras (y con eso lenguajes).

Hacia el final del curso tendremos algunos conocimientos sobre una jerarquía de lenguajes y las equivalencias entre:

- Lenguajes Tipo 3, Gramáticas Regulares y Autómatas Finitos,
- Lenguajes Tipo 2, Gramáticas Libres de Contexto y Autómatas Finitos con Pila,
- Lenguajes Tipo 1, Gramáticas Sensitivos al Contexto y Autómatas Linealmente Acotados,
- Lenguajes Tipo 0, Gramáticas Generales y Máquinas de Turing.

Dicha clasificacón es algo gruesa considerando avances modernas en la teoría de lenguajes formales: los lenguajes tipo 2 se dividen en dos: deterministas y no-deterministas que se aceptan con autómatas de pila deterministas y no-deterministas respectivamente. Luego, los lenguages Tipo 1 ya se dividen en tres subclases que se aceptan con autómatas con cierto tipo de multi-pila.

4.4. Máquinas de Turing universales

Observamos la siguiente máquina muy simple:

mturing

Como máquina matemática se suele describir con siete componentes

$$M = (\Sigma, \Gamma, Q, \delta, B, q_0, F)$$

con (mucho de las notaciones siguientes se entiende más tarde)

1. Σ es un conjunto finito de símbolos (alfabeto) para escribir la entrada
2. Γ es un conjunto finito de símbolos (alfabeto) para escribir sobre la cinta ($\Sigma \subset \Gamma$, $|\Gamma| < \infty$)
3. Q es un conjunto finito de estados ($|Q| < \infty$)
4. δ es la función de transición

   $$\delta : Q \times \Gamma \longrightarrow W_{<\infty}(Q \times \Gamma \times \{L, R, N\}); \delta(q, \sigma) = (p, \sigma', m)$$

   donde L significa que la cabeza de lectura va una posición a la izquierda, R que va a la derecha, y N que no se mueve (este último se puede obviar).
5. q_0 es el estado inicial ($q_0 \in Q$)
6. F es un conjunto de estados finales ($F \subset Q$)
7. B es un símbolo que se usa para marcar una casilla vacía en la cinta ($B \in \Gamma$)

donde se asumen ciertas cosas implícitamente

- la salida es el contenido de la cinta una vez haber llegado a un estado final
- la cinta al comienzo contiene a parte de la entrada solamente casillas con el símbolo del blanco (un número infinito)
- la entrada se presenta de forma consecutiva en la cinta
- la cabeza está situada al principio a la izquierda de la entrada

Dada una entrada la máquina realiza transiciones hasta que llegue a un estado final.

Con el concepto de simulación, es decir, una máquina M simula otra máquina M' reproduciendo la misma salida con la misma entrada se puede construir una máquina de Turing universal. Dicha máquina, que aparte de la entrada también dispone de la descripción (en una codificación adecuada) de la máquina M a simular en la cinta, es capaz de simular cualquier otro máquina.

Con unos arreglos técnicos se puede construir tal máquina de Turing universal (MTU) con solo 2 símbolos de cinta y 7 estados, o con 5 símbolos de cinta y 2 estados. Es decir, pueden ser sorprendentemente pequeños teniendo en cuenta que sirven para simular cualquier otra máquina. (En 2007 se comprobó que también existe una MTU con solo 3 símbolos de cinta y 2 estados, pero la comprobación todavía está discutida.)

Más información sobre máquinas de Turing pequeñas, por ejemplo, en
`http://www.mathrix.org/experimentalAIT/TuringMachine.html`

donde hay una MTU en C/C++ con 285 characteres de código fuente.

5. Conceptos básicos

5.1. Alfabetos

Un **alfabeto** es un conjunto finito no vacío de símbolos.

$$\begin{aligned}
\Sigma_1 &= \{0, 1\} \\
\Sigma_2 &= \{a, b\} \\
\Sigma_3 &= \{\text{na}, \text{pa}, \text{bra}, \text{la}\} \\
\Sigma_4 &= \{\texttt{<HTML>}, \texttt{</HTML>}, \texttt{<BODY>}, \texttt{</BODY>}, \ldots\} \\
\Sigma_5 &= \{|\} \\
\Sigma_6 &= \{a, ab, aab\}
\end{aligned}$$

- Usamos meta-símbolos (tal como $\{$, $\}$, $=$, y la coma) para escribir sobre lo que hablamos. Desde el contexto siempre será claro, si se trata de un símbolo del alfabeto o si se trata de un meta-símbolo.
- Usamos subíndices para distinguir diferentes alfabetos.
- Usamos normalmente las minúsculas como alfabeto $\Sigma = \{a, \ldots, z\}$, en los ejemplos normalmente letras desde el principio del alfabeto.
- Cardinalidad del alfabeto (número de elementos del alfabeto): $|\Sigma| > 0$, $|\Sigma| < \infty$

5.2. Palabras

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

$$\begin{aligned}
\Sigma_1 &: 0, 1, 00, 01, 11, 000, 1001101 \\
\Sigma_2 &: a, aa, abb, ababa \\
\Sigma_3 &: \text{napa}, \text{palabra} \\
\Sigma_6 &: a, ab, aab, aaab, abab
\end{aligned}$$

Escribimos la **palabra vacía**, es decir, la palabra que no contiene ningún símbolo, como ε.

- Usamos normalmente letras minúsculas para anotar palabras, preferiblemente desde el final del alfabeto.
- El símbolo ε no pertenece a ningún alfabeto, $\varepsilon \notin \Sigma$

La **longitud** de una palabra sobre un alfabeto es el número de símbolos que contiene.

$$\begin{aligned}
\Sigma_1 &: \quad w = 0 \Longrightarrow |w| = 1, \quad w = 1001101 \Longrightarrow |w| = 7 \\
\Sigma_2 &: \quad w = a \Longrightarrow |w| = 1, \quad w = ababa \Longrightarrow |w| = 5 \\
\Sigma_3 &: \quad w = \text{napa} \Longrightarrow |w| = 2, \quad w = \text{palabra} \Longrightarrow |w| = 3 \\
\Sigma_6 &: \quad w = ab \Longrightarrow |w| = 2, \quad w = aab \Longrightarrow |w| = 1 \; o \; |w| = 2 \;\; ??
\end{aligned}$$

- Dependiendo del alfabeto puede resultar difícil dividir una palabra en sus símbolos.
- Si se puede dividir todas las palabras sobre un alfabeto solamente de una manera en sus símbolos, se llama tal alfabeto libre.
- Solemos usar solamente alfabetos libres.
- $|\varepsilon| = 0$

El conjunto de todas las palabras que se pueden formar sobre un alfabeto Σ más la palabra vacía se llama el **universo** del alfabeto $W(\Sigma)$.

- $W(\Sigma) = \{\varepsilon\} \cup \{w \mid w \text{ es palabra sobre } \Sigma\}$
- $\Sigma \subset W(\Sigma)$
- ε es palabra de cualquier universo, $\varepsilon \in W(\Sigma)$.
- La cardinalidad del universo es infinito (pero contable o enumerable, vemos más adelante lo que significa).
- Si el alfabeto es libre (o mejor decir, un generador libre), escribimos Σ^* por $W(\Sigma)$.

Podemos **concatenar** palabras, entonces sean w, v y u palabras en Σ^*.

- $w.v = wv$, es decir, usamos el . como símbolo de concatenación, pero muchas veces obviamos de él (igual como se suele hacer con el $\cdot$ de la multiplicación).
- $\varepsilon w = w = w\varepsilon$, es decir, ε se comporta como el elemento neutro (o elemento de intentidad) respecto a la concatenación.

- $|w.v| = |w| + |v|$
- $w.v \neq v.w$ para cualquier w y v, por ejemplo:

  $$w = abc \quad v = dec \qquad wv = abcdec \neq decabc = vw$$

  es decir, la concatenación no es conmutativa.
- $(w.v).u = w.(v.u)$ para cualquier palabras w, v y u, por ejemplo:

  $$w = abc \quad v = dec \quad u = fad$$
  $$(wv)u = (abcdec)fad = abcdecfad = abc(decfad) = w(vu)$$

  es decir, la concatenación es asociativa (usamos arriba las paréntesis como metasímbolos).
- Con dichas propiedades la estructura algebraica $(\Sigma^*,\ .\ )$ forma un monoide libre (es decir, un semigrupo con elemento de intentidad).

Si $xy = w$, llamamos x **prefijo** de w e y **sufijo** de w.

- Por $\varepsilon w = w$ y $w\varepsilon = w$, ε es por lo tanto prefijo y sufijo (trivial) de cualquier palabra, y w es prefijo y sufijo trivial de si mismo. (Normalmente no consideramos estos casos triviales.)
- Si x es prefijo de w entonces $|x| \leq |w|$.
- Si y es sufijo de w entonces $|y| \leq |w|$.
- Si x es prefijo de w, e y es sufijo de w y $x = y$, entonces $x = y = w$, ¿es verdad?

Si concatenamos siempre la misma palabra w, obtenemos **potencias** de w.

- $ww = w^2, \quad www = w^3, \quad \underbrace{w \ldots w}_{i\text{-veces}} = w^i, i \in \mathbb{N} = \{0, 1, 2, \cdots\}$
- $w^1 = w, \quad w^0 = \varepsilon$
- $|w^i| = i \cdot |w|$
- $|w^0| = |\varepsilon| = 0 = 0 \cdot |w| = |w^0|$
- $w^{m+n} = w^m.w^n$
- $|w^{m+n}| = (m+n) \cdot |w| = m \cdot |w| + n \cdot |w| = |w^m| + |w^n|$

La **reflexión** de una palabra w (o la palabra reversa) anotamos como w^R.

- $|w| = |w^R|$
- $\varepsilon = \varepsilon^R$

5.3. Lenguajes

Un **lenguaje** es cualquier subconjunto del universo sobre algún alfabeto, es decir, $L \subset W(\Sigma)$, o también $L \subset \Sigma^*$.

Ejemplo:

- Lenguajes triviales
  - $L = \emptyset$ es el lenguaje vacio (que no contiene ninguna palabra), $|L| = 0$
  - $L = \{\varepsilon\}$ es el lenguaje que solamente contiene la palabra vacio, $|L| = 1$

  son independientes del alfabeto y por eso son lenguajes sobre cualquier alfabeto.
- sea $\Sigma = \{a, b\}$
  - $L_1 = \{\varepsilon, a, b\}$
  - $L_{ab} = \{a^n b^n \mid n \in \mathbb{N}\}$ es decir, el lenguaje que contiene todas las palabras con un número de as seguidos por el mismo número de bs.
  - $L_{pal} = \{ww^R \mid w \in \Sigma^*\}$ es decir, palíndromos
  - $L_{quad} = \{a^{n^2} \mid n \in \mathbb{N}_{>0}\}$

Si $|L| < \infty$ para un lenguaje $L \subset \Sigma^*$, entonces se llama L **lenguaje finito**.

Operaciones sobre/con lenguajes, sean $L, L_1, L_2, L_3 \subset \Sigma^*$ lenguajes (igual para $W(\Sigma)$):

Unión:

$$L_1 \cup L_2 = \{w \mid w \in L_1 \text{ o } w \in L_2\}$$

Propiedades (unos ejemplos):

Conmutatividad:	$L_1 \cup L_2 = L_2 \cup L_1$
Asociatividad:	$(L_1 \cup L_2) \cup L_3 = L_1 \cup (L_2 \cup L_3)$
Idempotencia:	$L \cup L = L$
Operación con $\emptyset$:	$L \cup \emptyset = L = \emptyset \cup L$
Operación con Σ^*:	$L \cup \Sigma^* = \Sigma^* = \Sigma^* \cup L$

Intersección:

$$L_1 \cap L_2 = \{w \mid w \in L_1 \text{ y } w \in L_2\}$$

Propiedades (unos ejemplos):

Conmutatividad:	$L_1 \cap L_2 = L_2 \cap L_1$
Asociatividad:	$(L_1 \cap L_2) \cap L_3 = L_1 \cap (L_2 \cap L_3)$
Idempotencia:	$L \cap L = L$
Operación con $\emptyset$:	$L \cap \emptyset = \emptyset = \emptyset \cap L_1$
Operación con Σ^*:	$L \cap \Sigma^* = L = \Sigma^* \cap L$

Complemento:

$$\overline{L} = \{w \mid w \in \Sigma^* \text{ y } w \notin L\}$$

Propiedades (unos ejemplos):

Reglas de DeMorgan: $\overline{L_1 \cup L_2} = \overline{L}_1 \cap \overline{L}_2$
$\overline{L_1 \cap L_2} = \overline{L}_1 \cup \overline{L}_2$

Con estas tres operaciones la estructura $(\Sigma^*, \cup, \cap, \_)$ forma un álgebra booleana.

Diferencia:

$$L_1 - L_2 = \{w \mid w \in L_1 \text{ pero } w \notin L_2\}$$

Propiedades (unos ejemplos):

$\overline{L}_1 = \Sigma^* - L_1$
$L_1 - L_2 = L_1 \cap \overline{\Sigma^* \cap L_2}$

Concatenación:

$$L_1.L_2 = \{w \mid w = w_1.w_2 \text{ y } w_1 \in L_1 \text{ y } w_2 \in L_2\}$$

Propiedades (unos ejemplos):

No-Conmutatividad: $L_1.L_2 \neq L_2.L_1$ (en general)
Operación con $\emptyset$: $L_1.\emptyset = \emptyset = \emptyset.L_1$
Operación con $\{\varepsilon\}$: $L_1.\{\varepsilon\} = L_1 = \{\varepsilon\}.L_1$

Potencia:

$$L^i = \underbrace{L \ldots L}_{i\text{-veces}} \quad i \in \mathbb{N}$$

Propiedades (unos ejemplos):

Cero-Potencia: $L^0 = \{\varepsilon\}$
Inducción: $L^i.L = L^{i+1} = L.L^i$

Clausura positiva:

$$L^+ = \bigcup_{i=1}^{\infty} L^i = L^1 \cup L^2 \cup L^3 \cup \ldots$$

Clausura (de Kleene):

$$L^* = \bigcup_{i=0}^{\infty} L^i = L^0 \cup L^1 \cup L^2 \cup \ldots$$

Propiedades (unos ejemplos):

Reflexión (o inverso):

$$L = \{w \mid w^R \in L\}$$

Homomorfismo: Sean Σ, Γ dos alfabetos. Sea $\varphi : \Sigma \longrightarrow \Gamma^*$ una función que asigna a cada símbolo de Σ una palabra sobre Γ. Podemos ampliar la función φ a un homomorfismo $\varphi : \Sigma^* \longrightarrow \Gamma^*$, es decir, una función que asigna a cada palabra sobre Σ una palabra sobre Γ, con

$$\begin{aligned} \varphi(\varepsilon) &= \varepsilon \\ \varphi(w\sigma) &= \varphi(w)\varphi(\sigma) \end{aligned}$$

Ejemplo:

$$\begin{aligned} \Sigma &= \{a, b, c, d\} \\ \Gamma &= \{0, 1\} \\ & \varphi(a) = 00 \quad \varphi(b) = 1 \quad \varphi(c) = \varepsilon \quad \varphi(d) = 0110 \\ \varphi(abcd) &= 0010110 \end{aligned}$$

Entonces si $L \subset \Sigma^*$ es un lenguaje sobre Σ

$$\varphi(L) = \{\varphi(w) \mid w \in L\} \subset \Gamma^*$$

es un lenguaje sobre Γ y si $L \subset \Gamma^*$ es un lenguaje sobre Γ, entonces

$$\varphi^{-1}(L) = \{w \mid \varphi^{-1}(w) \in L\} \subset \Sigma^*$$

es un lenguaje sobre Σ.

¿Cuál es el orden de prioridad de estos operadores?

5.4. Producciones y Derivaciones

Definimos algunas notaciones para describir reglas de sustitución, es decir, como derivar una palabra con las producciones de la gramática:

Una **producción** p es una dupla (pareja) de un conjunto cartesiano sobre dos universos (que pueden ser el mismo), es decir, $p = (A, B) \in {\Sigma^*}_1 \times {\Sigma^*}_2$.

Sea (A, B) una producción, en vez de duplas también escribimos: $A \longrightarrow B$.

Un conjunto de producciones se llama **sistema de producciones** (o sistema de reglas). A este nivel todavía no decimos mucho sobre los alfabetos involucrados, más adelante concretaremos.

Una **derivación directa** $v \longrightarrow w$ es una conversión de una palabra en otra aplicando una producción, es decir, sea por ejemplo $v = aAb$ una palabra, y sea $A \longrightarrow B$ una producción, entonces se puede derivar la palabra $w = aBb$ directamente desde v sustituyendo la subpalabra A por la palabra B como indica la producción.

Ejemplo: Sean $000 \longrightarrow 010$ y $10 \longrightarrow 01$ dos producciones. Desde $v = 1000$ se puede derivar $w_1 = 1010$ aplicando la primera producción, y $w_2 = 0100$ aplicando la segunda.

Una **derivación** $v \longrightarrow^* w$ es una secuencia de derivaciones directa aplicando sucesivamente producciones de un sistema. La **longitud de una derivación** es el número de producciones aplicadas.

Ejemplo: Sean $000 \longrightarrow 010$ y $10 \longrightarrow 01$ dos producciones. Desde $v = 1000$ se puede derivar $w_1 = 0011$, es decir, $v \longrightarrow^* w_1$ aplicando $v = 1000 \longrightarrow 1010 \longrightarrow 0110 \longrightarrow 0101 \longrightarrow 0011 = w_1$, o también $w_2 = 0001$ aplicando $v = 1000 \longrightarrow 0100 \longrightarrow 0010 \longrightarrow 0001 = w_2$. En el primer caso la longitud de la derivación es 4, en el segundo caso 3.

Comentario importante: muchas de las comprobaciones en el ámbito de la teoría de los lenguajes formales se realiza mediante inducción sobre: longitud de la palabra, longitud de la derivación, (o luego también longitud del cálculo).

Dado un sistema de producciones, si sustituimos siempre la primera posibilidad a la izquierda de la palabra de partida, se llama una **derivación más a la izquierda**, e igual, si sustituimos siempre la primera posibilidad a la derecha de la palabra de partida, se llama una **derivación más a la derecha**.

5.5. Relaciones de equivalencia

Un conjunto $R \subset \Sigma^* \times \Sigma^*$ es una **relación (binaria sobre Σ^*)**.

Escribimos los pares siendo elementos de R como (x, y), o como $x \longrightarrow y$, o como xRy. Esta última notación es aquella a la cual estamos acostumbrados, por ejemplo, con relaciones como $>, <, \geq, \leq$ etc. en situaciones como $5 > 3$, o $x \leq y$.

Sean R y S dos relaciones. Definimos

$$\begin{aligned} R^0 &= \{(x,x) \mid x \in \Sigma^*\} \\ RS &= \{(x,y) \mid \exists z \in \Sigma^* : xRz \text{ y } zSy\} \\ R^{n+1} &= RR^n \end{aligned}$$

es decir, R^0 es la relación de identidad, y la operación nos permite crear nuevas relaciones a partir de dos relaciones dadas, y R^{n+1} es una relación construida de tal manera recursivamente. Con eso definimos:

$$R^* = \bigcup_{n \geq 0} R^n$$

$$R^+ \;=\; \bigcup_{n\geq 1} R^n$$

es decir, xR^*y (o en otra notación $x \longrightarrow^* y$, o con palabras: la pareja (x, y) está en la clausura transitiva de la relación R) si $x = y$ o si existe una secuencia $z_1, z_2, \ldots, z_n$ con $n \geq 1$ y $xRz_1, z_1Rz_2, \ldots, z_nRy$.

Una relación R es

- **reflexiva**, si $\forall x : xRx$, es decir, la relación de identidad R^0 es subrelación de R,
- **transitiva**, si $xRy, yRz \Longrightarrow xRz$, es decir, si los pares (x, y) y (y, z) son elementos de R entonces (x, z) también lo es,
- **simétrica**, si $\forall x, y : xRy \Longleftrightarrow yRx$, es decir, con (x, y) también (y, x) es elemento de la relación.

Observamos que para R

- R^* es una relación reflexiva y transitiva, llamada la clausura reflexiva y transitiva de R (porque es la relación más pequeña con tal propiedad).
- R^+ es una relación transitiva, llamada la clausura transitiva de R (porque es la relación más pequeña con tal propiedad).
- R^+ es también reflexiva si R ya lo es.
- R^* y R^+ son simétricas si R ya lo es.

Una relacion R es una **relación de equivalencia** si R es reflexiva, simétrica, y transitiva.

Sea R una relación de equivalencia sobre Σ^*. A cada elemento de Σ^* podemos asignar el conjunto de los elementos que son equivalentes a él. Basta con anotar un representente de dicho conjunto y escribimos

$$[x]_R = \{y \mid yRx\} = \{y \mid xRy\}$$

(si desde el contexto ya conocemos R, obviamos del subíndice R).

Si xRy entonces $[x] = [y]$ porque ambos caen en la misma clase de equivalencia. Se suele usar como representante una de las palabras más cortas de la clase.

Si $x, y \in [z]$ escribimos también $x \equiv y$ que significa que xRy e yRx.

Una relación de equivalencia divide Σ^* en clases, es decir,

$$\Sigma^* = [x_1] \cup [x_2] \cup \ldots \cup [x_k] \cup \ldots$$

cuyo número es finito o infinito. La intersección de dos clases es vacía, es decir, $[x_i] \cap [x_j] = \emptyset$ si $i \neq j$ porque si tuviesen un elemento en común, ambas clases serían iguales.

Ejemplo: Sea $\Sigma = \{\sigma_1, \ldots, \sigma_k\}$ un alfabeto (por ejemplo el alfabeto de toda la vida).

La relación

$$R = \{(x, y) \mid x \text{ comienza con el mismo símbolo que } y\}$$

es una relación de equivalencia y nos divide Σ^* en

$$\Sigma^* = [\sigma_1] \cup [\sigma_2] \cup \ldots \cup [\sigma_k] \cup [\epsilon]$$

es decir, en todas las clases de palabras que empiezan con la misma letra más la clase para la palabra vacía (que no empieza con ninguna letra).

Entonces hay tantas clases como símbolos en Σ más una clase.

