Índice general

1 Introducción

1.1. La computadora hoy en día

Sin lugar a dudas la computación se ha convertido en un campo que ha impactado el desarrollo de nuestra sociedad, hoy en día encontramos computadoras en varios aspectos de la cotidianidad de nuestro actividades diarias. De hecho algunas de ellas pasan desapercibidas, las más fáciles de identificar son las tipo escritorio, o tipo laptop o recientemente en nuestros dispositivos móviles e "inteligentes". Sin embargo si ponemos atención también encontramos computadoras en: tarjetas bancarias, en los coches y hasta en las lavadoras. Sus capacidades no nos dejan de sorprender, no sólo nos apoyan en nuestras tareas diarias a través de editores de texto, hojas de cálculo, editores de imágenes y en redes sociales sino que también son una fuente principal de entretenimiento, son base de la economía con sistemas que facilitan el comercio electrónico y el manejo de la banca, y nos ayudan a encontrar nuevas fronteras literalmente del universo o de las matemáticas a través de la exploración numérica. No solo eso, sino que hoy en día las computadoras están comenzando a romper barreras en tareas antes pensadas exclusivas del humano: los mejores jugadores de ajedrez son computadoras, el mejor jugador de Go también fue una computadora. Además ahora podemos conversar con las computadoras, estas comienzan a entendernos cuando les hablamos, crean imágenes y hay un esfuerzo gigantesco para que comiencen a manejar coches de forma autónoma.

Ante todo este avance, es importante preguntarse sobre la naturaleza de la computadora/computación:

1. ¿Qué es una computadora?
2. ¿Cuales son las propiedades esenciales de una computadora?
3. ¿Cómo serán las computadoras en 10 años, en 100 años o 1,000 años?
4. ¿Qué problemas estarán resolviendo las computadoras en 100 años o 1,000 años?
5. ¿Qué problemas no podrán resolver aún en 1,000 o 10,000 años?

1.2. ¿Por qué de lenguajes formales?

Hay varias formas de tratar de responder a las preguntas anteriores, uno podría concentrarse en el aspecto tecnológico como lo son hoy en día los circuitos electrónicos y tratar analizar de ahí que elementos son básicos y qué hacen a una computadora. A partir de ahí uno podría extrapolar como serán las computadoras electrónicas, a lo mejor siguiendo leyes como la de Moore. Por otro lado, también podríamos estudiar nuevas tendencias como la computación cuántica que trata de cambiar el paradigma de la computación y podríamos predecir como lucirá una computación basada en esta nueva tecnología.

También uno podría fijarse en los algoritmos y su complejidad; uno podría concentrarse en tratar de identificar nuevos algoritmos que resuelvan problemas hasta ahora difíciles y de forma eficiente, de ahí uno podría extrapolar nuevos algoritmos para las computadoras del futuro, y a lo mejor familias de estos.

Este material ofrece otra perspectiva para responder a estas preguntas y en particular a la número dos. Para poder, responderla asumiremos un marco matemático-computacional. Partiremos de conceptos de la teoría de conjuntos para explorar los alcances y propiedades de máquinas que computan, e iremos escalando los elementos de estas máquinas para llegar a un modelo teórico de la computadora. Este mismo modelo nos abrirá las puertas para encontrar los límites de la computación.

Para lograr alcanzar a vislumbrar las respuestas a dichas preguntas iremos ahondando en conceptos como:

1. Conjuntos
2. Alfabetos
3. Cadenas
4. Lenguajes
5. Máquinas
6. Gramáticas
7. Decidibilidad
8. Paradojas

1.3. El origen de la idea de la computación

La humanidad ha buscado tomar ventaja de su ambiente y no solo adaptarse a él, sino adaptar el ambiente en beneficio de la sociedad. A través del tiempo encontramos ejemplos de uso de herramientas y su aplicación para mejorar la situación de vida desde una persona, hasta pueblos

y civilizaciones. En esta búsqueda han surgido máquinas que ayudan con tareas específicas. Fue durante el periodo que denominamos como revolución industrial que vemos un nivel nunca antes visto en el diseño y uso de diferentes materiales, máquinas y procesos. En particular se creó maquinaria para crear objetos de forma masiva. Esto implico un cambio socio-económico, donde por ejemplo uno de los fenómenos observados fue que la población se desplazó de estar dispersa en un ambiente rural a concentrarse en focos urbanos.

La introducción de la nueva maquinaria implicó el incremento del conocimiento necesario para los procesos de producción de la misma maquinaria. En particular se requirió avanzar en múltiples frentes como la química, la metalurgia, el diseño de materiales, etc. De forma interesante este avance recayó en el poder del cálculo numérico, esto se debió a que se requería de una flexibilidad para calcular valores específicos a un problema particular. Como es imposible predecir qué calculo se necesitaría para un problema particular se asumió una estrategia de precalcular muchos valores todo hasta cierta precisión, un ejemplo de esto son las tablas logarítmicas,esto permitiría después hacer el cálculo requerido tomando como base los cálculos hechos con anterioridad. Estos cálculos masivos fueron hechos por humanos, y los resultados incluían algunos errores.

En este contexto que surge la idea de qué si ya se automatizaban procesos que antes hacían humanos, por ejemplo la elaboración de textiles, porque ahora no se automatizaba el proceso de hacer cálculos. Es Charles Babbage el primero en proponer una primer máquina calculadora a este nivel: la máquina diferencial, mientras construía comenzó a idear otra: la máquina analítica, que asemeja a nuestras computadoras actuales pero que sin embargo no alcanzó a concluir. Ada Lovelace al conocer el diseño de esta máquina se convierte en la primera programadora de la historia de la humanidad, al programar una fórmula para números de Bernulli que nunca pudo ejecutar (ya que no existía la computadora). Sin embargo el marco teórico en el que funcionaría dicho calculo, hace posible ver que su propuesta fue correcta y así ella ideo el primer programa en toda la historia de la humanidad.

Esta no sería la última vez en que se dé un avance fundamental en la computación de forma teórica aun cuando no se tenga la máquina en su forma física. Alan Turing antes de comenzar a diseñar elementos físicos de computadoras, identificó uno de los límites más duros de la matemáticas y de paso creó el campo de la computación. Todo esto cuando todavía no existía una computadora, pero ya sabíamos sus límites.

1.3.1. El rol de las paradojas

En el corazón de estos límites están conceptualizaciones que producen contradicciones y que en esencia son paradójicas. Por ejemplo la siguiente frase

- Esta frase es una mentira

¿Qué nos está tratando de decir esta frase? Si la frase es verdadera, la misma frase nos está diciendo que es falsa; si la frase es falsa, significa que lo dice no es verdad sino lo contrario, osea que la frase es verdadera, pero partimos que no lo era. Existe un elemento paradójico.

Otro ejemplo de paradoja, es la paradoja del barbero que va más o menos así:

Imaginen un pueblo donde todos los habitantes hombres están rasurados, todos. Entonces, podemos dividir a los habitantes del pueblo en dos grupos, aquellos que se rasuran a si mismo y aquellos que los rasura el barbero del pueblo ¿en qué grupo debe ir el barbero del pueblo? 😵

1.4. Dude dónde está mi paradoja

La siguiente es una lista de sentencias que establece el valor de verdad para cada sentencia en relación con la celda denominada realidad:

No.	Sentencia sobre hecho	Realidad	Valor de verdad
0	El círculo es amarillo	🟡	Verdadero
1	El círculo es amarillo	🔴	Falso
2	No hay nada en la siguiente celda		Verdadero
3	No hay nada en la siguiente celda	I	Falso

Por supuesto la “realidad” no se limita a una celda, y la podemos extender a la página o al mundo completo:

No.	Sentencia sobre hecho	Valor de verdad
4	Esta página tiene varias palabras	Verdadero
5	La página está en blanco	Falso
6	Las mascotas son fabulosas	Verdadero

No.	Sentencia sobre hecho	Valor de verdad
7	Los gatos tienen alas	Falso

Un caso interesante es cuando la sentencia habla de si misma:

8	Esta sentencia contiene la palabra verde	Verdadero
9	Esta sentencia contiene tiene la letra z	Verdadero
10	Esta sentencia contiene tiene dos letras z	Falso
11	Esta es una sentencia	

Verdadero

12 | Esto es una novela larga |

Falso

13 | Esto es un color |

Falso

14 | Esta sentencia es verdadera |

Verdadera

15 | Esta sentencia es falsa |

🧐…🤯

¿Por qué es difícil esta última, la número 15?

Respuesta

Existen dos casos:

- Si la sentencia es verdadera, nos dice que es falsa
- Si la sentencia es falsa, entonces lo que dice no es cierto y en realidad nos diría que es verdadera

Es decir, es una paradoja

1.4.1. Enlaces para explorar

- Babbage's Difference Engine No.~2. (2008, mayo 2). [Video]
- The greatest machine that never was - John Graham-Cumming. (2013, junio 19). [Video]
- A demo of Charles Babbage's Difference Engine. (2014, julio 10). [Video]
- Russell's Paradox - A Ripple in the Foundations of Mathematics. (2019, marzo 25). [Video]
- List of paradoxes (2020). [Wikipedia]
- A retrospective on The History of Work(2020). [Sitio web].
- The History of the Workplace(2020). [Sitio web].
- List of pioneers in computer science (2020). [Wikipedia]
- A History of Computing. (s/f).
- Computer History Museum. (s/f).
- Más sobre logaritmos: cómo usar las tablas y algunas curiosidades sobre el tema. (2019). [Texto de blog]
- Logarithm Table. (s/f) [Sitio web]
- Interactive Logarithm Table (s/f) [Sitio interactivo]

1.4.2. Lecturas recomendadas

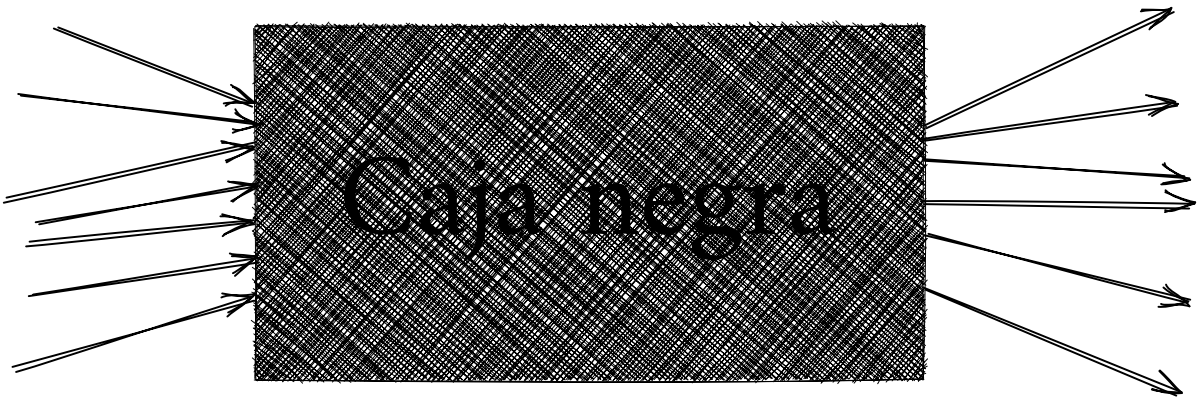
- Greibach, S. A. (1979, October). Formal languages: Origins and directions. In 20th Annual Symposium on Foundations of Computer Science (sfcs 1979) (pp. 66-90). IEEE Computer Society. [BiDi/UNAM], [Liga]
- O'Regan, G. (2012). A brief history of computing (Second edition). Springer. [BiDi/UNAM], [Liga]
- Haigh, T., & Ceruzzi, P. E. (2021). A new history of modern computing. MIT Press. [BiDi/UNAM], [Liga]
- Coello, C. A. C. (2004). Breve historia de la computación y sus pioneros. México: Fondo de cultura económica. [Liga]

2 De lenguajes y palabras

2.1. Un modelo de computadora

Recordemos que queremos responder a la pregunta de ¿Cuáles son las propiedades esenciales de una computadora? Para lograr esto debemos definir una forma de hablar de cualquier máquina capaz de hacer cálculos (computar). Sin embargo hay muchas formas que puede tomar una máquina que computa ¿cómo vamos a poder generalizar sobre todas estas máquinas? Necesitamos un modelo general de máquina computacional. Para lograr esto vamos a concentrarnos en su funcionalidad y no tanto en sus partes, por lo que podemos pensar que es una caja negra.

Podemos comenzar con que una máquina que computa es un objeto físico 🖳 que interactúa con su ambiente: el mundo 🌎. Para lograr esa interacción necesita datos/información de entrada que internamente serán procesados para producir datos/información de salida. Esta es una generalización habitual para nosotros, sin embargo ¿de cuántas entradas estamos hablando? 🧐 las computadoras que habitualmente usamos tiene muchas: teclado, ratón, pantalla táctil, antena internet, joystick, etc.; igualmente para las salidas tenemos muchas opciones, como la pantalla, la impresora, las bocinas, la misma antena de internet, etc. La representación gráfica de esta conceptualización sería la siguiente:



{#ca-janegra, height=250px }

Para simplificar este modelo vamos asumir que nuestra máquina de cómputo recibe una sola entrada en forma secuencial, es decir todas las entradas habituales de nuestra computadora las

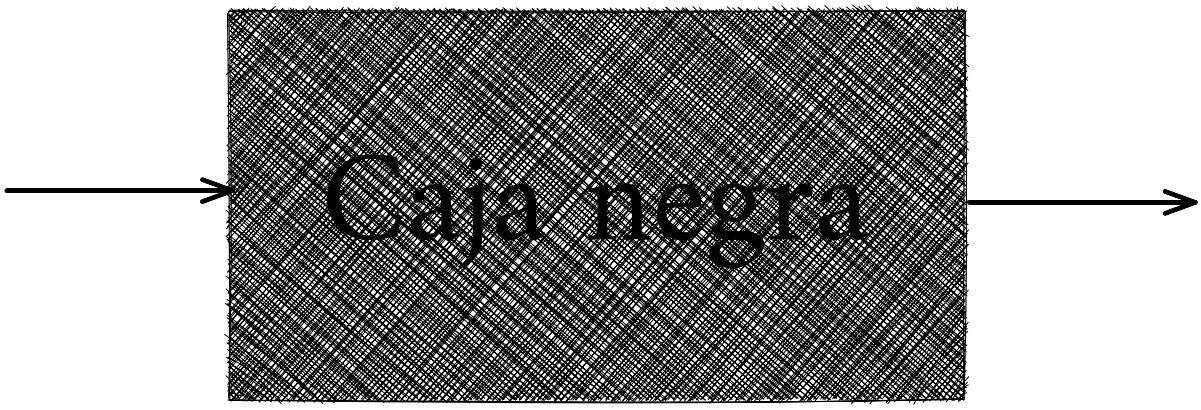
podemos poner una tras otra de forma serial. Existe otro problema con los tipos de entradas, muchas veces nuestros datos tienen diferente 'forma' como números, cadenas, booleanos. Vamos a volver asumir una posición simplista y nada más vamos a dejar que nuestra única entrada reciba únicamente cadenas, los números y booleanos los podríamos codificar como cadenas y de esa forma incluirlos. A lo mejor se preguntan cómo podemos poner todas las entradas una tras otra, sin embargo podemos sincronizar las entradas o podemos usar marcas conformadas por cadenas que identifique de qué dispositivo viene la entrada. Por ejemplo:

MOUSE:200,300,click;KEYBOARD:k;INTERNET:{name:ivan}...

Con respecto a la salida podríamos asumir la misma convención y determinar que sólo hay una salida y es de tipo cadena. Sin embargo, vamos a ir un poco más allá y vamos a poner de lado la mayor parte de la información generada por el proceso, y nos vamos a concentrar en si la máquina de cómputo dada la entrada que le dimos pudo hacer algo 'útil' con ella o no. Esto deja del lado muchos aspectos: como si el resultado se obtuvo de forma rápido, si se ve bonito o si el programa que lo genera fue "bien" hecho ¿Esto quiere decir que la salida no es importante para una máquina de cómputo? No, quiere decir que para lo que resta de esta abstracción de máquina computacional el resultado lo vamos a considerar como un efecto secundario de ejecutar el proceso codificado en la máquina. Lo importante es si la entrada sirvió para generar el resultado o no. Al resultado "que sirvió" lo marcaremos como "verdadero" y al que no como "falso".

2.1.1. Una abstracción de máquina computacional

En el resto del material asumiremos que una máquina computacional es aquella que podemos abstraer como una caja negra que sólo tiene una entrada de tipo cadena y sólo tiene una salida que toma la forma de verdadero (true) o falsa (false).



{#cajanegra2, height=250px }

2.1.2. Experimentando con nuestra abstracción

Ya que tenemos nuestra abstracción de tipo caja negra podemos imaginarnos que podemos experimentar con ella. Podemos tomar una entrada y podemos ver si puede hacer algo 'útil' o no. Esto nos va a permitir ligar el comportamiento de la caja negra con el exterior, de hecho algo que podríamos hacer es pasar todas las entradas posibles y ver su comportamiento para todas las entradas. Por supuesto, esta línea de pensamiento tiene el problema que existen un infinito número de entradas posibles entonces nunca acabaríamos el análisis. Sin embargo, a lo mejor existe la posibilidad de establecer una liga entre el funcionamiento de una máquina computacional con la bastedad de posibilidades de entradas. Pronto veremos que esto es posible y el infinito no es un límite de las máquinas computacionales.

2.1.3. Ventajas de nuestra abstracción

Nuestra abstracción tienen las siguientes ventajas:

- Aunque representa un objeto físico, no depende de elementos físicos tecnología. La caja podría ser: eléctrica, electrónica, óptica, cuántica o mecánica.
- No dependemos del tipo de la información, todo se pone en forma secuencial y de salida sólo se codifica si la máquina pudo hacer algo 'útil' o no con dicha entrada.
- El comportamiento de la máquina está configurado internamente dentro de esta.

2.2. Conceptos básicos

En este momento estamos en la posición de explorar cómo podremos experimentar con nuestra abstracción de máquina de cómputo. Tenemos que definir una forma sistemática para poder presentar todas las entradas posibles que podría aceptar nuestra máquina de cómputo. Para lograr esto vamos a definir unos cuantos conceptos basados en teoría de conjuntos.

2.2.1. Alfabeto

Un alfabeto es un conjunto finito de símbolos básicos. Habitualmente vamos a usar la notación Σ para referirnos a ellos

2.2.1.1. Ejemplos

Algunos ejemplos de alfabetos son:

1. $\{a, b\}$
2. $\{0, 1\}$
3. $\{import, print, for, in, v1, v2, 1, 2, 3, [,], ', ', m, ' '\}$

El primero es un alfabeto de dos símbolos a y b; el segundo se parece pero los símbolos son 0 y 1; finalmente, el último es un alfabeto con 14 símbolos básicos, es interesante que algunos de ellos no son tan básicos, como import o print, pero a consideración del diseñador de estos alfabetos fueron suficientemente "básicos" y como podrán notar corresponden a palabras reservadas del lenguaje de programación python.

2.2.2. Cadenas

Una cadena es una secuencia finita de elementos de un alfabeto Σ. Habitualmente vamos usar la notación w para referirnos a una cadena.

2.2.2.1. Ejemplos

Algunos ejemplos de cadenas son:

1. $bbbbaaa, bab, bbabababbbab$
2. $0, 1, 000, 110, 11100, 001, 00000$
3. $import\ m, for\ v1\ in\ [1, 2, 3]$

El primer inciso ejemplifica tres cadenas usando el primer alfabeto, el segundo inciso muestra siete cadenas del segundo y el tercero dos para nuestro alfabeto inspirado en python. #### Longitud de la cadena

De las cadenas podemos saber la cantidad de elementos básicos que la componen:

$$|w| \rightarrow n$$

Algunos ejemplos de longitud son:

1. $|bbbbaaa| = 7$
2. $|bab| = 3$
3. $|bbabababbbbab| = 15$
4. $|0| = 1$
5. $|1| = 1$
6. $|11100| = 5$
7. $|import\ m| = ??$

Respuesta

3, import es un símbolo básico, m otro y el espacio ' ' el tercer.

2.2.2.2. La cadena vacía

La cadena dónde se elije no elegir ningún elemento del alfabeto se le conoce como la cadena vacía, se le denota con el símbolo ε (epsilón). Notar que su longitud es cero:

$$|\varepsilon| = 0$$

Algunos notas o libros de texto usan el símbolo λ (lambda) en lugar del epsilón. #### Concatenación de dos cadenas

La concatenación es una operación que toma dos argumentos de tipo cadena y construye una nueva cadena donde se ponen los símbolos de la primera cadena seguidos por los de la segunda cadena. Si tenemos dos cadenas, w_1 y w_2, con símbolos representados por aes (que siguen la notación $a_{i,j}$ donde la i representa la cadena y j la posición del símbolo en la cadena) $w_1 = a_{1,1}a_{1,2} ... a_{1,n}$; $w_2 = a_{2,1}a_{2,1} ... a_{2,m}$; entonces la concatenación de w_1 y w_2 es:

{{< katex display>}}w_c=a_{1,1}a_{1,2}...a_{1,n}a_{2,1}a_{2,1}...a_{2,m}{{< /katex>}}.

Algunos ejemplos con cadenas específicas:

1. b y $a = ba$
2. a y $b = ab$
3. ab y $aa = abaa$
4. 0 y 1 = 01
5. ε y 1 = 1
6. 0 y ε = 0
7. 001 y ε = ??

Respuesta

001

2.2.2.2.1. Propiedades de la concatenación La concatenación entre dos cadenas tiene las siguientes propiedades

1. $w_1w_2 \neq w_2w_1$ (no conmutativa)
2. $w_1w_2w_3 = (w_1w_2)w_3 = w_1(w_2w_3)$ (asociativa)

3. $w\varepsilon = \varepsilon w$ (elemento neutro)
4. $|w_1 w_2| = |w_1| + |w_2|$ (suma de longitudes)

2.2.3. Lenguajes

Conjunto de cadenas basado en un alfabeto Σ. Habitualmente vamos a usar la notación L para referirnos a ellos.

2.2.3.1. Ejemplos

Algunos ejemplos de lenguajes son:

1. $\{a, aa, aaa, aaaa, aaaaa, aaaaaa\}$
2. $\{ba, aaab, aaba, aabb, aaaaaabbb\}$
3. $\{a, b, aa, ab, ba, bb, aaa, aba, ...\}$
4. $\{a, b\}$

Ojo, ¿por qué es interesante el tercer lenguaje? ¿y, el cuarto?

Respuesta

El tercer lenguaje es infinito, dado que solo se señala que un lenguaje es un conjunto se permite que estos sean infinitos. El cuarto lenguaje se podría confundir con alfabeto tal como luce; lo único que lo hace lenguaje es el hecho que aparece en una lista de lenguajes.

2.2.3.2. Lenguajes notables

2.2.3.2.1. El lenguaje de la cadena vacía Es el lenguaje que contiene únicamente a la cadena vacía

$L_\varepsilon = \{\varepsilon\}$

Hay que notar que $|L_\varepsilon| = 1$ aún cuando la longitud de ese elemento sea cero (i.e., $|\varepsilon| = 0$)

2.2.3.2.2. El lenguaje vacío Es el lenguaje que no contiene ninguna cadena

- $L_\emptyset = \{\ \}$
- $|L_\emptyset| = 0$

2.2.4. Potencia de un alfabeto

La potencia n de un alfabeto (Σ^n) resulta en un lenguaje conformado por las cadenas que se puedan componer concatenando n símbolos del alfabeto.

2.2.4.1. Ejemplos de potencias de un alfabeto

Considere el alfabeto $\Sigma = \{a, b\}$

1. $\Sigma^1 = \{a, b\}$
2. $\Sigma^2 = \{aa, ab, ba, bb\}$
3. $\Sigma^3 = \{aaa, baa, aba, aab, abb, bab, bba, bbb\}$
4. $\Sigma^4 = \{aaaa, baaa, abaa, aaba, \dots, bbbb\}$

Considere la potencia cero del mismo alfabeto:

1. $\Sigma^0 = \{\varepsilon\}$

Este resultado parecería extraño ¿por qué la potencia cero, genera un lenguaje con un elemento y no el lenguaje vació? Hay que recordar que la potencia indica el número símbolos a concatenar en las cadenas de ese lenguaje, y la cadena que se hace concatenando cero símbolos, es la cadena vacía.

2.2.4.2. Otro lenguaje notable

Si unimos todas las potencias de un alfabeto obtenemos el lenguaje de todas las cadenas posibles de ese alfabeto para el cual usamos la notaciónΣ^*:

$$\Sigma^* = \{\varepsilon\} \cup \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \Sigma^4 \cup \dots$$

Una consecuencia de definir a este lenguaje es que cualquier L basado en un Σ es un subconjunto de Σ^*. En notación de conjuntos esta circunstancis se escribe como:

$$L \subset \Sigma^*$$

2.3. Operaciones con lenguajes

2.3.1. Concatenación de dos lenguajes

Esta operación toma como argumentos dos lenguajes y da como resultado un nuevo lenguaje que contiene todas las cadenas posibles al concatenar cadenas del primer lenguaje con cadenas del segundo lenguaje.

$L_1L_2 = \{w_1w_2 | w_1 \in L_1, w_2 \in L_2\}$

Si suponemos $L_1 = \{aa\}$, $L_2 = \{a, b\}$, $L_3 = \{\varepsilon\}$ y $L_4 = \{a, aa, aaa, aaaa, ...\}$

1. $L_1L_2 = \{aaa, aab\}$
2. $L_2L_1 = \{aaa, baa\}$
3. $L_2L_2 = \{aa, ab, ba, bb\}$
4. $L_2L_3 = \{a, b\}$
5. $L_1L_4 = \{aaa, aaaa, aaaaa, aaaaaa, ...\}$
6. $L_4L_3 = ??$

Respuesta

$\{a, aa, aaa, aaaa, ...\}$

2.3.2. Potencia de un lenguaje

La potencia n de un lenguaje (representado como L^n) toma como argumento un lenguaje y resulta en un nuevo lenguaje conformado por las cadenas que se puedan componer concatenando n veces las cadenas del lenguaje L original.

2.3.2.0.1. Ejemplos de potencias de un lenguaje

Suponiendo $L = \{0, 11\}$

1. $L^1 = \{0, 11\}$
2. $L^2 = \{00, 011, 110, 1111\}$

Considere el siguiente caso:

1. $L^0 = ??$

¿El lenguaje que se crea si se considera no concatenar ninguna de las cadenas del lenguaje L?

Respuesta

$\{\varepsilon\}$

2.3.3. Cerraduras de lenguajes

De manera análoga como se creo la potencia estrella de un alfabeto, podemos generar la potencia estrella de un lenguaje L^* de hecho le llamaremos cerradura y tenemos dos opciones:

- Cerradura de Kleene estrella $L^* = \bigcup_{i=0}^{\infty} L^i$
- Cerradura de Kleene más $L^+ = \bigcup_{i=1}^{\infty} L^i$

Notar que la diferencia es que cerrdura _*_ (estrella) comienza del índice cero y + (más) del índice 1. Esta diferencia sutil garantiza que cerradura _*_ siempre incluye a la cadena vacía ya que esta cerradura incluye a la potencia cero de un lenguaje.