Llamamos el número de clases que produce una relación de equivalencia el **índice de la relación** $\mathrm{Indice}(R)$.

En el ejemplo tenemos $\mathrm{Indice}(R) = k + 1 = |\Sigma| + 1$, es decir, un índice finito.

5.6. Relación de equivalencia de lenguajes

Para cada lenguaje $L \subset \Sigma^*$ podemos construir una relación de equivalencia sobre Σ^*:

$$xR_Ly \iff (\forall z \in \Sigma^* : xz \in L \iff yz \in L)$$

es decir, x es equivalente a y, si, añadiendo cualquier sufijo, ambas palabras resultantes o bien están en L o bien no están en L.

Observa: $z = \epsilon : x \in L \iff y \in L$, es decir, o bien todas las palabras de una clase están en L o bien ninguna palabra de una clase está en L.

Ejercicio:¡Verifica que R_L es una relación de equivalencia!

6. Gramáticas generativas

Una **gramática** es una cuádrupla

$$G = (\Sigma_N, \Sigma_T, P, \$)$$

donde

- Σ_N es un alfabeto de símbolos no-terminales.
- Σ_T es un alfabeto de símbolos terminales.
- Se exige $\Sigma_N \cap \Sigma_T = \emptyset$ y se suele usar $\Sigma = \Sigma_N \cup \Sigma_T$.
- P es un sistema de producciones finitos, donde se distingue varios casos, ejemplos son:
  - $P \subset (\Sigma_N \cup \Sigma_T)^* \times (\Sigma_N \cup \Sigma_T)^*$
    caso muy general, (así no haría falta distinguir los dos alfabetos a la primera vista, es decir, $P \subset \Sigma^* \times \Sigma^*$)
  - $P \subset \Sigma^*.\Sigma_N^+.\Sigma^* \times \Sigma^*$
    es decir, a la derecha existe por lo menos un símbolo no-terminal
  - $P \subset \Sigma_N \times \Sigma^*$
    es decir, se sustitue solamente símbolos (palabras) no-terminales
  - $P \subset \Sigma_N \times (\Sigma_N^* \cup \Sigma_T^*)$
    es decir, se sustitue solamente símbolos (palabras) no-terminales, pero por símbolos (palabras) o bien terminales o bien no-terminales
  - Repetimos: se exige que $|P| < \infty$, es decir, el conjunto de reglas es finito.

  ¡Más adelante vemos en detalle qué tipos de sistemas de producciones se suele usar!
- $\$$ es el símbolo inicial (o de partida, o de comienzo, o axioma) que pertenece al alfabeto no-terminal, es decir, $\$ \in \Sigma_N$.

El **lenguaje generado** por una gramática es

$$L(G) = \{w \mid w \in \Sigma_T^* \text{ y } \$ \longrightarrow^* w\}$$

es decir, se puede derivar la palabra $w \in \Sigma_T$ desde el símbolo inicial aplicando las reglas del sistema de producciones. Dichas palabras derivables que consisten solamente de símbolos terminales se llaman **sentencias**.

6.1. Ejemplos

¿Es posible derivar lenguajes infinitos con sistemas de producciones finito?

Sí, por ejemplo, es posible generar el lenguaje $L(G) = \Sigma_T^*$ con un sistema de producciones finitos:

$$G = (\{\$\}, \{a, b\}, \{\$ \longrightarrow \epsilon, \$ \longrightarrow a\$, \$ \longrightarrow b\$\}, \$)$$

$$\begin{aligned} L_1 &= \{\epsilon, a, b\} \\ G_1 &= (\{\$\}, \{a, b\}, \{\$ \longrightarrow \epsilon, \$ \longrightarrow a, \$ \longrightarrow b\}, \$) \end{aligned}$$

- obviamente $L(G_1) = L_1$
- para lenguajes finitos es fácil generar una gramática, basta con derivar directamente cada palabra desde el símbolo inicial (aunque se puede usar un sistema de producciones más sofisticado)

Una **gramática recursiva** sobre la palabra $v \in \Sigma^*$ es una gramática donde se puede derivar desde v una palabra que contiene v de nuevo, es decir, existe la posibilidad de una derivación: $v \longrightarrow^* uvw$ (con $|v| < |uvw|$).

El lenguaje generado por una gramática es infinito, si la gramática es recursiva sobre una palabra v y que a su vez es derivable desde el símbolo inicial.

$$\begin{aligned} L_{ab} &= \{a^n b^n \mid n \in \mathbb{N}\} \\ G_{ab} &= (\{a, b\}, \{\$\}, \{\$ \longrightarrow a\$b, \$ \longrightarrow \epsilon\}, \$) \end{aligned}$$

$$\begin{aligned} L_{abc} &= \{a^n b^n c^n \mid n \in \mathbb{N}\} \\ &\quad \epsilon, abc, aabbcc, aaabbbccc, \ldots \in L_{abc} \\ G_{abc} &= (\{\$, \ldots\}, \{a, b, c\}, P, \$) \end{aligned}$$

- ¿Cuáles son las producciones necesarias?
- Una vez sabiendo eso, podemos completar el alfabeto no-terminal Σ_N

- Una primera idea:

$$\begin{aligned} P &= \{\$ \longrightarrow \epsilon, \$ \longrightarrow ABC, A \longrightarrow \epsilon, A \longrightarrow aA, \\ &\quad\ B \longrightarrow \epsilon, B \longrightarrow bB, C \longrightarrow \epsilon, C \longrightarrow cC\} \\ \Sigma_N &= \{\$, A, B, C\} \end{aligned}$$

  Obviamente podemos derivar cualquier elemento de L_{abc} con esa gramática, por ejemplo:

$$\begin{aligned} \$ &\longrightarrow ABC \longrightarrow aABC \longrightarrow aaABC \longrightarrow aaBC \longrightarrow aabBC \longrightarrow aabbBC \\ &\longrightarrow aabbC \longrightarrow aabbcC \longrightarrow aabbccC \longrightarrow aabbcc \end{aligned}$$

  Pero también podemos derivar palabras como $aaabcccc$, es decir, el lenguaje es

$$L(G_{\text{Test}}) = \{a^i b^j c^k \mid i, j, k \in \mathbb{N}\} \supset L_{abc}$$

  Parece que la gramática G_{Test} es demasiado amplia. De alguna manera deberíamos construir un sistema de producciones que permite mantener un número igual de letras a, b y c (o en otras palabras, necesitamos "contar")...

- **Idea 1**: Si somos capaz de derivar desde $a^k X b^k c^k$ la secuencia $a^{k+1} X b^{k+1} c^{k+1}$, hemos ganado.

  Idea 2: Tenemos que pasar la "información" que hemos añadido por ejemplo un ab en un lado hacia el otro lado donde tenemos que añadir entonces una c (o en un lado la a y en el otro lado un bc).

  El truco consiste en usar unos símbolos no-terminales cuales se van a sustituir dependiendo del contexto en el cual se encuentran.

  Entonces, construimos P y Σ_N:

$$\begin{aligned} P &= \{ \\ &\quad \$ \longrightarrow \epsilon, && \text{para obtener la palabra vacía} \\ &\quad \$ \longrightarrow aXbc, && \text{para iniciar la construcción} \\ &\quad Xb \longrightarrow bY, && \text{para empezar "ir" hacia las } c\text{'s} \\ &\quad Yb \longrightarrow bY, && \text{para "ir" hacia las } c\text{'s} \\ &\quad Yc \longrightarrow Zcc, && \text{para añadir una } c \text{ y empezar "volver"} \\ &\quad bZ \longrightarrow Zb, && \text{para "volver" hacia las } a\text{'s} \\ &\quad aZ \longrightarrow aaXb, && \text{para añadir una } a \text{ y una } b \\ &\quad X \longrightarrow \epsilon && \text{para terminar} \\ &\} \end{aligned}$$

$$\Sigma_N = \{\$, X, Y, Z\}$$

- Se puede comprobar formalmente con inducción sobre k que la gramática dada genera exactamente el lenguaje deseado, es decir $L(G_{abc}) = L_{abc}$.

  La comprobación sigue la construcción y se observa que no hay ambigüedad en el momento de elegir una producción.

- Existe también una gramática que usa un símbolo no-terminal menos y también una producción menos:

$$
\begin{array}{rcl}
P & = & \{ \\
 & & \begin{array}{lcll}
\$ & \longrightarrow & \epsilon, & \text{para obtener la palabra vacía} \\
\$ & \longrightarrow & aXbc, & \text{para iniciar la construcción} \\
Xb & \longrightarrow & bX, & \text{para “ir” hacia las } c\text{'s} \\
Xc & \longrightarrow & Ybcc, & \text{para añadir una } b \text{ y una } c \\
bY & \longrightarrow & Yb, & \text{para “volver” hacia las } a\text{'s} \\
aY & \longrightarrow & aaX, & \text{para añadir una } a \\
aY & \longrightarrow & aa & \text{para terminar}
\end{array} \\
 & & \} \\
\Sigma_N & = & \{\$, X, Y\}
\end{array}
$$

  Se observa:

  - tenemos ambigüedad en elegir producciones para sustituir y dónde aplicarlas
  - aquí hemos decidido añadir a la derecha una b y una c
  - generalmente se nota que hay muchas gramáticas que generan el mismo lenguaje

6.2. Abreviación de Backus

Para abreviar la notación de las producciones usamos la **forma normal de Backus (BNF)**. Agrupamos las producciones cuyas partes izquierdas coincidan, escribiendo las partes derechas separadas por $|$, por ejemplo:

$$
\begin{array}{rcl}
P & = & \{ \\
 & & \begin{array}{lcl}
\$ & \longrightarrow & \epsilon \mid aXbc, \\
Xb & \longrightarrow & bX, \\
Xc & \longrightarrow & Ybcc, \\
bY & \longrightarrow & Yb, \\
aY & \longrightarrow & aaX \mid aa,
\end{array} \\
 & & \}
\end{array}
$$

Definimos una gramática que genere lo que se usa en programas, por ejemplo:

$$((a+b)*(c+d))*(e+f)$$

$$L_{expr} \quad = \quad \{w \mid w \text{ es expresión algebraica}\}$$

donde nos limitamos a variables que consisten de una sola letra. Entonces

$$\begin{aligned} \Sigma_T &= \{(,),+,*,a,\ldots,z\} \\ P &= \$ \longrightarrow E, E \longrightarrow E*E \mid (E*E) \mid (E+E) \mid a \mid \ \ldots \ \mid z \\ G_{expr} &= (\{\$, E\}, \Sigma_T, P, \$) \end{aligned}$$

- se puede ampliar la gramática que incluye también $-$ y $/$
- se puede ampliar la gramática que genere también expresiones con variables de más de una letra, por ejemplo: $ancho * altura$
- más tarde veremos como se define las expresiones de tal estilo un poco más completo

6.3. Árbol de derivación

Para las gramáticas podemos visualizar la aplicación de las producciones que derivan desde el símbolo inicial una palabra como un árbol, el **árbol de derivación**:

arbol

El lugar con el símbolo inicial se llama **raíz** del árbol (aunque se suele dibujarlo arriba de todo).

Como se ve, cada símbolo es la raíz de un subárbol.

La palabra que se puede leer desde la izquierda hacia la derecha en las hojas del árbol y solamente consiste de símbolos terminales será una **sentencia**.

6.4. Jerarquia de Chomsky

Según Chomsky se clasifica las gramáticas en cuatro tipos (cuales son, como vemos más adelante, entre si verdaderamente diferentes).

Entonces sea $G = (\Sigma_N, \Sigma_T, P, \$)$ una gramática (y $\Sigma = \Sigma_N \cup \Sigma_T$). Las gramáticas se destinguen solamente en el sistema de producciones que siempre será un conjunto finito y que se clasifica en los siguientes tipos:

Tipo 0: gramáticas generales sin restricciones

$$P \subset \Sigma^*.\Sigma_N.\Sigma^* \times \Sigma^*$$

es decir, se sustituye por lo menos un símbolo no-terminal.

Tipo 1: gramáticas sensibles al contexto

$$P \subset \{xAy \longrightarrow xvy \mid x, y \in \Sigma^*, A \in \Sigma_N, v \in \Sigma^+\} \cup \{\$ \longrightarrow \epsilon\}$$

es decir, se sustituye un símbolo no-terminal por algo manteniendo el contexto; entonces una derivación siempre produce palabras más largas o igual de larga ($u \longrightarrow^* v \Longrightarrow |u| \leq |v|$)

Tipo 2: gramáticas libres de contexto

$$P \subset \Sigma_N \times \Sigma^+ \cup \{\$ \longrightarrow \epsilon\}$$

es decir, se sustituye solo símbolos no-terminales por palabras no vacías

Tipo 3: gramáticas regulares (o lineales)

$$P \subset \Sigma_N \times (\Sigma_N.\Sigma_T \cup \Sigma_T) \cup \{\$ \longrightarrow \epsilon\}$$

es decir, lineales a la izquierda (porque los símbolos no-terminales aparecen en una derivación siempre a la izquierda de la palabra)

$$P \subset \Sigma_N \times (\Sigma_T.\Sigma_N \cup \Sigma_T) \cup \{\$ \longrightarrow \epsilon\}$$

es decir, lineales a la derecha (porque los símbolos no-terminales aparecen en una derivación siempre a la derecha de la palabra)

- Se ha introducido explícitamente la regla $\$ \longrightarrow \epsilon$ en las gramáticas de tipos 1, 2, y 3 para permitir que el lenguaje $\{\epsilon\}$ puede ser generado dado que las reglás solo permiten un crecimiento de la longitud de las palabras a lo largo de las derivaciones.
- Retomamos la clasificación de las gramáticas hacia final del curso (por ejemplo, respondemos a la pregunta si son de verdad clases separadas).

Observación: si permitimos para las gramáticas de libre contexto reglas del tipo $\Sigma_N \longrightarrow \Sigma^*$, es decir, permitimos reglas como $A \longrightarrow \epsilon$, podemos sustituir todas las reglas que tengan una A a la derecha, por ejemplo $B \longrightarrow xAy$ por $B \longrightarrow xy$, y conseguir así una eliminación de las producciones compresoras.

6.5. Equivalencia y ambigüedad

Dos **gramáticas son equivalentes** si generan el mismo lenguaje, es decir, $G_1 \equiv G_2$ si $L(G_1) = L(G_2)$.

(Adelanto: averiguar en general si dos gramáticas son equivalentes es un problema no computable.)

Sea $G = (\{\$, A\}, \{1\}, \{\$ \longrightarrow 1A, \$ \longrightarrow 11, A \longrightarrow 1\}, \$)$ una gramática. Tanto $\$ \longrightarrow 11$ como $\$ \longrightarrow 1A \longrightarrow 11$ es una derivación para la palabra 11.

Una **sentencia es ambigua** si existen más que una derivación para ella en una gramática.

Una **gramática es ambigua** si su lenguaje contiene una sentencia ambigua, es decir, se puede derivar la misma sentencia con dos (o más) derivaciones distintas.

Un **lenguaje es ambiguo** (o incluso se dice inherentemente ambiguo) si todas las gramáticas que generan el lenguaje son ambiguas.

Ejemplo: $G = (\{\$\}, \{1\}, \{\$ \longrightarrow 11\}, \$)$ no es ambigua, entonces $L(G)$ no es ambiguo.

Si una sentencia es ambigua (en el caso de las gramáticas libres de contexto) tenemos dos árboles de derivación para la misma sentencia.

Ejemplo:

$$E \longrightarrow E + E \mid E * E \mid (E) \mid a \mid \ \ldots \ \mid z, \$ \longrightarrow E$$

ambitree

La ambigüedad introduce cierto grado de no-determinismo para derivar palabras, por eso, en la práctica se intenta evitar gramáticas ambiguas.

(Pero: el problema de decisión, si existe para una gramática ambigua una gramática equivalente no-ambigua es un problema no-computable.)

Investigamos de nuevo las expresiones aritméticas:

- sabemos que tanto la adición como la multiplicación son asociativas, entonces podemos acordar generar siempre con derivaciones más a la izquierda
- sabemos que hay prioridades (acordadas) entre las operaciones: $()$ antes que $*$ antes que $+$, entonces podemos acordar generar primero las operaciones con *menos* prioridad
- podemos introducir varables adicionales que nos garantizan una derivación única

Usamos E para expresiones (va a ser también el símbolo inicial), T para termios (con prioridad asociado a $+$), F para factores (con prioridad asociado a $*$, y V para variables (que ya no tendrán operaciones):

$$
\begin{aligned}
E &\longrightarrow E + T \mid T \\
T &\longrightarrow T * F \mid F \\
F &\longrightarrow (E) \mid V \\
V &\longrightarrow a \mid b \mid \ \ldots \ \mid z
\end{aligned}
$$

La gramática con este sistema de producciones no es ambigua.

exprnoamb

7. Autómatas finitos

Describimos autómatas finitos con unas definiciones matemáticas. Nos limitamos al principio a autómatas solamente con entrada.

7.1. Autómatas finitos deterministas (AFD)

Un **autómata finito determinista** (AFD) es una quíntupla

$$M = (\Sigma, Q, \delta, q_0, F)$$

donde

- Σ es un alfabeto (sabemos $\varepsilon \notin \Sigma$)
- Q es un conjunto finito no vacío de estados, es decir, $0 < |Q| < \infty$.
- δ es una *función* de transición:

  $$\delta : Q \times \Sigma \longrightarrow Q \; ; \; \delta(q, \sigma) = p$$

  es decir, si el autómata se encuentra en el estado q y 'lee' el símbolo σ va al estado p.
- $q_0 \in Q$ es el estado inicial.
- $F \subset Q$ es el conjunto de estados finales.

Podemos pensar de un autómata como un dispositivo que lee desde una cinta con símbolos y que realiza cambios de estados internamente:

auto

Dibujamos los autómatas como grafos dirigidos (no introducimos el concepto matemático de grafos formalmente), los estados representan los nodos del grafo, y dibujamos una arista atribuida con un símbolo entre dos nodos si existe una transisión correspondiente:

compauto

es decir, el estado inicial está marcado por una flecha y los estados finales están marcados con doble círculo.

Ejemplo: Un AFD que 'acepta' las cadena de 0s y 1s donde los números de ceros y unos es par:

zeroonepar

entonces

$$M = (\{0,1\}, \{q_0, q_1, q_2, q_3\}, \delta, q_0, \{q_0\})$$

¿Cómo describimos cómodamente δ?

Observamos: $|Q| < \infty$ y $|\Sigma| < \infty$, entonces podemos hacer una tabla con los estados como filas y con los símbolos como columnas:

$$\delta(q_0, 0) = q_3, \delta(q_0, 1) = q_1, \delta(q_1, 0) = \ldots$$

o más breve una tabla:

δ	0	1
$\Longrightarrow \star q_0$	q_3	q_1
q_1	q_2	q_0
q_2	q_1	q_3
q_3	q_0	q_2

- 'Determinista' significa que no tenemos opción ninguna para eligir, δ es una **función**.
- Si δ es una **función total** llamamos el autómata **completo**, es decir, existe para cada estado y cada símbolo una transición.
- Abreviamos los dibujos para reducir el número de aristas:

  aristas

  es decir, permitimos escribir más de un símbolo por arista, pero el cambio de estado se realiza con leer solo uno de la lista.

Para definir el lenguaje aceptado por un AFD ampliamos la función δ a una función δ^* para que trabaja sobre palabras:

$$\begin{aligned} \delta^* : Q \times \Sigma^* &\longrightarrow Q \\ \delta^*(q, \varepsilon) &= q \\ \delta^*(q, \sigma w) &= \delta^*(\delta(q, \sigma), w) \quad \sigma \in \Sigma, w \in \Sigma^* \end{aligned}$$

es decir, δ^* refleja el movimiento de la cabeza de lectura del autómata, o en otras palabras, δ^* marca el camino que se está yiendo en el autómata para aceptar la palabra (hilo rojo).

Un autómata finito determinista $M = (\Sigma, Q, \delta, q_0, F)$ **acepta** una palabra $w \in \Sigma^*$ si $\delta^*(q_0, w) \in F$ donde δ^* es la ampliación de la funcion de transición δ.

O en otras palabras, M acepta w, si $\delta^*(q_0, w)$ es un estado final del autómata.

El **lenguaje aceptado por un autómata finito determinista** M es el conjunto de palabras aceptadas por M:

$$L(M) = \{w \mid w \in \Sigma^*, M \text{ acepta } w\}$$

En el grafo podemos observar: si $w \in L(M)$ entonces existe un camino en el grafo desde el estado inicial q_0 hasta algún estado final de tal manera que podemos 'leer' la palabra w a lo largo de las aristas visitadas.

Ejemplo: Un autómata que acepta números reales (en Pascal):

afdreal

Curiosidades de C/C++:

- Comprueba con un compilador de C/C++ (o de Java) si `a=000;` o `a=0011.0;` son sentencias correctas, sino no lo son, modifica el autómata adecuadamente (¿Qué pasa con `a=009` o `a=007`?).
- Comprueba con un compilador de C/C++ (o de Java) si `a=3E000;` es una sentencia correcta, sino no lo es, modifica el autómata adecuadamente.
- `a=.1+ +1.;` es una sentencia correcta en C/C++ (se asigna a `a` el valor `1.1` siendo la suma de dos constantes flotantes), pero importante es el espacio entre los dos +

Vemos que estámos confrontados con diferentes problemas:

- deberíamos saber antemano: ¿Qué es una constante flotante?
- deberíamos traducir dicho conocimiento en un autómata
- deberíamos comprobar si dicho autómata de verdad acepta lo que debe aceptar
- si implementásemos tal autómata de forma real, deberíamos comprobar adicionalmente si la implementación refleja la descripción matemática

Observamos, cada AFD se puede completar:

- añadimos un estado e a Q (pero $e \notin F$)
- añadimos las transiciones que faltan, es decir, $\delta(q, \sigma) = e$ para todos los $q \in Q$ (incluyendo e) y $\sigma \in \Sigma$
- con eso δ se convierte en una función total

Observamos:

- si $q_0 \in F$ entonces $\varepsilon \in L(M)$ y al revés, si $\varepsilon \in L(M)$ entonces $q_0 \in F$.
- puede ocurrir que hay estados no acesibles desde q_0, incluso pueden ser aislados, es decir, no existe un camino desde q_0 hacia tal estado.

7.2. Autómatas finitos no-deterministas (AFND)

Ampliamos un poco las posibilidades de las transiciones de un autómata finito, es decir, cambiamos la función δ.

Un **autómata finito no-determinista** (AFND) es una quíntupla

$$M = (\Sigma, Q, \delta, q_0, F)$$

donde

- Σ es un alfabeto.
- Q es un conjunto finito no vacío de estados, es decir, $0 < |Q| < \infty$.
- δ es (una de las dos definiciones, que entre si son equivalentes)
  - una relación, es decir $\delta \subset (Q \times \Sigma) \times Q$
  - o una función, es decir, $\delta \ : \ Q \times \Sigma \longrightarrow \mathcal{P}(Q)$ siendo $\mathcal{P}(Q)$ el conjunto de las partes de Q
- $q_0 \in Q$ es el estado inicial.
- $F \subset Q$ es el conjunto de estados finales.

Ejemplo: un AFND para el lenguaje

$$L_{dos} = \{w \mid w \in \{0,1\}^*, w \text{ contiene dos 0s ó dos 1s}\}$$

afnd

Representamos la función δ también con una tabla, solo que ahora aparece más de un estado en cada celda de la tabla, por eso usamos la notación de conjuntos:

δ	0	1
$\Longrightarrow q_0$	$\{q_0, q_3\}$	$\{q_0, q_1\}$
q_1	$\emptyset$	$\{q_2\}$
$*q_2$	$\{q_2\}$	$\{q_2\}$
q_3	$\{q_4\}$	$\emptyset$
$*q_4$	$\{q_4\}$	$\{q_4\}$

Ampliamos de nuevo δ para definir el lenguaje aceptado por un AFND

$$\begin{aligned}
\delta^* : Q \times \Sigma^* &\longrightarrow \mathcal{P}(Q) \\
\delta^*(q, \varepsilon) &= \{q\} \\
\delta^*(q, \sigma w) &= \{p \mid p \in Q, \exists r \in \delta(q, \sigma) : p \in \delta^*(r, w)\} \\
&\quad \sigma \in \Sigma, w \in \Sigma^* \\
&= \bigcup_{r \in \delta(q,\sigma)} \delta^*(r, w)
\end{aligned}$$

es decir, δ^* coincide con δ para símbolos del alfabeto y en general enumera los estados alcanzables con la palabra.

Un autómata finito no-determinista $M = (\Sigma, Q, \delta, q_0, F)$ **acepta** una palabra $w \in \Sigma^*$ si $\delta^*(q_0, w) \cap F \neq \emptyset$ donde δ^* es la ampliación de la relación de transición δ.

O en otras palabras, M acepta w, si $\delta^*(q_0, w)$ contiene un estado final del autómata.

El **lenguaje aceptado por un autómata finito no-determinista** M es el conjunto de palabras aceptadas por M:

$$L(M) = \{w \mid w \in \Sigma^*, M \text{ acepta } w\}$$

7.3. Equivalencia entre AFD y AFND

Dos **autómatas** M_1 **y** M_2 **son equivalentes** si aceptan el mismo lenguaje, $M_1 \equiv M_2$ si $L(M_1) = L(M_2)$.

- Si eliminamos todos los estados no acesibles (o aislados) de un autómata, obtenemos un autómata equivalente al autómata original.
- Obviamente tal autómata se representa con un grafo conexo.

Dos **estados** q_1 **y** q_2 **de dos autómatas** M_1 **y** M_2 **son equivalentes**, es decir, $q_1 \equiv q_2$, si para $q_1 \in Q_1$ y $q_2 \in Q_2$ $\delta^*(q_1, w) \in F_1 \Leftrightarrow \delta^*(q_2, w) \in F_2$.

Entonces dos autómatas son equivalentes si sus estados iniciales son equivalentes.

Está claro que cualquier AFD también es un AFND, es decir, si L es un lenguaje aceptado por un AFD, también está aceptado por un AFND. Simplemente existe como mucho una sola transición para cada símbolo del alfabeta y para cada estado.

Pero también podemos construir para cada AFND un AFD equivalente, es decir, un autómata determinsta que acepta el mismo lenguaje.

Ejemplo: convertimos el AFND que acepta L_{dos} en un AFD equivalente:

afndafd

Para el caso general tenemos:

Sea $M = (\Sigma, Q, \delta, q_0, F)$ un AFND, construimos un AFD $M' = (\Sigma, Q', \delta', q_0', F')$ con

- $Q' \subset \mathcal{P}(Q)$, es decir, es—como mucho—el conjunto de todos los subconjuntos de Q.
- $q_0' = \{q_0\}$, es decir, es el conjunto que contiene el estado inicial del AFND.
- $\delta'(Q_i, \sigma) = P_j \iff \forall p \in P_j \exists q \in Q_i$ con $\delta(q, \sigma) = p$, (por incluir, escribe texto)
- $F' \subset \mathcal{P}(Q)$ con si $f \in F'$ entonces existe un $q \in f$ con $q \in F$, es decir, el conjunto de estados finales son todos aquellos estados del AFD que contienen por lo menos un estado final del AFND.

Se suelen construir los estados necesarios del AFD a lo largo de la construcción en vez de coger por defecto todos los posibles subconjuntos, para evitar—en caso que sea posible—la construcción de muchos estados que finalmente no se alcanzan desde el estado inicial.

¿Por qué es correcta la construcción?

Tenemos que comprobar formalmente que si M (siendo un AFND) acepta w, entonces M' (siendo el AFD construido) también lo acepta; y si M' acepta w, entonces M también lo hace, es decir, que $L(M) = L(M')$.

Pues, sea M un AFND y M' el AFD correspondiente.

Sea $w = x_0 x_1 x_2 \dots x_n \in L(M)$ cualquier palabra aceptada por M.

Comprobamos que $w \in L(M')$, es decir, $L(M) \subset L(M')$:

Definimos los siguientes diagramas

$$\begin{array}{ccc} p & \xrightarrow{x_i} & q \\ \cap & & \cap \\ P & \xrightarrow{x_i} & Q \end{array}$$

es decir, si hacemos la transición en M desde p a q leyendo x_i, en otras palabras, usamos $\delta(p, x_i) = q$, entonces existe (según construcción) una transición en M' de P (con $p \in P$) a Q (con $q \in Q$) leyendo x_i, en otras palabras, existe $\delta(P, x_i) = Q$.

Para la palabra w obtenemos:

$$\begin{array}{ccccccccccccc} q_0 & \xrightarrow{x_0} & q_1 & \xrightarrow{x_1} & q_2 & \xrightarrow{x_2} & \cdots & \xrightarrow{x_{n-1}} & q_n & \xrightarrow{x_n} & q_{n+1} & \in & F \\ \cap & & \cap & & \cap & & & & \cap & & \cap & & \\ Q_0 & \xrightarrow{x_0} & Q_1 & \xrightarrow{x_1} & Q_2 & \xrightarrow{x_2} & \cdots & \xrightarrow{x_{n-1}} & Q_n & \xrightarrow{x_n} & Q_{n+1} & \in & F' \end{array}$$

donde la construcción va *desde la izquierda*, es decir, del estado inicial, hacia la derecha, es decir, a un estado final. Dado que M acepta w, q_{n+1} es un estado final y siendo miembro de un conjunto Q_{n+1}, este será un estado final de M'.

Entonces hemos comprobado que M' acepta w, y por eso $L(M) \subset L(M')$.

Ahora, sea $w = x_0 x_1 x_2 \ldots x_n \in L(M')$ cualquier palabra aceptada por M'.

Comprobamos que $w \in L(M)$, es decir, $L(M) \supset L(M')$:

Definimos los siguientes diagramas

$$\begin{array}{ccc} P & \xrightarrow{x_i} & Q \\ \cup & & \cup \\ p & \xrightarrow{x_i} & q \end{array}$$

es decir, si hacemos la transición en M' desde P leyendo x_i a Q, en otras palabras, usamos $\delta(P, x_i) = Q$, entonces existe (según construcción) una transición en M de algún p (con $p \in P$) leyendo x_i a algún q (con $q \in Q$), en otras palabras, existe $\delta(p, x_i) = q$.

Para la palabra w obtenemos:

$$\begin{array}{ccccccccccccc} Q_0 & \xrightarrow{x_0} & Q_1 & \xrightarrow{x_1} & Q_2 & \xrightarrow{x_2} & \cdots & \xrightarrow{x_{n-1}} & Q_n & \xrightarrow{x_n} & Q_{n+1} & \in & F' \\ \cup & & \cup & & \cup & & & & \cup & & \cup & & \\ q_0 & \xrightarrow{x_0} & q_1 & \xrightarrow{x_1} & q_2 & \xrightarrow{x_2} & \cdots & \xrightarrow{x_{n-1}} & q_n & \xrightarrow{x_n} & q_{n+1} & \in & F \end{array}$$

donde la construcción va ahora *desde la derecha*, es decir, un estado final, hacia la izquierda, es decir, al estado inicial. Dado que M' acepta w, Q_{n+1} es un estado final y un conjunto no vacío, entonces existe un miembro q_{n+1} que también es elemento de F y por consecuencia un q_n aplicando el diagrama y asi succesivamente hasta llegar a q_0.

Entonces hemos comprobado que M acepta w, y por eso $L(M) \supset L(M')$.

Finalmente tenemos $L(M) \subset L(M')$ y $L(M) \supset L(M')$ y por eso $L(M) = L(M')$.

Como se observa en la construcción puede ser que se usa $2^{|Q|}$ estados en el autómata determinista si el autómata no-determinista tenía $|Q|$ estados, es decir, el crecimiento del número de estados puede ser exponencial.

Surgen dos preguntas:

1. ¿Existen AFNDs que producen un AFD de tal tamaño grande?
2. ¿Son necesarios tantos estados (o existe una mejor forma de realizar la conversión)?

Un ejemplo para una respuesta a la segunda:

Usamos $\Sigma = \{a, b\}$ como alfabeto. Definimos los siguientes lenguajes (que dependen del número $n \in \mathbb{N}$):

$$L_n = \{w \mid w \in \Sigma^*, w = w_1 w_2, w_1 \neq w_2, |w_1| = |w_2| = n, n \in \mathbb{N}\}$$

es decir, todas las palabras con $2n$ letras donde la primera mitad se distingue de la segunda.

Es bastante claro que para cualquier n existe un autómata que acepta L_n porque el lenguaje es finito ($|L_n| = 2^{2n} - 2^n$).

En un libro (HotzEstenfeld) se encuentra el siguiente AFND que acepta L_3 (dejan la comprobación al lector)

afndln

Bueno, con un poco de trabajo se puede comprobar (enumerando todos los caminos desde el estado inicial hasta el estado final) que en cada uno de los caminos siempre existe en la primera parte una arista con una a (o una b) donde en la misma posición de la segunda parte hay una b (o una a).

El AFND dado tiene 22 estados que (sin que ellos lo dicen) está en el orden de n^2 (si inspeccionamos la construcción 'vemos' la suma de 1 hasta $2n$).

También construeron un AFD para L_3:

afdln

Manifestan que dicho autómata es mínimo, y teniendo más de 2^n estados concluyen que la construcción de un AFND a un AFD puede incrementar el número de estados exponencialmente.

Veremos: ¡Ambas construcciones tienen sus deficiencias, aunque el hecho en si es correcto!

Primero, no dan un esquema cómo construir un autómata que reconozca L_n para cualquier n (puede ser que hay 'buena suerte' en el caso de L_3).

Segundo, el AFD dado no es mínimo, una simplificación sería:

afdlns

Pero, el nuevo autómata sigue necesitando un número exponencial de estados, porque se tiene que construir en el 'lado izquierdo' todas las posibles palabras w_1.

Entonces: ¿Creemos o sabemos?, si no lo hemos comprobado o si no hemos entendido una comprobación presentada, entonces solamente creemos. El saber va más allá. Hay que mantenerse crítico, siempre.

Construimos un AFND para L_n sistemáticamente.

Idea: En cada uno de los caminos reconociendo w_1 siempre tiene que existir una arista con una a (o una b) donde en la misma posición para reconocer w_2 hay una b (o una a).

Este principio nos lleva a una construcción inductiva:

afdln1

afdln2

afdln3

afdlnn

El número de estados entonces es:

$$\begin{aligned} |Q| &= 1+2+4+6+\ldots+2n+(2n-2)+\ldots+4+2+1 \\ &= 1+2\sum_{i=1}^{n} i+1+2\sum_{i=1}^{n-1} i \\ &= 1+n(n+1)+1+(n-1)n \\ &= 2(n^2+1) \end{aligned}$$

Como vemos, incluso hemos reducido el número de estados comparando con el autómata del libro: el AFND para aceptar L_3 tiene solamente 20 estados.

La construcción de un AFD sigue el mismo argumento dado arriba: se necesita construir todas las posibles palabras w_1 en 'el lado izquierdo' y por eso el AFD tiene por lo menos 2^n estados (los $2^n - 1$ para enumerar los w_1 y por lo menos un estado final en 'el lado derecho'.

Hasta ahora sólo hemos comprobado la explosión del número de estados para lenguajes finitos. ¿Existe tal crecimiento exponencial también para lenguajes infinitos?

Otro ejemplo para mostrar las capacidades de un AFND (y el crecemiento exponencial necesario del AFD equivalente):

Usamos $\Sigma = \{0, 1\}$ como alfabeto. Definimos los siguientes lenguajes (que dependen del número $n \in \mathbb{N}$):

$$L_n = \{w \mid w \in \Sigma^*, w \text{ contiene un 1 en la } n\text{-nésima posición desde la derecha}\}$$

Es bastante fácil construir un AFND que acepte L_n:

afndlr

No es tan obvio como construir directamente un AFD. Pero es posible con la construcción (¡Hazlo!).

Observamos en la construcción:

- Sea $w = x_n x_{n-1} \dots x_2 x_1 \in \{0,1\}^*$.
- Para todos los $i \in \{1, \dots, n\}$ tenemos:

$$q_i \in \delta^*(q_0, w) \Longrightarrow x_i = 1$$

  es decir:

  - Si $x_i = 1$ (el i-ésimo símbolo desde la derecha es un 1), entonces existe un camino desde q_0 a q_i (es decir, $q_i \in \delta^*(q_0, w)$) porque podemos usar dicho x_i para pasar el 'puente' y
  - si existe un camino desde q_0 a q_i leyendo w ($q_i \in \delta^*(q_0, w)$), entonces w tiene un 1 como i-ésimo símbolo desde la derecha (es decir, $x_i = 1$) porque hemos pasado el 'puente'.

- Entonces, existe en la construcción para cada subconjunto $P \in \mathcal{P}(Q)$ con $q_0 \in P$ una palabra w tal que tenemos que construir un camino desde $Q_0 = \{q_0\}$ hacia P.
- Entones el AFD contiene por lo menos $2^{|Q|-1} = 2^n$ estados (todos aquellos que codifican subconjuntos conteniendo q_0).

Construimos un AFD directamente:

afdlr

Este autómata (y siguiendo el esquema de la construcción) contiene 2^n estados.

En ambos ejemplos parece que el número de estados necesarios en un AFD tenga algo que ver con la 'capacidad de contar o enumerar' hasta cierto número.

7.4. Autómatas finitos no-deterministas con transiciones ε (AFND-ε)

Queremos construir un autómata que acepta el lenguaje

$$L = \{a^i b^j c^k \mid i, j, k \in \mathbb{N}\}$$

Si fuesemos capaz de saltar mágicamente, es decir, sin consumir una letra de la entrada, de un estado a otro, sería fácil la construcción:

AUTaibjckeps

Es decir, hemos introducido aristas marcados con la palabra vacía ε.

Un **autómata finito no-determinista con transiciones** ε (AFND-ε) es una quíntupla

$$M = (\Sigma, Q, \delta, q_0, F)$$

donde

- Q, Σ, q_0, y F están definidos igual como en el caso de un AFND
- δ es
  - una relación, es decir $\delta \subset (Q \times (\Sigma \cup \{\varepsilon\})) \times Q$
  - o una función, es decir, $\delta \ : \ Q \times (\Sigma \cup \{\varepsilon\}) \longrightarrow \mathcal{P}(Q)$ siendo $\mathcal{P}(Q)$ el conjunto de las partes de Q

Observamos que añadir más aristas con ε obviamente no cambia el comportamiento del autómata:

AUTaibjckepstrans

Podemos tratar las transiciones con ε como una relación T sobre el conjunto de estados, es decir

$$T = T_1 = \{(q, p) \mid \delta(q, \varepsilon) = p\} \subset Q \times Q$$

En el ejemplo tenemos

$$T_1 = \{(q_0, q_1), (q_1, q_2)\}$$

Esta relación podemos ampliar para que sea reflexiva, es decir, que todas las parejas (q, q) con $q \in Q$ formen parte de la relación, es decir, formamos

$$T_0 = \{(q, q) \mid q \in Q\}$$

y con eso

$$T = T_0 \cup T_1$$

entonces T por construcción es una relación reflexiva. En el ejemplo tenemos

$$T_0 = \{(q_0, q_0), (q_1, q_1), (q_2, q_2)\}$$

y con eso

$$T = \{(q_0, q_0), (q_0, q_1), (q_1, q_1), (q_1, q_2), (q_2, q_2)\}$$

Podemos ampliar la relación aun más considerando el efecto transitivo de las transiciones ε, es decir, formamos en un primer paso

$$T_2 = \{(q, p) \mid \exists r \in Q \ : \ (q, r), (r, p) \in T_0 \cup T_1 \text{ y } (q, p) \notin T_0 \cup T_1\}$$

y con eso

$$T = T_0 \cup T_1 \cup T_2$$

en el ejemplo tenemos

$$T_2 = \{(q_0, q_2)\}$$

y así sucesivamente

$$T_i = \{(q,p) \mid \exists r \in Q \ : \ (q,r),(r,p) \in \bigcup_{j=0}^{i-1} T_j \text{ y } (q,p) \notin \bigcup_{j=0}^{i-1} T_j\}$$

Finalmente definimos

$$T^* = T_0 \cup T_1 \cup T_2 \cup \ldots = \bigcup_{i=0}^{\infty} T_i$$

como **clausura (o ciero, o cerradura) transitiva de la relación de las transiciones** ε o más breve **clausura-**ε.

El proceso termina en nuestro caso de autómatas finitos, es decir, la unión va solamente sobre un número finito de i's, porque T^* sigue siendo un subconjunto del conjunto finito $Q \times Q$ (es decir, $T^* \subset Q \times Q$).

Con la clausura-ε podemos definir la **clausura-**ε **de un estado**, como todos aquellos estados que se puede alcanzar con caminos de transisiones ε, es decir

$$cl(q) = \{p \mid (q,p) \in T^*\}$$

En el ejemplo:

$$\begin{aligned} cl(q_0) &= \{q_0, q_1, q_2\} \\ cl(q_1) &= \{q_1, q_2\} \\ cl(q_2) &= \{q_2\} \end{aligned}$$

AUTaibjckafnd

- hemos añadido q_0 a los estados finales F porque existe un estado final que pertenece a la clausura-ε de q_0, es decir, $\varepsilon \in L$
- hemos marcado las aristas de la clausura-ε con símbolos del alfabeto

Entonces podemos formalizar el lenguaje aceptado por un AFND-ε (parecido a lo que hicimos para un AFND).

Primero definimos la ampliación de δ para autómatas con transiciones ε. $\delta^*(q, w)$ va a ser el conjunto de estados (igual como en el caso de δ^* para AFNDs) que podemos alcanzar desde q leyendo la palabra. Entonces:

$$\delta^* : Q \times \Sigma^* \longrightarrow \mathcal{P}(Q)$$

1. $$\delta^*(q, \varepsilon) = cl(q)$$

   es decir, nos quedamos en la clausura-ε si hemos alcanzado el final de la palabra

2. $$\begin{aligned} \delta^*(q, w\sigma) &= \{p \mid p \in Q \text{ y } \exists r \in \delta^*(q, w) \text{ tal que } p \in cl(\delta(r, \sigma))\} \\ &= \bigcup_{r \in \delta^*(q,w)} cl(\delta(r, \sigma)) \end{aligned}$$

   es decir, $\delta^*(q, w\sigma)$ es el conjunto de estados alcanzables desde un estado r siendo miembro de la clausura-ε de un estado alcanzable desde q sin haber leído el último símbolo σ.

$\delta^*(q_0, w)$ enumera entonces todos los estados alcanzables desde q_0 leyendo la palabra w.

Observa: Hemos dado una definición recursiva desde la izquierda, es decir, añadimos un símbolo a la derecha. Hubiese sido posible definir δ^* para un AFND de la misma manera.

Un **autómata no-determinista con transiciones** ε M **acepta una palabra** w sobre el alfabeto Σ, es decir, $w \in \Sigma^*$, si

$$\delta^*(q_0, w) \cap F \neq \emptyset$$

donde δ^* sea la ampliación de la funciòn δ dada arriba.

El lenguaje aceptado por M es (como siempre)

$$L(M) = \{w \mid M \text{ acepta } w\}$$

7.5. Equivalencia entre AFND y AFND-ε

Primero observamos que cualquier AFND es obviamente también un AFND-ε (pues uno que, por casualidad, no tenga transiciones ε).

Luego podemos construir a partir de un AFND-ε un AFND equivalente.

Entonces, sea $M = (\Sigma, Q, \delta, q_0, F)$ un AFND-ε.

Un AFND equivalente es el autómata $M' = (\Sigma, Q', \delta', q_0', F')$ donde

- $Q' = Q$
- $\delta'(q,\sigma) = \bigcup_{r \in cl(q)} cl(\delta(r,\sigma))$ (podemos escribir solo q porque $Q' = Q$)
- $q'_0 = q_0$
- $F' = \begin{cases} F & \text{si} \quad F \cap cl(q_0) = \emptyset \\ F \cup q_0 & \text{si} \quad F \cap cl(q_0) \neq \emptyset \end{cases}$
  es decir, añadimos q_0 como estado final, si algún estado final del AFND-ε pertenece a la clausura-ε del estado inicial.