2.3.3.1. Ejemplos de cerraduras

Suponiendo $L = \{aa\}$

1. $L^* = \{\varepsilon, aa, aaaa, aaaaaa, aaaaaaaa, ...\}$
2. $L^+ = \{\ \ aa, aaaa, aaaaaa, aaaaaaaa, ...\}$

2.3.3.2. Cerraduras notables

Algunas cerraduras que son relevantes conocer:

1. $\{\varepsilon\}^* = \{\varepsilon\}$
2. $\{\varepsilon\}^+ = \{\varepsilon\}$
3. $\emptyset^* = \{\varepsilon\}$
4. $\emptyset^+ = \{\ \}$

2.3.3.3. La importancia del lenguaje de la cadena vacía

1. $\{\varepsilon\}L = L$
2. $L\{\varepsilon\} = L$
3. $\{\epsilon, ...\}L = (\{\epsilon\} \cup L_r)L = (\{\epsilon\}L) \cup (L_rL) = L \cup L_rL$

2.3.4. Operaciones sobre conjuntos

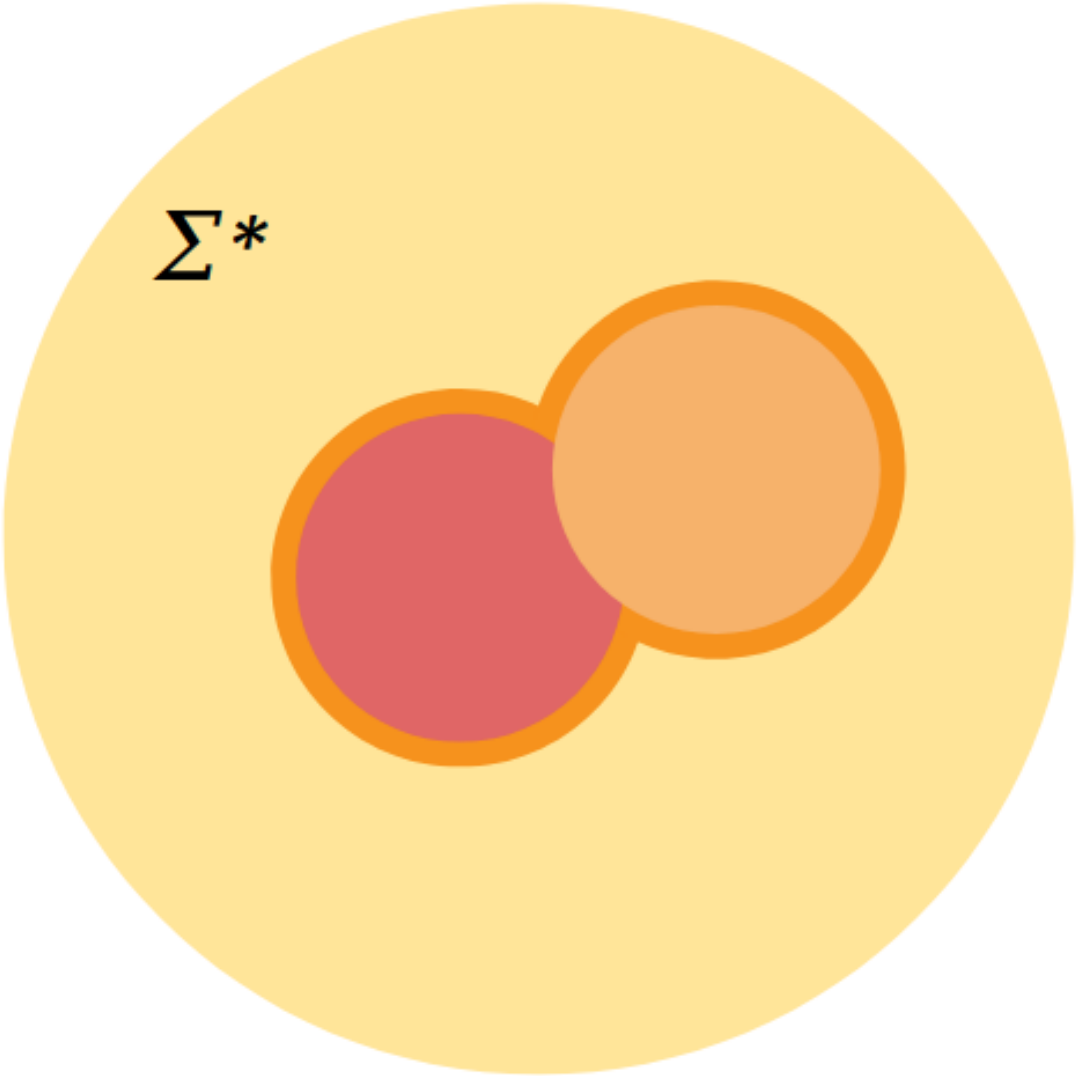
Como los lenguajes son conjuntos, podemos usar las operaciones sobre los conjuntos

1. Union: $L_1 \cup L_2$

2. Intersección: $L_2 \cap L_2$
3. Diferencia: $L_1 - L_2$
4. Complemento: $\overline{L} = \Sigma^* - L$

2.3.5. Visualización de operaciones con lenguajes

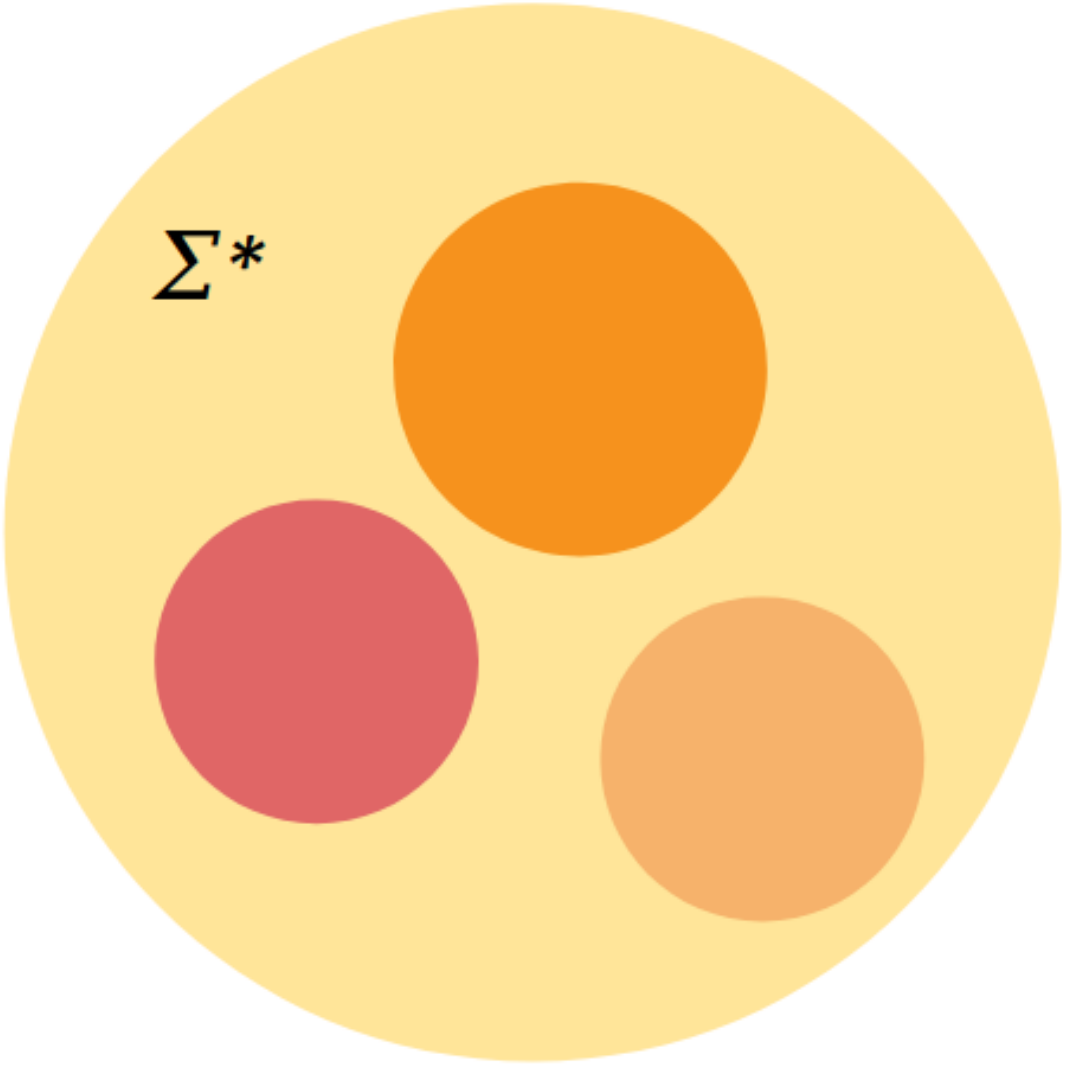
2.3.5.1. Unión



{#unión, height=250px }

El resultado de la unión entre dos lenguajes es un nuevo conjunto (● naranja fuerte) que contiene todas las cadenas del lenguaje uno (● rojo) y lenguaje dos (● naranja suave).

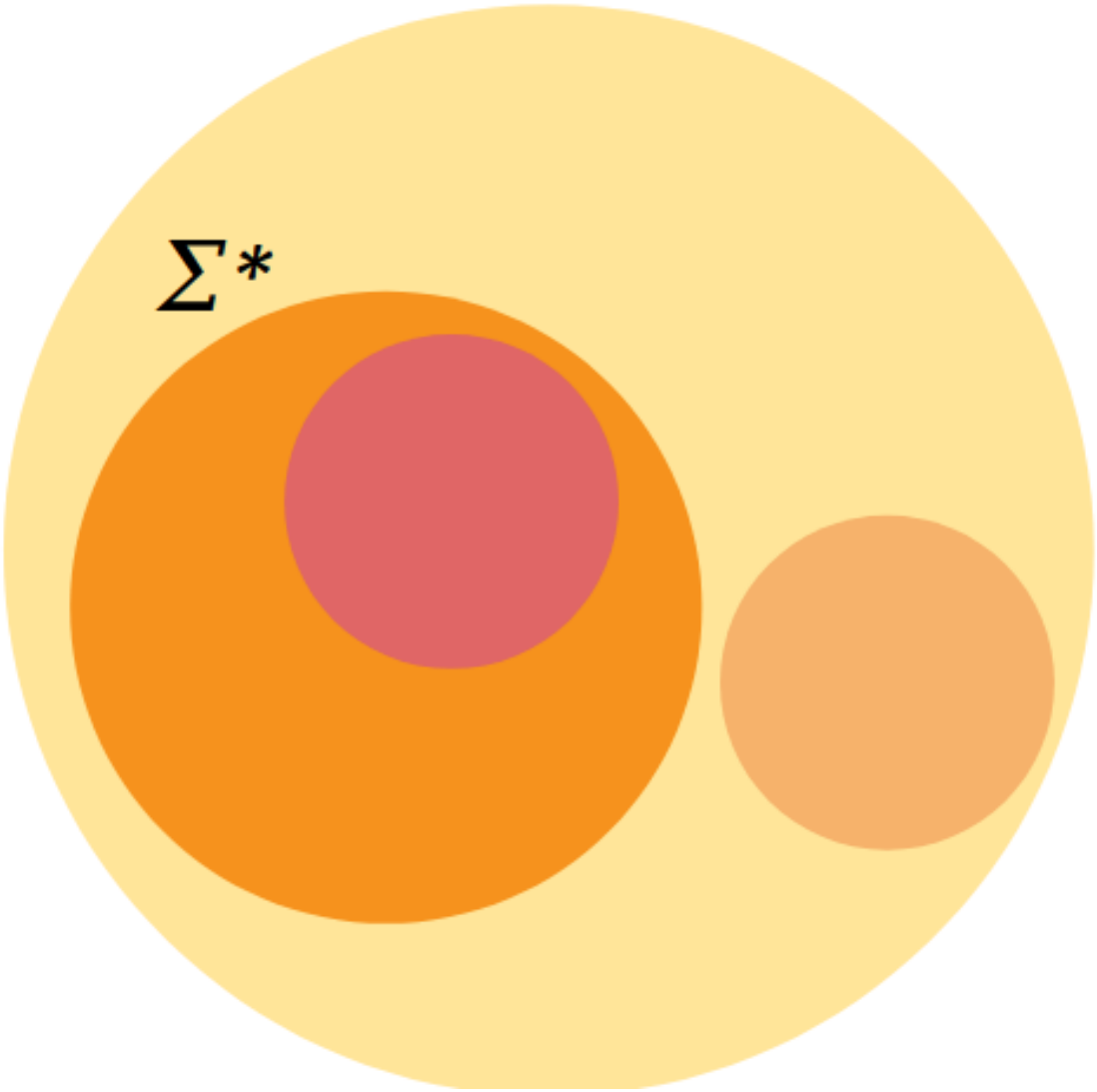
2.3.5.2. Concatenación



{#concatenación, height=250px }

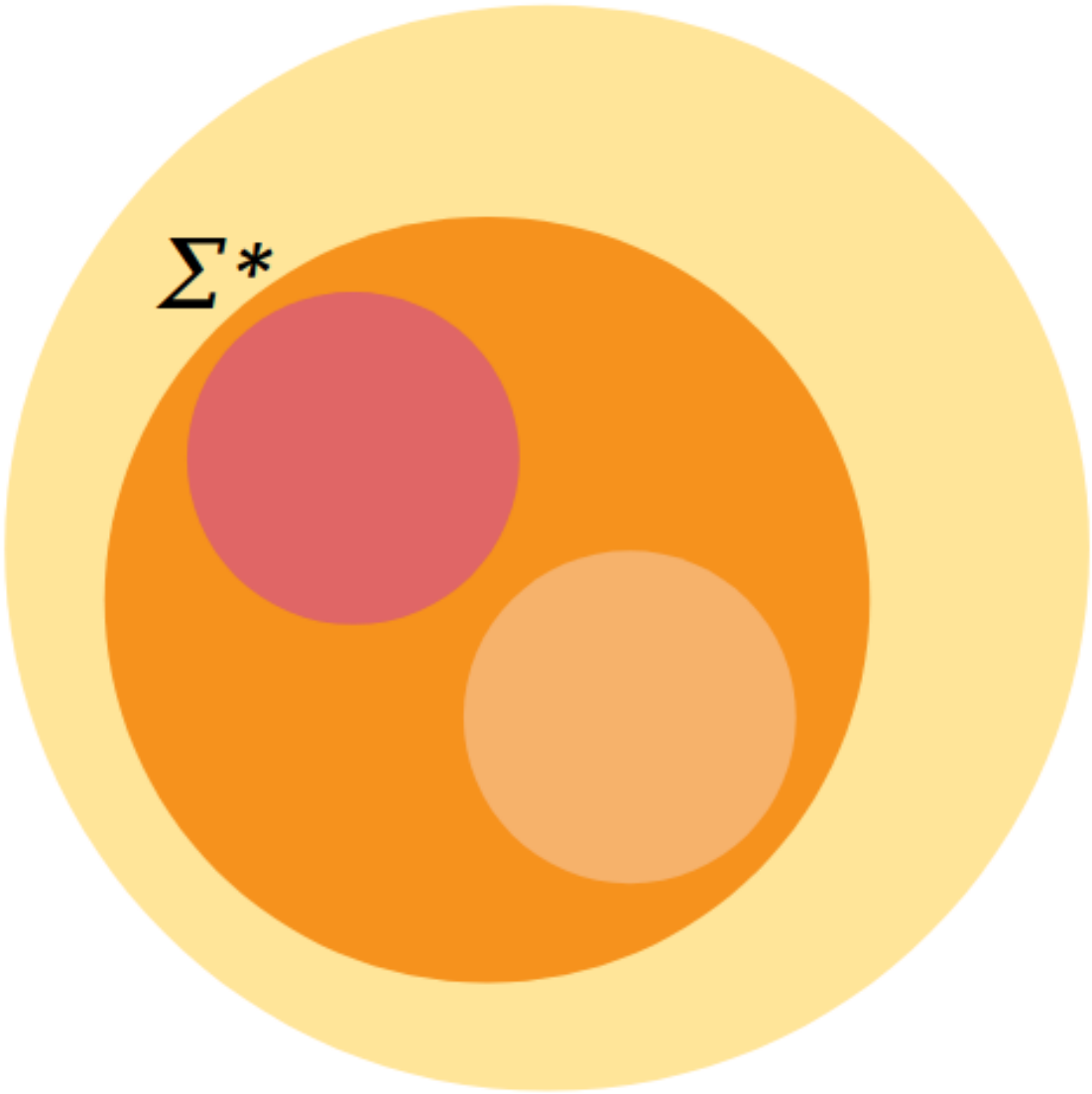
El resultado de la concatenación entre dos lenguajes es un nuevo conjunto (● naranja fuerte) que contiene todas las cadenas que se pueden formar de concatenar cadenas del lenguaje uno (● rojo) con cadenas del lenguaje dos (● naranja suave).

Sin embargo es importante notar, que si uno de los lenguajes contiene a la cadena vacía (ε) entonces, el lenguaje contrario está contenido en el resultado.



{#con-
catenación2, height=250px }

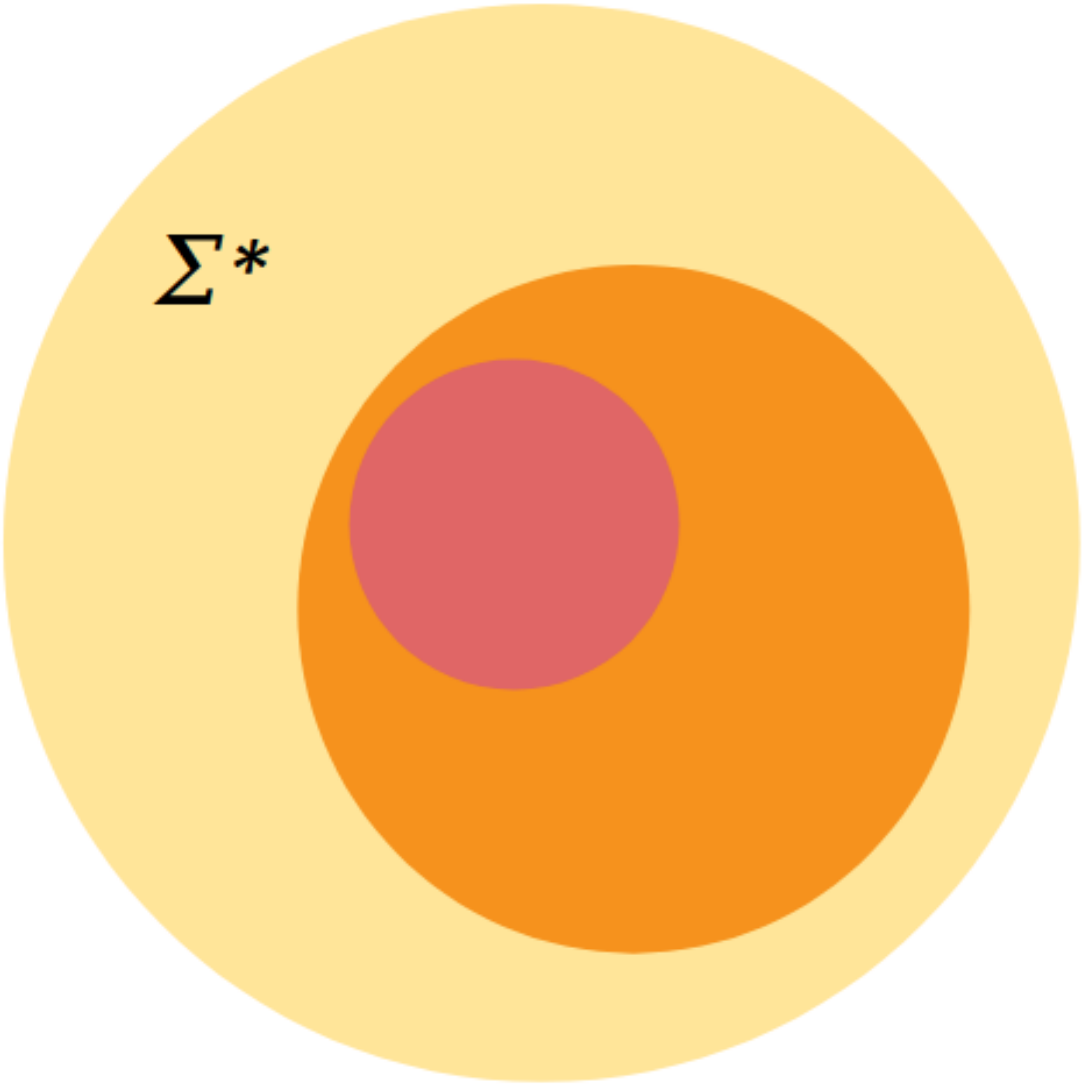
Por otro lado, si ambos lenguajes contienen a la cadena vacía (ε) ambos son parte del resultado.



{#con-

catenación3, height=250px }

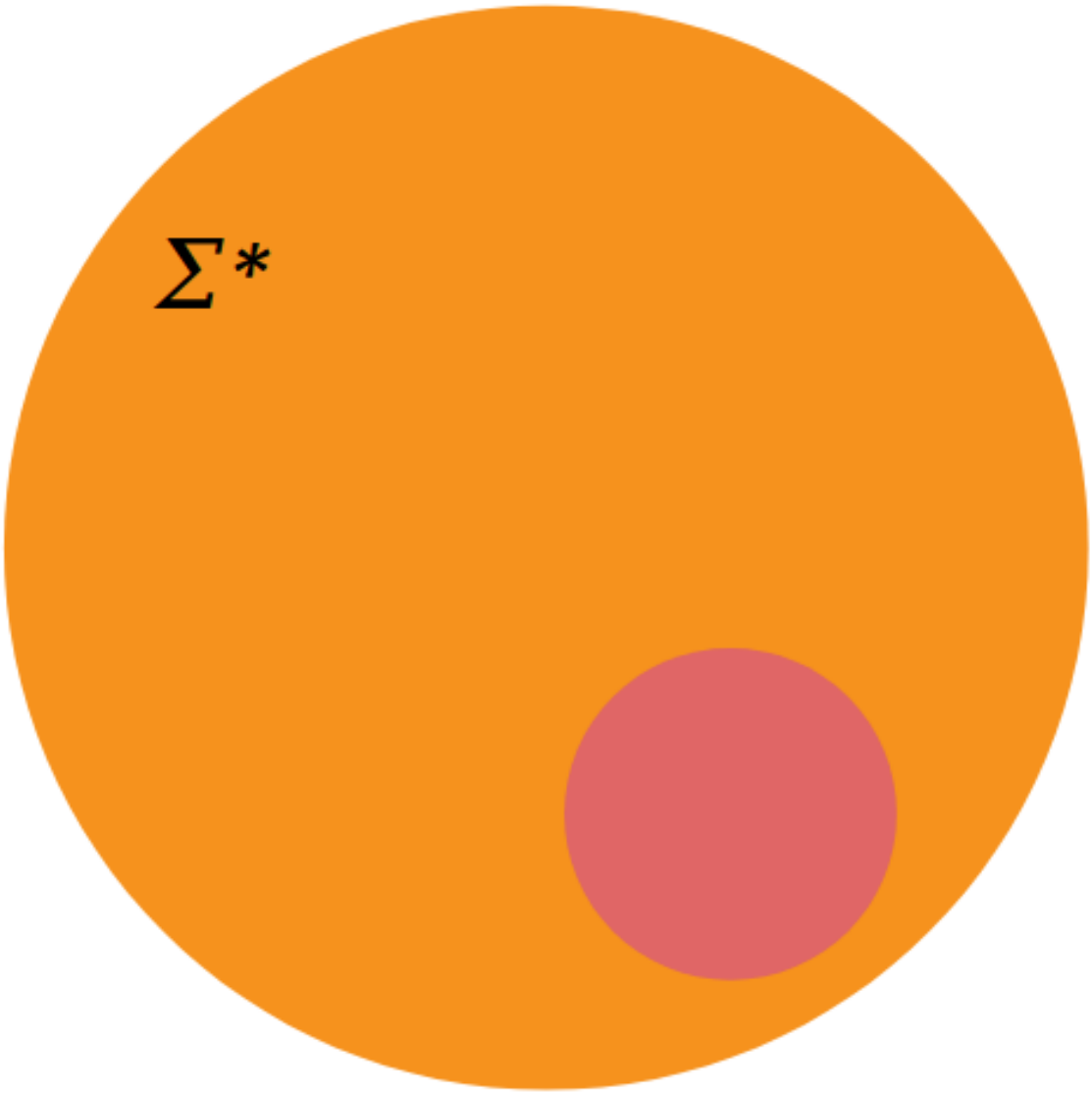
2.3.5.3. Cerradura



{#cerradura, height=250px }

El resultado de la cerradura de un lenguaje es un nuevo lenguaje (● naranja fuerte) que contiene todas las cadenas que se pueden formar de concatenar de cero a más veces las cadenas de lenguaje original (● rojo). Como una de las concatenaciones que se permite es la de una vez de las cadenas del lenguaje, entonces el resultado siempre contiene al lenguaje original.

2.3.5.4. Complemento



{#complemento, height=250px }

El resultado del complemento de un lenguaje es un lenguaje (● naranja fuerte) que contiene todas las cadenas que no están en el lenguaje original (● rojo).

2.4. Formas de hablar de los lenguajes

Hasta ahora hemos visto diferentes formas de hablar sobre los lenguajes:

1. Enumerando los elementos del lenguaje

> $L = \{a, b, aa, ab, ba, bb, ...\}$

2. Usando una combinación de operaciones

> $\{a, b\}^*$

3. Describiendo una propiedad

> $\{w | w$ esta compuesta por los símbolos a y b$\}$

4. De forma descriptiva

> El lenguaje cuyas cadenas están compuesta por aes y bes

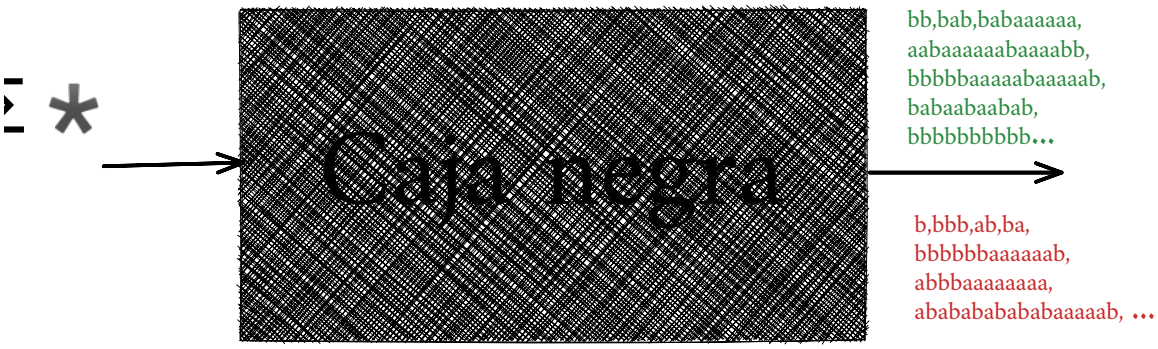
Durante el material aprenderemos nuevas formas de hablar de los lenguajes.