Convertimos el ejemplo:

La tabla de transiciones para M con las transiciones de la clausura-ε es:

	a	b	c	ε	cl
q_0	$\{q_0\}$	$-$	$-$	$\{q_1\}$	$\{q_0, q_1, q_2\}$
q_1	$-$	$\{q_1\}$	$-$	$\{q_2\}$	$\{q_1, q_2\}$
q_2	$-$	$-$	$\{q_2\}$	$-$	$\{q_2\}$

entonces la tabla con transiciones desde la claurura-ε es

	a	b	c	cl
$cl(q_0)$	$\{q_0\}$	$\{q_1\}$	$\{q_2\}$	$\{q_0, q_1, q_2\}$
$cl(q_1)$	$-$	$\{q_1\}$	$\{q_2\}$	$\{q_1, q_2\}$
$cl(q_2)$	$-$	$-$	$\{q_2\}$	$\{q_2\}$

y con eso la tabla final del AFND es

	a	b	c
q'_0	$\{q_0, q_1, q_2\}$	$\{q_1, q_2\}$	$\{q_2\}$
q'_1	$-$	$\{q_1, q_2\}$	$\{q_2\}$
q'_2	$-$	$-$	$\{q_2\}$

Además tenemos $F \cap cl(q_0) \neq \emptyset$ y por eso $F' = F \cup \{q_0\} = \{q_0, q_2\}$.

Finalmente resulta el siguiente grafo:

afnde

¿Por qué es correcto la construcción?

Pues los argumentos (y la comprobación) siguen los mismos pasos como lo vimos en el caso de AFND y AFD. Siempre cuando hay una transición en el AFND-ε leyendo un símbolo encontramos (según construcción) una transición en el AFND correspondiente porque consideramos

toda la clausura-ε, y vice versa, si hay una transición en el AFND, tiene que haber existido una transición en el AFND-ε o bien con o bien sin una secuencia de transiciones ε.

¿Cuánto ha crecido esta vez el autómata?

El número de estados queda igual, solo se amplia (si hace falta) F por un estado. Pero ha crecido el número de aristas (es decir, transisiones). Dicho crecimiento puede llegar como mucho a $|\Sigma||Q|^2$ porque como mucho tantas aristas se pueden incorporar entre los nodos del grafo.

Finalmente hemos comprobado la equivalencia entre autómatas no-deterministas y autómatas no-deterministas con transiciones ε.

7.6. Existencia de autómatas finitos mínimos

Ya vimos que hay varias posibilidades para construir un autómata finito determinista que acepte un lenguaje (regular), por ejemplo, por construcción directa, o por el paso de un AFND a un AFD.

Surge la pregunta: ¿existe un autómata finito determinista (AFD) mínimo que acepta tal lenguaje?

Nos referimos al número de estados que tiene el AFD, es decir $|Q|$, dado que el número de transiciones por estado está determinado por el número de símbolos en Σ multiplicado por $|Q|$ si el AFD es completo.

La respuesta es: ¡por supuesto que sí!

Con el siguiente argumento: cada subconjunto de los números enteros $\mathbb{N}$ tiene un mínimo, y los números de estados de todos los posibles AFDs que aceptan L forman tal subconjunto.

Para la construcción del autómata mínimo necesitamos el formalismo de las relaciones de equivalencia.

Ya vimos que para cada lenguaje $L \subset \Sigma^*$ podemos construir una relación de equivalencia sobre Σ^*:

$$xR_Ly \iff (\forall z \in \Sigma^* : xz \in L \iff yz \in L)$$

es decir, x es equivalente a y, si, añadiendo cualquier sufijo, ambas palabras resultantes o bien están en L o bien no están en L.

Un lenguaje $L \subset \Sigma^*$ es regular, si y solo si el índice de la relación R_L es finito, es decir, la relación tiene solamente un número finito de clases de equivalencia (Teorema de Myhill y Nerode).

Comprobamos primero la dirección "$\Longrightarrow$", es decir, si el lenguaje es regular, entonces el índice de la relación es finito:

L es regular, entonces existe un AFD que acepta L.

Sea $M = (\Sigma, Q, \delta, q_0, F)$ un AFD con $L(M) = L$.

Definimos una relación de equivalencia sobre M:

$$xR_My \quad \text{si} \quad \delta^*(q_0, x) = \delta^*(q_0, y)$$

es decir, llegamos al mismo estado leyendo x o y empezando en el estado inicial.

Veremos a continuación que $R_M \subseteq R_L$, es decir, que R_M es un refinamiento de R_L, o en otras palabras, si dos elementos caen en una misma clase de equivalencia respecto a la relación R_M, también caen en una misma clase respecto a R_L.

Entonces, sea xR_My, es decir $\delta^*(q_0, x) = \delta^*(q_0, y)$.

Sea $z \in \Sigma^*$ cualquier palabra. Miramos:

$$\begin{aligned} xz \in L &\iff \delta^*(q_0, xz) \in F \\ &\iff \delta^*(\delta^*(q_0, x), z) \in F \\ &\iff \delta^*(\delta^*(q_0, y), z) \in F \\ &\iff \delta^*(q_0, yz) \in F \\ &\iff yz \in L \end{aligned}$$

es decir, si xR_My entonces también xR_Ly, y por eso:

$$\begin{aligned} \text{Indice}(R_L) &\leq \text{Indice}(R_M) \\ &= \text{número de estados acesibles desde } q_0 \\ &\leq |Q| \\ &< \infty \end{aligned}$$

Comprobamos ahora la dirección "$\Longleftarrow$", es decir, si el índice de la relación es finito, entonces el languaje es regular. Dicha comprobación va a ser una comprobación constructiva muy útil:

Sea R_L la relación de equivalencia de L con $\text{Indice}(R_L) < \infty$.

Entonces hay palabras $x_1, x_2, \ldots, x_k \in \Sigma^*$ con $k < \infty$, es decir k es finito, cuyas clases cubren Σ^*:

$$\Sigma^* = [x_1] \cup [x_2] \cup \ldots \cup [x_k]$$

Construimos un AFD que contiene justamente tantos estados como hay clases:

$$M = (\Sigma, Q, \delta, q_0, F)$$

donde

- $Q = \{[x_1], [x_2], \ldots, [x_k]\}$

- $\delta([x], \sigma) = [x\sigma]$, es decir, se hace la transición de la clase a la cual pertenece x leyendo σ a la clase a la cual pertenece $x\sigma$
- $q_0 = [\varepsilon]$, es decir, el estado inicial es la clase a la cual pertenece la palabra vacía
- $F = \{[x] \mid x \in L\}$, es decir, existen tantos estados finales como hay clases de equivalencia perteneciendo a L.

Entonces: $\delta^*([\varepsilon], x) = \delta^*(q_0, x) = [x]$ y vemos

$$\begin{aligned} x \in L(M) &\iff \delta^*(q_0, x) \in F \\ &\iff \delta^*([\varepsilon], x) \in F \\ &\iff [x] \in F \\ &\iff x \in L \end{aligned}$$

7.7. Ejemplos de uso del teorema de Myhill y Nerode

Investigamos de nuevo el lenguaje

$$L = \{a^n b^n \mid n \in \mathbb{N}, n > 0\}$$

anotamos unas clases de equivalencia de L:

$$\begin{aligned} [ab] &= L \\ [a^2b] &= \{a^2b, a^3b^2, a^4b^3, \ldots\} \\ &\ldots \\ [a^kb] &= \{a^{k+i-1}b^i \mid i \geq 1\} \end{aligned}$$

verificamos que son clases de equivalencia, porque si $a^{k+j-1}b^j \in [a^kb]$ y $a^{k+l-1}b^l \in [a^kb]$ entonces o bien $a^{k+j-1}b^jz, a^{k+l-1}b^lz \in L$ (si $z = b^{k-1}$) o bien $a^{k+j-1}b^jz, a^{k+l-1}b^lz \notin L$ (si $z \neq b^{k-1}$).

Por eso el número de clases de R_L es infinito, es decir, $\mathrm{Indice}(R_L) = \infty$.

Observa que no hemos clasificado todas las palabras de Σ^*, sino solamente algunas palabras posibles:

$$\Sigma^* = L \cup \underbrace{[a^2b] \cup \ldots \cup [a^kb] \cup \ldots}_{\text{ya son un número infinito}} \underbrace{\cup \ldots}_{\text{las demás clases}}$$

es decir, para comprobar que un lenguaje no es regular basta con encontrar un número infinito de clases de equivalencia (respecto a la relación R_L).

Investigamos el lenguaje

$$L = \{w \mid w \in \{0,1\}^* \text{ y } w \text{ termina con } 00\}$$

Pensamos en las posibles clases de equivalencia. Obviamente hay tres, o bien una palabra no termina en 0, o bien termina en un 0, o bien termina por lo menos en dos 0, es decir:

$$\begin{aligned}[\varepsilon] &= \{w \mid w \text{ no termina en } 0\} \\ [0] &= \{w \mid w \text{ termina en un solo } 0\} \\ [00] &= \{w \mid w \text{ termina en } 00\}\end{aligned}$$

Con $\Sigma^* = [\varepsilon] \cup [0] \cup [00]$ seguimos la construcción de arriba y obtenemos la tabla de transiciones para el autómata:

	0	1
$\Longrightarrow [\varepsilon]$	$[0]$	$[\varepsilon]$
$[0]$	$[00]$	$[\varepsilon]$
$\star[00]$	$[00]$	$[\varepsilon]$

o como diagrama:

equiafd

7.8. Algoritmo de minimización

La comprobación del teorema de Myhill y Nerode nos proporciona un hecho muy importante: el autómata basado en las clases de equivalencia es el autómata mínimo dentro de todos los posibles autómatas finitos deterministas y completos que aceptan el mismo lenguaje, porque un tal autómata M' definiría un refinamiento de $R_{M'} \subseteq R_L$, es decir, $\text{Indice}(R_{M'}) \geq \text{Indice}(R_L)$ y el AFD de las clases de equivalencia M representa las mismas clases $R_L = R_M$, entonces $\text{Indice}(R_{M'}) \geq \text{Indice}(R_L) = \text{Indice}(R_M)$.

Una pregunta surge: ¿Cómo sabemos si un AFD M ya es mínimo?

Pues, M no es mínimo, si

$$\forall w \in \Sigma^* \exists p, q \in Q, p \neq q : \delta^*(p, w) \in F \Longleftrightarrow \delta^*(q, w) \in F$$

es decir, llegamos con alguna palabra w desde ambos estados siempre o bien a un estado final, o bien a un estado no-final.

En tal caso, podemos unir los dos estados en un único estado.

Basta con 'realizar las pruebas' con todas las palabras w con $|w| < |Q|$ porque no hace falta visitar un estado dos veces.

Con dicho argumento describimos el algoritmo de minimización (sin comprobación) a continuación.

Decimos que dos estados p y q son distinguibles (o no-equivalentes) si existe una palabra w que nos lleva desde p a un estado final pero no desde q, o al revés, es decir:

$$p \not\equiv q \iff (\delta^*(p,w) \in F \text{ y } \delta^*(q,w) \notin F) \text{ o } (\delta^*(p,w) \notin F \text{ y } \delta^*(q,w) \in F)$$

El algoritmo calculará la relación de distinguibilidad (o no-equivalencia) entre los estados y contiene 5 pasos.

1. Se elimina todos los estados no acesibles desde el estado inicial.
2. Se forma una tabla de todas las parejas de estados (p,q) con $p \neq q$.
3. Se marca en la tabla todas las parejas (p,q) con $p \in F, q \notin F$ o $p \notin F, q \in F$ (porque dichos estados seguro son distinguibles).
4. Mientras haya cambio en la tabla:

   para cada pareja (p,q) no marcada y para cada símbolo σ

   si $(\delta(p,\sigma), \delta(q,\sigma))$ está marcada, también se marca (p,q).
5. Las parejas (duplas) no marcadas se une en un sólo estado.

Ejemplo: partimos del siguiente AFD completo:

afdc

1. Todos los estados son acesibles desde a, por eso, no hay que eliminar nada.
2. La tabla es:

	a	b	c	d	e
a	-				
b	-	-			
c	-	-	-		
d	-	-	-	-	
e	-	-	-	-	-

3. Las marcas iniciales son (en vez de simple marcas, usamos números para visualizar en el siguiente apartado los cambios en la tabla en cada paso):

	a	b	c	d	e
a	-				1
b	-	-			1
c	-	-	-		1
d	-	-	-	-	1
e	-	-	-	-	-

4.

	a	b	c	d	e
a	-	2		3	1
b	-	-	4		1
c	-	-	-	5	1
d	-	-	-	-	1
e	-	-	-	-	-

5. El autómata mínimo es:

   afdcmin

   Observa que en la construccón del autómata podemos comprobar de cierta manera la corrección de la tabla: cuando recorremos todas las aristas del autómata original, tenemos que o bien añadir o bien encontrar su homóloga en el autómata en construcción.

El paso 4 se puede implementar más eficiente. En vez de mirar tantas veces las parejas no marcados, se mantiene listas de espera que se marcan recursivamente. Observamos:

- Si tenemos que marcar (p, q), es porque $(r, s) = (\delta(p, \sigma), \delta(q, \sigma))$ ya está marcado.
- Entonces de alguna manera la pareja (p, q) depende de la pareja (r, s).
- Es decir, si en un futuro marcamos en algun momento (r, s), directamente podemos marcar (p, q) también.

Para llevar eso a cabo, añadimos a cada celda una lista de parejas que dependen de la la pareja en cuestión. Si se marca una pareja, recursivamente se marcan también todas las entradas en las listas.

Con está mejora el algoritmo tiene complejidad $O(|Q|^2|\Sigma|)$.

8. Expresiones regulares

Hasta ahora era difícil describir lenguajes aceptados por autómatas. Siempre teníamos que aprovechar de una notación como

$$L(M) = \{w \mid \text{ alguna propiedad de } w\}$$

Por ejemplo, si queríamos desarrollar un autómata que comprobase que una cadena codificase una dirección de correo electrónico válida tendríamos como propiedades:

1. los símbolos permitidos son: `a-z, A-Z, 0-9, @ . - _`
2. debe contener exactamente una `@`
3. por lo menos un `.` detrás de la `@`
4. detrás del último `.` deben venir entre 2 y 4 letras
5. detrás de cada `.` y de la `@` debe venir por lo menos una letra
6. delante de la `@` por lo menos una palabra que empieza con una letra,

es decir, $L(M) = \{w \mid w$ cumple las condiciones de arriba $\}$.

Ejercicio: ¡Intenta construir un autómata!

Sería conveniente tener un meta-lenguaje que nos permitiese describir fácilmente lenguajes (por lo menos de cierto tipo).

8.1. Sintaxis y semántica

Sea Σ un alfabeto. Una **expresión regular** α sobre Σ se define con las siguientes reglas (inductivas):

1. *a*) $\emptyset$ es una expresión regular
   b) ε es una expresión regular
   c) si $\sigma \in \Sigma$, entonces σ es una expresión regular
2. si α y β son expresiones regulares, entonces también
   a) $\alpha.\beta$ es una expresión regular (obviamos del punto muchas veces)

b) $(\alpha + \beta)$ es una expresión regular

3. si α es una expresión regular, entonces también
   a) (α) es una expresión regular
   b) $(\alpha)^*$ es una expresión regular

Como observamos: hemos introducido meta-símbolos ('(',')','*','+','.','$\emptyset$'). Si alguno de ellos aparece en Σ tenemos un problema (Houston) que resolveremos al final de esta sección.

Ejemplos:

Sea $\Sigma = \{a, b, c\}$. Posibles expresiones regulares son:

$$((a.b)^* + b.c.(a)^*) \qquad ((a.a.a + b.c) + (c.b)^*.(b)^*)$$

Con eso hemos definido una sintaxis de expresiones regulares, pero ¿cuál será su semántica?

Para cada expresión regular definimos un lenguaje correspondiente (basado en las reglas).

El **lenguaje** $L(\alpha)$ **definido por una expresión regular** α se define:

1. *a*) $L(\emptyset) = \emptyset$
   b) $L(\varepsilon) = \{\varepsilon\}$
   c) si $\sigma \in \Sigma$, entonces $L(\sigma) = \{\sigma\}$
2. si α y β son expresiones regulares, entonces
   a) $L(\alpha.\beta) = L(\alpha).L(\beta)$
   b) $L((\alpha + \beta)) = L(\alpha) \cup L(\beta)$
3. si α es una expresión regular
   a) $L((\alpha)) = L(\alpha)$
   b) $L((\alpha)^*) = (L(\alpha))^*$

Ejemplos: sobre $\Sigma = \{0, 1\}$:

- el lenguaje que contiene una subcadena 11:

$$((0 + 1))^*.1.1.((0 + 1))^*$$

- todas las cadenas que alternan 0 y 1:

$$(((0.1)^* + (0.1)^*.0) + ((1.0)^* + (1.0)^*.1))$$

  o también con la expresión

$$(1 + \varepsilon).(0.1)^*.(0 + \varepsilon)$$

8.2. Equivalencia entre autómatas finitos y expresiones regulares

La semántica de una expresión regular define un lenguaje.

Dado una expresión regular α (sobre un alfabeto Σ). ¿Qué tiene que ver el lenguaje $L(\alpha)$ con un lenguaje $L(M)$ aceptado por un autómata finito M?

Veremos: para cada expresión regular α existe un autómata no-determinista con transiciones ε M, o sea un AFND-ε, que acepta el mismo lenguaje (es decir, $L(\alpha) = L(M)$).

Ya sabemos: entonces también existe un autómata finito determinista, o sea un AFD, aceptando el mismo lenguaje.

De hecho, comprobaremos algo más: para cada α sobre Σ existe un AFND-ε $M = (\Sigma, Q, \delta, q_0, F)$ con $L(\alpha) = L(M)$ y

- no existe ninguna transición hacia el estado inicial, es decir

$$\forall q \in Q, \sigma \in \Sigma : q_0 \notin \delta(q, \sigma) \cup \delta(q, \varepsilon)$$

- M tiene exactamente un estado final del cual no sale ninguna transición, es decir,

$$|F| = 1 \text{ y } \forall \sigma \in \Sigma, f \in F : \delta(f, \sigma) \cup \delta(f, \varepsilon) = \emptyset$$

La comprobación sigue la definición inductiva de la expresión regular, lo describimos solamente con los grafos de los autómatas. Entonces, sean α, β, y γ expresiones regulares sobre algún alfabeto Σ.

1. *a*) $\alpha = \emptyset$
   regexprafnde1
   b) $\alpha = \varepsilon$
   regexprafnde2
   c) $\alpha = a$
   regexprafnde3
2. *a*) $\alpha = \beta\gamma$
   regexprafnde4
   b) $\alpha = (\beta + \gamma)$
   regexprafnde5
3. *a*) $\alpha = (\beta)$
   regexprafnde6

b) $\alpha = (\beta)^*$
regexprafnde7

Ejemplo: construimos el AFND-ε para $\alpha = (((a.b)^* + a) + b.b)$

regexprafndeej

La otra dirección, es decir, comprobando que para cada autómata finito existe una expresión regular que describe el mismo lenguaje, nos costará un poco más de trabajo.

Sea $M = (\Sigma, Q, \delta, q_0, F)$ un AFD (sabemos que cualquier AFND o AFND-ε se puede convertir en un AFD).

Describimos un algoritmo que sucesivamente construye la clausura transitiva del autómata dado y así construye finalmente—como atributos de las aristas entre q_0 y un nuevo estado f—la expresión regular.

Por eso permitimos que se pueden escribir expresiones regulares a las aristas de un autómata, es decir, para $\delta(p, \sigma) = q$ escribimos (p, σ, q) (pues, la arista del estado p al estado q con atributo a), o teniendo expresiones regulares (p, α, q) (pues, una arista de p a q con atributo α), o con dibujo:

aristaexpr

1. añadimos un nuevo estado f y conectamos todos los estados en F con transiciones ε a f, es decir, cambiamos M por $M' = (Q \cup \{f\}, \Sigma, \delta', q_0, \{f\})$ donde $\delta' = \delta$ para estados en Q y además $\forall q \in F : \delta'(q, \varepsilon) = f$. Así no hemos cambiado el lenguaje aceptado por M. (Pero siguimos escribiendo abajo simplemente M, δ, y Q para simplificar la notación.)

2. para todos los estados $q \neq q_0$ y $q \neq f$

   a) para cada pareja de aristas (p, β, q) y (q, γ, r) y arista reflexiva (q, φ, q) (nota, puede ser $p = r$)
   añade arista $(p, \beta\varphi^*\gamma, r)$

   b) elimina q con todas sus aristas adyacentes
   pqr

   c) agrupa las aristas construidas $(p, \alpha_1, r), \ldots, (p, \alpha_k, r)$ escribiendo $(p, \alpha_1 + \ldots + \alpha_k, r)$

3. cuando termina el proceso, es decir, solamente existen aristas entre q_0 y f, precisamente (q_0, α, q_0) y/o (q_0, β, f), la expresión regular final es $\alpha^*\beta$.

(Observa: si $q_0 \in F$ entonces existe una arista con ε entre q_0 y f, por eso, $\varepsilon \in L(\beta)$, y entonces no hay que considerar un caso especial para contemplar lazos reflexivos en q_0 porque $\alpha^*\beta + \alpha^* = \alpha^*\beta$.)

Una comprobación formal de la corrección del algoritmo es bastante técnica. Principalmente hay que realizar una inducción estructural con propiedades de dichos autómatas extendidos (que tienen expresiones regulares en sus aristas). No lo detallamos aquí, cae en la categoría: lo creemos (en estos momentos).

Como vimos en el ejemplo, hemos construido una expresión regular totalmente diferente a la de partida. Debemos transformar dicha expresión regular sin cambiar el lenguaje que define para conseguir finalmente una expresión regular igual a la de partida. Por eso:

Dos **expresiones regulares** α **y** β **son equivalentes** ($\alpha \equiv \beta$) si definen el mismo lenguaje, es decir, si $L(\alpha) = L(\beta)$.