3 La máqina sin memoria

3.1. Revisando nuestro modelo

En el capítulo anterior presentamos nuestra abstracción de máquina computacional como una caja negra que recibe un entrada compuesta de una cadena y produce una salida que determina si para esa cadena se pudo hacer algo "útil" o no. En el capítulo anterior también presentamos el concepto de cadena. No sólo eso sino que se presentó el lenguaje Σ^* que contiene a todas las cadenas posibles compuestas por símbolos de un alfabeto Σ.

¿Qué pasa si pasamos todo el conjunto Σ^* a una máquina computacional? La máquina, según como esté programada va a determinar que unas cadenas de este conjunto serán útiles mientras que otras no.



{#ca-janegra, height=250px }

El resultado de ese experimento es que efectivamente nuestra abstracción estaría separando al lenguaje Σ^* en dos lenguajes:

1. El lenguaje L conformado por cadenas que fueron útiles para una caja específica.
2. El lenguaje $\overline{L}$ conformado por cadenas que no fueron útiles para una caja específica.

Esta observación es muy importante porque permite apreciar que podemos establecer una relación entre el conjunto de palabras que son útiles para una máquina específica y las piezas internas de esta, por lo que serán estas piezas internas las que van a determinar si una cadena es útil o no.

Si ponemos atención es fácil ver que $\overline{L}$ es el complemento de L.

{#ca-janegra2, height=250px }

3.2. Lenguajes regulares

Vamos a definir a un tipo de lenguajes que denominaremos como Lenguaje Regulares (Lreg). Para lograr esto se va usar una definición recursiva.

3.2.1. Lenguaje Regular

Hay dos tipos de lenguajes regulares:

- Lenguajes regulares básicos
- y los resultantes de operaciones sobre lenguajes regulares.

3.2.1.1. Lenguajes regulares básicos

Un lenguaje L es regular si se trata de uno de los siguientes casos:

1. El lenguaje $\emptyset$, es decir, el lenguaje vacío.
2. El lenguaje $\{\varepsilon\}$, es decir, el lenguaje de la cadena vacía.
3. El lenguaje $\{a\}$ donde $a \in \Sigma$. A este lenguaje se le conoce como el lenguaje de un símbolo del alfabeto.

3.2.1.2. Lenguajes regulares resultantes de operaciones

Adicionalmente un lenguaje es regular si resulta de una de estas operaciones:

1. Unión

   Si L_1 y L_2 son regulares, entonces $L_1 \cup L_2$ es regular también.

2. Concatenación

   Si L_1 y L_2 son regulares, entonces $L_1 L_2$ es regular también.

3. Cerradura estrella

   Si L es regular, entonces L^* es regular también.

4. Priorización por paréntesis

   SiL es regular, entonces (L) es regular también.

En resumen un lenguaje L es regular si corresponde con uno de los casos básicos: vacío, de la cadena vacía o lenguaje de un símbolo ; o se puede generar a través de una secuencia finita de operaciones sobre lenguajes regulares a través de las operaciones de unión, concatenación, cerradura estrella y priorización por paréntesis.

3.2.2. Ejemplo: número de *bes* pares

¿Cómo lucen cadenas de este lenguaje?

$\{bb, bab, babaaaaaaa, aabaaaaabaaaabb, bbbbbbaaaaaaaabaaaaab, bababaabaabab, bbbbbbbb, ...\}$

¿Cómo lucen cadenas que no están en este lenguaje?

$\{b, bbb, ab, ba, bbbbbaaaaaaaaaaaaabb, ...\}$

¿Cómo podríamos generar este lenguaje a través de nuestras operaciones?

Partiendo de $\{b\}$ como regular, por ser uno de los casos básicos:

1. $\{b\}\{b\}$ es regular por concatenación, dos bes seguidas
2. $\{b\}\{a\}\{b\}$es regular por concatenación, puede aparecer una a entre las dos bes
3. $\{b\}\{a\}^*\{b\}$ es regular por cerradura, pueden aparecer cualquier cantidas de aes entre las dos bes
4. $\{a\}^*\{b\}\{a\}^*\{b\}\{a\}^*$ es regular por concatenación, en realidad pueden aparecer cualquier cantidad de aes antes, en medio y después de las bes
5. $(\{a\}^*\{b\}\{a\}^*\{b\}\{a\}^*)^*$ pueden aparecer cualquier cantidad de veces el patrón de dos bes con cualquier cantidad de aes antes, en medio o después.

Construcción erronea

La construcción anterior cubre la mayoría de los aspectos sin embargo deja fuera el caso en que la cadena esté conformada por puras aes; dado que una vez en el patrón de repetición que forza la cerradura estrella más externa, siempre tienen que aparecer dos bes; por lo tanto necesitamos una opción dónde aparezcan solo aes, este efecto lo podemos lograr si unimos este lenguaje con el lenguaje de puras áes resultando en la siguiente operación:

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

3.2.2.1. Algunas alternativas

Reflexionar por qué las siguientes operaciones también generan el mismo lenguaje.

1. $\{a\}^*(\{b\}\{a\}^*\{b\}\{a\}^*)^*$
2. $(\{a\}^*(\{b\}\{a\}^*\{b\})^*\{a\}^*$

Respuesta

1. Las repeticiones provocadas por la cerradura estrella comienzan con be, como la repetición incluye opcionalmente aes al final, a partir del primer par, podrán aparecer aes entre las segundas bes y la siguiente be; pero esto deja sin la opción de que aparezcan aes al comienzo de la cadena por lo que se concatena estas aes de forma opcional antes de la repetición.
2. En este patrón, la repetición que comience con aes y que estas aparezcan entre las segundas bes y las siguiente be, sin embargo la repetición no garantiza que las cadenas puedan acabar con aes por lo que se concatenan de forma opcional al final de esta y no como parte de la repetición.

3.3. Expresiones regulares

Las expresiones regulares son una relajación en la notación de lenguajes regulares. El objetivo es hacer más visible el patrón de concatenación, para lograr esto se omiten las llaves de la notación de conjuntos y se cambia el símbolo de la unión por un símbolo $+$.

De nuevo para la definición formal recurrimos a una definición recursiva.

3.3.1. Expresiones regulares básicas

1. $\emptyset$ representa al lenguaje vacío
2. ε representa al lenguaje de la cadena vacía.
3. a donde $a \in \Sigma$ representa al lenguaje de un símbolo del alfabeto.

Cabe recordar que estos casos por definición son lenguajes regulares, en su notación como expresión regular

3.3.2. Expresiones regulares para operaciones

1. $L_1 + L_2$ representa la unión de dos lenguajes.
2. $L_1 L_2$ representa la concatenación.
3. L^* representa la cerradura estrella de un lenguaje.
4. (L) representa la priorización por paréntesis de un lenguaje.

3.3.2.1. Ejemplo: número de *bes* pares

Con esto en mente, la expresión regular para las cadenas que tienen número par de bes con $\Sigma = \{a, b\}$ queda como:

$$(a^*ba^*ba^*)^* + a^*$$

Alternativamente

$$a^*(ba^*ba^*)^*$$

o

$$(a^*ba^*b)^*a^*$$

3.3.2.2. Ejemplo: Penúltimo símbolo una *a*

La notación de expresiones regulares nos permite enfocarnos en el patrón que define a las cadenas de nuestro lenguaje. El siguiente ejemplo muestra como se puede construir una expresión regular para el lenguaje de todas las cadenas cuyo último símbolo sea una a con $\Sigma = \{a, b\}$:

1. Cualquier cosa • dos últimas posiciones (para que exista penúltima posición)

2. Cualquier cosa • penúltima posición • última posición
3. Cualquier cosa • a • última posición
4. a o b tantas veces queramos • a • a o b
5. $a + b$tantas veces queramos • a • $a + b$
6. $(a + b)^*$• a • $a + b$

Finalmente este patrón lo podemos representar con la ER:

$$(a + b)^* a(a + b)$$

¿Cómo luciría la ER para todas las cadenas que comienzan con una a con $\Sigma = \{a, b\}$?

Respuesta

$$a(a + b)^*$$

3.4. Autómatas finitos

En este momento vamos a hacer un cambio de paradigma, y vamos a presentar nuestra primera máquina computacional que denominaremos Autómata Finito (determinístico) (AF o AFD). El propósito de un autómata finito es procesar una cadena símbolo del alfabeto por símbolo del alfabeto hasta consumir toda la cadena; dependiendo en qué estado se encuentra se determina si la cadena se acepta (es decir, si fue útil).

3.4.1. Autómata finito (determinístico)

Definición 1. Un autómata finito es una tupla $(Q, \Sigma, q_0, A, \delta)$ donde:

- Q es un conjunto de estados finitos
- Σ es un alfabeto
- q_0 es un estado que denominaremos inicial donde $q_0 \in Q$
- A es un conjunto de estados que denominaremos finales donde $A \subset Q$
- δ es una función de transición que cumple con: $\delta : Q \times \Sigma \to Q$

3.4.2. Ejemplo: Número par de bes

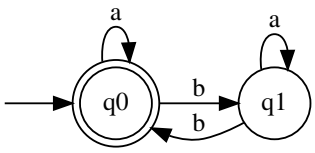
En forma de tupla el autómata finito que acepta a cadenas con un número par de bes queda definido como:

$$(\{q_0, q_1\}, \{a, b\}, q_0, \{q_0\}, \delta)$$

Donde

$$\delta = \begin{cases} (q_0, a) \to q_0 \\ (q_0, b) \to q_1 \\ (q_1, a) \to q_1 \\ (q_1, b) \to q_0 \end{cases}$$

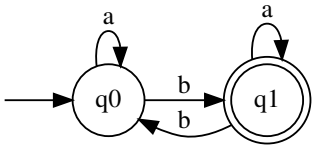
De forma gráfica el autómata se representa de esta forma:



{#bes_pares, height=250px }

Los círculos y sus etiquetas representan a los estados, en este caso dos. Uno de estos estados tiene la propiedad de ser inicial, marcado aquí con una flecha que no proviene de ningún lado. Todos los estados potencialmente pueden tener la propiedad de ser final, marcado aquí por un doble círculo. Los símbolos del alfabeto lo usamos para marcar las transiciones (arcos). Las transiciones son la definen la función de transición, se lee: "Si estoy en estado q y llega un símbolo a, transicionó de ese estado al que me indique la flecha".

3.4.2.1. Variantes



{#bes_pares1, height=250px }

$$(\{q_0, q_1\}, \{a, b\}, q_0, \{q_1\}, \delta)$$

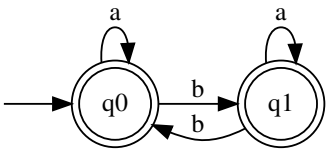
Donde

$$\delta = \begin{cases} (q_0, a) \to q_0 \\ (q_0, b) \to q_1 \\ (q_1, a) \to q_1 \\ (q_1, b) \to q_0 \end{cases}$$

¿Qué cadenas acepta este autómata finito?

Respuesta

Acepta a las cadenas con número impar de cantidad de bes, es decir el complemento de nuestro lenguaje número par de bes



{#bes_pares2, height=250px }

$$(\{q_0, q_1\}, \{a, b\}, q_0, \{q_0, q_1\}, \delta)$$

Donde

$$\delta = \begin{cases} (q_0, a) \to q_0 \\ (q_0, b) \to q_1 \\ (q_1, a) \to q_1 \\ (q_1, b) \to q_0 \end{cases}$$

¿Qué cadenas acepta este autómata finito?

Respuesta

Acepta todas las posibles cadenas, porque no importa si tiene par o impar, de todas formas la cadena resulta “útil”.

3.4.3. Procesando cadenas con AF

Un problema con nuestra definición hasta este momento es que la función de transición solo procesa un símbolo a la vez. Para que un AF pueda procesar una cadena tenemos que extenderlo con la función de transición extendida para la que usaremos la notación:

$$\delta^*$$

Esta se define de forma recursiva de la siguiente forma:

$$\delta^* = \begin{cases} \delta^*(q, \epsilon) = q & q \subseteq Q \\ \delta^*(q, wa) = \delta(\delta^*(q, w), a) & q \subseteq Q, w \subseteq \Sigma^*, a \subseteq \Sigma \end{cases}$$

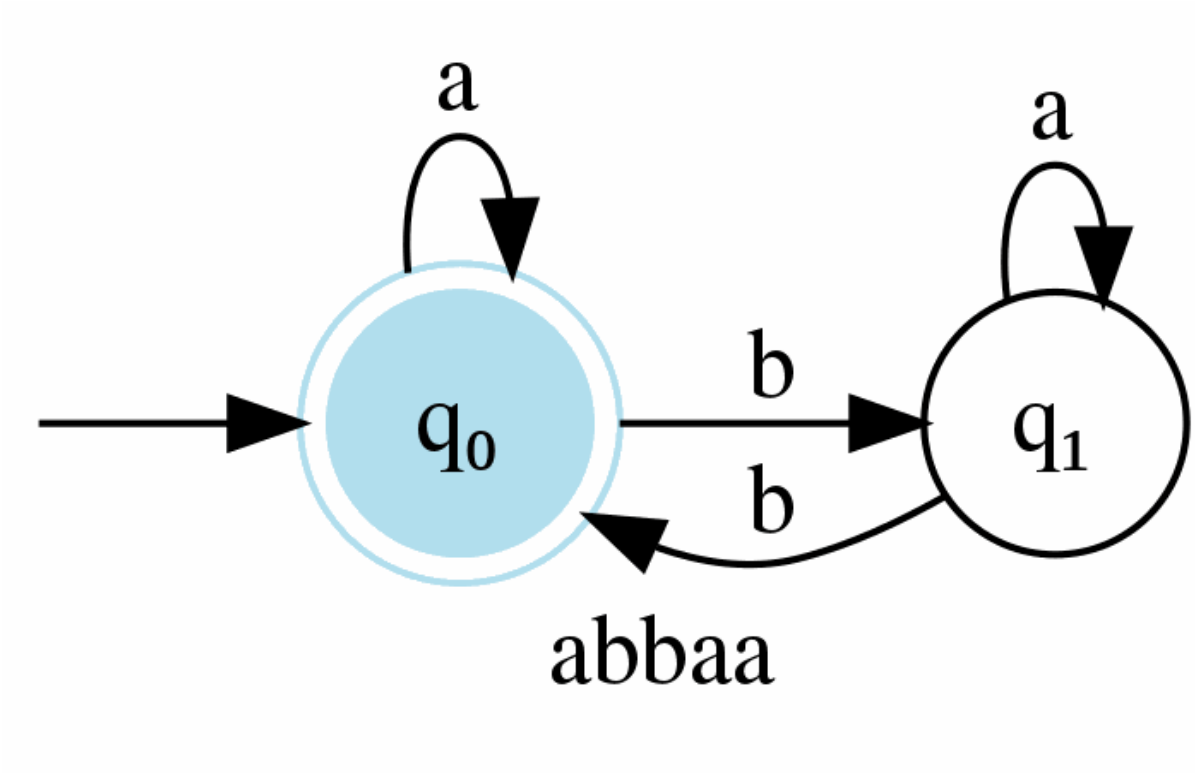
Si el $\delta^*(w) \in A$ entonces w es aceptada por el autómata.

3.4.4. Ejemplo: AF par de *bes* con la cadena *abbaa*

$$\begin{aligned} \delta^*(q_0, abbaa) &= \delta(\delta^*(q_0, abba), a) \\ &= \delta(\delta(\delta^*(q_0, abb), a), a) \\ &= \delta(\delta(\delta(\delta^*(q_0, ab)b), a), a) \\ &= \delta(\delta(\delta(\delta(\delta^*(q_0, a), b), b), a), a) \\ &= \delta(\delta(\delta(\delta(\delta(\delta^*(q_0, \epsilon), a), b), b), a), a) \\ &= \delta(\delta(\delta(\delta(\delta(q_0, a), b), b), a), a) \\ &= \delta(\delta(\delta(\delta(q_0, b), b), a), a) \\ &= \delta(\delta(\delta(q_1, b), a), a) \\ &= \delta(\delta(q_0, a), a) \\ &= \delta(q_0, a) \\ &= q_0 \end{aligned}$$

En el caso de la cadena abbaa $\delta^*(abbaa) = q_0$ donde $q_0 \in A$ entonces la cadena es aceptada por el autómata.

3.4.4.1. De forma gráfica

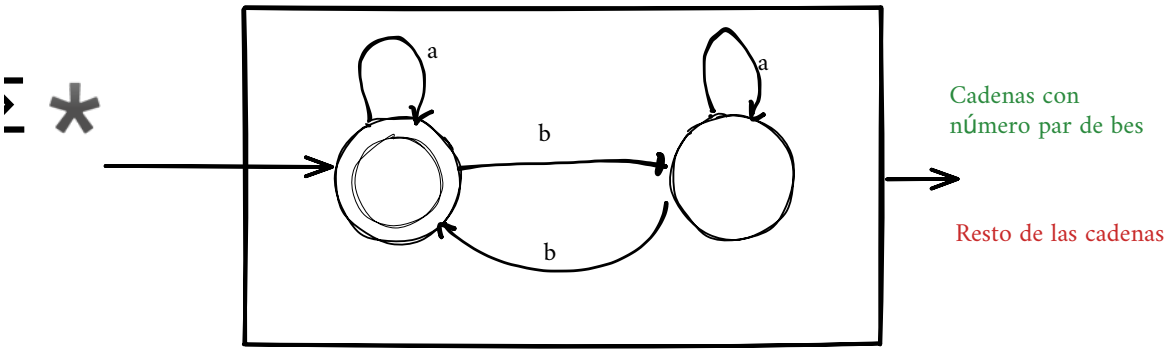


{#bes_pares, height=250px }

3.5. El lenguaje aceptado por un AF

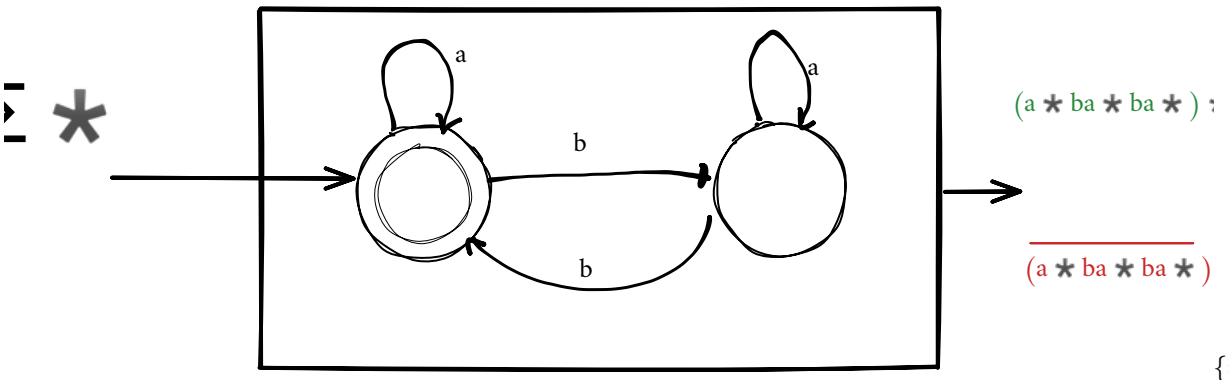
Un autómata finito A acepta al lenguaje L si:

1. Si $w \in L$ es aceptada por A
2. Si $w \notin L$ no es aceptada por A



{#cajanegra3, height=250px }

Es decir:



{#ca-janegra4, height=250px }

3.6. Teorema de Kleen

Teorema 1. Un lenguaje L sobre el alfabeto Σ es regular si y sólo si existe un Autómata Finito con un alfabeto Σ que acepte L.

3.7. Reflexión sobre autómatas finitos

3.7.1. ¿Qué sabe un AF?

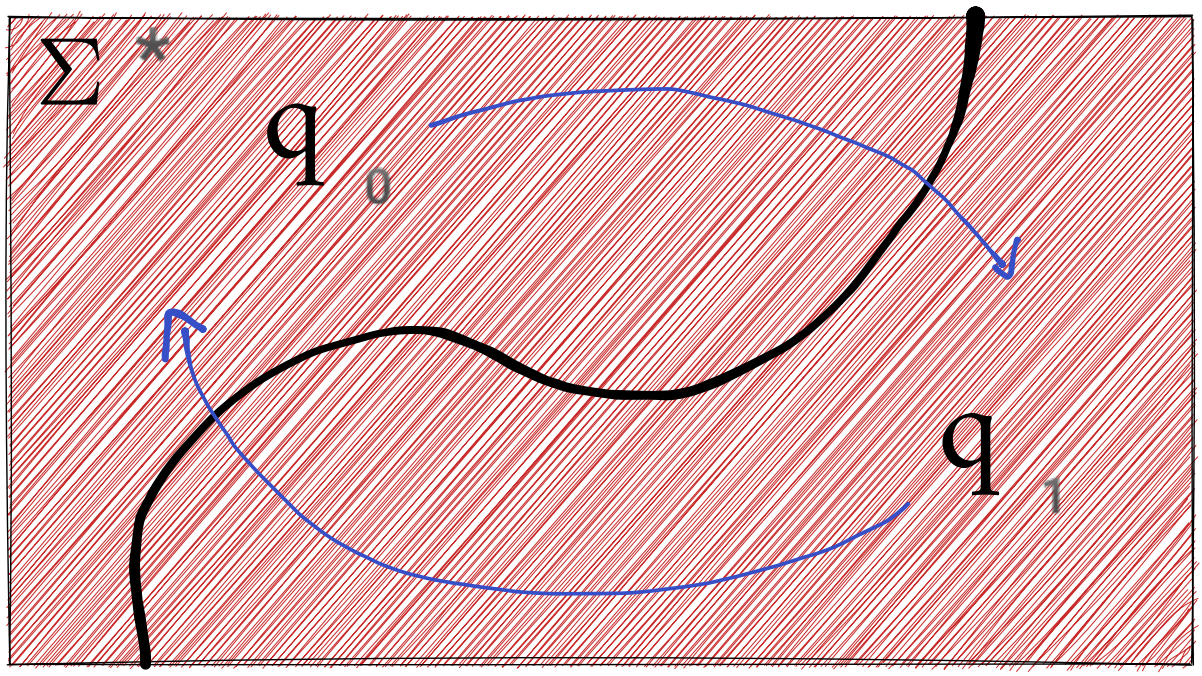
Para procesar una cadena lo único que sabe un autómata es:

1. La posición de la cadena que ya analizó
2. El estado en el que se encuentra

3.7.1.1. ¿Qué le pasa al espacio de Σ^* mientras procesamos una cadena?

Nos movemos entre los conjuntos de cadenas aceptadas y cadenas rechazadas. El espacio lo podemos dividir en dos partes: cuando estamos en una estado no final y cuando estamos en un estado final.

Por ejemplo, para el lenguaje de cadenas con número par de bes (aquí como expresión regular y aquí como autómata finito), habría el conjunto de estados se separaría en dos uno asociado a q_0 y otro a q_1.



{#sigma, height=250px }

3.8. Formas de saber si un lenguaje es regular

Hasta este momento estás son las formas que hemos visto para saber que un lenguaje es regular:

1. Es un lenguaje regular básico o lo podemos componer a través de las operaciones de unión, concatenación, cerradura estrella o priorización con paréntesis.

2. Diseñamos una expresión regular que lo represente

3. Diseñamos un Automata finito que lo acepte

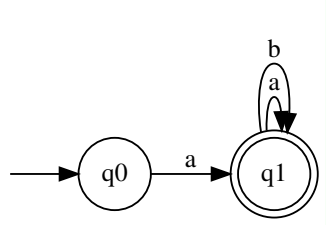
Considerando estas opciones, cómo sabemos que el lenguaje compuesto por cadenas que comienzan con a con $\Sigma = \{a, b\}$ es regular:

Respuesta

1. Surge de la secuencia de operaciones $\{a\}$ concatenada con $\{a, b\}^*$
2. La representa la expresión regular:

$$a(a+b)^*$$

3. Lo acepta el autómata



{#a_loquesea, height=250px }

4 La máqina que están en varios lugares

4.1. Especificación contra proceso

Por el Teorema de Kleene sabemos que tanto Expresiones Regulares (una relajación en la notación de los Lenguajes Regulares) son equivalentes a los Autómatas Finitos (no determinístico). El que sean equivalentes quiere decir que representan exactamente al mismo tipo de lenguajes, los Lenguajes Regulares. Este es una característica muy importante de nuestro modelo computacional.

Esta correspondencia tiene varias implicaciones:

- Si sabemos de un Lenguaje Regular L está garantizado que debe existir un autómata finito.
- Si sabemos sobre un Autómata Finito A está garantizado que debe existir un Lenguaje Regular y por lo tanto una Expresión Regular que los represente.

4.1.1. ¿Cúal es la diferencia en *ER* y Autómata?

¿Si ambos ER y Autómatas representan el mismo tipo de lenguajes porque necesitamos a ambos, porque no quedarnos con una de la opciones? La respuesta tiene que ver con que ER y AF son dos aspectos diferentes de los Lenguajes Regulares. Las Expresiones Regulares especifican el patrón que debe seguir las cadenas que pertenecen al lenguaje, mientras que el Autómata Finito define el proceso que se tiene que seguir para aceptar a una cadena del lenguaje.

4.1.1.1. Ejemplo de especificación vs proceso

Supongamos el lenguaje de las cadenas conformadas por aes y bes en donde todas las cadenas contiene la subcadena abb.

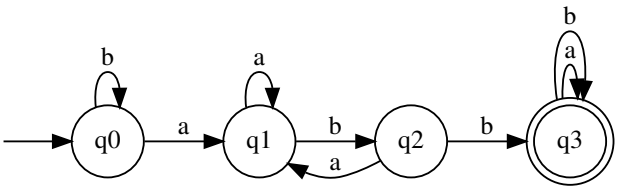
$$L = \{abb, aabba, babba, babbb, ...\}$$

La ER que representa a este lenguaje es $(a + b)^* abb(a + b)*$. Si nos detenemos a analizar esta ER podemos ver que nos está especificando la forma que tienen que ser las cadenas, nos dice que

puede venir cualquier cosa $(a+b)^*$, es decir cualquier cadena compuesta por repeticiones de aes o bes. Sin embargo, en algún momento debe aparecer la cadena abb para después ser seguida por cualquier cosa. El infijo de abb garantiza que nuestra cadena tenga esa subcadena, no nos dice nada de cómo será procesada, por ejemplo con la cadena abbabb no queda claro cual de las dos apariciones de abb se alinea con el infijo.

{#er_alineación, height=250px }

Por otro lado un autómata para este lenguaje es:



{#contiene_abb, height=250px }

Se puede apreciar que el autómata sigue una estrategia diferente, describe una secuencia de decisiones para determinar si en la cadena ya ocurrió la subcadena abb; al comenzar si viene una be no hay chances de que ocurra la cadena abb, pero si viene una a potencialmente hemos encontrado la a de la subcadena, sin embargo si llega otra a quiere decir que la a pasada no pertenece a nuestra subcadena ya que el patrón aa no es el que buscamos confirmar; sin embargo la nueva a podría ser el inicio de nuestro patrón abb; si llega una b se confirmaría que llevamos dos símbolos de nuestro patrón, sólo faltará una siguiente b y podríamos estar seguros que ocurrió el patrón que buscábamos y después de eso ya no importa lo que llegue, siempre estaremos en un estado final. Sin embargo, si llegaba una a se rompe el patrón y tenemos que regresar a considerar que el patrón buscado apenas está comenzado con esa nueva a, por eso regresamos al estado q_1 en lugar que el inicial.