Obviamente hay operaciones con expresiones regulares que mantienen la equivalencia, por ejemplo:

Asociatividad:

$$\begin{aligned} (\alpha + (\beta + \gamma)) &\equiv ((\alpha + \beta) + \gamma) \\ \alpha.(\beta.\gamma) &\equiv (\alpha.\beta).\gamma \end{aligned}$$

Conmutatividad:

$$(\alpha + \beta) \equiv (\beta + \alpha)$$

Elementos neutros:

$$\begin{aligned} (\alpha + \emptyset) &\equiv (\emptyset + \alpha) \\ &\equiv \alpha \\ (\alpha.\varepsilon) &\equiv (\varepsilon.\alpha) \\ &\equiv \alpha \end{aligned}$$

Eliminación:

$$\begin{aligned} (\alpha.\emptyset) &\equiv (\emptyset.\alpha) \\ &\equiv \emptyset \end{aligned}$$

Distributividad:

$$\begin{aligned} \alpha.(\beta + \gamma) &\equiv (\alpha.\beta + \alpha.\gamma) \\ (\alpha + \beta).\gamma &\equiv (\alpha.\gamma + \beta.\gamma) \end{aligned}$$

Simplificación:

$$\begin{aligned}((\alpha)^*)^* &\equiv (\alpha)^* \\ (\emptyset)^* &\equiv \emptyset \\ (\varepsilon)^* &\equiv \varepsilon\end{aligned}$$

Con eso y un poco de ímpetu podemos transformar sucesivamente la expresión regular obtenida para obtener al final la expresión regular que era la base para el autómata finito inicial.

El problema de comprobar en general si dos expresiones regulares son equivalentes no es nada fácil. Dicho problema cae en la clase de los problemas PSPACE que contiene problemas aún más complejos que los problemas de la clase NP que (a lo mejor) veremos hacia el final del curso (un problema NP es el problema del viajante). Aquí nos basta constatar que existen EE.RR. tal que un algoritmo determinista que resuelve el problema necesita un tiempo que crece por lo menos exponencial en la longitud de las expresiones regulares.

8.3. Abreviaciones para el uso de expresiones regulares

Para simplificar más el uso de expresiones regulares, introducimos prioridades para eliminar parentesis, atorgamos

- a la operación 'asterisco de Kleene' máxima prioridad (parecido a la exponenciación en expresiones algebraicas)
- a la operación 'concatenación' segunda prioridad (parecido a la multiplicación en expresiones algebraicas) y
- a la operación 'adición' la mínima prioridad (parecido a la adición en expresiones algebraicas)

Adicionalmente describimos algunos ejemplos de abreviaciones de uso común para expresiones regulares (¡puede ser que dicha notación describe lenguajes que ya no son lenguajes regulares!):

Sea $\Sigma = \{\sigma_0, \sigma_1, \ldots, \sigma_n\}$ un alfabeto, donde los símbolos implícitamente estén ordenados, es decir, si $i < j$ para $i, j \in \{1, \ldots, n\}$ entonces σ_i viene antes en el orden de todos los símbolos que σ_j (pues, entonces es tal cual como estamos acostrumbados de tratar nuestro alfabeto del lenguaje natural).

- $[\sigma_i - \sigma_j] \; : \; \sigma_i + \ldots + \sigma_j$, es decir, todo el rango de símbolos entre (y incluyendo) σ_i y σ_j. Si $j < i$, $[\sigma_i - \sigma_j] = \varepsilon$.

- $\alpha?$: $(\alpha + \varepsilon)$, es decir, una o ninguna vez α.
- ., cualquier símbolo del alfabeto
- α^+ : $\alpha\alpha^*$, es decir, por lo menos una vez α.
- α^n : $\underbrace{\alpha \ldots \alpha}_{n-\text{veces}}$
  (usando tales n's en varias posiciones y exigiendo que tenga en todos los sitios el mismo valor se pueden describir lenguajes ya no regulares)
- $\alpha^{\{m,n\}}$ por lo menos m veces, pero como mucho n veces α.
  (igual como arriba, usando tales n's y m's en varias posiciones y exigiendo que tengan en todos los sitios los mismos valores se pueden describir lenguajes ya no regulares)

8.4. Símbolos y meta-símbolos

Resolvemos el problema de tener símbolos iguales en Σ y en el meta-alfabeto:

Se suele usar un símbolo de escape en el meta-lenguaje, normalmente el símbolo \. Si \ aparece delante de un meta-símbolo, entonces se considera un símbolo de Σ, y al revés, si \ aparece delante de un símbolo, se considera un meta-símbolo. (Entonces, si \ debe estar en Σ, se anotaría como \\.)

Entonces podemos escribir la expresión regular η que define una dirección de correo electrónico sintácticamente correta como:

$$\begin{aligned} \alpha &= [a-zA-Z][a-zA-Z0-9\backslash-\_]^* \\ \eta &= (\alpha\backslash.)^*\alpha@(\alpha\backslash.)^+[a-zA-Z]^{\{2,4\}} \end{aligned}$$

donde hemos unido adicionalmente varios rangos en uno.

9. Lenguajes regulares

9.1. Equivalencia entre gramáticas lineales por la derecha y autómatas finitos

Sea $M = (\Sigma, Q, \delta, q_0, F)$ un AFD.

Construimos una gramática lineal por la derecha G con $L(G) = L(M)$, es decir, genera el mismo lenguaje que el AFD acepta.

$$G = (\Sigma_N, \Sigma_T, P, \$) = (Q, \Sigma, P, q_0)$$

es decir

- $\Sigma_N = Q$, los estados del autómata determinan los símbolos no-terminales de la gramática
- $\Sigma_T = \Sigma$, los símbolos del autómata determinan los símbolos terminales de la gramática
- $\$ = q_0$, el estado inicial del autómata determina el símbolo inicial de la gramática

El sistema de producciones P está dado por:

- Si $\delta(q, \sigma) = p$ es una transición del AFD, con $p, q \in Q$ y $\sigma \in \Sigma$, entonces añadimos a P la producción $q \longrightarrow \sigma p$.
- Si $\delta(q, \sigma) = p$ es una transición del AFD, con $q \in Q$, $p \in F$ y $\sigma \in \Sigma$, entonces añadimos a P la producción $q \longrightarrow \sigma$.
- Si $q_0 \in F$, entonces añadimos a P la producción $q_0 \longrightarrow \epsilon$.

Ejemplo:

afdabc

	a	b	c
$\Longrightarrow \star q_0$	q_0	q_1	q_2
$\star q_1$	$-$	q_1	q_2
$\star q_2$	$-$	$-$	q_2

Entonces el sistema de producciones P de la gramática será:

$$P \;=\; \{q_0 \longrightarrow aq_0|a|bq_1|b|cq_2|c|\epsilon, q_1 \longrightarrow bq_1|b|cq_2|c, q_2 \longrightarrow cq_2|c\}$$

Sea $G = (\Sigma_N, \Sigma_T, P, \$)$ una gramática lineal por la derecha, es decir, $P \subset \Sigma_N \times (\Sigma_T.\Sigma_N \cup \Sigma_T) \cup \{\$ \longrightarrow \epsilon\}$.

Construimos una autómata finito M con $L(M) = L(G)$, es decir, el autómata acepta el mismo lenguaje que la gramática genera.

$$M = (\Sigma, Q, \delta, q_0, F) = (\Sigma_T, \Sigma_N \cup \{f\}, \delta, \$, \{f\})$$

es decir,

- $\Sigma_T = \Sigma$, los símbolos terminales de la gramática determinan los símbolos del autómata
- $Q = \Sigma_N \cup \{f\}$, los símbolos no-terminales de la gramática determinan los estados del autómata, y añadimos un nuevo estado f, es decir, $f \notin \Sigma_N$
- $q_0 = \$$, el símbolo inicial de la gramática determina el estado inicial del autómata

Las transiciones δ están dadas por:

- Si $A \longrightarrow \sigma B$ es una producción de G, con $A, B \in \Sigma_N$ y $\sigma \in \Sigma_T$, entonces añadimos la transición $\delta(A, \sigma) = B$.
- Si $A \longrightarrow \sigma$ es una producción de G, con $A \in \Sigma_N$ y $\sigma \in \Sigma_T$, entonces añadimos la transición $\delta(A, \sigma) = f$.
- Si $\$ \longrightarrow \epsilon$ es una producción de G, entonces añadimos la transición $\delta(\$, \epsilon) = f$.

Observamos que el autómata construido es un autómata finito no-determinista (AFND) que podemos convertir en un AFD si hace falta.

Ejemplo:

Para la gramática de arriba—renombrando los símbolos—convertimos

$$P \;=\; \{\$ \longrightarrow a\$|a|bA|cB|c|\epsilon, A \longrightarrow bA|b|cB|c, B \longrightarrow cB|c\}$$

a la tabla de transiciones

	a	b	c
$\Longrightarrow \$$	$\{\$, f\}$	$\{A\}$	$\{B, f\}$
A	–	$\{A, f\}$	$\{B, f\}$
B	–	–	$\{B, f\}$
$\star f$	–	–	–

graafd

9.2. Equivalencia entre gramáticas lineales por la derecha y lineales por la izquierda

Como era de esperar, gramáticas lineales por la derecha y gramáticas lineales por la izquierda describen el mismo 'fenómeno', es decir, generan los lenguajes regulares.

Sea $G = (\Sigma_N, \Sigma_T, P, \$)$ una gramática lineal por la derecha, es decir, $P \subset \Sigma_N \times (\Sigma_N.\Sigma_T \cup \Sigma_T) \cup \{\$ \longrightarrow \epsilon\}$.

Construimos una gramática $G' = (\Sigma'_N, \Sigma_T, P', \$)$ lineal por la izquierda con el siguiente algoritmo en cuatro pasos:

1. Si el símbolo inicial $\$$ de G aparece a la derecha en una producción de P, se sustitue $\$$ en dichas reglas de la siguiente manera:

   - Se introduce un nuevo símbolo no-terminal $\$'$, es decir, $\Sigma'_N = \Sigma_N \cup \{\$'\}$.
   - Por cada regla de forma $\$ \longrightarrow \alpha$ con $\alpha \in \Sigma_T.\Sigma_N \cup \Sigma_T$ se crea una nueva regla $\$' \longrightarrow \alpha$.
   - Cada regla de forma $X \longrightarrow \sigma\$$ ($X \in \Sigma_N, \sigma \in \Sigma_T$) se sustitue por $X \longrightarrow \sigma\$'$.
   - Si $\$ \longrightarrow \epsilon \in P$, se añade para cada regla $X \longrightarrow \sigma\$$ ($X \in \Sigma_N, \sigma \in \Sigma_T$) la regla $X \longrightarrow \sigma$.

   Con esas modificaciones obtenemos un nuevo sistema de producciones $\overline{P}$ y un alfabeto de variables o bien $\Sigma'_N = \Sigma_N$ o bien $\Sigma'_N = \Sigma_N \cup \{\$'\}$.

2. Se crea un grafo dirigido con las siguientes propiedades:

   - El conjunto de nodos es $\Sigma'_N \cup \{\epsilon\}$.
   - Se añade una arista entre los nodes A y B con atributo σ, si existe una regla $A \longrightarrow \sigma B$ en $\overline{P}$.
   - Se añade una arista entre los nodes A y ϵ con atributo σ, si existe una regla $A \longrightarrow \sigma$ en $\overline{P}$.
   - Se añade una arista entre los nodes $\$$ y ϵ con atributo ϵ, si existe la regla $\$ \longrightarrow \epsilon$ en $\overline{P}$.

3. Se 'inverte' el grafo, más preciso:

   - Se intercambian los nodos $\$$ y ϵ.
   - Se invierte la dirección de todas las aristas.

4. Se transforma el grafo obtenido en el conjunto de reglas P':

   - Para cada arista entre A y B con atributo α se crea una regla $A \longrightarrow B\alpha$ ($A \in \Sigma'_N, B \in \Sigma'_N \cup \{\epsilon\}$ y $\alpha \in \Sigma_T \cup \{\epsilon\}$).

Ejemplo: Partimos de la gramática

$$G = (\{\$, A\}, \{0, 1\}, \{\$ \longrightarrow \epsilon|1A, A \longrightarrow 0\$|0\}, \$)$$

1. el símbolo inicial $\$$ aparece a la derecha entonces:
   - Introducimos un nuevo símbolo no-terminal $\$'$.
   - Añadimos la regla $\$' \longrightarrow 1A$.
   - Sustituimos la regla $A \longrightarrow 0\$$ por $A \longrightarrow 0\$'$
   - siendo $\$ \longrightarrow \epsilon \in P$, añadimos la regla $A \longrightarrow 0$ (pero que ya está en P)

   Queda el sistema de producciones intermedio como

   $$\overline{P} = \{\$ \longrightarrow \epsilon|1A, A \longrightarrow 0\$'|0, \$' \longrightarrow 1A\}$$

2. El grafo reflejando dichas reglas es:
   graphldli
3. Y el grafo invertido es:
   igraphldli
4. con lo cual obtenemos el conjunto de reglas:

   $$P' = \{\$ \longrightarrow \epsilon|A0, A \longrightarrow \$'1|1, \$' \longrightarrow A0\}$$

9.3. Lema de bombeo

Siendo a^*b^* una expresión regular, podemos construir un autómata finito que acepta el lenguaje así definido, también podemos construir para cualquier $n \in \mathbb{N}$ **fijo** un autómata finito adecuado (a^nb^n sería una expresión regular extendida que define el lenguaje correspondiente que contiene una sola palabra).

Pero no podemos construir un autómata finito que acepte el lenguaje:

$$L_{ab} = \{a^nb^n \mid n \in \mathbb{N}\} = \{\epsilon, ab, aabb, aaabbb, \ldots\}$$

donde el parámetro n no es fijo, sino se quiere que haya tantas a's como b's.

¿Por qué no podemos construir tal autómata?

- asumimos que tengamos un autómata finito M con k estados que acepta L_{ab}

- anotamos los estados de M después de haber leído las palabras a^i para $i = 0, \ldots, k$ (son $k+1$ palabras)
- pues serán (usando la ampliación de la función δ):

$$\delta^*(q_0, \epsilon), \delta^*(q_0, a), \delta^*(q_0, aa), \delta^*(q_0, aaa), \ldots, \delta^*(q_0, a^k)$$

- Entonces, un estado tiene que aparecer por lo menos dos veces (se llama principio de los cajones (*pigeonhole principle*): si se quiere poner más calcetines que hay cajones en los cajones, por lo menos en un cajón acaban por lo menos dos calcetines)
- es decir: $\delta^*(q_0, a^i) = \delta^*(q_0, a^j)$ para algunos $i \neq j$
- Entonces:

$$\begin{aligned} \delta^*(q_0, a^i b^j) &= \delta^*(\delta^*(q_0, a^i), b^j) \\ &= \delta^*(\delta^*(q_0, a^j), b^j) \\ &= \delta^*(q_0, a^j b^j) \in F \end{aligned}$$

  pues, el autómata también acepta $a^i b^j, i \neq j$ que no debe hacer. ¡Una contradicción!
- Entonces asumimos mal, es decir, no existe un autómata que acepte L_{ab}, o en otras palabras, L_{ab} no es regular.

Observamos el comportamiento del siguiente autómata:

afdcpl

$$\begin{array}{lcccc} w_0 & = & 110 & & 10 \\ w_1 & = & 110 & 010 & 10 \\ w_2 & = & 110 & 010010 & 10 \\ w_3 & = & 110 & 010010010 & 10 \\ & \ldots & & & \\ w_k & = & x & y^k & z \end{array}$$

Lema (de bombeo para lenguajes regulares): Sea L un lenguaje regular (infinito). Entonces existe un $n \in \mathbb{N}$ de tal manera que cada palabra $w \in L$ con $|w| \geq n$ se puede dividir en tres partes, $w = xyz$ cumpliéndose las tres propiedades:

1. $y \neq \epsilon$
2. $|xy| \leq n$
3. para todos los $k \geq 0 : xy^k z \in L$

Comprobación (ideas principales):

- Sea L un lenguaje regular (infinito).
- Entonces existe un autómata finito determinista M que acepta L.
- Sea n el número de estados de M ($n = |Q|$).
- Sea w una palabra con $|w| \geq n$ (tal palabra existe porque L es infinito).
- Entonces se visita un estado de M por lo menos dos veces.
- Escogemos el estado que se visita la primera vez dos veces, le llamamos q.
- La parte de w que se lee hasta llegar la primera vez a q llamamos x (puede ser que $x = \epsilon$).
- La parte de w que se lee hasta llegar la segunda vez a q llamamos y ($y \neq \epsilon$ porque un bucle en un AFD tiene aristas con símbolos).
- La longitud $|xy| \leq n$ porque hemos recorrido un camino dondo solo un estado aparece dos veces.
- La parte que sobra para terminar aceptando w llamamos z.
- Entonces dividimos w en tres partes, es decir, $w = xyz$.
- M acepta tanto xz, como xyz, como cualquier xy^kz para todos los $k \geq 0$ porque podemos recorrer el bucle de y tantas veces como queremos (esto se debe comprobar formalmente con inducción).

Entonces, comprobamos de nuevo que L_{ab} no es regular, ahora usando el lema de bombeo:

- Asumimos que L_{ab} sea regular.
- El lema de bombeo nos garantiza la existencia de un n tal que se cumplen las propiedades para palabras w con $|w| \geq n$.
- (Pensamos...): Elegimos $w = a^nb^n$. Obviamente $w \in L_{ab}$ y $|w| = 2n \geq n$.
- El lema de bombeo nos garantiza la existencia de una partición $w = xyz$ con $y \neq \epsilon$, $|xy| \leq n$, y $\forall k \geq 0 : xy^kz \in L_{ab}$. (No conocemos la partición en concreto, pero sus propiedades.)
- Porque $|xy| \leq n$ el prefijo xy no contiene ninguna b.
- Porque $y \neq \epsilon$ la subpalabra y contiene por lo menos una a.

- Todos las bs están en z.
- Tanto $xz = xy^0z \in L_{ab}$ como $xy^1z \in L_{ab}$ pero xz contiene por lo menos una a menos que xyz (hemos quitado y).
- Eso es una contradicción porque xz no debe estar dentro L_{ab}.
- Entonces L_{ab} no puede ser regular.

Receta para el uso del lema de bombeo:

- Dado un lenguaje L.
- Queremos comprobar que L *no* es regular.
- Comprobación con contradicción.
- Asumimos que L sea regular.
- El lema de bombeo garantiza la existencia de un n (pero no lo conocemos).
- Buscamos $w \in L$ (con un poco de sabiduría) que depende de n, tal que $|w| \geq n$
- El lema de bombeo garantiza la existencia de la partición de $w = xyz$ cumpliendo las 3 propiedades.
- Comprobamos (con un poco de sabiduría) que, independiente de la partición de w en concreto (en los límites de las primeras dos propiedades), se produce una contradicción con la tercera propiedad.

Podemos describir dicha 'receta' también como un juego para dos personas:

- Juego para un lenguaje L.
- Jugador 1 selecciona n.
- Jugador 2 selecciona $w \in L$ con $|w| \geq n$.
- Jugador 1 selecciona partición $w = xyz$ con $y \neq \epsilon$ y $|xy| \leq n$.
- Jugador 2 selecciona k.
- si $xy^kz \notin L$ gana jugador 2, sino gana jugador 1.
- si jugador 2 puede ganar siempre, entonces L no es regular.

El lema de bombeo es el jugador 1, ¿quién es el jugador 2?

Otro **Ejemplo:**

$$L_{quad} = \{0^m \mid m \text{ es número cuadrado}\}$$

es decir, todas las cadenas de ceros que tienen un número cuadrado de símbolos.

- Asumimos que L_{quad} sea regular.
- El lema de bombeo nos garantiza la existencia de un n tal que se cumplen las propiedades para palabras w con $|w| \geq n$.
- (Pensamos...): Elegimos $w = 0^{n^2}$. Obviamente $w \in L_{quad}$ y $|w| = n^2 \geq n$.
- El lema de bombeo nos garantiza la existencia de una partición $w = xyz$ con $y \neq \epsilon$, $|xy| \leq n$, y $\forall k \geq 0 : xy^kz \in L_{quad}$. (No conocemos la partición en concreto, pero sus propiedades.)
- Tanto $xyz \in L_{quad}$ como $xy^2z \in L_{quad}$ con $y \neq \epsilon$.
- Tenemos entonces:

$$\begin{aligned} n^2 &= |xyz| \quad \text{porque es } w \\ &< |xy^2z| \quad \text{porque } y \text{ tiene una longitud} > 0 \\ &= |xyz| + |y| \\ &\leq n^2 + n \quad \text{porque si } |xy| \leq n \text{ también } |y| \leq n \\ &< n^2 + 2n + 1 \\ &= (n+1)^2 \end{aligned}$$

- Eso es una contradicción porque xy^2z no puede ser una palabra cuya longitud es un número cuadrado entre dos números cuadrados consecutivos.
- Entonces L_{quad} no puede ser regular.

Dos comentarios más:

- Este lema de bombeo solo garantiza una propiedad para lenguajes regulares, es decir, todos los lenguajes regulares (infinitos) la tienen, pero pueden existir más lenguajes que la tengan, o en otras palabras, pueden existir lenguajes L donde encontramos tal n y la división de w en xyz con todas las propiedades, pero L no es regular.

- Con el lema de bombeo también se puede derivar: si tal $w \notin L$ entonces $xy^kz \notin L$ (el argumento es fácil: no hace falta que lleguemos a una estado final en la comprobación, lo importante eran los caminos recorridos).

Reglas de mano:

- Un lenguaje es regular si independientemente de la longitud de la palabra de entrada, hay que memorizar solamente una cantidad constante de información (en el caso de L_{ab} deberíamos memorizar el número de a's que no es constante).
- La comprobación formal se realiza con el lema de bombeo.
- El lema de bombeo se usa para comprobar que un lenguaje *no* es regular, para comprobar que es regular, por ejemplo, se construye un autómata finito, una expresión regular, o una grámatica de tipo 3.

Pero ojo, existen lenguajes regulares que tienen que ver con números:

$$L_{tres} = \{w \mid w \text{ es codificación de un número divisible por 3}\}$$

Ejercicio: contruye un autómata finito que acepte L_{tres}.

10. Propiedades, algoritmos de decisión, y aplicaciones para lenguajes regulares

La clase de los lenguajes regulares es una clase de alguna manera muy 'robusta': hay muchas posibilidades de describir los lenguajes y exhiben un gran número de propiedades de clausura, como vemos ahora.