4.2. Autómata Finito No Determinístico (AFND)

Una limitante del Autómata Finito su diseño es complicado y algunas veces no intuitivo. El diseñador de autómatas finitos se tiene concentrar en la esencia de lo que representa cada estado

de un autómata, y tiene que identificar aquellos que permitan realizar el proceso de aceptación de una cadena de forma correcta.

Por otro lado como vimos algunas veces diseñar una especificación de un lenguaje puede resultar muy directo. Surge la duda de por qué no tener un mecanismo más cercano a las Expresiones Regulares, para esto vamos a recurrir a aumentar la flexibilidad de los autómatas finitos, el primer paso será permitir al autómata estar en más de un estado, mientras que en el segundo caso permitiremos un caso especial en la transición de un autómata.

4.2.1. Definición

Definición 2. Un Autómata Finito No Determinístico es una tupla $(Q, \Sigma, q_0, A, \delta)$ donde:

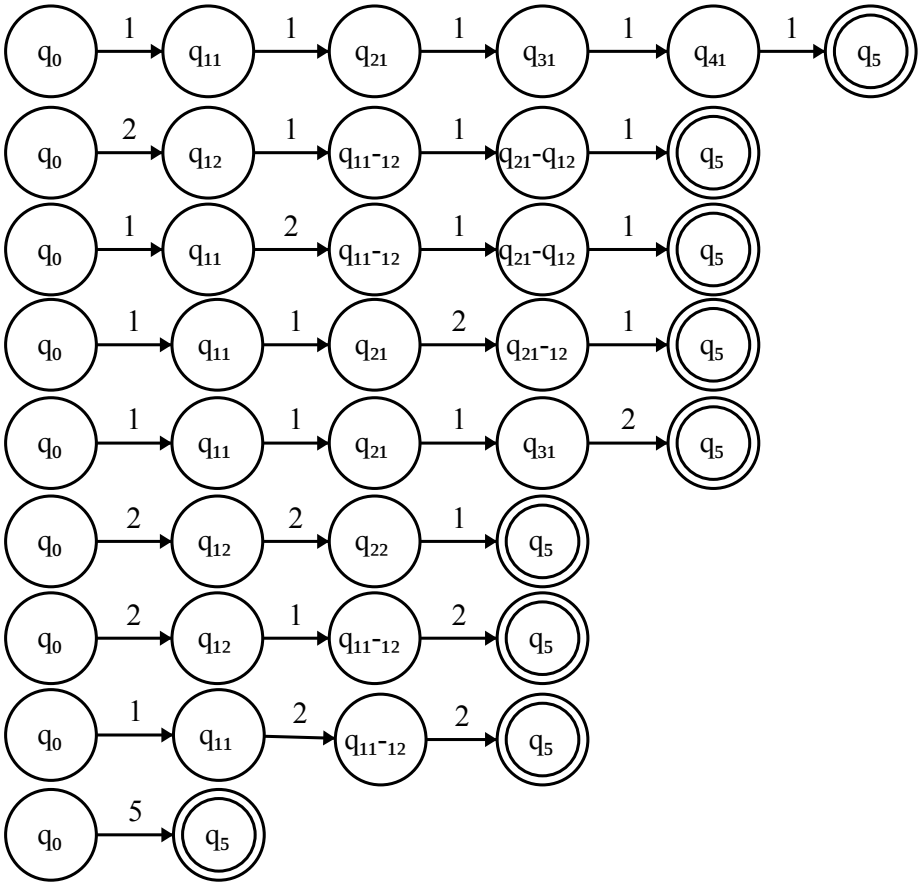
- Q es un conjunto de estados finitos
- Σ es un alfabeto
- q_0 es un estado que denominaremos inicial donde $q_0 \in Q$
- A es un conjunto de estados que denominaremos finales donde $A \subset Q$
- δ es una función de transición que cumple con: $\delta : Q \times \Sigma \to 2^Q$

4.2.1.1. El conjunto potencia

El conjunto 2^Q representa a todas los conjuntos posibles que podamos hacer con los estados (i.e., conjunto potencia). Es decir la función de transición ya no regresa un sólo estado, sino que en la definición de un AFND la función de transición regresa un conjunto de estados, cualquier conjunto de estados que se puedan formar con los estados.

4.2.1.2. Ejemplo: Máquina de chicles

Supongamos que queremos diseñar una máquina dispensadora de chicles, dónde estos cuestan 5 pesos. Supongamos que para esa máquina solo existen monedas con valor de 1, 2 y 5 pesos. Un AFND que acepta todos los posibles pagos que suman 5 y que por lo tanto tendrían que hacer que la máquina dispensadora otorgara un chicle sería:



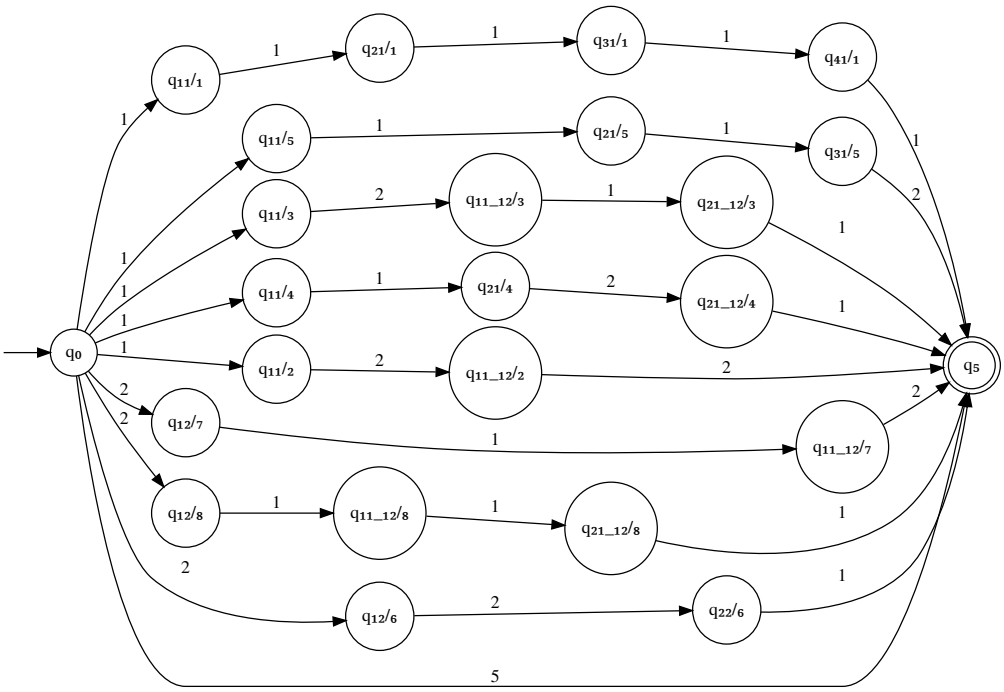
{#chi-

cles_independientes, height=250px }

Sin embargo esta solución no resulta en una sola máquina, sino en ¡¡¡nueve máquinas!!!. Un "cambio" pequeño que permite cambiar esta situación sería proponer la siguiente máquina:

Nombres de estados

Para cada una de estas máquinas el estado codifica la cantidad de monedas que ha visto la máquina hasta cierto momento por ejemplo: q21 señala que se han visto dos monedas de 1 peso, q11-12 señala que se han visto una moneda de 1 peso y una moneda de 2 pesos.



{#full, height=250px }

Como es evidente este AFND no puede confundirse con AF, porque del estado q_0 podemos llegar a varios estados, ya sea con una moneda de un peso o de dos pesos.

Nombres de estados

En esta ocasión los nombres de los estados agregan información sobre qué máquina independiente es la original. Por ejemplo el estado q21/4 señala el estado donde ya han pagado con dos monedas de un peso que originalmente pertenecía a la cuarta máquina independiente.

4.2.2. Función de transición extendida para AFNDs

Al igual que hicimos para los Autómatas Finitos es necesario definir una función de transición extendida que tome como entrada una cadena y determine a que estados finales se llegan. Para lograr esto volvemos a recurrir a una definición recursiva de la siguiente forma:

$$\delta^* = \begin{cases} \delta^*(q, \epsilon) = \{q\} & q \in Q \\ \delta^*(q, wa) = \bigcup_{r \in \delta^*(q,w)} \delta(r, a) & q, r \subseteq Q, w \subseteq \Sigma^*, a \subseteq \Sigma \end{cases}$$

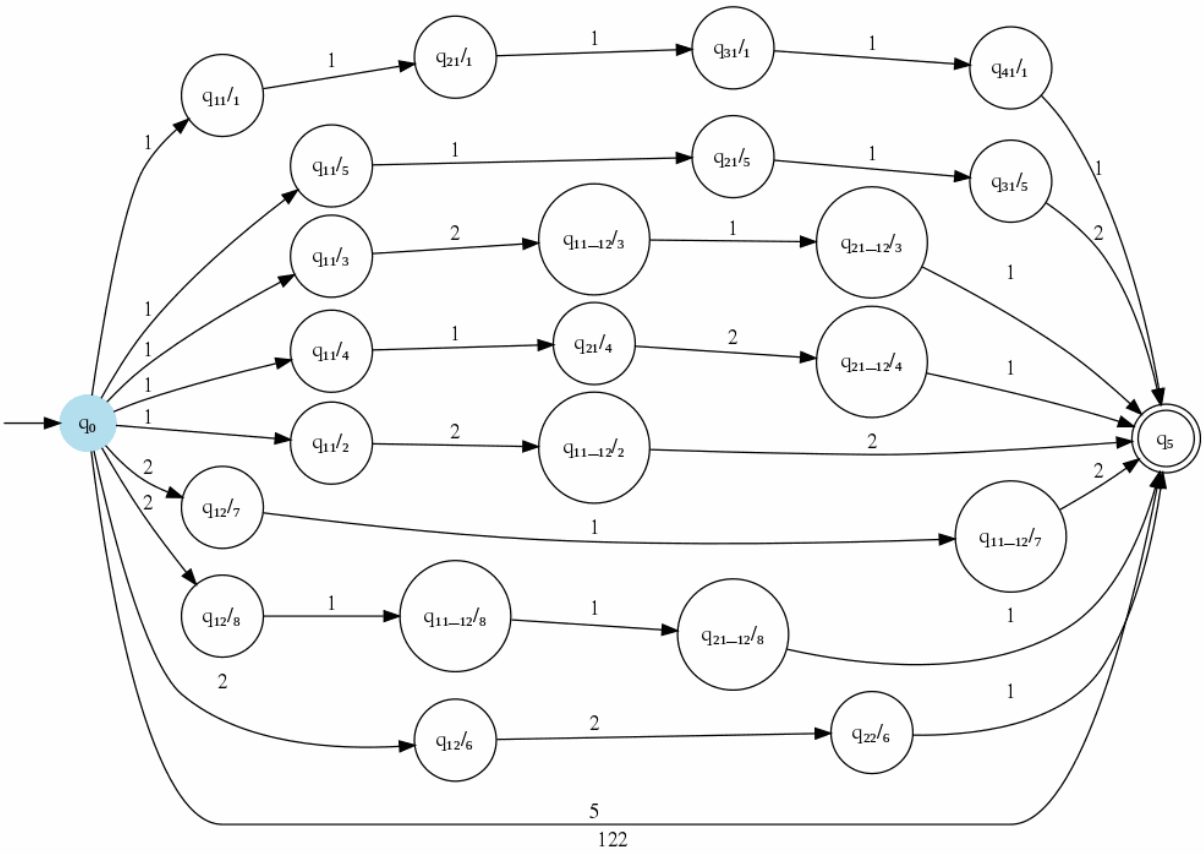
Otra vez tenemos un caso base (con la cadena vacía) y un caso recursivo (con una cadena que podemos dividir en prefijo y sufijo de un sólo símbolo). El caso base simplemente nos regresa como conjunto el estado inicial, mientras que el caso recursivo pospone la evaluación del último símbolo con los estados posibles que se llegan a través del sufijo.

4.2.2.1. Ejemplo función de transición extendida

¿Nuestro autómata de Máquina de chicles acepta el pago 1, 2 y 2?

$$
\begin{aligned}
\delta^*(q_0, 122) &= \\
&= \bigcup_{r_1 \in \delta^*(q_0, 12)} \delta(r_1, 2) \\
&= \bigcup_{r_1 \in \bigcup_{r_2 \in \delta^*(q_0, 1)} \delta(r_2, 2)} \delta(r_1, 2) \\
&= \bigcup_{r_1 \in \bigcup_{r_2 \in \bigcup_{r_3 \in \delta^*(q_0, \epsilon)} \delta(r_3, 1)} \delta(r_2, 2)} \delta(r_1, 2) \\
&= \bigcup_{r_1 \in \bigcup_{r_2 \in \bigcup_{r_3 \in \{q_0\}} \delta(r_3, 1)} \delta(r_2, 2)} \delta(r_1, 2) \\
&= \bigcup_{r_1 \in \bigcup_{r_2 \in \bigcup_{r_3 \in \{q_0\}} \delta(r_3, 1)} \delta(r_2, 2)} \delta(r_1, 2) \\
&= \bigcup_{r_1 \in \bigcup_{r_2 \in \delta(q_0, 1)} \delta(r_2, 2)} \delta(r_1, 2) \\
&= \bigcup_{r_1 \in \bigcup_{r_2 \in \{q_{11/1}, q_{11/2}, q_{11/3}, q_{11/4}, q_{11/5}\}} \delta(r_2, 2)} \delta(r_1, 2) \\
&= \bigcup_{r_1 \in \{\delta(q_{11/1}, 2) \cup \delta(q_{11/2}, 2) \cup \delta(q_{11/3}, 2) \cup \delta(q_{11/4}, 2) \cup \delta(q_{11/5}, 2)\}} \delta(r_1, 2) \\
&= \bigcup_{r_1 \in \{\emptyset \cup \{q_{11-12/2}\} \cup \{q_{11-12/3}\} \cup \emptyset \cup \emptyset\}} \delta(r_1, 2) \\
&= \bigcup_{r_1 \in \{q_{11-12/2}, q_{11-12/3}\}} \delta(r_1, 2) \\
&= \delta(q_{11-12/2}, 2) \bigcup \delta(q_{11-12/3}, 2) \\
&= \{q_5\} \bigcup \emptyset \\
&= \{q_5\}
\end{aligned}
$$

De manera gráfica se ve así:



{#full, height=250px }

4.2.3. Lenguaje aceptado por un AFND

Para el AFND $M = (Q, \Sigma, q_0, A, \delta)$

La cadena $w \in \Sigma^*$se acepta si:

- $\delta^*(q_0, w) \bigcap A \neq \emptyset$

L(M) es el lenguaje conformado por cadenas aceptadas por M

4.3. AFND → AF

Un aspecto sorprendente de los AFND es que son equivalentes a los AF; esto quiere decir que todo AFND podrá ser reducido a un AF.

4.3.1. Reducción

La reducción recae en el hecho que todo AFND tiene un número finito de estados, y podemos imaginarnos que una combinación de estados de un AFND sera equivalente a un estado de un AF. De esta forma cuando en un AFND estamos en una combinación de estados y llega un símbolo, y este nos lleva a otra combinación, en realidad nos estamos moviendo de un estado a otro en el AF a través del mismo símbolo.

Por supuesto, existe un problema exponencial ya que los posibles estados en AF dado un AFND son todas las posibles combinaciones de los estados de este. Tratar de hacer el análisis de dicho número de combinaciones no es práctico. Sin embargo, existe un proceso más directo que no requiere explorar todo este espacio y es a través de codificar los estados del AF con un código posicional de los estados del AFND. Un 1 en el código del AF representa que el estado en el AFND está activo, si hay más unos en el código todos estos estados se encuentran activos, mientas que un 0 representa que el estado correspondiente en el AFND no está activo.

4.3.1.1. El procedimiento

Suponiendo un AFND $(Q, \Sigma, q_0, A, \delta)$ el siguiente procedimiento genera la tabla de transición de un AF $(Q', \Sigma, q_0', A', \delta')$ equivalente:

1. Definir el código para los estados nuestro AF, cada estado $q \in Q$ define una posición de esa codificación.
2. Agregamos en nuestra tabla una fila para donde estado origen es q_0 cuyo código es la posición del estado inicial activada con un 1 mientras que el restos de las posiciones son cero.
3. Por cada símbolo $a \in \Sigma$ construir la codificación en la columna correspondiente al símbolo a dónde se activan los estados a los que se llega con dicho símbolo partiendo del estado origen correspondiente a la fila.
4. Por cada codificación nueva que aparezca en las columnas agregar una fila.
5. Si existe un código que no haya sido analizada regresar a 3.

Al final podemos renombrar las codificaciones de los estados con nombres más compactos.

Finalmente, Q será conformado por todos los estados identificados, q_0' será el estado correspondiente a la codificación creada en el paso 2; A estará conformada por todos los estados cuya codificación tiene activa una posición correspondiente a un estado final del original AFND; y finalmente la tabla de transición representa q_0'.

4.3.1.2. Ejemplo

Supongamos el siguiente orden para nuestros estados del AFND:

$q_0, q_{11/1}, q_{11/5}, q_{11/3}, q_{11/4}, q_{11/2}, q_{12/7}, q_{12/8}, q_{12/6}, q_5, q_{21/1}, q_{31/1}, q_{41/1}, q_{11_12/2}, q_{11_12/3}, q_{21_12/3}, q_{21/4}, q_{21_12/4}, q$

Con este orden podemos codificar el estado inicial como:

100000000000000000000000

Con este estado inicial podemos averiguar a dónde llegamos si recibimos cada uno de los símbolos del alfabeto:

Estado	1	2	5
100000000000000000000000	011111000000000000000000	000000111000000000000000	000000000100000000000000

En este caso las tres codificaciones resultantes son nuevas, por lo que hay que agregarlas y calcular a dónde podemos llegar si llega uno de los símbolos del alfabeto:

Estado	1	2	5
100000000000000000000000	011111000000000000000000	000000111000000000000000	000000000100000000000000
011111000000000000000000			
000000111000000000000000			
000000000100000000000000			

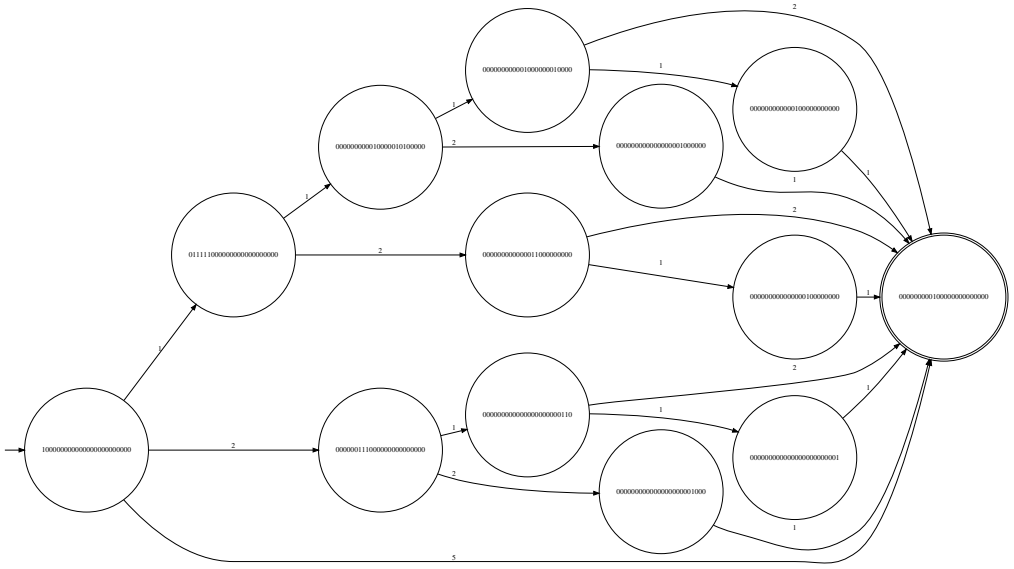
Si hacemos el calculo para estas nuevas filas obtenemos:

estado	1	2	5
100000000000000000000000	011111000000000000000000	000000111000000000000000	000000000100000000000000
011111000000000000000000	000000000010000010100000	000000000000011000000000	000000000000000000000000
000000111000000000000000	000000000000000000000110	000000000000000000001000	000000000000000000000000
000000000100000000000000	000000000000000000000000	000000000000000000000000	000000000000000000000000

Es importante notar que la codificación con sólo ceros, representa que con ese símbolos y el estado en el que se encuentra no se puede llegar ningún estado, es un estado de error.

La tabla final resultante es:

estado	1	2	5
10000000000000000000000	01111100000000000000000	00000011100000000000000	00000000010000000000000
01111100000000000000000	00000000001000001010000	00000000000011000000000	00000000000000000000000
00000011100000000000000	00000000000000000000110	00000000000000000001000	00000000000000000000000
00000000010000000000000	00000000000000000000000	00000000000000000000000	00000000000000000000000
00000000001000001010000	00000000000100000010000	00000000000000001000000	00000000000000000000000
00000000000100000010000	00000000000010000000000	00000000010000000000000	00000000000000000000000
00000000000010000000000	00000000010000000000000	00000000000000000000000	00000000000000000000000
00000000000011000000000	00000000000000100000000	00000000010000000000000	00000000000000000000000
00000000000000100000000	00000000010000000000000	00000000000000000000000	00000000000000000000000
00000000000000001000000	00000000010000000000000	00000000000000000000000	00000000000000000000000
00000000000000000001000	00000000010000000000000	00000000000000000000000	00000000000000000000000
00000000000000000000110	00000000000000000000001	00000000010000000000000	00000000000000000000000
00000000000000000000001	00000000010000000000000	00000000000000000000000	00000000000000000000000



{#dfa_full, height=250px }

Por supuesto la tabla es incomprensible, pero podemos renombrar los estados de la siguiente forma:

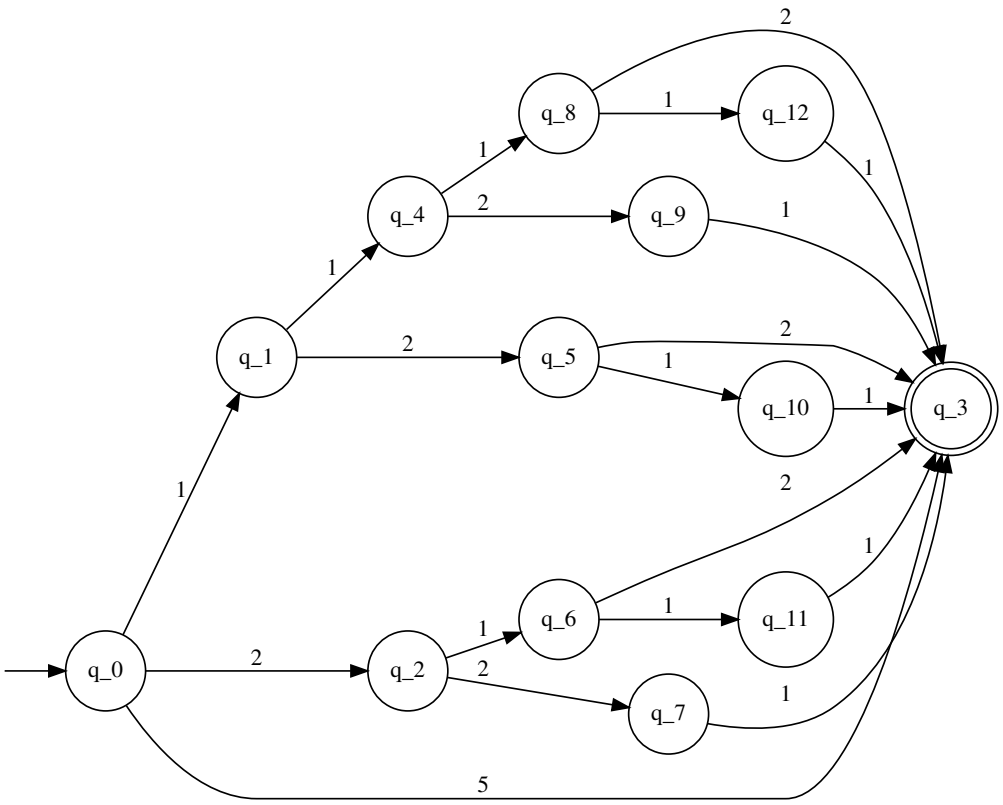
estado	nuevo nombre
10000000000000000000000	q_0
01111100000000000000000	q_1
00000011100000000000000	q_2

estado	nuevo nombre
000000000100000000000000	q_3
000000000010000010100000	q_4
000000000001000000010000	q_5
000000000000100000000000	q_6
000000000000011000000000	q_7
000000000000000100000000	q_8
000000000000000001000000	q_9
000000000000000000001000	q_10
000000000000000000000110	q_11
000000000000000000000001	q_12

Resultando:

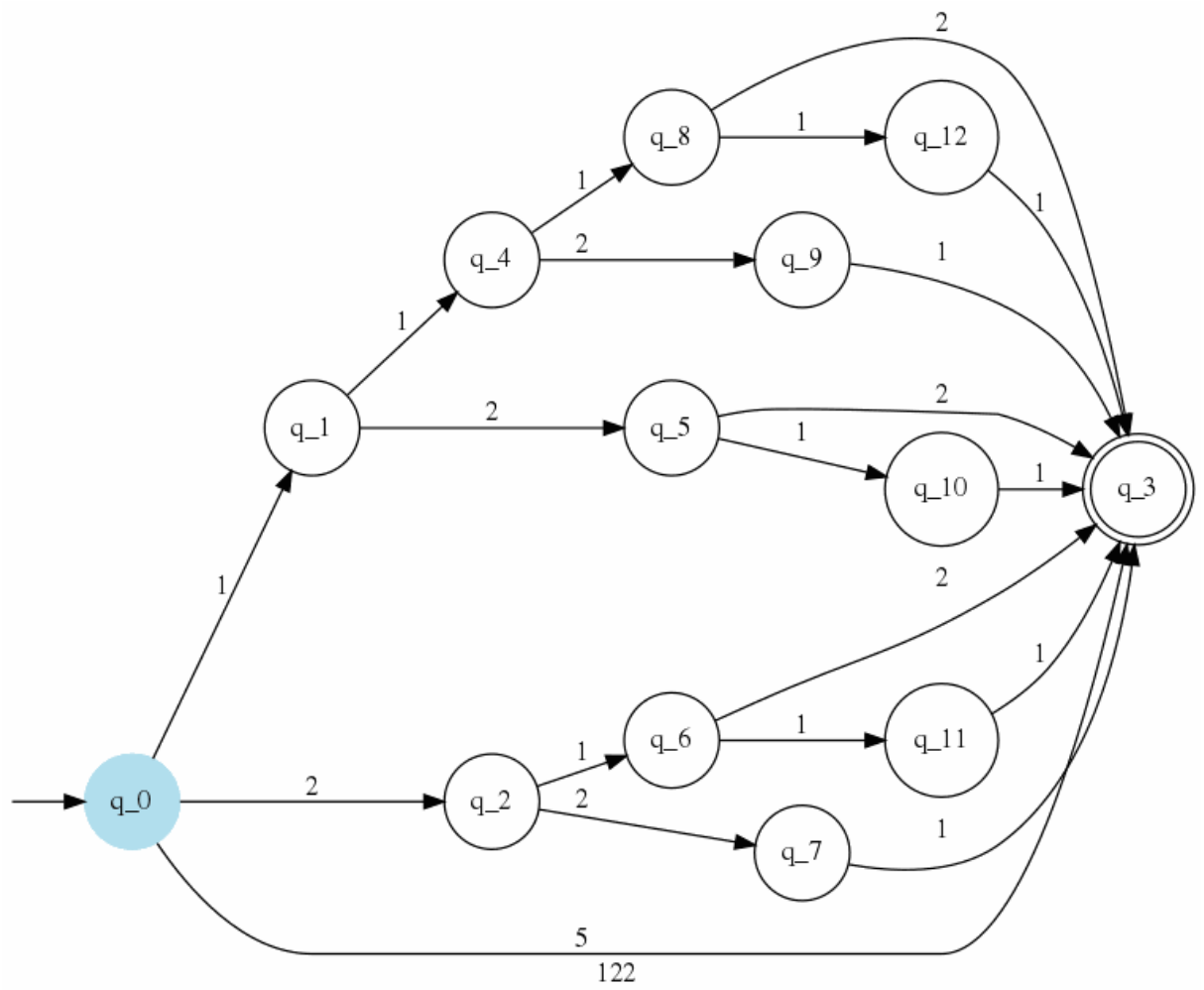
estado	1	2	5
$\longrightarrow$ q_0	q_1	q_2	q_3
q_1	q_4	q_5	
q_2	q_6	q_7	
q_3			
q_4	q_8	q_9	
q_5	q_10	q_3	
q_6	q_11	q_3	
q_7	q_3		
q_8	q_12	q_3	
q_9	q_3		
q_10	q_3		
q_11	q_3		
q_12	q_3		

Finalmente, si lo graficamos queda de la siguiente forma:



{#dfa_chi-

cles, height=250px }



{#dfa, height=250px }

4.4. Autómata Finito No Determinístico con transición épsilon (AFND-)

La flexibilidad lograda por el AFND fue muy buena, sin embargo al diseñar un AFND tenemos que seguir pensando el procesamiento de la cadenas. Sería deseable todavía mayor flexibilidad, esto lo lograremos al permitir una transición a través de la cadena vacía, es decir podremos transiciones entre estados aún cuando no hayamos visto un símbolo en la cadena.

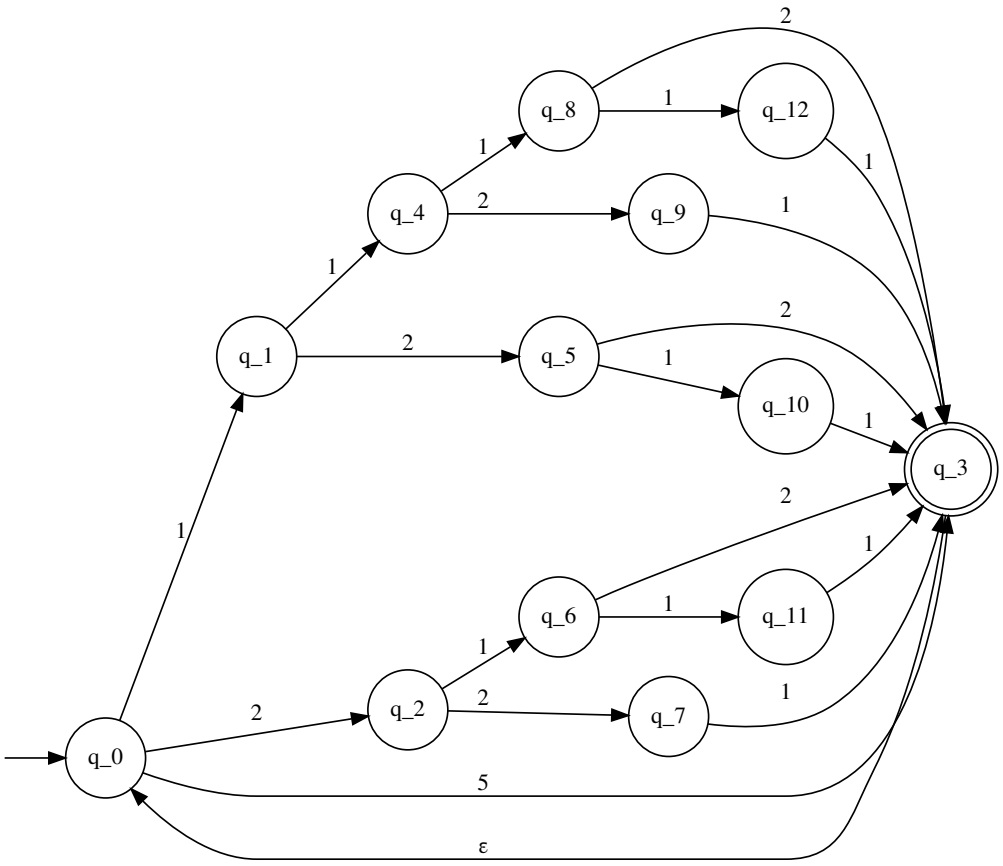
4.4.1. Definición

Definición 3. Un Autómata Finito No Determinístico con transición épsilon es una tupla $(Q, \Sigma, q_0, A, \delta)$ donde:

- Q es un conjunto de estados finitos
- Σ es un alfabeto
- q_0 es un estado que denominaremos inicial donde $q_0 \in Q$
- A es un conjunto de estados que denominaremos finales donde $A \subset Q$
- δ es una función de transición que cumple con: $\delta : Q \times (\Sigma \cup \{\epsilon\}) \to 2^Q$

4.4.1.1. Ejemplo: Máquina de chicles con múltiples pagos

Qué pasa ahora si deseamos que nuestra máquina de chicles ya no sólo acepte un sólo pago de un chicle sino de múltiples pagos ¿Tendríamos que rediseñar nuestra máquina? Una solución es utilizar la extensión de la cadena vacía como transición, como se ilustra a continuación:



{#nd-fa_e_multiple, height=250px }

¿Cuál es la diferencia con el AF de la sección anterior?

Respuesta

Existe una nueva transición de q_3 a q_0 a través de la cadena vacía, ε. Esto permite que una vez realizado un pago por un chicle se pueda proceder a otro.

4.4.2. Función de transición extendida para AFND-

Al igual que hicimos para los Autómatas Finitos y los Autómatas Finitios No Determinísticos es necesario definir una función de transición extendida que tome como entrada una cadena y determine a que estados finales se llegan. Para lograr esto volvemos a recurrir a una definición recursiva de la siguiente forma:

$$\delta^* = \begin{cases} \delta^*(q, \epsilon) = exp_\epsilon(\{q\}) & q \in Q \\ \delta^*(q, wa) = exp_\epsilon(\bigcup_{r \in \delta^*(q,w)} \delta(r, a)) & q, r \subseteq Q, w \subseteq \Sigma^*, a \subseteq \Sigma \end{cases}$$

Otra vez tenemos un caso base (con la cadena vacía) y un caso recursivo (con una cadena que podemos dividir en prefijo y sufijo de un sólo símbolo). sin embargo ahora tenemos que definir una nueva función auxiliar exp (expansión por épsilon)

4.4.3. Calculo de expansión por épsilon

El objetivo de la función de expansión es recolectar los estados extras a los que se pueda llegar a través de la cadena vacía (ε).

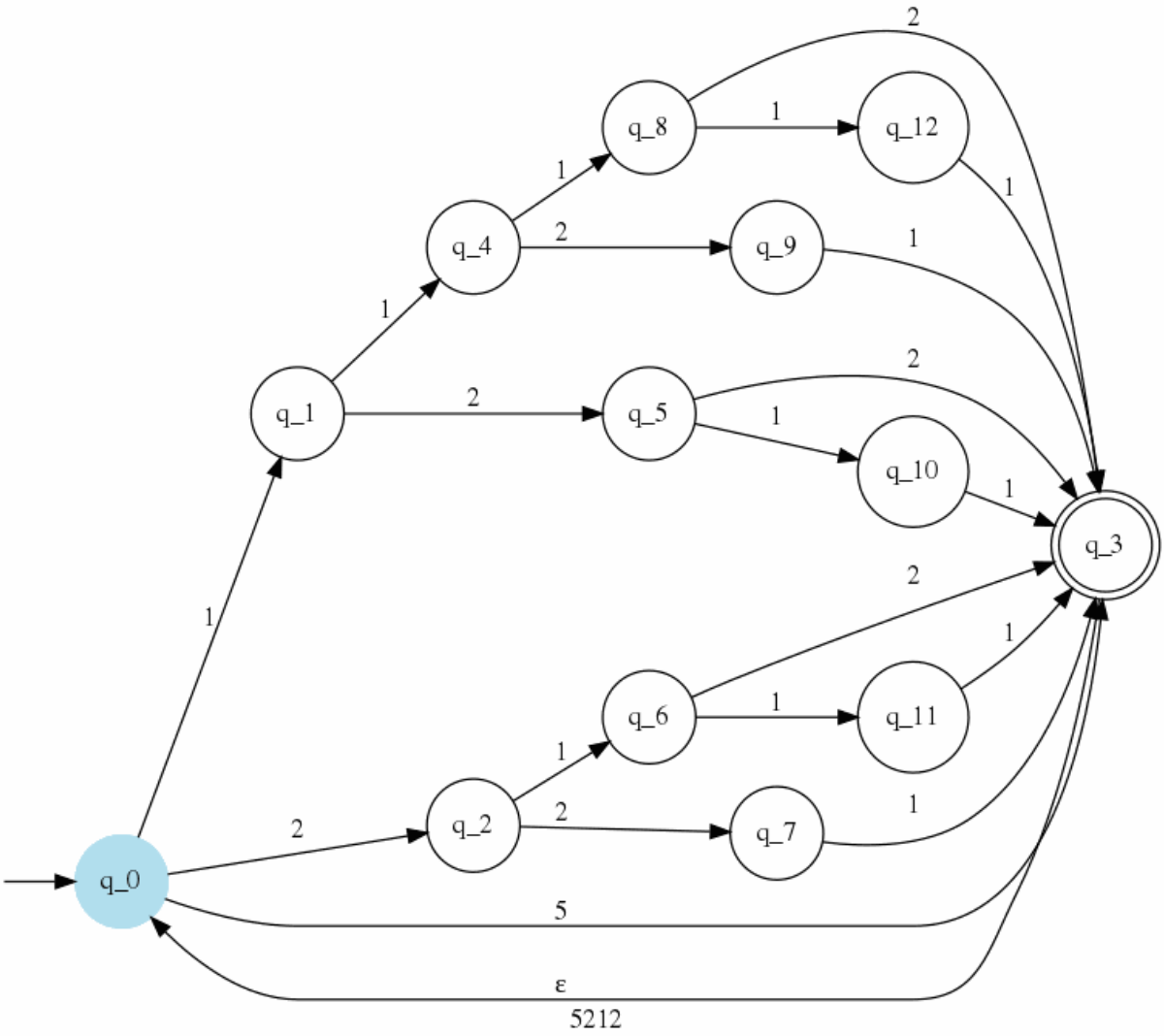
1. Para cada $q \in S$ incluir in S todo $\delta(q, \epsilon)$
2. Repetir hasta que S no cambie

4.4.3.1. Ejemplo función de transición extendida

¿Nuestro autómata de Máquina de chicles acepta el pago 5, 2, 1 y 2?

$$
\begin{aligned}
\delta^*(q_0, 5212) &= exp_\epsilon\Big(\bigcup_{r_1 \in \delta^*(q_0,521)} \delta(r_1, 2)\Big) \\
&= exp_\epsilon\Big(\bigcup_{r_1 \in exp_\epsilon\left(\bigcup_{r_2 \in \delta^*(q_0,52)} \delta(r_2,1)\right)} \delta(r_1, 2)\Big) \\
&= \exp_\epsilon\Big(\bigcup_{r_1 \in exp_\epsilon\left(\bigcup_{r_2 \in exp_\epsilon\left(\bigcup_{r_3 \in \delta^*(q_0,5)} \delta(r_3,2)\right)} \delta(r_2,1)\right)} \delta(r_1, 2)\Big) \\
&= \exp_\epsilon\Big(\bigcup_{r_1 \in \exp_\epsilon\left(\bigcup_{r_2 \in exp_\epsilon\left(\bigcup_{r_3 \in exp_\epsilon\left(\bigcup_{r_4 \in \delta^*(q_0,\epsilon)} \delta(r_4,5)\right)} \delta(r_3,2)\right)} \delta(r_2,1)\right)} \delta(r_1, 2)\Big) \\
&= \exp_\epsilon\Big(\bigcup_{r_1 \in \exp_\epsilon\left(\bigcup_{r_2 \in exp_\epsilon\left(\bigcup_{r_3 \in exp_\epsilon\left(\bigcup_{r_4 \in exp_\epsilon(\{q_0\})} \delta(r_4,5)\right)} \delta(r_3,2)\right)} \delta(r_2,1)\right)} \delta(r_1, 2)\Big) \\
&= \exp_\epsilon\Big(\bigcup_{r_1 \in \exp_\epsilon\left(\bigcup_{r_2 \in exp_\epsilon\left(\bigcup_{r_3 \in exp_\epsilon\left(\bigcup_{r_4 \in \{q_0\}} \delta(r_4,5)\right)} \delta(r_3,2)\right)} \delta(r_2,1)\right)} \delta(r_1, 2)\Big) \\
&= \exp_\epsilon\Big(\bigcup_{r_1 \in \exp_\epsilon\left(\bigcup_{r_2 \in exp_\epsilon\left(\bigcup_{r_3 \in exp_\epsilon(\delta(q_0,5))} \delta(r_3,2)\right)} \delta(r_2,1)\right)} \delta(r_1, 2)\Big) \\
&= \exp_\epsilon\Big(\bigcup_{r_1 \in \exp_\epsilon\left(\bigcup_{r_2 \in exp_\epsilon\left(\bigcup_{r_3 \in exp_\epsilon(\{q_3\})} \delta(r_3,2)\right)} \delta(r_2,1)\right)} \delta(r_1, 2)\Big) \\
&= \exp_\epsilon\Big(\bigcup_{r_1 \in \exp_\epsilon\left(\bigcup_{r_2 \in exp_\epsilon\left(\bigcup_{r_3 \in \{q_0,q_3\}} \delta(r_3,2)\right)} \delta(r_2,1)\right)} \delta(r_1, 2)\Big) \\
&= \exp_\epsilon\Big(\bigcup_{r_1 \in \exp_\epsilon\left(\bigcup_{r_2 \in exp_\epsilon(\delta(q_0,2)\cup\delta(q_3,2))} \delta(r_2,1)\right)} \delta(r_1, 2)\Big) \\
&= \exp_\epsilon\Big(\bigcup_{r_1 \in \exp_\epsilon\left(\bigcup_{r_2 \in exp_\epsilon(\{q_2\}))} \delta(r_2,1)\right)} \delta(r_1, 2)\Big) \\
&= \exp_\epsilon\Big(\bigcup_{r_1 \in \exp_\epsilon\left(\bigcup_{r_2 \in \{q_2\}} \delta(r_2,1)\right)} \delta(r_1, 2)\Big) \\
&= \exp_\epsilon\Big(\bigcup_{r_1 \in \exp_\epsilon(\delta(q_2,1))} \delta(r_1, 2)\Big) \\
&= \exp_\epsilon\Big(\bigcup_{r_1 \in \exp_\epsilon(\{q_6\})} \delta(r_1, 2)\Big) \\
&= \exp_\epsilon\Big(\bigcup_{r_1 \in \{q_6\}} \delta(r_1, 2)\Big) \\
&= \exp_\epsilon(\{q_3\}) \\
&= \{q_0, q_3\}
\end{aligned}
$$

De forma gráfica luce de la siguiente forma:



{#nd-fa_e_multiple, height=250px }

4.4.4. Lenguaje aceptado por un AFND-

Para el AFND-ε $M = (Q, \Sigma, q_0, A, \delta)$

La cadena $w \in \Sigma^*$se acepta si:

- $\delta^*(q_0, w) \bigcap A \neq \emptyset$

L(M) es el lenguaje conformado por cadenas aceptadas por M

4.5. AFND- $\rightarrow$ AFND

Un aspecto interesante de los AFND-ε es que son equivalentes a los AFND; esto quiere decir que todo AFND-ε podrá ser reducido a un AFND, y por la equivalencia de AFND entre AF, todo AFND-ε podrá ser reducido a un AF

4.5.1. Reducción

En esta reducción, lo único que tenemos que hacer es quitar la transición épsilon. En este momento nuestra función de transición luce así:

estado	1	2	5	ε
$\longrightarrow$q_0	{q_1}	{q_2}	{q_3}	∅
q_1	{q_4}	{q_5}	∅	∅
q_2	{q_6}	{q_7}	∅	∅
q_3	∅	∅	∅	{q_0}
q_4	{q_8}	{q_9}	∅	∅
q_5	{q_10}	{q_3}	∅	∅
q_6	{q_11}	{q_3}	∅	∅
q_7	{q_3}	∅	∅	∅
q_8	{q_12}	{q_3}	∅	∅
q_9	{q_3}	∅	∅	∅
q_10	{q_3}	∅	∅	∅
q_11	{q_3}	∅	∅	∅
q_12	{q_3}	∅	∅	∅

Para lograr esto solo tenemos que evaluar la función de transición extendida por cada uno de los símbolos y por cada uno de los estados en nuestro AFND-ε, de la siguiente forma:

estado	δ*(q,“1”)	δ*(q,“2”)	δ*(q,“5”)
$\longrightarrow$q_0	δ*(q_0,“1”)	δ*(q_0,“2”)	δ*(q_0,“3”)

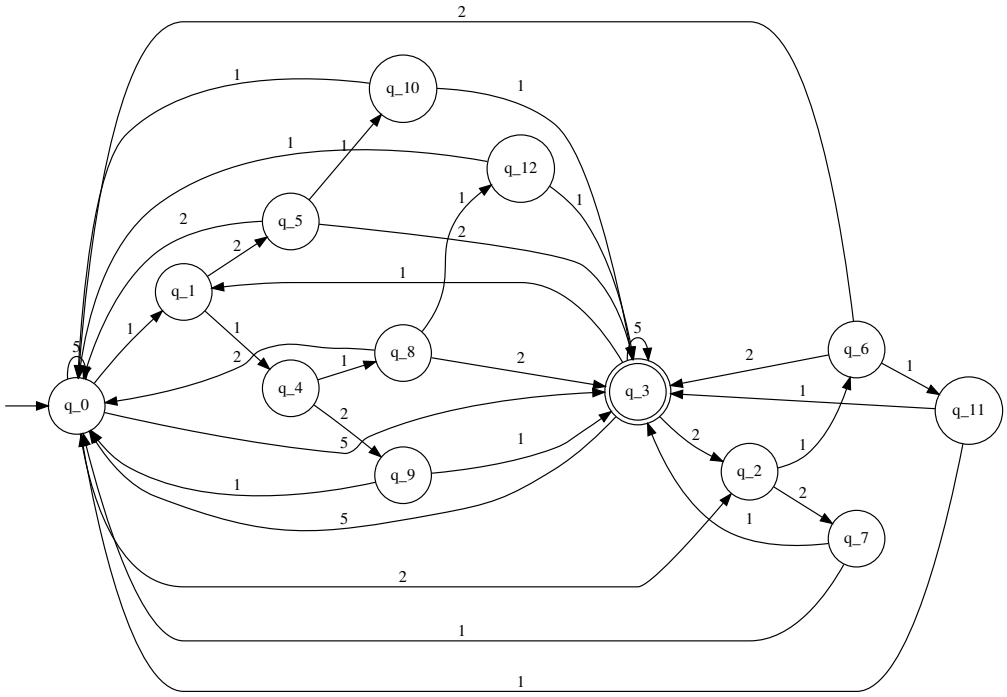
Es decir suponemos estar en el estado de la fila, y tratamos al símbolo como si fuera una cadena y registramos a dónde podemos llegar:

estado	δ*(q,“1”)	δ*(q,“2”)	δ*(q,“5”)
⟶q_0	{q_1}	{q_2}	{q_3,q_0}

Esto se hace para todas las filas resultando en el AFND

estado	1	2	5
⟶q_0	{q_1}	{q_2}	{q_0,q_3}
q_1	{q_4}	{q_5}	∅
q_2	{q_6}	{q_7}	∅
q_3	∅	∅	∅
q_4	{q_8}	{q_9}	∅
q_5	{q_10}	{q_3,q_0}	∅
q_6	{q_11}	{q_3,q_0}	∅
q_7	{q_3}	∅	∅
q_8	{q_12}	{q_3,q_0}	∅
q_9	{q_3,q_0}	∅	∅
q_10	{q_3,q_0}	∅	∅
q_11	{q_3,q_0}	∅	∅
q_12	{q_3,q_0}	∅	∅

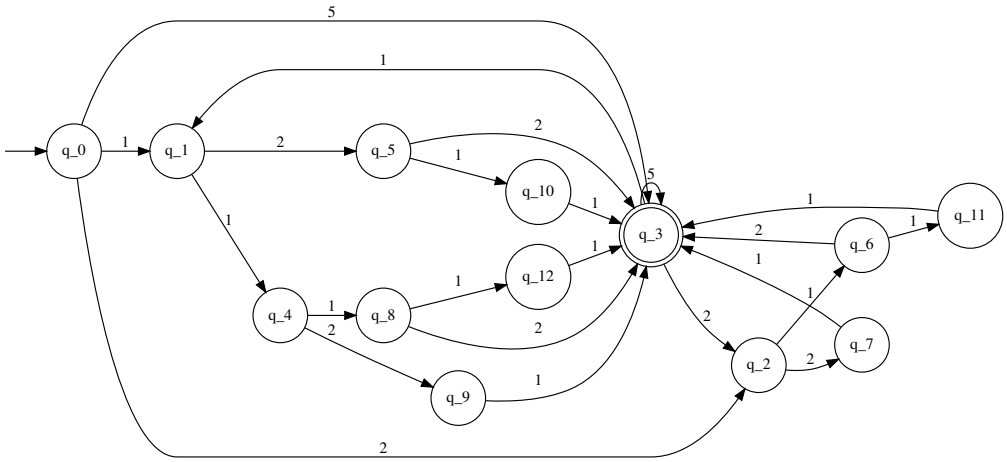
Que de forma gráfica se puede visualizar así:



{#nd-fa_multiple, height=250px }

4.6. Extra

Por supuesto si tenemos un AFND, podemos reducirlo a un AF siguiendo el procedimiento en AFND → AF. El resultado es:

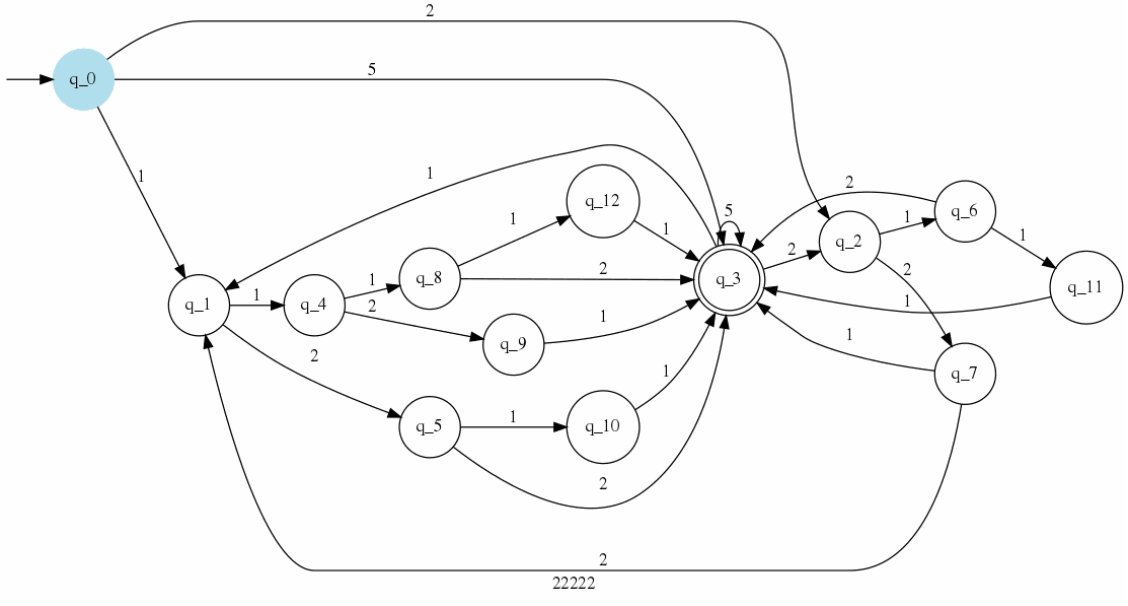


{#dfa_multiple, height=250px }

Existe un error con nuestra máquina, no se puede pagar con 5 monedas de dos pesos ¿Cómo se podría arreglar?

Respuesta

Agregando una transición de q_7, donde se han pagado con dos monedas de 2 pesos, a q_1 a través de la tercera moneda de 2 pesos, de esta forma en la segunda ronda lleva un equivalente a 6 pesos. La siguiente animación muestra el pago con cinco monedas de 2 pesos.