10.1. Propiedades de lenguajes regulares

Sean L_1 y L_2 dos lenguajes regulares.

Unión: $L = L_1 \cup L_2$ es regular, porque podemos construir una expresión regular para L, teniendo las expresiones regulares para L_1 y L_2, más preciso: con $L_1 = L(\alpha)$ y $L_2 = L(\beta)$ tenemos $L = L((\alpha + \beta))$

Concatención: $L = L_1.L_2$ es regular, porque podemos construir una expresión regular para L, teniendo las expresiones regulares para L_1 y L_2, más preciso: con $L_1 = L(\alpha)$ y $L_2 = L(\beta)$ tenemos $L = L(\alpha\beta)$

Clausura: $L = L_1^*$ es regular, porque podemos construir una expresión regular para L, teniendo la expresión regular para L_1, más preciso: con $L_1 = L(\alpha)$ tenemos $L = L((\alpha)^*)$

Complemento: $L = \overline{L_1} = \Sigma^* - L_1$ es regular, porque podemos construir, dado un AFD completo M_1 que acepta L_1, un AFD M que acepta L simplemente 'invertiendo' sus estados finales, es decir, los estados no finales de M_1 serán los estados finales de M y los finales se convierten en los no finales, entonces, si $M_1 = (\Sigma, Q, \delta, q_0, F)$ construimos $M = (\Sigma, Q, \delta, q_0, Q - F)$.

Intersección: $L = L_1 \cap L_2$ es regular, porque con las reglas de DeMorgan obtenemos $L = L_1 \cup L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$. Complemento y unión producen lenguajes regulares, como visto antes. Dicha construcción es bastante laborosa, abajo vemos una construcción directa y simple.

Diferencia: $L = L_1 - L_2$ es regular, porque se puede expresar la diferencia como $L = L_1 - L_2 = L_1 \cap \overline{L_2} = L_1 \cap (\Sigma^* - L_2)$ y las operaciones usadas mantienen la regularidad.

En vez de usar la lógica booleana, es decir, aplicando las reglas de DeMorgan, se puede construir directamente un autómata que acepta el lenguaje $L = L_1 \cap L_2$.

La idea principal es, simular en paralelo en un solo autómata (digamos autómata de producto) las transiciones de los dos autómatas (por ejemplo finitos deterministas y completas) para L_1 y L_2.

Entonces sean $M_1 = (\Sigma_1, Q_1, \delta_1, q_1, F_1)$ y $M_2 = (\Sigma_2, Q_2, \delta_2, q_2, F_2)$ los dos AFDs completos que aceptan L_1 y L_2, es decir, $L_1 = L(M_1)$ y $L_2 = L(M_2)$.

Construimos el AFD completo M que acepta $L = L_1 \cap L_2 = L(M)$ como

$$M = (\Sigma, Q, \delta, q_0, F)$$

donde

- asumimos que $\Sigma = \Sigma_1 = \Sigma_2$, es decir, usamos solamente los símbolos comunes. Es fácil eliminar en M_1 y en M_2 todas las dependencias de símbolos superflues antemano en caso que haya.
- $Q = Q_1 \times Q_2$, es decir, el producto cartesiano de los estados de M_1 y M_2.
- δ es la función de transición con

  $$\delta((p, q), \sigma) = (\delta_1(p, \sigma), \delta_2(q, \sigma))$$

  para $p \in Q_1$, $q \in Q_2$ y $\sigma \in \Sigma$.

- $q_0 = (q_1, q_2)$, es decir, la pareja de los dos estados iniciales
- $F = F_1 \times F_2$, es decir, todas las parejas donde ambos estados son estados finales de ambos autómatas.

Ejemplo:

afdprod

Obviamente la construcción funciona igual con autómatas finitos no–deterministas (AFND).

Homomorfismo: a lo mejor lo incluyo.

10.2. Algoritmos de decisión de lenguages regulares

Pertenencia: ¿$w \in L$? sí, se puede contestar la pregunta (es decir, es un problema computable), porque

- construimos un AFD M que acepte L
- simulamos su comportamiento leyendo la palabra w
- si acabamos en un estado final, w está en L, sino w no está en L

Vaciedad: ¿$L = \emptyset$? sí, se puede contestar la pregunta (es decir, es un problema computable) porque

- construimos un autómata que acepte L
- analizamos el grafo del autómata para averiguar si existe un camino desde el estado incial hacia un estado final (dicho proceso resulta más fácil, si se añade un estado y se conecta todos los estados finales a este estado, así basta buscar un camino entre el estado inicial y el estado añadido)
- si existe tal camino, entonces L no es vacío, sino L es vacío lrempty

Cardinalidad: ¿$|L| < \infty$? sí, se puede contestar la pregunta (es decir, es un problema computable) porque

- construimos un autómata que acepte L
- analizamos el grafo del autómata para averiguar si existe un ciclo en un camino entre el estado inicial y algún estado final
- si existe tal ciclo, entonces L es infinito, sino L es finito

Igualidad: ¿$L_1 = L_2$? sí, se puede contestar la pregunta (es decir, es un problema computable) porque

- construimos los autómatas finitos deterministas y mínimos que acepten L_1 y L_2
- comparamos la estructura de los grafos de ambos autómatas
- si son iguales, entonces ambos lenguajes son iguales, sino son diferentes
  (**Nota:** comparar dos grafos generales y decidir si son iguales ,es decir, el problema de isomorfía de grafos, es un problema que se puede calcular, pero todavía no se conoce el algoritmo óptimo. Para los AFD mínimos se reduce la complejidad porque las aristas llevan atributos y se conoce ambos estados iniciales que tengan que coincidir.)

10.3. Aplicaciones para lenguajes regulares

Análisis sintáctico: El trabajo de un compilador es la traducción de algún código fuente (escrito en un lenguaje de programacón, es decir, el código fuente no es nada más que una palabra (larga) sobre el alfabeto de los símbolos permitidos) en una serie de instrucciones para un procesador.

En la primera fase, el compilar analiza lexicográficamente el texto, es decir, transforma con la ayuda de autómatas finitos, el texto continuo en una secuencia de entidades, llamadas *token*, por ejemplo, palabras claves, valores numéricos, operadores, comentarios, etc.

Existen herramientas para el desarrollador que ayudan en la construcción de dichos autómatas, cuyas entradas suelen ser expresiones regulares y código de acción y cuyas salidas son programas que realizan la tarea del análisis lexicográfico. Un ejemplo es `lex` para un sistema Unix.

Búsqueda de palabras en texto: Existen herramientas para el desarrollador que ayudan en la búsqueda de secuencias de texto descritas por expresiones regulares. Un ejemplo es `grep` para un sistema Unix o comandos internos a editores como el `vi` o `emacs`.

Diagramas de estados en UML: El lenguaje UML (*unified modeling language*) se usa para la descripción durante el proceso de desarrollo de software. El lenguaje gráfico usa diferentes tipos de diagramas para este fin. Uno de ellos son los diagramas de estados que visualizan el cambio de estados de los objetos debido al paso de mensajes entre ellos.

11. Lenguajes libres de contexto

Ya hemos visto los siguientes lenguajes:

$$\begin{aligned}
L_{ab} &= \{a^n b^n \mid n \geq 0\} \\
L_{abc} &= \{a^n b^n c^n \mid n \geq 0\} \\
L_{pal} &= \{w \mid w \in \{0,1\}^*, w = vv^R\} \\
L_{dup} &= \{w \mid w \in \{0,1\}^*, w = vv\} \\
L_{quad} &= \{0^{n^2} \mid n \text{ número cuadrado}\} \\
L_{prim} &= \{0^n \mid n \text{ número primo}\} \\
L_{()} &= \{w \mid w \in \{(,)\}^*, w \text{ correcto}\}
\end{aligned}$$

Una gramática *libre de contexto* es una cuádrupla

$$G = (\Sigma_N, \Sigma_T, P, \$)$$

donde

- Σ_N es un alfabeto de símbolos no-terminales (o variables)
- Σ_T es un alfabeto de símbolos terminales (o constantes)
- $P \subset \Sigma_N \times (\Sigma_N \cup \Sigma_T)^+ \cup \{\$ \longrightarrow \epsilon\}$ es un sistema de producciones (o reglas)
- $\$ \in \Sigma_N$ es el símbolo inicial

Es decir, la definición de las gramáticas libres de contexto nos da mucha libertad para el sistema de producciones.

Por eso (y también para otros objetivos como por ejemplo mostrar que existe un tipo de autómata que justamente acepta lenguajes libres de contexto como veremos en adelante) se ha desarrollado formas normales de la representación de gramáticas libres de contexto, es decir, se transforma el sistema de producciones de la gramática de tal manera que no se varía el lenguaje generado pero las reglas tengan cierta propiedad.

Especialmente la definición arriba exluye reglas de forma $X \longrightarrow \epsilon$ siendo X un símbolo no-terminal diferente a $, sin embargo, si permitesemos tales producciones, es decir, permitir $P \subset \Sigma_N \times (\Sigma_N \cup \Sigma_T)^*$, obtendríamos los mismos lenguajes, porque, como veremos a continuación, dichas producciones se pueden eliminar sin cambiar el lenguaje que genera la gramática.

11.1. Forma Normal de Chomsky

Sea $G = (\Sigma_N, \Sigma_T, P, \$)$ una gramática con $P \subset \Sigma_N \times (\Sigma_N \cup \Sigma_T)^*$ y $X \in \Sigma_N$ un símbolo no-terminal (o una variable). Podemos clasificar tales símbolos X en tres clases:

variables acesibles: si existe una derivación desde el símbolo inicial que contiene X, es decir, existe $\$ \longrightarrow^* \alpha X \beta$ donde $\alpha, \beta \in \Sigma^*$.

variables generativas: si existe una derivación desde el la variable que produce una sentencia w, es decir, existe $X \longrightarrow^* w$ donde $w \in \Sigma_T^*$.

variables útiles: si existe una derivación desde el símbolo inicial usando X que produce una sentencia w, es decir, existe $\$ \longrightarrow^* \alpha X \beta \longrightarrow^* w$ donde $\alpha, \beta \in \Sigma^*$ y $w \in \Sigma_T^*$.

Una **gramática está en forma normal de Chomsky** (FNC)

- si G (es decir, su Σ_N) solamente contiene variables útiles y
- si todas las producciones de G (es decir, en su P) son
  - o bien de la forma $X \longrightarrow YZ$ con $X, Y, Z \in \Sigma_N$
  - o bien de la forma $X \longrightarrow \sigma$ con $X \in \Sigma_N$ y $\sigma \in \Sigma_T$
- si $\$$ (es decir, el símbolo inicial de G) no aparece al lado derecho de ninguna producción, también está permitido que $\$ \longrightarrow \epsilon \in P$.

La tercera condición es necesaria para poder derivar ϵ. Si $\$$ aparece a la derecha, primero habrá que sustituir las producciones implicadas adecuadamente como lo vimos en la conversión de una gramática lineal por la derecha a una gramática lineal por la izquierda.

Observamos:

- la primera condición garantiza que todas las variables son necesarias para derivar por lo menos una sentencia.
- la segunda condición garantiza que un árbol de derivación es un árbol binario.

Obviamente cualquier gramática en forma normal de Chomsky es una gramática libre de contexto que se verifica directamente analizando la forma de producciones permitidas.

Pero también es valido la otra dirección: para cualquier lenguaje libre de contexto existe una gramática en forma normal de Chomsky, que genera el mismo lenguaje.

La comprobación de este hecho detallamos con la siguiente construcción donde a partir de una gramática libre de contexto dada, elaboramos una nueva gramática en forma normal de Chomsky.

Sea L un lenguaje libre de contexto y $G = (\Sigma_N, \Sigma_T, P, \$)$ una gramática que genere L (es decir $L = L(G)$).

La construcción sigue 5 pasos (asumimos que $\epsilon \notin L$, eso remediamos al final):

1. eliminamos las variables inútiles
2. modificamos las reglas para que no haya mezcla de variables y constantes en las partes derechas de las producciones y para que todas las reglas con constantes tengan la forma $X \longrightarrow \sigma$
3. sustituimos las reglas cuya longitud de su parte derecha es > 2
4. sustituimos las reglas de tipo $X \longrightarrow \epsilon$
5. sustituimos las reglas de tipo $X \longrightarrow Y$, las reglas unitarias.

Las gramáticas después de cada paso llamamos $G = G_0, G_1, G_2, \ldots, G_5 = G_{FNC}$ respectivamente.

Usamos la siguiente gramática inicial

$$G_0 = (\{\$, A, B, C, D, E, F\}, \{a, b, c\}, P_0, \$)$$

donde P_0 contenga las siguientes producciones:

$$\begin{array}{rcl} \$ & \longrightarrow & bDD \mid Ca \mid bc \\ A & \longrightarrow & B \mid aCC \mid baD \\ B & \longrightarrow & cBD \mid \epsilon \mid AC \\ C & \longrightarrow & bD \mid aBA \\ D & \longrightarrow & CD \mid a \mid EF \\ E & \longrightarrow & Eb \\ F & \longrightarrow & a \end{array}$$

como ejemplo para realizar todos los pasos.

1. Sabiendo que una variable es inútil si es no-generativa o inacesible realizamos dos subpasos:

   a) eliminamos primero las variables no-generativas N (y todas las reglas con ellas) llamando a la gramática resultante G'_1,

   b) eliminamos después las variables inacesibles I (y todas las reglas con ellas).

   Para ello recorremos en forma estructurada las variables y reglas:

a) para calcular N empezamos con aquellas variables que producen directamente sentencias (incluyendo ϵ) y seguimos el uso de reglas con dichas variables para producir así sucesivamente sentencias (o en otras palabras: 'seguimos las reglas desde el lado derecho hacia el lado izquierdo para obtener así la información sobre las variables').
Una forma de realizar dicho recorrido es empezar con $N = \Sigma_N$ y borrar del conjunto todas aquellas variables que o bien directamente deriven una sentencia o bien lo hacen indirectamente.
Se observa que solamente E es un símbolo no-generativo, es decir, $N = \{E\}$, P_1' entonces es:

$$\begin{array}{rcl} \$ & \longrightarrow & bDD \mid Ca \mid bc \\ A & \longrightarrow & B \mid aCC \mid baD \\ B & \longrightarrow & cBD \mid \epsilon \mid AC \\ C & \longrightarrow & bD \mid aBA \\ D & \longrightarrow & CD \mid a \\ F & \longrightarrow & a \end{array}$$

b) para calcular I empezamos con el símbolo inicial y veremos a cuales de las variables se puede llegar directamente y seguimos el uso de reglas con dichas variables para llegar así sucesivamente a nuevas variables (o en otras palabras: 'seguimos las reglas para obtener así la información sobre las variables acesibles'). Dicho algoritmo es una exploración de un grafo de dependencia parecido al algoritmo que vimos para detectar estados no-acesibles en un autómata finito.
Se observa que solamente F es un símbolo inacesible, es decir, $I = \{F\}$, P_1 entonces es:

$$\begin{array}{rcl} \$ & \longrightarrow & bDD \mid Ca \mid bc \\ A & \longrightarrow & B \mid aCC \mid baD \\ B & \longrightarrow & cBD \mid \epsilon \mid AC \\ C & \longrightarrow & bD \mid aBA \\ D & \longrightarrow & CD \mid a \end{array}$$

Entonces G_1 no contiene símbolos inútiles.

2. Añadimos para cada símbolo terminal σ una regla W_σ y sustituimos σ en todas las reglas de P_1, P_2 entonces es:

$$\begin{array}{rcl} \$ & \longrightarrow & W_bDD \mid CW_a \mid W_bW_c \\ A & \longrightarrow & B \mid W_aCC \mid W_bW_aD \\ B & \longrightarrow & W_cBD \mid \epsilon \mid AC \end{array}$$

$$\begin{aligned} C &\longrightarrow W_bD \mid W_aBA \\ D &\longrightarrow CD \mid W_a \\ W_a &\longrightarrow a \\ W_b &\longrightarrow b \\ W_c &\longrightarrow c \end{aligned}$$

Entonces P_2 solamente contiene reglas con partes derechas siendo ϵ, un símbolo terminal, o una palabra de variables.

3. Sustituimos cada regla del tipo $X \longrightarrow Y_1Y_1 \ldots Y_k$ con $k > 2$ por las reglas:

$$\begin{aligned} X &\longrightarrow Y_1X_1 \\ X_1 &\longrightarrow Y_2X_2 \\ \vdots &\longrightarrow \vdots \\ X_{k-3} &\longrightarrow Y_{k-2}X_{k-2} \\ X_{k-2} &\longrightarrow Y_{k-1}Y_k \end{aligned}$$

donde las X_i son nuevas variables, P_3 entonces es:

$$\begin{aligned} \$ &\longrightarrow W_b\$_1 \mid CW_a \mid W_bW_c \\ \$_1 &\longrightarrow DD \\ A &\longrightarrow B \mid W_aA_1 \mid W_bA_2 \\ A_1 &\longrightarrow CC \\ A_2 &\longrightarrow W_aD \\ B &\longrightarrow W_cB_1 \mid \epsilon \mid AC \\ B_1 &\longrightarrow BD \\ C &\longrightarrow W_bD \mid W_aC_1 \\ C_1 &\longrightarrow BA \\ D &\longrightarrow CD \mid W_a \\ W_a &\longrightarrow a \\ W_b &\longrightarrow b \\ W_c &\longrightarrow c \end{aligned}$$

Entonces P_3 solamente contiene reglas con partes derechas siendo ϵ, un símbolo terminal, o una palabra de una o dos variables.

4. Eliminamos las reglas que producen ϵ, ¡ojo! tenemos que distinguir entre variables que solamente producen ϵ y aquellas que también producen ϵ.

   Entonces, el paso se realiza en 3 partes:

   - Calculamos los conjuntos de variables
     $E = \{V \mid V \longrightarrow^* \epsilon\}$ (las variables que posiblemente producen ϵ) y
     $E_\epsilon = \{V \mid V \longrightarrow^* \epsilon$ y no existe $V \longrightarrow^* w$ con $w \neq \epsilon\} \subset E$ (las variables que solo producen ϵ).

     Se calculan los conjuntos aplicando el mismo algoritmo que usamos en el primer paso para detectar variables no-generativas.
   - Añadimos para cada regla del tipo $X \longrightarrow YZ$
     - $X \longrightarrow Y$ si $Y \notin E_\epsilon$ y $Z \in E$
     - $X \longrightarrow Z$ si $Y \in E$ y $Z \notin E_\epsilon$.
   - Eliminamos
     - todas las reglas de tipo $X \longrightarrow \epsilon$,
     - todas las reglas de tipo $X \longrightarrow Y$ con $Y \in E_\epsilon$, y
     - todas las reglas de tipo $X \longrightarrow YZ$ con $Y, Z \in E_\epsilon$.

   En el ejemplo tenemos: $E = \{A, B, C_1\}, E_\epsilon = \emptyset$, y por eso P_4 es:

$$
\begin{array}{rcl}
\$ & \longrightarrow & W_b\$_1 \mid CW_a \mid W_bW_c \\
\$_1 & \longrightarrow & DD \\
A & \longrightarrow & B \mid W_aA_1 \mid W_bA_2 \\
A_1 & \longrightarrow & CC \\
A_2 & \longrightarrow & W_aD \\
B & \longrightarrow & W_cB_1 \mid AC \mid C \\
B_1 & \longrightarrow & BD \mid D \\
C & \longrightarrow & W_bD \mid W_aC_1 \mid W_a \\
C_1 & \longrightarrow & BA \mid A \mid B \\
D & \longrightarrow & CD \mid W_a \\
W_a & \longrightarrow & a \\
W_b & \longrightarrow & b \\
W_c & \longrightarrow & c
\end{array}
$$

5. Para eliminar las reglas unitarias, es decir, reglas de tipo $X \longrightarrow Y$ procedemos:

   - Calculamos el conjunto de las reglas unitarias $U = \{(X, Y) \mid X \longrightarrow^* Y\}$ (¡ojo! no basta con $(X, Y) \in U$ si $X \longrightarrow Y$, hay que calcular la clausura transitiva). Dicho

cálculo se realiza de forma parecida como lo vimos para el cálculo de la clausura transitiva de las transiciones ϵ en los AFND-ϵ.

Partimos del conjunto que contiene todas las reglas unitarias del sistema de producciones, es decir

$$U_1 = \{(X,Y) | X \longrightarrow Y \in P\}$$

Después construimos U_2 insertando para cada par de parejas $(X,Y),(Y,Z) \in U_1$ la pareja (X,Z), o en general, construimos U_i insertando para cada par de parejas $(X,Y),(Y,Z) \in \bigcup_{j<i} U_j$ la pareja (X,Z).

Seguimos con el procedimiento hasta que encontramos U_i vacía para cierto i, es decir, no se ha añadido nada más. (El índice i de U_i nos indica con cuantos 'saltos' como mucho podemos llegar.) El algoritmo termina (porque el número de producciones es finito) y finalmente obtenemos

$$U = \bigcup_{i=1}^{\infty} U_i$$

Nota: El cálculo de la clausura transitiva en un grafo dirigido es una operación que se necesita en muchos ámbitos de la informática (por ejemplo en bases de datos relacionadas). Nosotros ya lo usamos en el contexto de los autómatas finitos. El algoritmo especificado arriba tiene un tiempo de ejecución cuadrático en el número de producciones iniciales. Los mejores algoritmos tienen un comportamiento de $O(ne+n+e)$ en tiempo de ejecución (donde n es el número de nodos y e el número de aristas en el grafo), pero algunos de ellos se comportan en casos prácticos linealmente respecto al tamaño de entrada (número de arista).