{#dfa_multiple_22222, height=250px }

4.7. AF → AFND-

Un aspecto interesante de los AF es que son equivalentes a los AFND-ε; esto quiere decir que todo AF podrá ser reducido a un AFND-ε, y por la equivalencia de AFND con AF y AFND-ε con AFND, los tres tipos de máquinas finitas AFND-ε, AFNS y AF son equivalentes

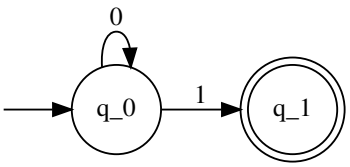
4.7.1. Reducción

Esta reducción es trivial, sólo hay que agregar una columna a nuestra tabla de transición para la transición ε donde toda la columna va al conjunto vacía.

estado	1	2	5	ε
q	{_}	{_}	{_}	∅

4.7.1.1. Ejemplo

Supongamos el siguiente autómata finito,



{#unos_cero, height=250px }

Con la siguiente tabla de transición

estado	0	1
q_0	q_0	q_1
q_1		

Su reducción como AFND-ε es:

estado	0	1	ε
q_0	{q_0}	{q_1}	∅
q_1	∅	∅	∅

4.8. ER → AFND-

4.8.1. Aplicando operaciones de Lenguajes regulares a AFND-

Si tengo dos AFND-ε:

- $(Q^A, \Sigma, q_0^A, A^A, \delta^A)$ asociado al lenguaje L^A
- $(Q^B, \Sigma, q_0^B, A^B, \delta^B)$ asociado al lenguaje L^B

Entonces es posible construir los siguientes AFND-ε basados en la operaciones de los Lenguajes Regulares

- $L^A + L^B$

$$\left(Q^A \bigcup Q^B \bigcup \{q_0^{union}\}), \Sigma, q_0^{union}, A^A \bigcup A^B, \delta^A \bigcup \delta^B \bigcup \{(q_0^{union}, \varepsilon) \rightarrow \{q_0^A, q_0^B\}\}\right)$$

- $L^A L^B$

$$\Big(Q^A \bigcup Q^B), \Sigma, q_0^A, A^A \bigcup A^B, \delta^A \bigcup \delta^B \bigcup \{(q, \varepsilon) \rightarrow \{q_0^B\} | q \in A^A\}\Big)$$

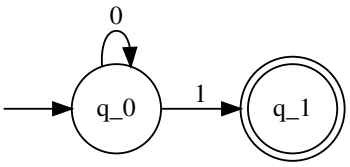
- L^*

$$\Big(Q, \Sigma, q_0^{cerradura}, A \bigcup \{q_f^{cerradura}\}, \delta \bigcup \{(q_0^{cerradura}, \varepsilon) \rightarrow \{q_0\}, (q_0^{cerradura}, \varepsilon) \rightarrow \{q_f^{cerradura}\}, (q_f^{cerrad}$$

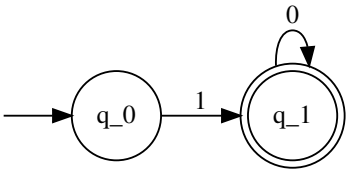
4.8.1.1. Ejemplo

Supongamos los lenguajes $L_1 : 0^*1$ y $L_2 : 10^*$

Con sus autómatas gráficos



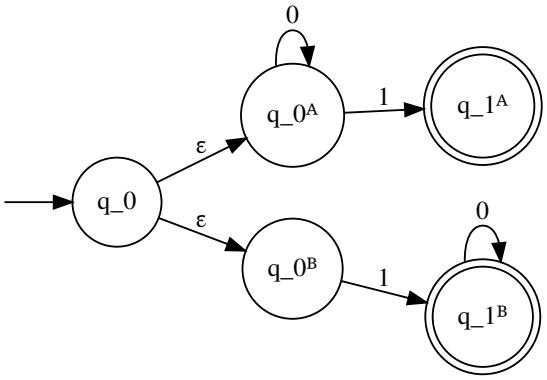
{#unos_cero, height=250px }



{#uno_ceros, height=250px }

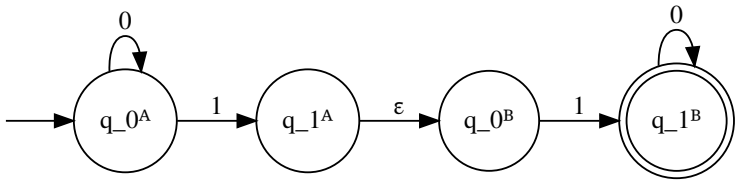
Si aplicamos cada una de las operaciones obtenemos:

- Unión $L_1 \bigcup L_2$



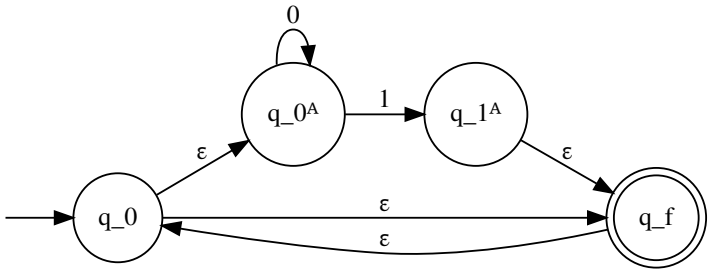
{#union, height=250px }

- Concatenación $L_1 L_2$



{#concat, height=250px
}

- Cerradura L_1^*

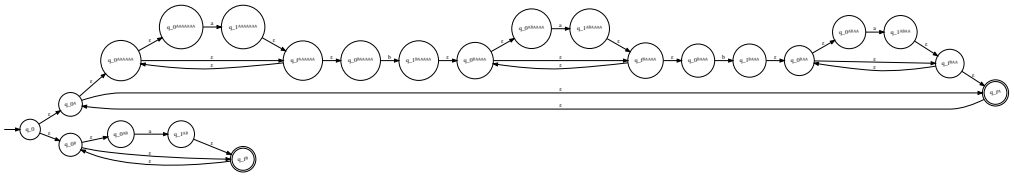


{#cerradura, height=250px
}

4.8.2. Ejemplo de AFND- a ER

Dado a que tenemos una forma de hacer las operaciones de lenguajes regulares con AFND-ε, podemos pensar en transformar un ER a un AFND, partiendo de AF básicos e ir aplicando las operaciones

- $(a^*ba^*ba^*) + a^*$



{#reg,
height=250px }

5 Abro paréntesis, abro paréntesis, cierro paréntesis

5.1. Reflexión infinito en LR

Como su nombre lo establece un autómata finito es finito, sin embargo dada la forma de cómo lo definimos estos pueden lidiar con lenguajes infinitos ¿Cuál es el mecanismo que hace esto posible?

Respuesta

Es la operación cerradura, ni concatenación ni unión general un lenguaje infinito de un finito; pero la cerradura sí.

Es el hecho de que los AF permiten ciclos, es decir que una transición regrese a un estado ya visitado abre la oportunidad de aceptar un número infinito de cadenas.

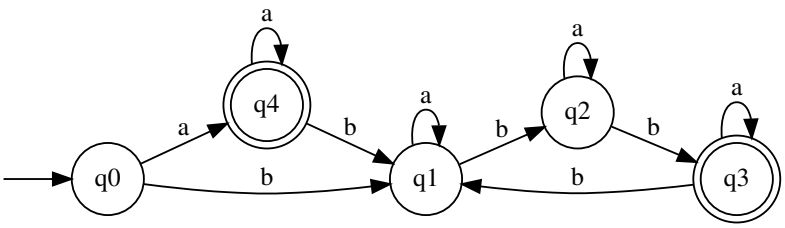
Una pregunta que surge es: ¿cuál es la relación entre el número de estados (n) de un AF en particular y el tamaño de una cadena que es aceptada por el autómata? Un análisis de esta situación, nos lleva a detectar dos casos:

- Si $|w| < n$ quiere decir que estamos seguros de que no se han usado todos los estados.
- Si $|w| >= n$ quiere decir que estamos seguros que se ha repetido uno de los estados.

En el primer caso no podemos asegurar que no se ha repetido un estado, ya que la cadena aunque menor que el número de estados puede que atraviese un ciclo; pero en el segundo caso estamos seguros que al menos un estado ha sido visitado dos veces.

5.1.1. Ejemplo con AF

El siguiente autómata tiene cinco estados acepta cadenas con número de bes divisibles entre tres o puras aes aes, en particular no acepta la cadena vacía.



{#tres_bes_o_puras_aes, height=250px }

Con esto es mente para cadenas aceptadas podemos llenar la siguiente tabla

Cadena	Longitud	Se repitió estado	# de ciclos	Posición de primer ciclo	Longitud primer ciclo
a	1	No	0		
aa	2	Sí	1	2, q4	1
bbb	3	No	0		
bbba	4	Sí	1	4, q3	1
abbb	4	No	0		
aaaaa	5	Sí	4	2, q4	1
bbbbbb	6	Sí	1	4, q1	3
abbbbbb	7	Sí	1	5, q1	3
ababbbbb	8	Sí	1	3, q1	1
abbabbbb	8	Sí	1	4, q2	1
abbbabbb	8	Sí	1	5, q3	1
bbbbbbbbb	9	Sí	2	4, q1	3
abaabbbbb	8	Sí	2	3, q1	1
abbaabbbb	9	Sí	2	4, q2	1
abbbaabbb	9	Sí	2	5, q3	1

Como podemos ver una vez que alcanza la longitud mayor o igual que el número de estados,

podemos estar seguros que al menos un estado se está repitiendo.

Por supuesto, suponiendo cualquier AF hay varias preguntas que surgen:

- ¿Si existe el ciclo, dónde está ese ciclo?

  Respuesta

  Si existe, puede ocurrir antes de alcanzar la longitud n; no sólo eso, si la cadena es mayor que el número de estados podemos estar seguro que el primer ciclo ocurre antes o al alcanzar la longitud n.

- ¿Si existe el ciclo, cuántos ciclos hay?

  Respuesta

  No sabemos con certeza, depende de la estructura del lenguaje, pero si estamos seguros que existe al menos uno.

- ¿Si existe el ciclo, qué longitud tiene?

  Respuesta

  No sabemos con certeza, depende de la estructura del lenguaje.

5.1.2. Dividiendo la cadena en tres

Supongamos que una cadena w pertenece a un lenguaje regular, supongamos que el AF asociado a este lenguaje tiene n estados y que $|w| \geq n$; es decir estamos seguros que al menos un estado se repite. Esto implica que también estamos seguros que el primer ciclo ocurre antes de alcanzar la longitud n. Considerando esto podemos dividir nuestra cadena w en tres partes xyz; a la primera parte la llamamos prefijo (x), la segunda infijo (y) y la tercera sufijo (z); en particular vamos buscar $|xy| \leq n$ mientras $|xyz| = |w|$. Además vamos a considerar que la parte y corresponde al ciclo, no sabemos dónde está pero dado que xyz es mayor que el número de estados, sabemos que un ciclo está antes o cuando la cadena alcanza la longitud igual al número de estados.

5.1.3. ¿El número de estados?

El objetivo de que podamos dividir una cadena de un lenguaje regular en tres partes xyz, nos permite explorar un lenguaje que supongamos regular, sin necesidad de tener que definirlo de las cinco formas que conocemos:

1. Identificar una secuencia de operaciones de lenguajes regulares
2. Diseñar una expresión regular
3. Diseñar un Autómata Finito
4. Diseñar un Autómata Finito No Determinístico
5. Diseñar un Autómata Finito No Determinístico con transición épsilon

Por ejemplo para un lenguaje L que supongo regular e infinito, podré asegurar lo siguiente:

1. Una cadena w de ese lenguaje L lo suficientemente grande, es decir con $|w| \geq n$, la podré dividir en tres partes.
2. Donde las dos primeras partes podrían tener estas longitudes $|xy| \leq n$
3. Entonces existirá un ciclo en esas dos primeras partes

Es importante notar que no importa el valor de n (es decir el número de estados asociados al AF que define el lenguaje), si este es muy elevado, siempre será posible encontrar una cadena de longitud n o mayor. La labor de identificar x, y y z es nuestra y dependerá de las propiedades del lenguaje, no sabemos de antemano como son; sólo que existen. Sin embargo la existencia nos brinda una forma de hablar de los lenguajes regulares, sin hacer el esfuerzo de un diseño.

5.2. Lema de bombeo para lenguajes regulares

Una consecuencia de qué podamos particionar en xyz a una cadena w perteneciente a un Lenguaje Regular con una longitud mayor que el número de estados del AF que la acepta, es que ese ciclo asociado al infijo, lo podríamos repetir múltiples veces y la cadena resultante también debería pertenecer al mismo lenguaje. Por ejemplo:

Considerando el ejemplo de la sección anterior; si w es abbbbbb, donde x es a, y es bbb y z es bbb; entonces si repetimos varias veces y obtendríamos las nuevas cadenas:

- Con una repetición, a bbbbbb bbb pertenece al mismo lenguaje (espaciado usado para señalar donde comienza la parte que se repite)
- Con dos repeticiones, a bbbbbbbbb bbb pertenece al mismo lenguaje
- etc

A esta consecuencia de la existencia de la partición se le conoce como el lema de bombeo.

5.2.1. Lema de bombeo

Sea L un Lenguaje Regular y su AF con n estados que lo acepta, entonces para todas la cadenas w donde $|w| \geq n$ se podrán descomponer en tres partes xyz tales que:

1. $|xy| \leq n$
2. $y \neq \varepsilon$, puesto de otra forma $y > 1$
3. Para todo $k \geq 0$, las cadenas con la forma $xy^k z$ también pertenecen a L

5.2.2. Aplicación del lema de bombeo

Algo que se podría hacer con el lema de bombeo es usarlo para demostrar que un lenguaje es regular de la siguiente forma:

1. Identificar un lenguaje regular L
2. Escoger una longitud n para xy
3. Proponer una forma de cadena que dependa de n pero que sea mayor que n
4. Particionar la cadena w en xyz
5. Verificar que la partición cumpla con las restricciones:
    1. $xyz = w$
    2. $|xy| \leq n$
    3. $y \neq \varepsilon$
6. Generar cadenas bombeadas $xy^k z \in L$

Si se cumple para todas las formas de la cadena del lenguaje L podríamos concluir que es un lenguaje regular.

5.2.3. Cómo realmente usamos el lema de bombeo

Si tratáramos de demostrar que un lenguaje es regular tendríamos que enumerar todas las formas y para todas demostrar que al bombear la cadena pertenece al lenguaje; esto no va a ser directo porque tendríamos que demostrar que tenemos una lista de todas las formas posibles de las cadenas del lenguaje, regresando a nuestro problema original. En realidad no usamos el lema de bombeo para demostrar que un lenguaje es regular... ¿entonces para qué lo usamos? Lo usamos para demostrar que un lenguaje no es regular, vamos a partir de qué es regular y si para una de las formas del lenguaje regular no se cumple el lema de bombeo, habremos demostrado que un lenguaje no es regular.

5.2.4. Ejemplo de uso del lema de bombeo

Dado $\Sigma = \{a, b\}$ demostrar que el lenguaje formado por cadenas con la forma $a^n b^n$ no es regular. Para esto, vamos a seguir la aplicación del lemma de bombeo:

1. Identificar un lenguaje regular $L = \{w|w = a^n b^n\}$
2. Escoger una longitud n
   Qué tal si en lugar de escoger un valor específico, escogemos cualquier valor, es decir n
3. Proponer una forma de cadena que dependa de n pero que sea mayor que n Propongo: $a^n b^n$
4. Particionar la cadena w en xyz $x = \varepsilon, y = a^n, z = b^n$
5. Verificar que la partición cumpla con las restricciones:
   1. $xyz = \varepsilon a^n b^n$ ✔
   2. $|xy| = n$ ✔
   3. $y \neq \varepsilon$ ✔
6. Generar cadenas bombeadas $xy^k zinL$
   1. Para k=0, $xyz = \varepsilon b^n$ pero $\varepsilon b^n \notin L$ ☒
   2. Para k=2, $xyz = \varepsilon a^{2n} b^n$ pero $\varepsilon a^{2n} b^n \notin L$ ☒

Dado que no se cumplió para cualquier valor de k que la cadena bombeada perteneciera al lenguaje original, algo hicimos mal. En este caso lo que hicimos mal fue haber dicho que L era regular. Por lo anterior concluimos que el lenguaje formado por cadenas con la forma $a^n b^n$ no es regular.

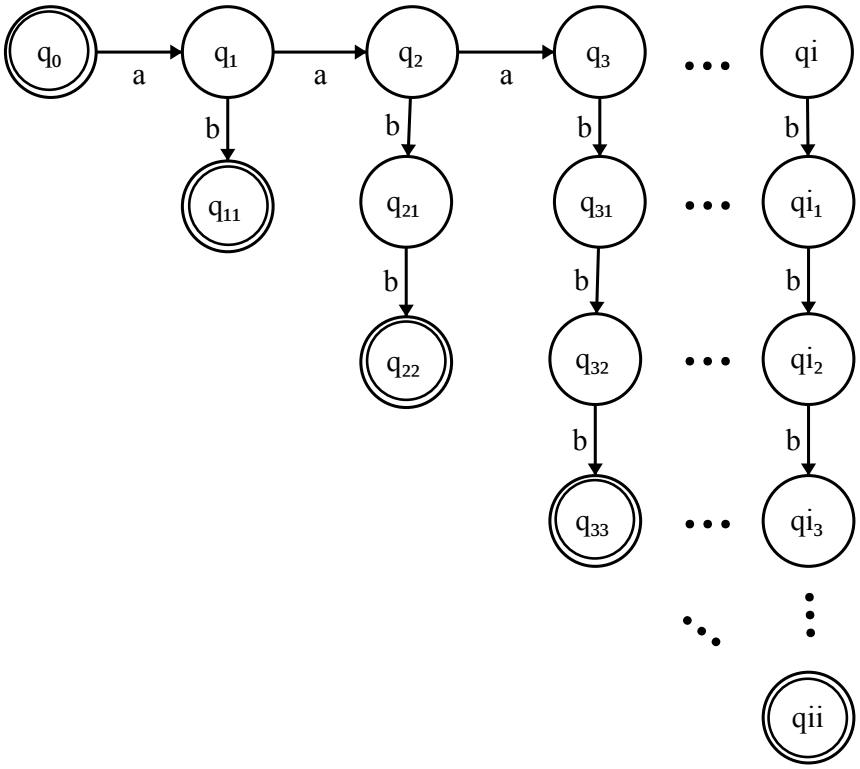
¿El procedimiento pudo haber sido hecho de forma diferente?

Respuesta

Sí, hay varios puntos de decisión; por ejemplo los siguientes a partir del paso 4.
Paso 4
$x = a^{n-1}, y = a, z = b^n$
Paso 5
$xyz = a^{n-1}ab^n = a^n b^n$ ✔
$|xy| = n$ ✔
$y \neq \varepsilon$ ✔
Paso 6
Para k=0,
$xyz = a^{n-1}b^n$ pero $a^{n-1}b^n \notin L$ ☒
Para k=2,
$xyz = a^{n-1}a^2 b^n = a^{n+1}b^n$ pero $a^{n+1}b^n \notin L$ ☒
Por lo tanto también llegamos a la conclusión que no es un lenguaje regular.

5.2.5. El lenguaje de las cadenas $a^n b^n$

A diferencia de los lenguajes que hasta este momento habíamos visto, este lenguaje pide algo que no habíamos visto antes. Una parte del lenguaje depende del otra parte. La parte de bes depende de la de aes ya que el número de aes define cuantas bes habrá; entonces necesitamos una forma de pasar esta información a la parte de bes, pero hasta ahora no tenemos una forma de pasar información entre los estados, pero los estados mismos. Con esto en mente, si tratáramos de diseñar un AF para este lenguaje quedaría de esta forma:



{#anbn, height=250px }

Como es evidente de ese diseño, ese autómata no es finito; por lo tanto no cabe dentro de nuestra definición de ningún autómata finito, por lo tanto tampoco es un lenguaje regular.

5.3. Gramaticas libres de contexto

En la sección anterior identificamos al lenguaje formado por cadenas del tipo $a^n b^n$ que no es regular ¿Entonces qué es? Antes de poder explicar más sobre este lenguaje vamos a presentar

el concepto de gramáticas a través del tipo denominado libres de contexto, que nos ayudarán a generar cadenas de este lenguaje.

5.3.1. GLC

Definición 4. Una Gramática Libre de Contexto (GLC) es una tupla (V, Σ, P, S) donde:

- V es un alfabeto de símbolos no terminales o símbolos auxiliares
- Σ es un alfabeto de símbolos terminales que conforman nuestras cadenas
- P es un conjunto de reglas con la forma $A \to \alpha$ donde $\alpha \in (\Sigma \cup V)^*$

- $S \in V$ lo denominamos símbolo inicial

Algunas preferencias en la notación

- Es usual definir los símbolos no terminales con mayúsculas.
- Es usual definir los símbolos terminales con minúsculas.
- Es usual usar S, de start, o R, de root, como símbolos iniciales.
- En las reglas, a la parte izquierda de estas (antes de la flecha) se le denomina cabeza de la regla, y la parte derecha (después de la flecha) se le denomina cuerpo de la regla.

5.3.1.1. Ejemplo de GLC

Un ejemplo de gramática es $(\{S\}, \{a, b\}, \{S \to aSb, S \to \varepsilon\}, S)$

También es correcto definirla de la siguiente forma:

$(\{S\}, \{a, b\}, P, S)$ donde P:

$$S \to aSb$$
$$S \to \varepsilon$$

Algunas preferencias en la notación

- Es común usar una | para agrupar reglas que tienen la misma cabeza. De tal forma que la definición puede ser:

$(\{S\}, \{a, b\}, P, S)$ donde P:

$$S \to aSb|\varepsilon$$

- También es común omitir la tupla y sólo poner las reglas; en este caso los elementos se infieren de las reglas. De tal forma que la definición de la gramática puede ser simplemente:

$$S \rightarrow \quad aSb|\varepsilon$$

5.3.2. Proceso de re-escritura

Las gramáticas se interpretan con un proceso de re-escritura; uno comienza del símbolo inicial y re-escribe símbolos no terminales hasta que lo que quede sea una cadena conformada por símbolos no terminales.

Al camino seguido de re-escritura para generar una cadena se denomina derivacion; para señalar los pasos intermedios se usa el símbolo $\Rightarrow$.

5.3.3. Ejemplo de derivación

Usando la gramática $S \rightarrow aSb|\varepsilon$ se puede generar la siguiente derivación:

$$\begin{aligned}
\underline{S} &\Rightarrow a\underline{S}b \\
&\Rightarrow aa\underline{S}bb \\
&\Rightarrow aaa\underline{S}bbb \\
&\Rightarrow aaaa\underline{S}bbbb \\
&\Rightarrow aaaaa\underline{S}bbbbb \\
&\Rightarrow aaaaa\varepsilon bbbbb = aaaaabbbbb
\end{aligned}$$

Algunas preferencias en la notación

- Es común usar el símbolo $\Rightarrow^*$ para significar que el siguiente paso en realidad requiere múltiples pasos intermedios. Por ejemplo, la cadena anteriormente generada puede escribirse como:

$$\underline{S} \quad \Rightarrow^* \quad aaaaabbbbb$$

5.3.4. El lenguaje generado por una GLC

Debe ser intuitivo que la gramática de ejemplo hasta este momento genera únicamente cadenas de la forma $a^n b^n$. En este momento tenemos un gramática que genera el lenguaje que no es regular. Como existen muchas gramáticas, hay muchos lenguajes generados por las GLC, y muchos de

estos lenguajes no son Lenguajes Regulares. A los lenguajes generados por las GLC se les denomina Lenguajes Libres de Contexto.

Formalmente un lenguje L generado por una gramática G se define como $L = \{w | S \Rightarrow^* w\}$ donde S es el símbolo inicial de la gramática $G = (V, \Sigma, P, S)$. Es decir el lenguaje de asociado a una gramática es el conjunto de todas las cadenas que se derivan partiendo del símbolo inicial y siguiendo las reglas de producción.

5.4. Ejemplo de gramática

La siguiente gramática genera cadenas que representan Expresiones Regulares.

$$
\begin{aligned}
R &\rightarrow B \\
R &\rightarrow R + R \\
R &\rightarrow R^* \\
R &\rightarrow RR \\
R &\rightarrow (R) \\
B &\rightarrow a \\
B &\rightarrow b \\
B &\rightarrow \epsilon \\
B &\rightarrow \emptyset
\end{aligned}
$$

5.4.1. Ejemplos de derivaciones

La siguiente es un ejemplo de una derivación para producir la cadena $(a^*ba^*ba^*)^*$, cabe notar que siempre se cambia el no terminal más a la izquierda.

$$
\begin{aligned}
\underline{R} \Rightarrow \quad & \underline{R^*} \\
\Rightarrow \quad & (\underline{R})^* \\
\Rightarrow \quad & (\underline{R}R)^* \\
\Rightarrow \quad & (\underline{R}RR)^* \\
\Rightarrow \quad & (\underline{R^*}RR)^* \\
\Rightarrow \quad & (\underline{B^*}RR)^* \\
\Rightarrow \quad & (a^*\underline{R}R)^* \\
\Rightarrow \quad & (a^*\underline{B}R)^* \\
\Rightarrow \quad & (a^*b\underline{R})^* \\
\Rightarrow \quad & (a^*b\underline{R^*})^* \\
\Rightarrow \quad & (a^*b\underline{R}R^*)^* \\
\Rightarrow \quad & (a^*b\underline{R}RR^*)^* \\
\Rightarrow \quad & (a^*b\underline{R^*}RR^*)^* \\
\Rightarrow \quad & (a^*b\underline{B^*}RR^*)^* \\
\Rightarrow \quad & (a^*ba^*\underline{R}R^*)^* \\
\Rightarrow \quad & (a^*ba^*B\underline{R^*})^* \\
\Rightarrow \quad & (a^*ba^*b\underline{R^*})^* \\
\Rightarrow \quad & (a^*ba^*b\underline{B^*})^* \\
\Rightarrow \quad & (a^*ba^*ba^*)^*
\end{aligned}
$$

A la derivación anterior se le conoce como derivación por la izquierda. También es posible hacerlo de derecha a izquierda:

$$
\begin{aligned}
\underline{R} \Rightarrow\ & \underline{R^*} \\
\Rightarrow\ & (\underline{R})^* \\
\Rightarrow\ & (R\underline{R})^* \\
\Rightarrow\ & (RR\underline{R})^* \\
\Rightarrow\ & (RR\underline{R^*})^* \\
\Rightarrow\ & (RR\underline{B^*})^* \\
\Rightarrow\ & (R\underline{R}a^*)^* \\
\Rightarrow\ & (R\underline{B}a^*)^* \\
\Rightarrow\ & (\underline{R}ba^*)^* \\
\Rightarrow\ & (R\underline{R}ba^*)^* \\
\Rightarrow\ & (R\underline{R^*}ba^*)^* \\
\Rightarrow\ & (R\underline{B^*}ba^*)^* \\
\Rightarrow\ & (\underline{R}a^*ba^*)^* \\
\Rightarrow\ & (R\underline{R}a^*ba^*)^* \\
\Rightarrow\ & (R\underline{B}a^*ba^*)^* \\
\Rightarrow\ & (\underline{R}ba^*ba^*)^* \\
\Rightarrow\ & (\underline{R^*}ba^*ba^*)^* \\
\Rightarrow\ & (\underline{B^*}ba^*ba^*)^* \\
\Rightarrow\ & (a^*ba^*ba^*)^*
\end{aligned}
$$

A esta estrategia para derivar se le conoce como derivación por la derecha.

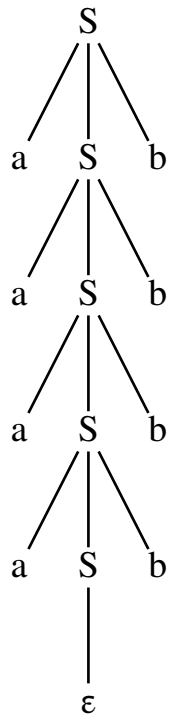
Lo anterior nos muestra que una cadena puede tener múltiples derivaciones.

5.5. Árboles de derivación

Un árbol de derivación es una forma de visualizar las relaciones que tienen las cabezas de reglas junto con los cuerpos de estas. En un árbol de derivación, los elementos del cuerpo de una regla se convierte en nodos hijos de la cabeza de la regla.

5.5.1. Ejemplo árbol de derivación para la cadena *aaabbb*

Para la gramática $S \rightarrow aSb|\varepsilon$ y la cadena aaabbb el árbol de derivación es:



{#arbol_aaabbb, height=250px }

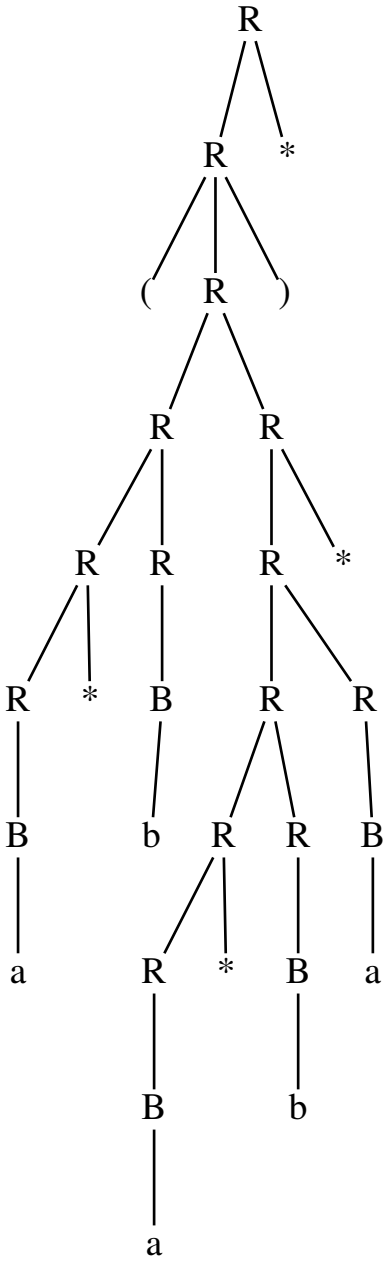
Para este ejemplo, sólo tenemos una derivación posible y sólo tenemos un árbol de derivación.

5.5.2. Ejemplo árbol de derivación para la cadena *(a*ba*ba*)**

Para la gramática

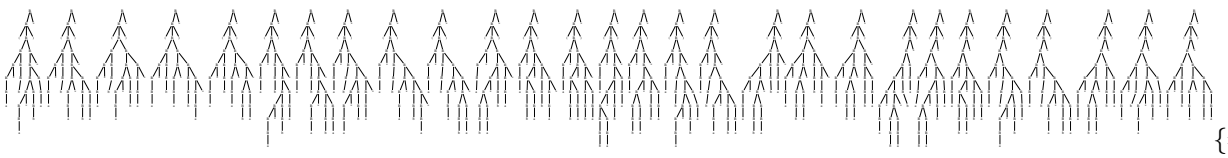
$$
\begin{aligned}
R &\rightarrow B \\
R &\rightarrow R + R \\
R &\rightarrow R^* \\
R &\rightarrow RR \\
R &\rightarrow (R) \\
B &\rightarrow a \\
B &\rightarrow b \\
B &\rightarrow \epsilon \\
B &\rightarrow \emptyset
\end{aligned}
$$

y las derivaciones de la cadena $(a^*ba^*ba^*)^*$ vistas en la sección anterior el árbol de derivación resulta:



{#arbol_re, height=250px }

En este caso varias derivaciones pueden resultar en un sólo árbol de derivación. Sin embargo, esta gramática tiene múltiples derivaciones y alguna otras generan otros árboles de derivación.



{#ar-

boles_re, height=250px }

En resumen

Para una gramática G un cadena puede tener múltiples derivaciones, dependiendo de la estrategia de sustitución que se elija; algunas de estas derivaciones resultarán en un solo árbol de derivación; sin embargo, es posible que algunas derivaciones de la cadena resulten en múltiples árboles de derivación, por lo tanto una cadena tenga asociadas múltiples árboles de derivación.

6 Gramáticas libres de contexto en su hábitat... y AP

6.1. ¿Dos tipos de lenguajes?

¿Por qué existen dos tipos de lenguajes? Hasta este momento hemos establecido que existen dos tipos de lenguajes: los Lenguajes Regulares (LR) están asociados a los autómatas finitos y Lenguajes Libres de Contexto (LLC) están asociados a las Gramáticas Libres de Contexto (GLC). En interesante notar, que mientras los LLC no pueden ser aceptados por automátas finitos, los LR si pueden ser generados por una gramática. Por ejemplo, la siguiente gramática genera las cadenas con número par de bes:

$$S \rightarrow aS| \rightarrow bB|\varepsilon$$
$$B \rightarrow aS| \rightarrow bS$$

¿Cual es la derivación para la cadena abbaaa?

Respuesta

$$\begin{aligned} S &\Rightarrow aS \\ &\Rightarrow abB \\ &\Rightarrow abbS \\ &\Rightarrow abbaS \\ &\Rightarrow abbaaS \\ &\Rightarrow abbaaaS \\ &\Rightarrow abbaaa \end{aligned}$$

¿Por qué se da la situación de existen dos tipos de lenguajes? Hasta ahora sabemos que esta es una propiedad del ecosistema de lenguajes. De la definición de los LR podemos notar que los lenguajes tienen que ver con las composicionalidad de los lenguajes, la composicionalidad permite construir

un sin fin de estructuras/patrones, una analogía serían los legos que nos permiten de consturuir un sin fin de formas; mientras que los LLC tienen que ver con la estructura en particular la de un árbol, una analogía sería la construcción de un puente; es importante notar que es posible construir un puente con legos, sin embargo las GLC permiten la construcción de un infinito número de puentes con una estructura finita, situación que los LR no permiten.

Sin embargo todavía hay varías preguntas sin responder:

- ¿Cuál es la relación entre los LR y LLC?
- ¿Hay una gramática para los LR?
- ¿Hay una máquina para los LLC?

6.2. Ambigüedad

En sección anterior vimos algunas particularidades sobre los árboles de derivación:

1. La gramática $S \to aSb|\varepsilon$ produce sólo una derivación y un solo árbol de derivación
2. La gramática para las Expresiones Regulares presentada en esa sección produce múltiples derivaciones, para algunas de estas derivaciones tiene mismo árbol de derivación, pero en general tienen múltiples árboles de derivación.

Hasta este momento tenemos:

- Una cadena puede tener múltiples derivaciones
- Varías derivaciones pueden corresponder a un árboles de derivación, por lo tanto una cadena aunque tenga múltiples derivaciones puede tener un sólo árbol de derivación.
- Sino se cumple lo anterior, una cadena puede tiene múltiples árboles de derivación

A la propiedad de que una cadena estar asociada a múltiples árboles se le conoce como ambigüedad. Decimos quela cadena es ambigua.

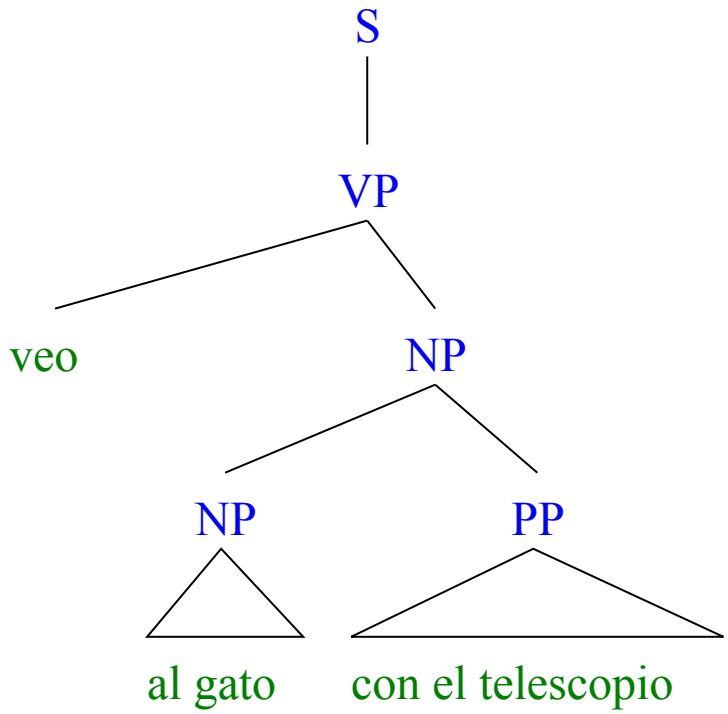
6.2.1. Ambigüedad para los humanos

La ambigüedad es muy importante para las humanos, en lenguaje natural es común encontrar para una misma oración (cadena) múltiples sentidos (árboles de derivación):

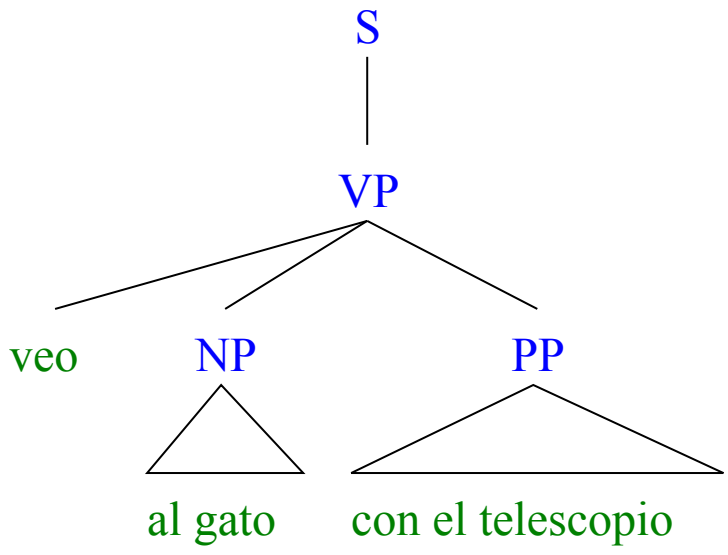
- Veo al gato con el telescopio

  ¿Quién tiene el telescopio, yo o el gato?

Para este caso estos son los dos árboles:



{#veo_gato_telescopio2, height=250px
}



{#veo_gato_telescopio1, height=250px
}

De hecho, la ambigüedad alguna veces crea una reacción en nosotros, risa:

- ¿Cómo te llamas?
  María de los Ángeles

Respuesta

Yo soy Carlos de Nueva York

- Oye, pues mi hijo en su nuevo trabajo se siente como pez en el agua
  ¿Qué hace?

Respuesta

Nada

De alguna forma la ambigüedad son las cosquillas del cerebro 🤯.

6.3. Gramáticas ambiguas

Hemos establecido que una cadena puede ser ambigua, y en particular se debe por la gramática que la genera; entonces de alguna forma la gramática es quien impone esa propiedad sobre la cadena. Con esto en mente, vamos a definir que una gramática es ambigua si genera una cadena ambigua, con una sola cadena que se genere que tenga dos árboles de derivación a la gramática se le considera ambigua. Tomando en cuenta esta definición ¿la gramática vista para las cadenas de la forma $a^n b^n$ es ambigua?

Respuesta

No, ninguna cadena generada genera dos árboles de derivación.

¿Y la gramática de cadenas de expresiones regulares?

Respuesta

Sí, al menos la cadena $(a^*ba^*ba*^a)^*$ tiene múltiples árboles de derivación.

6.4. ER como gramática no ambigua

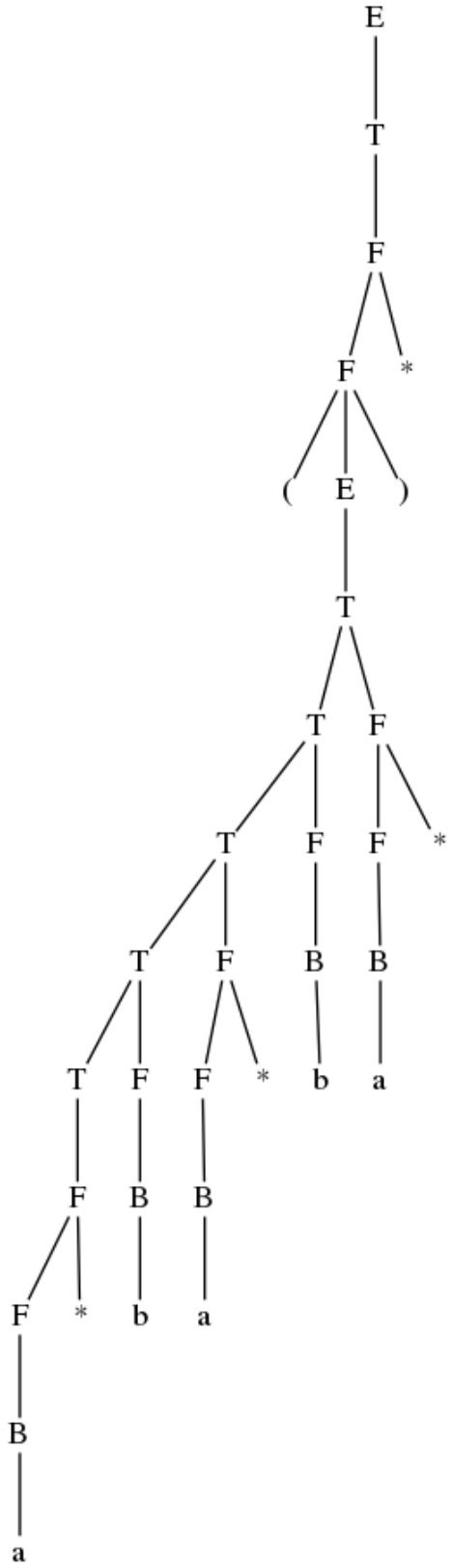
Como vimos la gramática del lenguaje de las Expresiones Regulares puede ser ambigua, sin embargo es posible generar una gramática para el lenguaje que no es ambigua:

$$
\begin{aligned}
E \rightarrow &\quad T|E+T \\
T \rightarrow &\quad F|TF \\
F \rightarrow &\quad B|F^*|(E) \\
B \rightarrow &\quad a|b|\varepsilon|\emptyset
\end{aligned}
$$

Con esta gramática se obtiene la siguiente derivación:

$$
\begin{aligned}
E \Rightarrow &\quad T \\
\Rightarrow &\quad F \\
\Rightarrow &\quad F^* \\
\Rightarrow &\quad (E)^* \\
\Rightarrow &\quad (T)^* \\
\Rightarrow &\quad (TF)^* \\
\Rightarrow &\quad (TFF)^* \\
\Rightarrow &\quad (TFFF)^* \\
\Rightarrow &\quad (TFFFF)^* \\
\Rightarrow &\quad (FFFFF)^* \\
\Rightarrow &\quad (F^*FFFF)^* \\
\Rightarrow &\quad (B^*FFFF)^* \\
\Rightarrow &\quad (a^*bFFF)^* \\
\Rightarrow &\quad (a^*ba^*FF)^* \\
\Rightarrow &\quad (a^*ba^*BF)^* \\
\Rightarrow &\quad (a^*ba^*bF)^* \\
\Rightarrow &\quad (a^*ba^*bF^*)^* \\
\Rightarrow &\quad (a^*ba^*bB^*)^* \\
\Rightarrow &\quad (a^*ba^*ba^*)^*
\end{aligned}
$$

Esta es la única derivación, y por lo tanto sólo tiene un árbol de derivación:



{#arboles_re, height=250px }

De hecho para todas las cadenas sólo habrá un árbol de derivación, es decir esta gramática es no ambigua.

Con esto se puede apreciar que un lenguaje puede ser generado por gramática ambigua o no ambigua.

6.5. Lenguaje ambiguo

Hasta ahora hemos visto a la ambigüedad como una propiedad de las gramáticas, pero considere los siguientes lenguajes:

1. El lenguaje formado por cadenas con la forma $a^n b^n c^m d^m$
2. El lenguaje formado por cadenas con la forma $a^n b^m c^m d^n$

La siguiente gramática genera el primer lenguaje:

$$\begin{aligned} X &\rightarrow YZ \\ Y &\rightarrow aYb|\varepsilon \\ Z &\rightarrow cZd|\varepsilon \end{aligned}$$

Un ejemplo de cadena que pertenece al lenguaje es aabbcd.

La siguiente gramática genera el segundo lenguaje:

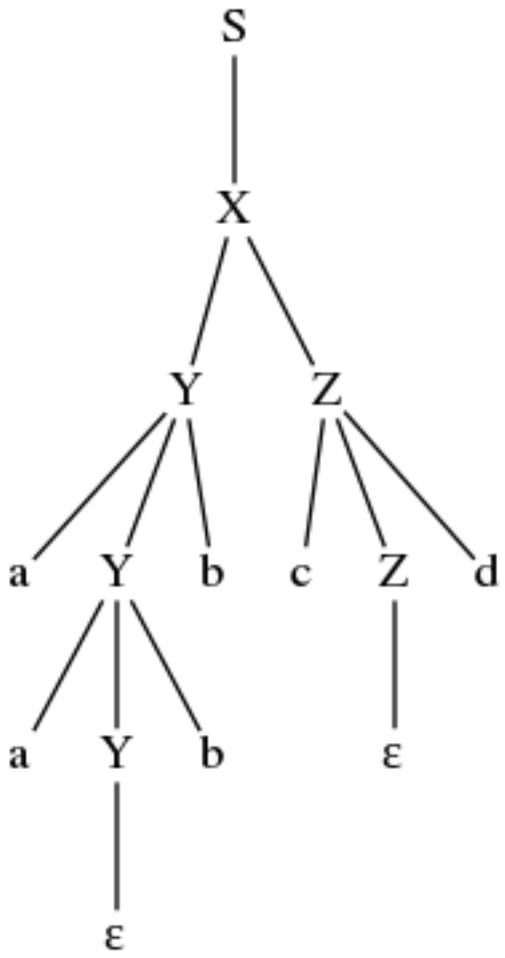
$$\begin{aligned} R &\rightarrow aRd|T \\ T &\rightarrow bTc|\varepsilon \end{aligned}$$

Un ejemplo de cadena que pertenece al lenguaje es aabcdd.

Si deseamos el lenguaje que genere ambos lenguajes, la siguiente gramática lo genera:

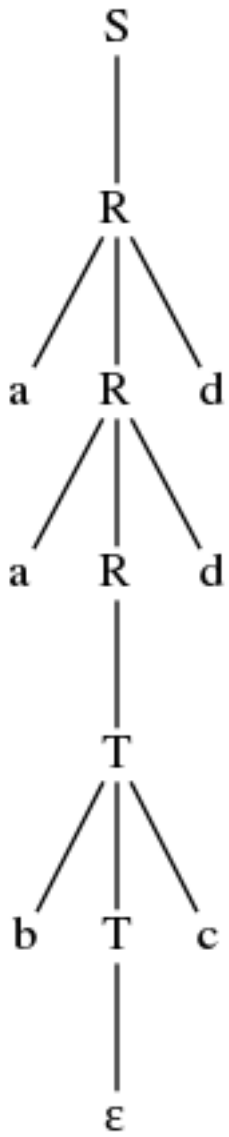
$$\begin{aligned} S &\rightarrow X|R \\ X &\rightarrow YZ \\ Y &\rightarrow aYb|\varepsilon \\ Z &\rightarrow cZd|\varepsilon \\ R &\rightarrow aRd|T \\ T &\rightarrow bTc|\varepsilon \end{aligned}$$

Usando esta gramática produce el siguiente árbol de derivación para la cadena aabbcd



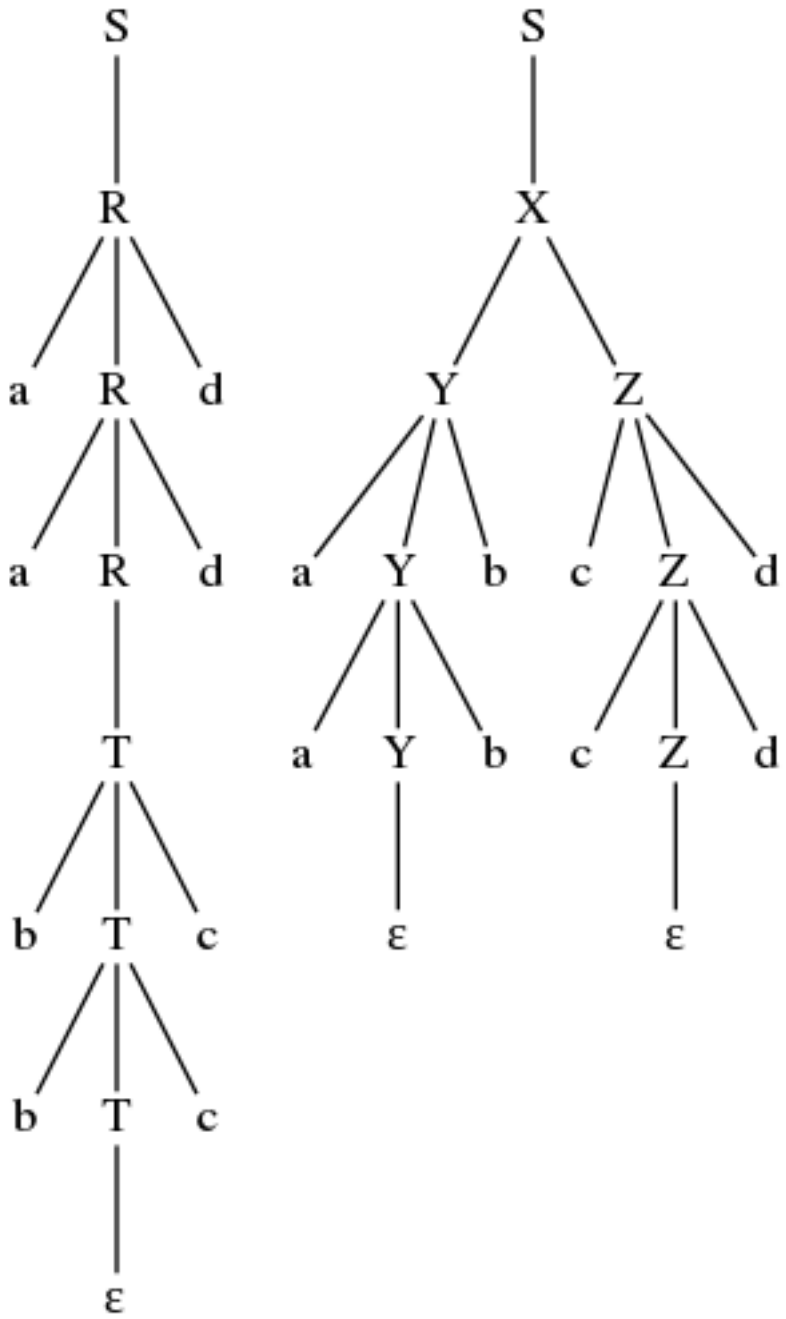
{#aabbcd, height=250px }

Y el siguiente árbol de derivación para la cadena aabcdd



{#aabcdd, height=250px }

Y los siguientes árboles de derivación para aabbccdd



{#aabbccdd, height=250px }

6.5.1. No hay gramática no ambigua

Como se puede ver, dado que para la cadena aabbccdd hay dos árboles de derivación, la gramática es ambigua. Sin embargo, no hay forma de organizar una gramática que genere cadenas de alguno de los lenguajes originales sin que sea ambigua. En este caso la propiedad de ambigüedad viene

del lenguaje, el que es ambiguo es el lenguaje y no la gramática.

6.6. Lenguajes regulares y GLC

En la sección anterior vimos que los lenguajes de dos gramáticas se pueden unir mediante la incorporación de un nuevo símbolo inicial y reglas de producción que vayan de ese nuevo símbolo a los símbolos iniciales de las gramáticas originales. La pregunta que surge inmediatamente es si podemos codificar algunas otras operaciones de lenguajes regulares.

6.6.1. Operaciones de lenguajes regulares como GLC

Para lenguajes con las siguientes dos gramáticas $G_1 = (V_1, \Sigma, P_1, S_1)$ y
$G_2 = (V_2, \Sigma, P_2, S_2)$ se pueden definir las siguientes gramáticas por operación de Lenguaje regular:

6.6.1.1. Unión

La gramática que hace la unión de los dos lenguajes basado en sus gramáticas es:

$$G_U = (V_1 \cup V_2, \Sigma, P_1 \cup P_2 \cup \{S_U \to S_1 + S_2\}, S_U)$$

6.6.1.2. Concatenación

La gramática que hace la concatenación de los dos lenguajes basado en sus gramáticas es:

$$G_C = (V_1 \cup V_2, \Sigma, P_1 \cup P_2 \cup \{S_C \to S_1 S_2\}, S_C)$$

6.6.1.3. Cerradura

La gramática que hace la cerradura de un lenguaje basado en sus gramáticas es:

$$G_* = (V_1, \Sigma, P_1 \cup \{S_* \to S_1 S_* | \epsilon\}, S_*)$$

6.6.2. Los lenguajes regulares básicos como GLC

Ya que es posible codificar las operaciones de lenguajes regulares a través de la creación de nuevas gramáticas (secuencia de operaciones) ¿Qué hay de los lenguajes básicos? Resulta que se pueden codificar como gramáticas

6.6.2.1. Lenguaje vació

$$G = (V, \Sigma, \emptyset, S)$$

6.6.2.2. Lenguaje de la cadena vacía

$$G = (V, \Sigma, \{S \to \epsilon\}, S)$$

6.6.2.3. Lenguaje de un símbolo del alfabeto

$$G = (V, \Sigma, \{S \to a\}, S)$$

6.6.3. Las GLC generan a cualquier LR

El hecho de que se puedan definir los lenguajes básicos como gramáticas y que existan las operaciones para construir lenguajes más complejos con base a estos nos indica que una GLC puede generar un Lenguaje Regular!

6.7. AF $\to$ GLC

Es posible transformar un AF a una GLC, recordemos que las GLC pueden generar a los lenguajes regulares, por lo tanto suena posible que un AF que acepta a un Lenguaje Regular pueda trasformarse en un AF.

6.8. Transformación

1. Cambiar los estados por símbolos auxiliares que conformaran V, un símbolo auxiliar por estado q del AF.

2. Por cada transición crear una regla de producción con el siguiente formato:

$$A_{origen} \rightarrow a_{simbolo} B_{destino}$$

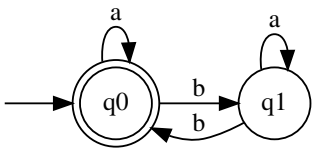
Donde A representa a un estado origen y B un estado destino a través del símbolo a.