- Para cada $(X,Y) \in U$ y para cada regla $Y \longrightarrow \alpha$ que no es regla unitaria, añadimos una regla $X \longrightarrow \alpha$.
- Eliminamos todas las reglas unitarias y controlamos que aquellas variables que aparecen a la derecha de las parejas siguen siendo acesibles.

$$\begin{aligned}
\$ &\longrightarrow W_b\$_1 \mid CW_a \mid W_bW_c \\
\$_1 &\longrightarrow DD \\
A &\longrightarrow B \mid W_aA_1 \mid W_bA_2 \\
A_1 &\longrightarrow CC \\
A_2 &\longrightarrow W_aD \\
B &\longrightarrow W_cB_1 \mid AC \mid C \\
B_1 &\longrightarrow BD \mid D \\
C &\longrightarrow W_bD \mid W_aC_1 \mid W_a \\
C_1 &\longrightarrow BA \mid A \mid B \\
D &\longrightarrow CD \mid W_a
\end{aligned}$$

$$\begin{aligned} W_a &\longrightarrow a \\ W_b &\longrightarrow b \\ W_c &\longrightarrow c \end{aligned}$$

En el ejemplo tenemos:

$$\begin{aligned} U_1 &= \{(A,B),(B,C),(B_1,D),(C,W_a),(C_1,A),(C_1,B),(D,W_a)\} \\ U_2 &= \{(A,C),(B,W_a),(B_1,W_a),(C_1,C)\} \\ U_3 &= \{(A,W_a),(C_1,W_a)\} \\ U_4 &= \emptyset \end{aligned}$$

y por eso P_5, el sistema de producciones final, queda en:

$$\begin{aligned} \$ &\longrightarrow W_b\$_1 \mid CW_a \mid W_bW_c \\ \$_1 &\longrightarrow DD \\ A &\longrightarrow W_cB_1 \mid AC \mid W_aA_1 \mid W_bA_2 \mid W_bD \mid W_aC_1 \mid a \\ A_1 &\longrightarrow CC \\ A_2 &\longrightarrow W_aD \\ B &\longrightarrow W_cB_1 \mid AC \mid W_bD \mid W_aC_1 \mid a \\ B_1 &\longrightarrow BD \mid CD \mid a \\ C &\longrightarrow W_bD \mid W_aC_1 \mid a \\ C_1 &\longrightarrow BA \mid W_aA_1 \mid W_bA_2 \mid W_cB_1 \mid AC \mid W_bD \mid W_aC_1 \mid a \\ D &\longrightarrow CD \mid a \\ W_a &\longrightarrow a \\ W_b &\longrightarrow b \\ W_c &\longrightarrow c \end{aligned}$$

Observamos en la construcción:

- En ningún paso hemos añadido variables inútiles.
- Si hemos borrado reglas, hemos asegurado que todas las variables siguen siendo útiles.
- Después de cada paso la gramática resultante genera el mismo lenguaje, es decir, $L(G_0) = L(G_1) = \ldots = L(G_5)$.
- Como se observa, la gramática G_5 es en forma normal de Chomsky.

Si el lenguaje de partida L contiene la palabra vacía ($\epsilon \in L$) entonces se detecta en el pasa 4 que el símbolo inicial pertenece a E (o incluso a E_ϵ), en este caso eliminamos con un nuevo símbolo, por ejemplo $\$'$, la aparencia de $\$ en los lados derechos y añadimos la regla $\$ \longrightarrow \epsilon$. Tal gramática sigue estando en forma normal de Chomsky y genera L.

Notas:

- El cálculo de los conjuntos N, I, E, E_ϵ, y U que se necesitan para sucesivamente modificar los sistemas de producciones se realiza con un recorrido estructurado sobre las variables y producciones.
- Dado que durante el proceso hemos eliminado producciones, puede ser que también en el alfabeto de los símbolos terminales Σ_T hay **símbolos superfluos**, es decir, que no se pueden producir con las producciones restantes. Dichos símbolos se pueden borrar de Σ_T sin que se cambie el lenguaje generado.
- Cuando eliminamos las reglas unitarias hemos eliminado implíctamente las reglas innecesarias de tipo $X \longrightarrow X$ que también se podría borrar ya antemano en un paso previo.
- Existen otras fuentes que primero realizan la eliminación de las reglas nulas y de las reglas unitarias antes de demezclar y reducir las partes derechas de las reglas. Eso es posible pero el cálculo de E y E_ϵ es más complejo y las reglas de ampliación y eliminación no se limitan a dos, respectivamente tres, casos simples como descritos arriba en el paso 4.

11.2. Forma Normal de Greibach

Veremos otra posible normalización de gramáticas que nos sirve más adelante para construir cierto tipo de autómatas.

Una **gramática es en forma normal de Greibach** (FNG) si

- G (es decir, su Σ_N) solamente contiene variables útiles
- todas las producciones de G (es decir, en su P) son de la forma $X \longrightarrow \sigma\Upsilon$ donde $\sigma \in \Sigma_T$ y $\Upsilon \in \Sigma_N^*$, es decir, todas las reglas tienen como primer símbolo en sus partes derechas un símbolo terminal que es seguido por una palabra de variables.
- (porque así no se podría derivar ϵ) si $\$ (es decir, el símbolo inicial de G) no aparece al lado derecho de una producción, también está permitido que $\$ \longrightarrow \epsilon \in P$

Obviamente cualquier gramática en forma normal de Greibach es una gramática libre de contexto que se verifica directamente analizando la forma de producciones permitidas.

Una interesante propiedad es: para cualquier lenguaje libre de contexto existe una gramática en forma normal de Greibach, que genera el lenguaje.

La comprobación de este hecho detallamos con la siguiente construcción, donde a partir de una gramática libre de contexto dada elaboramos una nueva gramática en forma normal de Greibach.

Sea L un lenguaje libre de contexto y $G = (\Sigma_N, \Sigma_T, P, \$)$ una gramática que genere L (es decir $L = L(G)$).

La construcción sigue 4 pasos (asumimos que $\epsilon \notin L$, eso remediamos al final):

1. construimos una gramática equivalente en forma normal de Chomsky
2. sustituimos las reglas recursivas a la izquierda, es decir, reglas de tipo $X \longrightarrow XY$; eso puede generar reglas con lados derechos mÃąs largos que dos
3. establecemos un orden en las variables, es decir $\Sigma_N = \{X_1, X_2, \ldots, X_n\}$ de tal manera que todas las reglas serán de tipo $X_i \longrightarrow X_j \Upsilon$ con $i < j$, $\Upsilon \in \Sigma_N^+$
4. sustituimos las reglas que no tengan un símbolo terminal como primer símbolo en su parte derecha.

Las gramáticas después de cada paso llamamos $G = G_0, G_1, G_2, \ldots, G_4 = G_{FNG}$ respectivamente.

Usamos la misma gramática inicial como en el apartado anterior

$$G_0 = (\{\$, A, B, C, D, E, F\}, \{a, b, c\}, P_0, \$)$$

donde P_0 contenga las siguientes producciones:

$$\begin{aligned}
\$ &\longrightarrow bDD \mid Ca \mid bc \\
A &\longrightarrow B \mid aCC \mid baD \\
B &\longrightarrow cBD \mid \epsilon \mid AC \\
C &\longrightarrow bD \mid aBA \\
D &\longrightarrow CD \mid a \mid EF \\
E &\longrightarrow Eb \\
F &\longrightarrow a
\end{aligned}$$

como ejemplo para realizar todos los pasos.

1. La transformación a FNC hicimos arriba. Entonces ya tenemos P_1 como

$$\begin{aligned}
\$ &\longrightarrow CW_a \mid W_b\$_1 \mid W_bW_c \\
\$_1 &\longrightarrow DD \\
A &\longrightarrow AC \mid W_cB_1 \mid W_aA_1 \mid W_bA_2 \mid W_bD \mid W_aC_1 \mid a \\
A_1 &\longrightarrow CC
\end{aligned}$$

$$
\begin{aligned}
A_2 &\longrightarrow W_aD \\
B &\longrightarrow AC \mid W_cB_1 \mid W_bD \mid W_aC_1 \mid a \\
B_1 &\longrightarrow BD \mid CD \mid a \\
C &\longrightarrow W_bD \mid W_aC_1 \mid a \\
C_1 &\longrightarrow AC \mid BA \mid W_aA_1 \mid W_bA_2 \mid W_cB_1 \mid W_bD \mid W_aC_1 \mid a \\
D &\longrightarrow CD \mid a \\
W_a &\longrightarrow a \\
W_b &\longrightarrow b \\
W_c &\longrightarrow c
\end{aligned}
$$

solo reordenado, para que aparezcan las partes derechas con variables al principio al comienzo de las listas.

2. Para cada producción recursiva a la izquierda, es decir, regla de tipo $X \longrightarrow X\alpha$ con $X \in \Sigma_N$ y $\alpha \in \Sigma$ se realiza los siguientes 3 pasos:

   - se sustitue $X \longrightarrow X\alpha$ por $X \longrightarrow \alpha Y$ siendo Y una nueva variable
   - se añade las reglas $Y \longrightarrow \alpha Y \mid \alpha$
   - para cada regla $X \longrightarrow \beta$ se añade $X \longrightarrow \beta Y$ si β no comienza con X

   En P_1 hay una regla recursiva a la izquierda: $A \longrightarrow AC$. Entonces, la sustituimos por $A \longrightarrow CA_3$, añadimos $A_3 \longrightarrow CA_3 \mid C$ y añadimos las demás reglas para A, y resulta el conjunto P_2:

$$
\begin{aligned}
\$ &\longrightarrow CW_a \mid W_b\$_1 \mid W_bW_c \\
\$_1 &\longrightarrow DD \\
A &\longrightarrow CA_3 \mid W_cB_1 \mid W_aA_1 \mid W_bA_2 \mid W_bD \mid W_aC_1 \mid a \mid \\
&\quad\ W_cB_1A_3 \mid W_aA_1A_3 \mid W_bA_2A_3 \mid W_bDA_3 \mid W_aC_1A_3 \mid aA_3 \\
A_1 &\longrightarrow CC \\
A_2 &\longrightarrow W_aD \\
A_3 &\longrightarrow CA_3 \mid C \\
B &\longrightarrow AC \mid W_cB_1 \mid W_bD \mid W_aC_1 \mid a \\
B_1 &\longrightarrow BD \mid CD \mid a \\
C &\longrightarrow W_bD \mid W_aC_1 \mid a \\
C_1 &\longrightarrow AC \mid BA \mid W_aA_1 \mid W_bA_2 \mid W_cB_1 \mid W_bD \mid W_aC_1 \mid a \\
D &\longrightarrow CD \mid a \\
W_a &\longrightarrow a \\
W_b &\longrightarrow b
\end{aligned}
$$

$$W_c \longrightarrow c$$

Entonces las reglas en P_2 tienen de nuevo diferentes longitudes en sus partes derechas (incluso puede ser que haya reglas unitarias).

3. (por incluir)
4. (por incluir)

Dado que con una gramática en forma normal de Greibach se genera con cada producción exactamente un símbolo terminal, cada palabra derivable con tal gramática tiene una derivación igual a la longitud de la palabra.

Ojo, eso no significa que se puede encontrar una derivación en tiempo lineal, porque es posible que en un momento se puede aplicar más de una regla.

11.3. Lema de bombeo para lenguajes libres de contexto

Igual como lo hemos visto para lenguajes regulares existe una propiedad que todos los lenguajes libres de contexto cumplen:

Lema (de bombeo para lenguajes libres de contexto): Sea L un lenguaje libre de contexto (infinito). Entonces existe un $n \in \mathbb{N}$ de tal manera que cada palabra $z \in L$ con $|z| \geq n$ se puede dividir en cinco partes, $z = uvwxy$ cumpliéndose las tres propiedades:

1. $|vx| \geq 1$
2. $|vwx| \leq n$
3. para todos los $k \geq 0 : uv^k wx^k y \in L$

Idea de la comprobación:

- partimos de la FNC de la gramática, es decir, las reglas son de las formas $X \longrightarrow YZ$ o $X \longrightarrow \sigma$
- el árbol para una palabra (suficientemente larga) será un árbol binario
- si $|z| \geq 2^{k+1}$ entonces el árbol tiene una altura por lo menos de $k+1$, es decir, se encuentran $k+1$ variables en un camino desde la raíz hacia alguna hoja
- entonces, si hay solamente k variables en el alfabeto Σ_N, se tiene que repetir una variable, sea X, en un camino desde la raíz hasta una hoja

- observamos los dos subárboles con dicha variable **desde abajo**

  lblibre0

- vemos: $|vx| \geq 1$ porque se tiene que derivar algo desde X dado que tenemos una FNC y el árbol se bifurca en X
- vemos: $|vwx| \leq n$ porque la altura del subárbol hacia el segundo X es como mucho k
- vemos: para todos los $k \geq 0 : uv^k wx^k y \in L$ porque podemos eliminar v y x o sustituir cuantas veces como queremos el subárbol debajo de X adecuadamente

  lblibre1

El uso del lema de bombeo es parecido a su uso en el caso de los lenguajes regulares, se puede comprobar que ciertos lenguajes no son libres de contexto.

Ejemplo: Investigamos $L_{abc} = \{a^n b^n c^n \mid n \geq 1\}$.

- Asumimos que L_{abc} sea libre de contexto.
- El lema de bombeo nos garantiza la existencia de un n tal que se cumplen las propiedades para palabras z con $|z| \geq n$. (No conocemos n en concreto, solo su existencia.)
- (Pensamos...): Elegimos $z = a^n b^n c^n$. Obviamente $z \in L_{abc}$ y $|z| = 3n \geq n$.
- El lema de bombeo nos garantiza la existencia de una partición $z = uvwxy$ con $|vx| \geq 1$, $|vwx| \leq n$, y $\forall k \geq 0 : uv^k wx^k y \in L_{abc}$. (No conocemos la partición en concreto, pero sus propiedades.)
- (Pensamos...): Porque $|vwx| \leq n$ el vwx no puede contener a's, b's, **y** c's al mismo tiempo.
- Entonces vx tampoco, es decir, vx contiene como mucho dos símbolos diferentes.
- Porque $|vx| \geq 1$ la subpalabra vx contiene por lo menos un símbolo.
- $uv^0 wx^0 y = uwy \in L_{abc}$ pero hemos borrado como mucho dos tipos de símbolos.
- Eso es una contradicción.
- Entonces L_{abc} no puede ser libre de contexto.

12. Autómatas finitos con pila (AFP)

12.1. Motivación

Ya sabemos $L_{ab} = \{a^n b^n \mid n \in \mathbb{N}\}$ no es regular (comprobamos con el lema de bombeo o con el teorema de Myhill–Nerode).

Pero L_{ab} es libre de contexto con la siguiente gramática:

$$\begin{aligned} G &= (\Sigma_N, \Sigma_T, P, \$) \\ &= (\{\$\}, \{a, b\}, \{\$ \longrightarrow a\$b|\epsilon\}, \$) \end{aligned}$$

Otro ejemplo parecido es: expresiones matemáticamente correctas de diferentes tipos de paréntesis $\Sigma_T = \{[,], \langle, \rangle, (,)\}$, por ejemplo, $(\,(\,[\,]\,]\,)\,\rangle\,)$ es incorrecto y $[\,(\,[\,]\,)\,\langle\,(\,)\,\rangle\,]$ es correcto.

$$L_{()} = \{w \mid w \in \Sigma_T^*, w \text{ es correcto}\}$$

es libre de contexto, con el sistema de producciones

$$P = \{\$ \longrightarrow \$\$ \mid (\$) \mid [\$] \mid \langle\$\rangle \mid \epsilon\}$$

$L_{()}$ no es regular, porque ya $[^n]^n$ no es regular.

¿Podemos construir un tipo de autómata que acepta una palabra de $L_{()}$?

Idea: usamos una pila para memorizar lo que se ha leído:

- Las paréntesis que abren ponemos en la pila.
- Si vemos una parentesis que cierre la cima de la pila tiene que ser su homóloga y la quitamos de la pila.
- Al final, la pila tiene que estar vacía.

Eso era bastante fácil, ampliamos las posibilidades algo más, permitimos

- que el autómata pueda tener varios (número finito) estados (parecido a los AFD, pero veremos que basta con un estado);
- que el autómata sea no-determinista (veremos que habrá una diferencia entre AFPDs y AFPNDs);
- que exista la posibilidad de transiciones ϵ;
- que acepte con pila vacía o con estados finales (veremos que ambas formas son equivalentes);

- que existan más símbolos para la pila;
- que se apile más de un símbolo a la vez;
- que se disponga de un símbolo inicial en la pila.

12.2. Autómatas finitos con pila no-deterministas (AFPND)

Un autómata finito con pila no-determinista (AFPND) es una séptupla

$$M = (\Sigma, \Gamma, Q, \delta, q_0, c_0, F)$$

donde

- Σ es un alfabeto de entrada.
- Γ es un alfabeto de pila ($\Gamma = \Sigma$ es posible).
- Q es un conjunto de estados, $|Q| < \infty$.
- $q_0 \in Q$ es el estado inicial.
- $c_0 \in \Gamma$ es el símbolo inicial de la pila.
- F es el conjunto de estados finales (puede ser el conjunto vacía).
- δ es la función de transición

  $$\delta : Q \times (\Sigma \cup \{\epsilon\} \times \Gamma \longrightarrow W_{<\infty}(Q \times \Gamma^*)$$

  donde $W_{<\infty}$ sea el conjunto de subconjuntos finitos.
  Es decir, el comportamiento del autómata depende en cada transición

  - del estado actual
  - posiblemente del siguiente símbolo de la entrada
  - del símbolo en la cima de la pila

  y se modifica el autómata en el sentido que

  - se cambia (posiblemente) del estado
  - se consume (posiblemente) el siguiente símbolo de la entrada
  - se modifica (posiblemente) el contenido de la cima de la pila.

Para el ejemplo de arriba obtenemos el autómata

$$M_{()} = (\{(,), \langle, \rangle, [,]\}, \{(, \langle, [, \#\}, \{q_0, q_1\}, \delta, q_0, \#, \emptyset)$$

con

$$\begin{array}{rcll} \delta(q_0, (, \gamma) & = & \{(q_0, (\gamma)\} & \forall \gamma \in \Gamma \\ \delta(q_0, \langle, \gamma) & = & \{(q_0, \langle\gamma)\} & \forall \gamma \in \Gamma \\ \delta(q_0, [, \gamma) & = & \{(q_0, [\gamma)\} & \forall \gamma \in \Gamma \\ \delta(q_0,), () & = & \{(q_0, \epsilon)\} & \\ \delta(q_0, \rangle, \langle) & = & \{(q_0, \epsilon)\} & \\ \delta(q_0,], [) & = & \{(q_0, \epsilon)\} & \\ \delta(q_0, \epsilon, \#) & = & \{(q_1, \epsilon)\} & \end{array}$$

Observa

- que escribimos en las expresiones arriba el contenido de la pila como los árabes: desde la derecha hacia la izquierda,
- que el autómata no está completo, pero se podría completar añadiendo transiciones adecuados en un estado "sin salida" que ya no varía la pila.

También podemos dibujar autómatas con pila, por ejemplo de la siguiente manera:

afp

Es decir, dibujamos el grafo parecido como lo hemos hecho para los AFND-ϵ: los vértices del grafo representan los estados del autómata y las aristas representan las transiciones. Ampliamos las etiquetas de las aristas con los cambios en la cima de la pila.

Podemos pensar de un autómata con pila como un dispositivo que lee desde una cinta con símbolos, realiza cambios de estados internamente, y maneja una pila de la forma descrita:

autopila

Otro ejemplo; construimos un AFP para el lenguaje

$$L_{vv^R} = \{w \mid w \in \{0,1\}^*, w = vv^R\}$$

es decir, los palíndromos con longitud par.

Idea:

- Adivinamos (no-determinismo) dónde acaba v.

- Copiamos toda la palabra v a la pila.
- Verificamos el resto de w, que debe ser v^R, con el contenido de la pila, es decir, la pila debe estar vacía una vez haber leído toda la palabra w.

Un AFPND será el siguiente:

$$M_{vv^R} = (\{0,1\}, \{0,1,\#\}, \{q_0,q_1,q_2\}, \delta, q_0, \#, \emptyset)$$

con

$$\begin{array}{rcll} \delta(q_0,0,\gamma) & = & \{(q_0,0\gamma)\} & \forall\gamma\in\Gamma \\ \delta(q_0,1,\gamma) & = & \{(q_0,1\gamma)\} & \forall\gamma\in\Gamma \\ \delta(q_0,\epsilon,\gamma) & = & \{(q_1,\gamma)\} & \forall\gamma\in\Gamma \\ \delta(q_1,0,0) & = & \{(q_1,\epsilon)\} & \\ \delta(q_1,1,1) & = & \{(q_1,\epsilon)\} & \\ \delta(q_1,\epsilon,\#) & = & \{(q_2,\epsilon)\} & \end{array}$$

afpvv

¿Cómo comprobamos que es correcto?

Dado que el contenido de la pila influye en el comportamiento del autómata necesitamos una notación para describir los cálculos del autómata.

La **configuración** (o descripción instantánea) C de un AFP $M = (\Sigma, \Gamma, Q, \delta, q_0, c_0, F)$ es la tripla (q, u, v) donde

- $q \in Q$ es el estado actual
- $u \in \Sigma^*$ es lo que queda por leer de la entrada
- $v \in \Gamma^*$ es el contenido actual de la pila

La **configuración inicial** C_0 entonces es (q_0, w, c_0).

Si el autómata está en configuración C podemos definir que es una posible siguiente configuración, es decir, después de haber realizado un paso en el cálculo.

$C' = (q', u, zv)$ es **configuración sucesora** de $C = (q, \sigma u, \gamma v)$ (es decir, σ es el siguiente símbolo de la entrada y γ la cima de la pila), si $(q', z) \in \delta(q, \sigma, \gamma)$ y, para las transiciones ϵ, $C' = (q', u, zv)$ es **configuración sucesora** de $C = (q, u, \gamma v)$ (es decir, no se lee un símbolo de la entrada y γ la cima de la pila), si $(q', z) \in \delta(q, \epsilon, \gamma)$.

Observa, si la pila está vacía, no existe configuración sucesora ninguna.

Escribimos $C \vdash\!\!\!- C'$ si C' es configuración sucesora de C. Si existe una secuencia de configuraciones sucesoras de C hasta C', es decir,

$$C = C_0 \vdash\!\!\!- C_1 \vdash\!\!\!- C_2 \vdash\!\!\!- \ldots \vdash\!\!\!- C_n = C'$$

llamamos la secuencia un **cálculo del autómata** y abreviamos con $C \vdash\!\!\!-^* C'$.

Un **AFPND acepta** una palabra w de entrada según modus:

- $F = \emptyset$, es decir, acepta con pila vacía

  $$M \text{ acepta } w \iff (q_0, w, c_0) \vdash\!\!\!-^* (q, \epsilon, \epsilon)$$

  para cualquier estado $q \in Q$
- $F \neq \emptyset$, es decir, acepta en estado final

  $$M \text{ acepta } w \iff (q_0, w, c_0) \vdash\!\!\!-^* (f, \epsilon, v)$$

  con $v \in \Gamma^*$ y $f \in F$.

El **lenguaje aceptado** por un autómata AFPND M es

$$L(M) = \{w \mid M \text{ acepta } w\}$$

En la siguiente sección comprobamos que ambos métodos de aceptación son equivalentes para los AFPND (pero no será el caso de ls AFPD, los autómatas finitos con pila deterministas, que veremos más adelante).

Comprobamos ahora que el M_{vv^R} es correcto, es decir, tenemos que comprobar que $L(M_{vv^R}) = L_{vv^R}$.

Primero verificamos que M_{vv^R} acepta para cualquier palabra $v \in \{0,1\}^*$ la palabra $w = vv^R$:

$$\begin{aligned}
(q_0, vv^R, \#) &\longrightarrow^* (q_0, v^R, v^R\#) \\
&\longrightarrow (q_1, v^R, v^R\#) \\
&\longrightarrow^* (q_1, \epsilon, \#) \\
&\longrightarrow (q_2, \epsilon, \epsilon)
\end{aligned}$$

es decir, hemos encontrado un cálculo y con eso sabemos que $L_{vv^R} \subset L(M_{vv^R})$.

Luego comprobamos que M_{vv^R} solamente acepta palabras en L_{vv^R}.

(por incluir)

12.3. Equivalencia entre AFPNDs aceptando con pila vacía y aceptando en estado final

Para cada AFPND M que acepta con pila vacía existe un AFPND M' que acepta en estado final.

Idea de la comprobación:

- M' simula M
- M' usa un nuevo símbolo c'_0 como símbolo inicial de la pila
- si después de la simulación de M dicho c'_0 está en la cima de la pila, M' sabe que M hubiese aceptado, es decir, M' acepta también yiendo a un estado final.

Para el ejemplo de antes

$$L_{vv^R} = \{w \mid w \in \{0,1\}^*, w = vv^R\}$$

con el siguiente autómata que acepta con pila vacía

afpndpv

obtenemos el nuevo autómata que acepta en estado final

afpndefpv

En general:

$$\begin{aligned} M &= (\Sigma, \Gamma, Q, \delta, q_0, c_0, \emptyset) \\ M' &= (\Sigma, \Gamma \cup \{c'_0\}, Q \cup \{q'_0, f\}, \delta', q'_0, c'_0, \{f\}) \end{aligned}$$

con

- $q'_0, f \notin Q$, es decir, son nuevos estados
- $c'_0 \notin \Gamma$, es decir, es un nuevo símbolo inicial
- $\delta'(q'_0, \epsilon, c'_0) = \{(q_0, c_0 c'_0)\}$, es decir, la primera transición apila el antiguo símbolo inicial y se va al antiguo estado inicial sin leer nada de la entrada
- $\forall q \in Q, \sigma \in \Sigma, \gamma \in \Gamma : \delta'(q, \sigma, \gamma) = \delta(q, \sigma, \gamma), \delta'(q, \epsilon, \gamma) = \delta(q, \epsilon, \gamma)$, es decir, se simula M
- $\forall q \in Q : \delta'(q, \epsilon, c'_0) = \{(f, c'_0)\}$, es decir, si la pila solamente contiene el nuevo símbolo inicial se va al estado final.

Para cada AFPND M que acepta en estado final existe un AFPND M' que acepta con pila vacía.

Idea de la comprobación:

- M' simula M
- M' vacía desde cualquier estado final de M su pila
- tenemos que tener cuidado si M no termina en estado final, pero su pila está vacía: colocamos antes de la simulación un nuevo símbolo c'_0 como símbolo inicial en la pila que no 'se toca' durante la simulación de M.

Para el ejemplo

$$L = \{a^i b^j \mid j \leq i\}$$

con el siguiente autómata que acepta en estado final

afpndef

(Primero observamos la consecuencia de la definición de un cálculo:

$$M \text{ acepta } w \iff (q_0, w, c_0) \vdash^* (f, \epsilon, v)$$

entonces, si sobran b's la pila estará vacía y no habrá transición ninguna, y por eso no llegamos a ϵ con la entrada.)

Siguiendo la idea, obtenemos el nuevo autómata que acepta con pila vacía

afpndpvef

En general:

$$\begin{aligned} M &= (\Sigma, \Gamma, Q, \delta, q_0, c_0, \emptyset) \\ M' &= (\Sigma, \Gamma \cup \{c'_0\}, Q \cup \{q'_0, q'\}, \delta', q'_0, c'_0, \emptyset) \end{aligned}$$

con

- $q'_0, q' \notin Q$, es decir, son nuevos estados
- $c'_0 \notin \Gamma$, es decir, es un nuevo símbolo inicial
- $\delta'(q'_0, \epsilon, c'_0) = \{(q_0, c_0 c'_0)\}$, es decir, la primera transición apila el antiguo símbolo inicial y se va al antiguo estado inicial sin leer nada de la entrada
- $\forall \gamma \in \Gamma \cup \{c'_0\} : \delta'(q', \epsilon, \gamma) = \{(q', \epsilon)\}$, es decir, una vez en estado q' se vacía la pila sin modificar la entrada
- $\forall q \in Q, \sigma \in \Sigma, \gamma \in \Gamma : \delta'(q, \sigma, \gamma) = \delta(q, \sigma, \gamma)$, es decir, pasos normales de la simulación
- $\forall q \in Q - F, \gamma \in \Gamma : \delta'(q, \epsilon, \gamma) = \delta(q, \epsilon, \gamma)$, es decir, se simula también las transiciones ϵ mientras M no esté en estado final
- $\forall q \in F, \gamma \in \Gamma : \delta'(q, \epsilon, \gamma) = \delta(q, \epsilon, \gamma) \cup \{(q', \gamma)\}$, es decir, saltamos al estado que vacía la pila si ya estamos en estado final

12.4. Equivalencia entre AFPNDs y gramáticas libres de contexto

Para cada gramática libre de contexto G existe un autómata finito con pila no-determinista M que acepta el mismo lenguaje, es decir, $L(M) = L(G)$.

La comprobación es constructiva.

Sea $G = (\Sigma_T, \Sigma_N, P, \$)$ una gramática libre de contexto.

Podemos convertir la gramática en su forma normal de Greibach (FNG), es decir todas las producciones son del tipo: $A \longrightarrow \sigma\Upsilon$ con $\sigma \in \Sigma_T$ y $\Upsilon \in \Sigma_N^*$ o la producción es $\$ \longrightarrow \epsilon$ si $\epsilon \in L(G)$.

Construimos un AFPND $M = (\Sigma_T, \Sigma_N, \{q\}, \delta, q, \$, \emptyset)$, (es decir, con un sólo estado) que acepta con pila vacía, donde

$$(q, \Upsilon) \in \delta(q, \sigma, A)$$

siempre que $A \longrightarrow \sigma\Upsilon$ sea una producción en P y

$$(q, \$) \in \delta(q, \epsilon, \epsilon)$$

siempre que $\$ \longrightarrow \epsilon$ sea una producción en P.

Entonces, el autómata **simula** en un cálculo la aplicación de las reglas de la gramática siempre siguiendo la derivación más a la izquierda para la palabra en cuestión.

Ejemplo:

$$G = (\{a, b\}, \{\$, A, B, C\}, P, \$)$$

con

$$P = \{\$ \longrightarrow aBBC, A \longrightarrow aAA|b, B \longrightarrow bBAC|b, C \longrightarrow b\}$$

que ya está en forma formal de Greibach, entonces el AFPND es:

$$M = (\{a, b\}, \{\$, A, B, C\}, \{q\}, \delta, q, \$, \emptyset)$$

con

$$\begin{aligned}
\delta(q, a, \$) &= \{(q, BBC)\} \\
\delta(q, a, A) &= \{(q, AA)\} \\
\delta(q, b, A) &= \{(q, \epsilon)\} \\
\delta(q, b, B) &= \{(q, BAC), (q, \epsilon)\} \\
\delta(q, b, C) &= \{(q, \epsilon)\}
\end{aligned}$$

Para cada autómata finito con pila no-determinista M existe una gramática libre de contexto G que genera el mismo lenguaje, es decir, $L(G) = L(M)$.

La comprobación es constructiva.

Sea $M = (\Sigma, \Gamma, Q, \delta, q_0, c_0, F)$ un AFPND.

Si $F \neq \emptyset$ podemos convertir el autómata en un AFPND que acepte con pila vacía.

Luego podemos asumir que todas las transiciones del autómata como mucho apilan dos símbolos a la pila, porque podemos introducir estados intermedios que apilan poco a poco todos los símbolos necesarios sin leer más de la entrada, en concreto,

- sea $(q, v) \in \delta(p, \sigma, \gamma)$ con $v = \gamma_1\gamma_2 \ldots \gamma_k$, $\sigma \in \Sigma$, y $k > 2$ una transición de tal tipo
- añadimos los nuevos estados $q_1, q_2, \ldots, q_{k-2}$ a Q y sustituimos la transición por

$$\begin{aligned} (q_1, \gamma_{k-1}\gamma_k) &\in \delta(p, \sigma, \gamma) \\ (q_2, \gamma_{k-2}\gamma_{k-1}) &\in \delta(q_1, \epsilon, \gamma_{k-1}) \\ &\ldots \\ (q, \gamma_1\gamma_2) &\in \delta(q_{k-2}, \epsilon, \gamma_2) \end{aligned}$$

- Observa que podemos realizar tal sustitución igual en caso que el autómata ejecute una transición-ϵ (es decir, arriba no se lee σ de la entrada sino ϵ); entonces, para simplificar escribimos σ' si leemos o bien un símbolo $\sigma \in \Sigma$ o bien ϵ.

Entonces, asumimos que tengamos un AFPND que acepta con pila vacía y que apile en una transición como mucho dos símbolos a la vez.

Construimos una gramática libre de contexto $G = (\Sigma, \Sigma_N, P, \$)$, es decir, con los mismos símbolos de entrada, y donde

- Σ_N está formado por las triplas $[p, A, q]$ siendo $p, q \in Q$ y $A \in \Gamma$, y el símbolo $\$$.
- P es el conjunto de producciones donde añadimos
  - para cada estado $q \in Q$ las reglas $\$ \longrightarrow [q_0, c_0, q]$
  - para cada transición $(q, \epsilon) \in \delta(p, \sigma', \gamma)$ la regla $[p, \gamma, q] \longrightarrow \sigma'$
  - para cada transición $(q, \gamma_1) \in \delta(p, \sigma', \gamma)$ y cada estado $r \in Q$ las reglas $[p, \gamma, r] \longrightarrow \sigma'[q, \gamma_1, r]$
  - para cada transición $(q, \gamma_1\gamma_2) \in \delta(p, \sigma', \gamma)$ y cada par de estados $r, s \in Q$ las reglas $[p, \gamma, r] \longrightarrow \sigma'[q, \gamma_1, s][s, \gamma_2, r]$

- observa que pueden existir reglas nulas en caso que $\sigma' = \epsilon$

Entonces, la gramática **simula** un cálculo del autómata con una derivación más a la izquierda para la palabra en cuestión.

Formalmente hay que comprobar la equivalencia

$$\$ \longrightarrow^* w \iff (q_0, w, c_0) \vdash^* (q, \epsilon, \epsilon)$$

es decir, si existe una derivación también existe un cálculo y al revés.

La comprobación del lado izquierdo al lado derecho se realice mediante una inducción sobre la longitud de una derivación más a la izquierda y la otra dirección mediante una inducción sobre la longitud del cálculo. El caso inicial, es decir, se aplica solamente una regla o se calcula solamente una configuración siguiente, se verifica directamente a partir de la construcción.

- Juntas ambas direcciones nos proporcionen la equivalencia entre las gramáticas libres de contexto y los autómatas finitos con pila no-deterministas.
- En la primera parte de la comprobación observamos que basta un solo estado en un AFPND (si un AFPND tiene más estados, podemos construir una gramática equivalente, y después un AFPND con un solo estado).
- En la segunda parte de la comprobación observamos que basta con una gramática en forma normal de Greibach donde las producciones tengan como mucho 2 símbolos no-terminales en sus partes derechas (es decir, también en la FNG los árboles de derivación pueden ser árboles binarios como en la FNC).

12.5. Autómatas finitos con pila deterministas (AFPD)

Los AFPND, como el propio nombre ya dice, no son deterministas, es decir, pueden existir varias posibles configuraciones siguientes, o en otras palabras, $\delta(q, \sigma, \gamma)$ o $\delta(q, \epsilon, \gamma)$ son conjuntos con—posiblemente—más de un elemento.

Para que un AFPND acepte una palabra de entrada w se ha exigido solamente la existencia de un cálculo que lee toda la palabra w y termina con pila vacía o en un estado final.

Este hecho no es adecuado en la práctica, porque de alguna manera hay que comprobar todos los posibles cálculos para ver si existe uno que acepta. Por eso limitamos los autómatas para que sean deterministas.

Podemos definir un **autómata finito con pila determinista AFPD**

$$M = (\Sigma, \Gamma, Q, \delta, q_0, c_0, F)$$

igual que un AFPND introduciendo las siguientes restricciones

1. para cada $q \in Q$, $\sigma \in \Sigma$, y $\gamma \in \Gamma$ permitimos como mucho una transición, es decir:

$$|\delta(q, \sigma, \gamma)| + |\delta(q, \epsilon, \gamma)| \leq 1$$

   Entonces, permitimos transiciones-ϵ que son deterministas si consideramos la pila.

2. $F \neq \emptyset$, es decir, el AFND acepta con estado final.

Dado que para un AFPD existe como mucho una configuración siguiente, es decir $\vdash$ es una función, los cálculos se convierten en cadenas deterministas, y decimos, que el AFPD acepta una palabra w si existe el cálculo $(q_0, w, c_0) \vdash (f, \epsilon, v)$ con $f \in F$.

Para AFPDs los dos criterios de parada no son equivalentes que se entiende analizando las comprobaciones donde era escencial disponer de transiciones no-deterministas para 'saltar' a un estado adicional con el fin de vaciar la pila.

Llamamos un lenguaje L **libre de contexto determinista** si L es aceptado por un autómata finito con pila determinista.

Los lenguajes libres de contexto deterministas son un verdadero subconjunto de los lenguajes libres de contexto, es decir, existen lenguajes que son libres de contexto pero no libres de contexto determinista.

Ejemplo:

El lenguaje $L = \{w \mid w \in \{0, 1, \#\}, w = v\#v^R$ y v no contiene $\#$ es libre de contexto determinista, porque se apila hasta encontrar el centro (que hemos marcado con $\#$) y después se verifica el resto de w con el contenido de la pila.

El lenguaje $L = \{w \mid w \in \{0, 1\}, w = vv^R\}$ es libre de contexto, como ya vimos, pero no es libre de contexto determinista, porque, para decirlo de alguna manera, se necesita el no-determinismo para encontrar el centro, o en otras palabras, hay que comprobar todos los posibles cálculos verificando si uno de ellos llega a una aceptación.

Obviamente los lenguajes regulares también son libres de contexto deterministas, porque si "no usamos la pila" justamente un AFP es un AFD.

13. Propiedades, algoritmos de decisión, y aplicaciones para lenguajes libres de contexto

13.1. Propiedades de lenguajes libre de contexto

Sean L_1 y L_2 dos lenguajes libres de contexto.

Unión: $L = L_1 \cup L_2$ es libre de contexto, porque podemos construir a partir de las gramáticas $G_1 = (\Sigma_T^1, \Sigma_N^1, P^1, \$^1)$ y $G_2 = (\Sigma_T^2, \Sigma_N^2, P^2, \$^2)$ con $L_1 = L(G_1)$ y $L_2 = L(G_2)$ la gramática $G = (\Sigma_T^1 \cup \Sigma_T^2, \Sigma_N^1 \cup \Sigma_N^2, P^1 \cup P^2 \cup \{\$ \longrightarrow \$^1, \$ \longrightarrow \$^2\}, \$)$ que es una gramática libre de contexto y obviamente genera todas las palabras tanto en L_1 como en L_2.

Eso no es el caso si nos limitamos a los lenguajes libres de contexto deterministas.

Concatención: $L = L_1.L_2$ es libre de contexto, porque podemos construir a partir de las gramáticas $G_1 = (\Sigma_T^1, \Sigma_N^1, P^1, \$^1)$ y $G_2 = (\Sigma_T^2, \Sigma_N^2, P^2, \$^2)$ con $L_1 = L(G_1)$ y $L_2 = L(G_2)$ la gramática $G = (\Sigma_T^1 \cup \Sigma_T^2, \Sigma_N^1 \cup \Sigma_N^2, P^1 \cup P^2 \cup \{\$ \longrightarrow \$^1\$^2\}, \$)$ que es una gramática libre de contexto y obviamente genera todas las palabras en L.

Eso no es el caso si nos limitamos a los lenguajes libres de contexto deterministas.

Clausura: $L = L_1^*$ es libre de contexto, porque podemos construir una gramática libre de contexto a partir de la gramática para L_1, simplemente añadimos las producciones $\{\$' \longrightarrow \$'\$', \$' \longrightarrow \$\}$ siendo $\$$ el nuevo símbolo inicial.

Intersección: $L = L_1 \cap L_2$ no es libre de contexto (en general), como demuestra el ejemplo: $L_1 = \{a^i b^i c^j \mid i, j > 0\}$ y $L_2 = \{a^i b^j c^j \mid i, j > 0\}$ nos lleva a $L = L_1 \cap L_2 = \{a^i b^i c^i \mid i > 0\}$ que no es libre de contexto como ya vimos.

Si confinamos L_2 a lenguajes regulares, entonces la intersección produce lenguajes libres de contexto. El argumento es igual de constructivo como en el caso de dos lenguajes regulares.

Complemento: $L = \overline{L_1} = \Sigma^* - L_1$ no es libre de contexto (en general), porque si asumimos que lo fuera y sabiendo que la unión lo es podríamos derivar $\overline{\overline{L_1} \cup \overline{L_2}} = L_1 \cap L_2$ como libre de contexto, pero ya sabemos que la intersección no genera siempre tal lenguajes.

Para los lenguajes libres de contexto determinista, el complemento genera un lenguaje libre de contexto determinista, porque es fácil *invertir* un autómata determinista.

Diferencia: $L = L_1 - L_2$ no es libre de contexto (en general), porque $L = \Sigma^* - L_2 = \overline{L_2}$ no es libre de contexto.

13.2. Algoritmos de decisión de lenguages libres de contexto

Pertenencia: ¿$w \in L$? sí, se puede contestar la pregunta (es decir, es un problema computable) porque

- construimos un autómata que acepte L (por ejemplo en estado final)
- simulamos su comportamiento leyendo la palabra w
- si acabamos en un estado final, w está en L, sino w no está en L

dicha simulación puede tener tiempo de cálculo exponential, otra posibilidad es

- construimos una gramática en forma normal de Chomsky
- aplicamos el algoritmo de Cocke-Younger-Kasami que resuelve el problema en tiempo de orden $O(n^3)$.

Vaciedad: ¿$L = \emptyset$? sí, se puede contestar la pregunta (es decir, es un problema computable) porque

- construimos una gramática que genere L
- analizamos si el símbolo inicial es útil, si es útil, entonces L no es vacío, sino L es vacío

Cardinalidad: ¿$|L| < \infty$? sí, se puede contestar la pregunta (es decir, es un problema computable) porque

- construimos una gramática que genere L en su forma normal de Chomsky
- analizamos el grafo de posisibles sustituciones de símbolos para averigüar si existe un ciclo
- si existe tal ciclo, entonces L es infinito, sino L es finito

Igualidad: ¿$L_1 = L_2$? no, no se puede contestar la pregunta (es decir, es un problema no computable).

- antes de entender dicha respuesta negativa, hay que estudiar más a fondo la teoría de las Máquinas de Turing.

13.3. Aplicaciones para lenguajes libres de contexto

Análisis semántico (basado en bloques estructurados): La aplicación más importante de los lenguajes libres de contexto es claramente la definición y el análisis de "información estructurado en bloques anidados" como suele ocurrir en los lenguajes de programación (recuerda los bloques `begin–end` en PASCAL) o las marcas en un fichero XML (*extended markup language*).

14. Bibliografía

14.1. Bibliografía básica

1. J.E. Hopcroft, R. Motwani, J.D. Ullman: *Introducción a la teoría de autómatas, lenguajes y computación*, Segunda edición, Addison–Wesley, 2002. (Signatura: OUR 681.34/46)

2. P. Isasi, P. Martínez, D. Borrajo. *Lenguajes, Gramáticas y Autómatas. Un enfoque práctico.* Addison-Wesley, ISBN 84-7829-014-1, 1997-2001. (Signatura: OUR 681.34/13)

3. M. Alfonseca, J. Sancho, M. Martínez Orga. *Teoría de Lenguajes, Gramáticas y Autómatas.* Universidad y Cultura, ISBN 84-7832-201-9, 1997. (Signatura: OUR 681.34/31)

14.2. Bibliografía, enlaces en la red

1. http://es.wikipedia.org/wiki/Teor%C3%ADa_de_aut%C3%B3matas

14.3. Bibliografía usada para la preparación de las clases

1. A. Sánchez López. *Teoría de Autómatas y Lenguajes Formales.* Apuntes de clase, comunicación personal, 2004.

2. R. Fernández Muñoz. *Teoría de Autómatas y Lenguajes Formales.* Proyecto fin de carrera, INX-380, Biblioteca de la Universidad de Vigo, Campus Ourense, 2003.

3. G. Hotz, K. Estenfeld. *Formale Sprachen.* B.I. Wissenschaftsverlag, ISBN 3-411-01626, 1981.

4. Th. Schwentick. *Theoretische Informatik.* Vorlesungsfolien, Internet, comunicación personal, 2004.

5. Uwe Schönfeld. *Theoretische Informatik kurz gefasst.* B.I. Wissenschaftsverlag, ISBN 3-411-15641-4, 1992.