3. Agregar una transición con la siguiente forma por cada transición del AF original que termine en un estado final.

$$A_{origen} \rightarrow a_{simbolo}$$

6.8.1. Ejemplo

Considere el siguiente AF:



{#../02lamáquinasinmemoria/bes_pares, height=250px }

1. Proponer símbolos auxiliares

$$V = \{A_{q_0}, B_{q_1}\}$$

2. Agregar reglas de producción por transición

$$\begin{aligned} A &\rightarrow aA \\ A &\rightarrow aB \\ B &\rightarrow aB \\ B &\rightarrow bA \end{aligned}$$

3. Agregar reglas de producción por transición que llega a estado final

$$\begin{aligned} A &\rightarrow a \\ B &\rightarrow b \end{aligned}$$

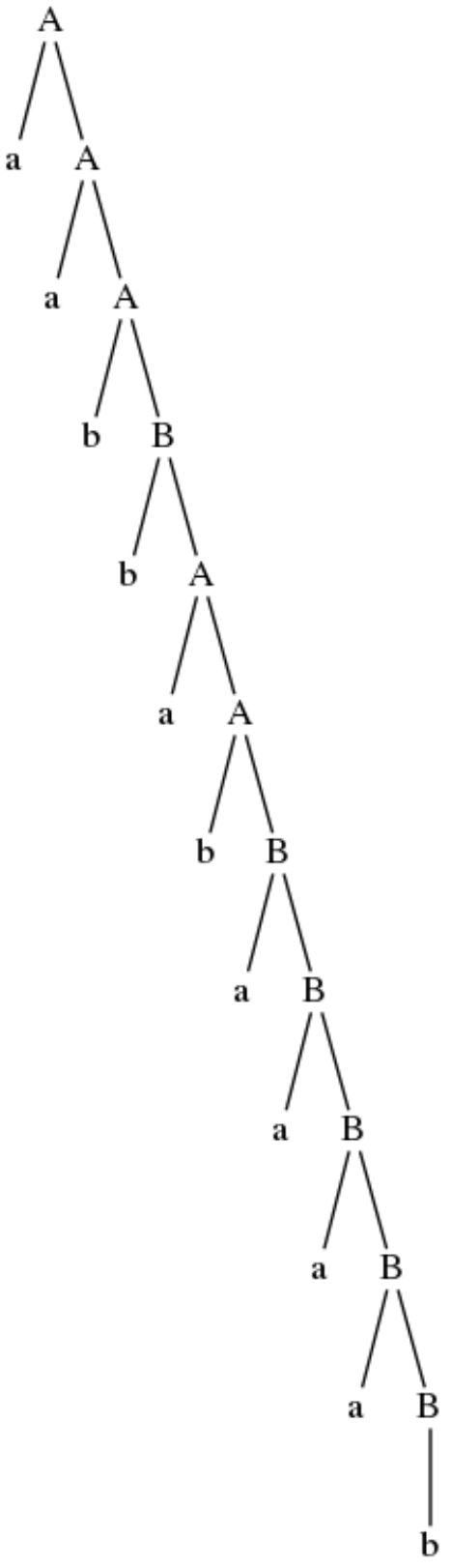
Nuestra gramática completa queda:

$$\begin{aligned} A &\rightarrow aA \\ A &\rightarrow aB \\ B &\rightarrow aB \\ B &\rightarrow bA \\ A &\rightarrow a \\ B &\rightarrow b \end{aligned}$$

La siguiente es la derivación para la cadena aabbabaaaaba:

$$\begin{aligned} A &\Rightarrow aA \\ &\Rightarrow aaA \\ &\Rightarrow aabB \\ &\Rightarrow aabbA \\ &\Rightarrow aabbaA \\ &\Rightarrow aabbabB \\ &\Rightarrow aabbabaB \\ &\Rightarrow aabbabaaB \\ &\Rightarrow aabbabaaaB \\ &\Rightarrow aabbabaaaaB \\ &\Rightarrow aabbabaaaab \end{aligned}$$

Dada la forma de las reglas es posible ver que siempre hay un sólo símbolo no terminal en lado derecho de la derivación, esto origina que sólo haya una derivación posible y además un árbol de derivación (que se muestra a continuación), es decir este tipo de gramáticas no son ambiguas.



{#gr, height=250px }

6.9. Gramáticas Regulares

En la sección anterior vimos que un AF se puede transformar en una GLC ¿La dirección contraria se puede? Es decir una GLC se pude transformar en un AF. Sabemos que no se pude para todos, porque existen GLC que generan lenguajes que no son regulares ¿Entonces bajo que condición se podría? Sólo se podrá para lo que denominaremos como:

6.9.1. GR

Definición 5. Una Gramática Regular (GLC) es una tupla (V, Σ, P, S) donde:

- Ves un alfabeto de símbolos no terminales o símbolos auxiliares
- Σ es un alfabeto de símbolos terminales, que conforman nuestras cadenas
- Pes un conjunto de reglas que con la forma $A \to bB|c$ donde $b \in \Sigma, c \in \Sigma, A \in V, B \in V$
- $S \in V$lo denominamos símbolo inicial

Si comparamos lo único que cambia con respecto a las GLC es la forma de la regla. Pasa de ser:

$$A \to \alpha$$

A esta forma que es mas restrictiva:

$$A \to bB|c$$

Con esto en mente, toda GR se puede convertir en un AF siguiendo un proceso inverso al presentado en la sección anterior.

6.10. Autómatas de pila

Para los Lenguajes Libres de Contextos, LLC, hemos visto que existe una gramática que los genera pero no hemos visto la máquina que los acepta, el Autómata de Pila será esta máquina.

6.10.1. AP

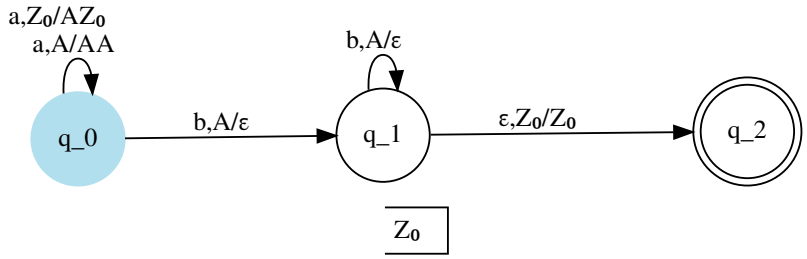
Definición 6. Un Autómata de Pila (AP) es una tupla $(Q, \Sigma, \Gamma, q_0, Z_0, A, \delta)$ donde:

- Q es un conjunto de estados finitos
- Σ es un alfabeto de símbolos terminales
- Γ es un alfabeto de la pila

- q_0 es un estado que denominaremos inicial donde $q_0 \in Q$
- Z_0 es un símbolo de la pila que denominaremos inicial de la pila donde $Z_0 \in \Gamma$
- A es un conjunto de estados que denominaremos finales donde $A \subset Q$
- δ es una función de transición que cumple con: $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \rightarrow Q \times \Gamma^*$

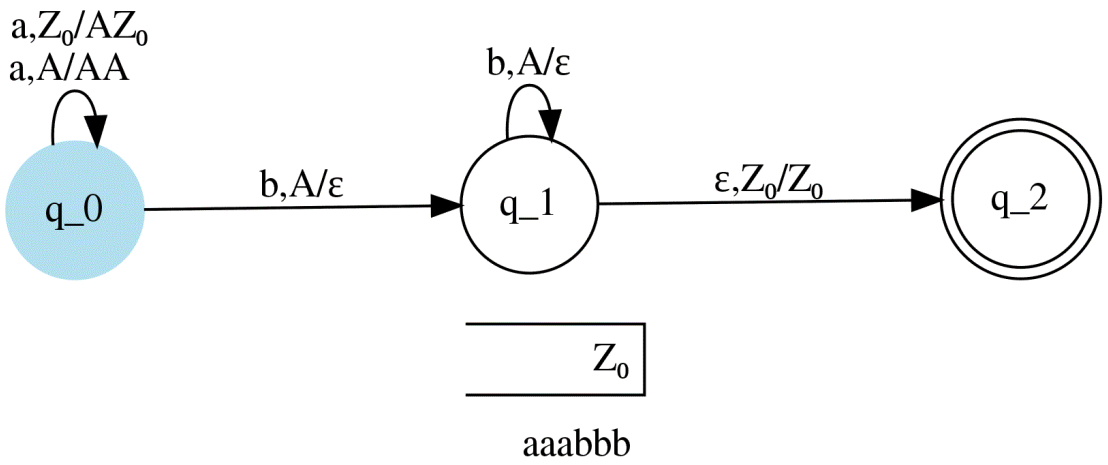
6.10.1.1. Ejemplo de AP

El siguiente autómata reconoce al lenguaje formado de cadenas $a^n b^n$ para cuando n>0:



{#anbn, height=250px }

Al igual que con los AF, al comenzar el análisis se comienza en el estado inicial y de ahí se tienen que poner atención a tres aspectos, el estado en el que se está, el símbolo en la cadena que está siendo analizado y el símbolo que se encuentra hasta arriba de la pila. El siguiente análisis muestra el proceso para la cadena aaabbb:



{#anbn, height=250px }

7 Depende del contexto

7.1. Resumen hasta ahora

Es un buen momento para hacer un recuento de los conceptos que hemos visto hasta el momento.

- Lenguajes Este es el concepto fundamental presentado en este curso, es un conjunto de cadenas.
- Máquinas Son mecanismos computacionales que permiten procesar una cadena y determinar si pertenece o no al lenguaje asociado a dicha máquina.
- Gramática Es un proceso de re-escritura basado en remplazo de reglas que permite generar cadenas de un lenguaje determinado.

Con esto en mente este es el mapa conceptual que sabemos en este momento:

Lenguaje	Gramática	Máquina	Ejemplo
Lenguaje Independiente del Contexto	$A \to \alpha$	Autómata de Pila	$a^n b^n$
Lenguaje Regular	$A \to bC \vert d$	AF, AFND y AFND-ε	a^n

Cabe recordar que los Lenguajes Independientes del Contexto contienen a los Lenguajes Regulares; una prueba de esto es en la definición de la forma de las reglas de producción de la Gramática Regular, estas cumplen también con la forma para reglas de producción de Gramáticas Independientes del Contexto, solo pasa que son más específicas.

7.2. Lenguajes de palíndromos

Un palíndromo es un texto que se lee igual de izquierda a derecha que de derecha a izquierda. Los palíndromos han sido explorado desde la literatura, un excelente expositor de los palíndromos en

español es el escrito argentino Juan Filloy, algunos ejemplos son:

- Acaso hubo búhos acá
- Adán y raza, azar y nada
- Allí va Ramón y no maravilla
- Al reparto, otra perla
- Amad a la dama
- Amada dama

Para más ejemplos hechos por Juan Filloy visitar Los palíndromos de Juan Filloy. Para ver más ejemplos de palíndromos buscar con el hashtag en la plataforma social Twitter.

7.2.1. Gramáticas para lenguajes de palíndromos

Desde el punto de vista de lenguajes formales, se pueden definir los siguientes lenguajes de palíndromos y sus gramáticas dado un alfabeto $\Sigma = \{a, b\}$, donde w^r es un cadena escrita en reversa:

- Palíndromos pares

$$ww^r$$

$$S \rightarrow \quad aSa|bSb|\varepsilon$$

- Palíndromos impares

$$wcw^r, c \in \Sigma$$

$$S \rightarrow \quad aSa|bSb|a|b$$

- Palíndromos pares e impares

$$w = w^r$$

$$S \rightarrow \quad aSa|bSb|a|b|\epsilon$$

- Palíndromos con marca en medio

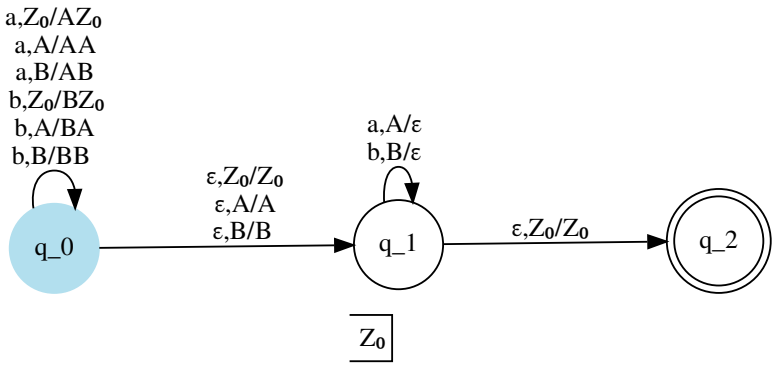
$$wmw^r$$

$$S \rightarrow \quad aSa|bSb|m$$

Como podemos ver todos las gramáticas son no ambiguas, ya que generan siempre un árbol de derivación para todos sus derivaciones, lo que muestra que estos lenguajes son no ambiguos.

7.2.1.1. AP para *ww*

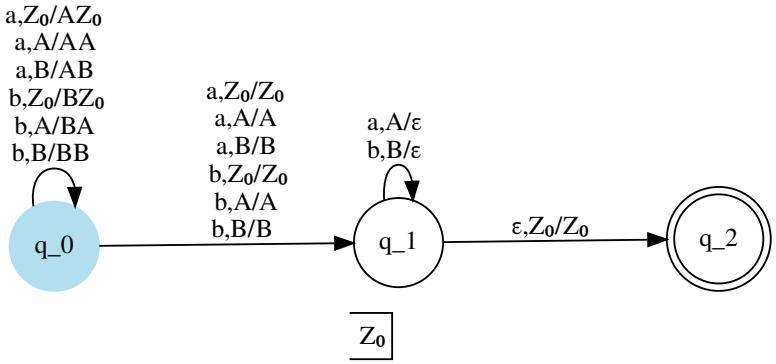
El autómata para este lenguaje luce de la siguiente forma, palíndromos pares



{#wwr, height=250px }

7.2.1.2. AP para *waw +wbw*

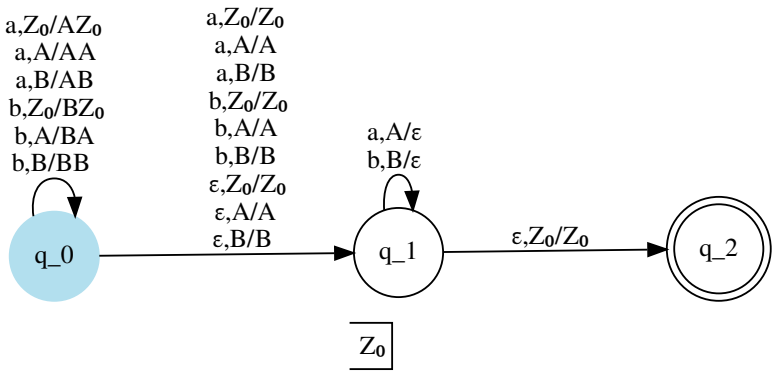
El autómata para este lenguaje luce de la siguiente forma, palíndromos impares



{#wawr, height=250px }

7.2.1.3. AP para *w=w*

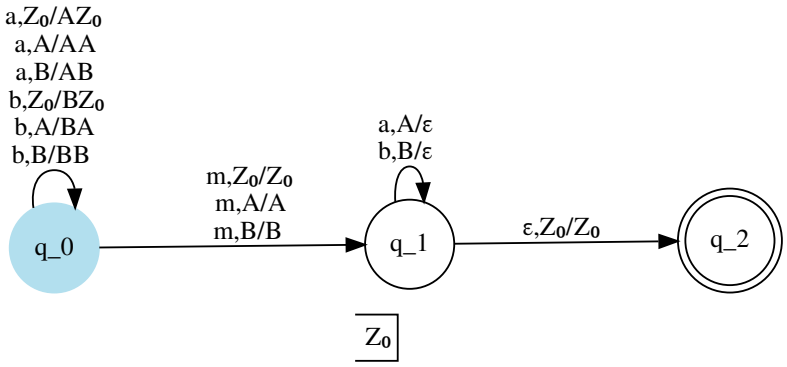
El autómata para este lenguaje luce de la siguiente forma, palíndromos



{#w_wr, height=250px }

7.2.1.4. AP para *wmw*

El autómata para este lenguaje luce de la siguiente forma, palíndromos



{#wmwr, height=250px }

7.3. Automata de Pila determinístico

En sección anterior vimos diferentes versiones de lenguajes de palíndromos, y aunque sus gramáticas fueron no ambiguas tres de estos lenguajes resultaron en un Autómata de Pila no determinístico; en este caso el no determinismo del autómata no solo hace que el autómata esté en varios

estados, pero estos están asociados a diferentes configuraciones de la pila; sin lugar a dudas el no determinismo complica más el análisis de una cadena.

Desafortunadamente para los lenguajes con formas: ww^r, waw^r y ww^r no es posible construir un AP no determinismo. Esto es derivado de la naturaleza del lenguaje, ya que dada la información que sabe el AP, estado, símbolo actual en la cadena y símbolo de la pila, no es posible saber que estrategia seguir: si seguir acumulando símbolos en la pila o ya llego a la mitad y tiene que comenzar a cancelarlos. Por supuesto, para nosotros con una cadena pequeña es fácil definir la estrategia, identificamos la mitad de la cadena y durante la primera mitad agregamos símbolos a la pila y durante la segunda quitamos símbolos. Sin embargo, el AP no tiene el lujo de ver toda la cadena sino símbolo por símbolo. Esto es diferente para el lenguaje con cadenas de la forma wmw^r, ya que la marca m nos especifica la mitad de la cadena, entonces no es necesario adivinar dónde está la mitad, como en los otros casos, donde como resultado que este lenguaje sí tiene un AP determinístico.

7.3.1. APD

Definición 7. Un Autómata de Pila Determinístico (APD) es una tupla $(Q, \Sigma, \Gamma, q_0, Z_0, A, \delta)$ donde:

- Q es un conjunto de estados finitos
- Σ es un alfabeto de símbolos terminales
- Γ es un alfabeto de la pila
- q_0 es un estado que denominaremos inicial donde $q_0 \in Q$
- Z_0 es un símbolo de la pila que denominaremos inicial de la pila donde $Z_0 \in \Gamma$
- A es un conjunto de estados que denominaremos finales donde $A \subset Q$
- δ es una función de transición que cumple con: $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \to Q \times \Gamma^*$

y además cumple con las siguientes restricciones:

- Solo hay una transición $\delta(q, a, x)$ por estado donde $q \in Q, a \in \Sigma, x \in \Gamma$
- Y si existe $\delta(q, \varepsilon, x)$ no existe ninguna transición con la forma $\delta(q, a, x)$ donde $q \in Q, a \in \Sigma, x \in \Gamma$

Esta definición es seguida por la definición AP las cadenas con la forma wmw^r. Para los otros AP vistos en esa sección no se cumplen estas restricciones:

- ww^r para el estado inicial rompen la segunda restricción en relación a épsilon, ya que existen transiciones por épsilon y símbolo usando usando el mismo símbolo de la pila.
- waw^r para el estado inicial rompen la primer restricción en relación a los símbolo, ya que existen más de una transición con el mismo símbolo del alfabeto y símbolo de la pila.

- $w=w^r$ para el estado inicial rompen la primer y segunda restricción.

7.4. Propiedad de determinismo asociado al lenguaje

Un lenguaje será determinístico si es posible construir un APD que lo acepte, sino es posible será no determinístico. De esta forma la propiedad de determinismo está asociada al lenguaje y no al AP.

7.4.1. Relación entre ambigüedad y no determinismo

En los ejemplos visto para los lenguajes de palíndromos vimos que no hay relación entre ambigüedad y no determinismo; un cuando los lenguajes son no ambiguos los lenguajes fueron no determinístico o determinístico. Sin embargo, si un lenguaje ambiguo será no determinístico, no habrá forma de procesar las cadenas sin explotar la ambigüedad; por otro lado, si el lenguaje es determinístico quiere decir que existe una gramática no ambigua que lo genera, y por lo tanto el lenguaje es no ambiguo. En resumen estos son los casos posibles:

- Lenguaje no ambiguo, lenguaje determinista o no determinista
- Lenguaje ambiguo, lenguaje no determinista
- Lenguaje determinista, lenguaje no ambiguo
- Lenguaje no determinista, lenguaje ambiguo o no ambiguo

7.5. Simulación de G como AP

Dada una gramática $G = (V, \Sigma, P, S)$ se puede crear una AP que lo simule $(Q, \Sigma, \Gamma, q_0, Z_0, A, \delta)$ donde:

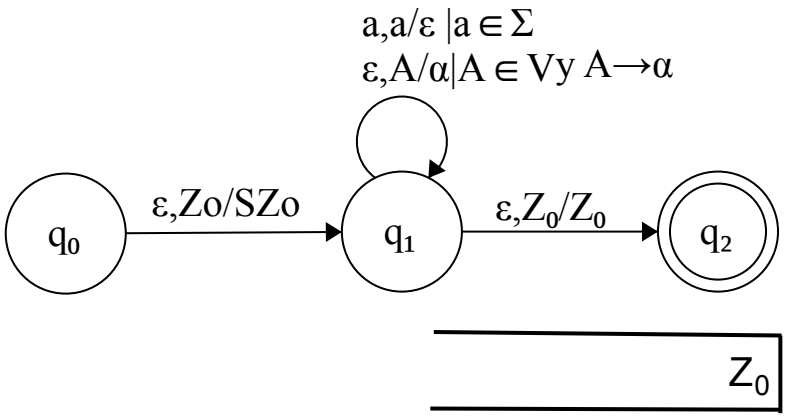
$$\begin{aligned} Q &= \{q_0, q_1, q_2\} \\ \Gamma &= V \cup \Sigma \cup \{Z_0\}, Z_0 \notin (V \cup \Sigma) \\ A &= \{q_2\} \end{aligned}$$

Donde las transiciones se definen de la sigueinte forma:

1. $\delta(q_0, \epsilon, Z_0) = \{(q_1, SZ_0\})\}$
2. $\delta(q_1, \epsilon, Z_0) = \{(q_2, Z_0\})\}$
3. Para todo $A \in V, \delta(q_1, \epsilon, A) = \{(q_1, \alpha\})|A \to \alpha\}$

4. Para todo$a \in \Sigma, \delta(q_1, a, a) = \{(q_1, \epsilon)\}$

De forma gráfica, el AP luce de la siguiente forma:



{#ap_glc_td, height=250px }

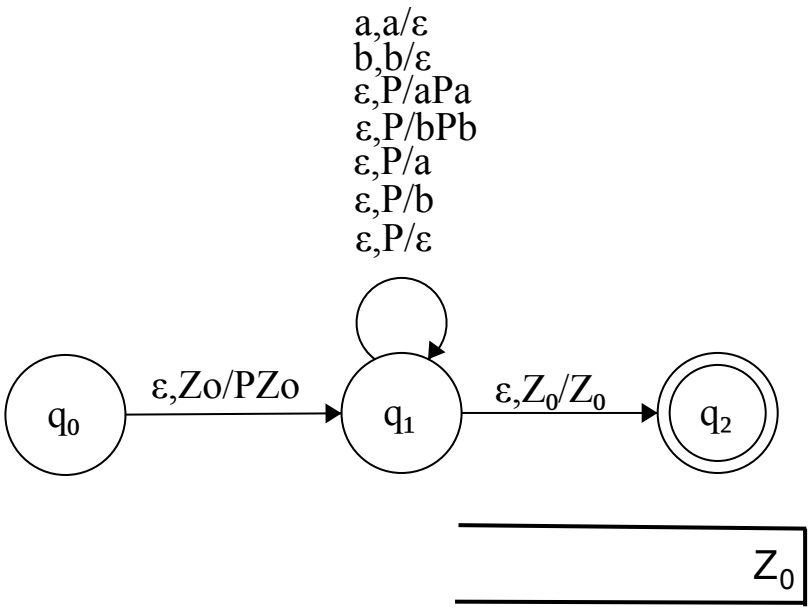
7.5.1. Ejemplo

Para la gramática de palíndromos

$$w = w^r$$

$$P \rightarrow \quad aPa|bPb|a|b|\epsilon$$

Este es su AP que lo simula:



{#ap_glc_tdxr, height=250px }

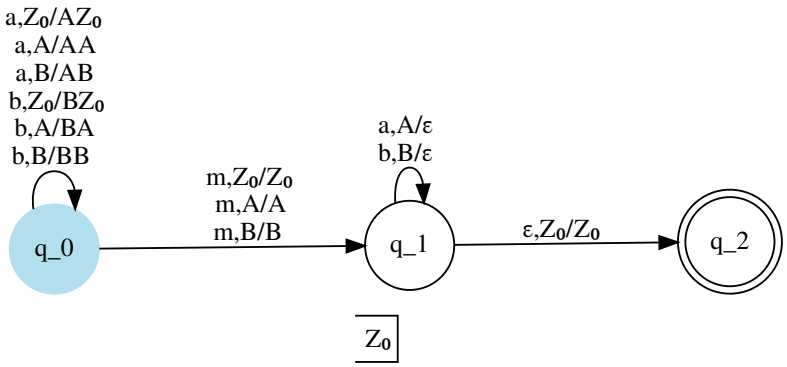
7.6. Simulación de AP como G

Para simular un AP con una gramática es necesario generar transformar las transiciones con forma $a, A/\alpha$ en la siguiente forma $A \to a\alpha$.

7.6.1. Casos dado el tipo de transición

1. Si la transición es un push la forma de la regla queda:
   - $Z_0 \to aAZ_0$
   - $A \to aAA$
2. Si la transición es un pop la forma de la regla queda:
   - $A \to a$
3. Si la transición que no modifica la pila la forma de la regla queda:
   - $Z_0 \to mZ_0$
   - $A \to mA$
4. Si es la transición de término del AP la forma de la regla queda:
   - $Z_0 \to \varepsilon$

7.6.1.1. Ejemplo



{#wmwr, height=250px }

La gramática que lo simula luce:

$$
\begin{aligned}
Z_0 &\rightarrow aAZ_0 \\
Z_0 &\rightarrow bBZ_0 \\
A &\rightarrow aAA \\
B &\rightarrow aAB \\
A &\rightarrow bBA \\
B &\rightarrow bBB \\
Z_0 &\rightarrow mZ_0 \\
A &\rightarrow mA \\
B &\rightarrow mB \\
A &\rightarrow a \\
B &\rightarrow b \\
Z_0 &\rightarrow \epsilon
\end{aligned}
$$

7.7. Lema de bombeo para lenguajes libres de contexto

Sea L un Lenguaje Libre de Contexto para todas la cadenas w se podrán descomponer en cinco partes uvwxy tales que:

1. $|vwx| \leq n$
2. $vx \neq \varepsilon$, puesto de otra forma $y > 1$
3. Para todo $k \geq 0$, las cadenas con la forma $uv^k wx^k y$ también pertenecen a L

7.7.1. Aplicación del lema de bombeo

El lema de bombeo se puede usar para demostrar que un lenguaje no es libre de contexto

1. Identificar el lenguaje libre de contexto L
2. Escoger una longitud n para xy
3. Proponer una forma de cadena que dependa de n pero que sea mayor que n
4. Particionar la cadena w en uvwxy
5. Verificar que la partición cumpla con las restricciones:
    1. $uvwxy = w$
    2. $|vwx| \leq n$
    3. $|vx| > 0$
6. Generar cadenas bombeadas $xy^k z \in L$

Si no cumple, se demuestra que el lenguaje no es libre de contexto.

7.7.1.1. Ejemplo de uso del lema de bombeo

Dado $\Sigma = \{a, b\}$ demostrar que el lenguaje formado por cadenas con la forma $a^n b^n c^n$ no es libre de contexto. Para esto, vamos a seguir la aplicación del lemma de bombeo:

1. Identificar un lenguaje regular $L = \{w | w = a^n b^n c^n\}$
2. Escoger una longitud n Qué tal si en lugar de escoger un valor específico, escogemos cualquier valor, es decir n
3. Proponer una forma de cadena que dependa de n pero que sea mayor que n Propongo: $a^n b^n c^n$
4. Particionar la cadena w en uvwxy, varios casos:
    - vwx son aes $u = \varepsilon, v = a^{\frac{n}{2}}, w = \varepsilon, x = a^{\frac{n}{2}}, y = b^n c^n$, no cumple porque al bombear de desbalancean las aes
    $$(a^{\frac{n}{2}})^k (a^{\frac{n}{2}})^k b^n c^n$$

- vwx son aes y bes $u = a^{\frac{n}{2}}, v = a^{\frac{n}{2}}, w = \varepsilon, x = b^{\frac{n}{2}}, y = b^{\frac{n}{2}}c^n$, no cumple porque al bombear de desbalancean las aes y bes

$$a^{\frac{n}{2}}(a^{\frac{n}{2}})^k(b^{\frac{n}{2}})^k b^{\frac{n}{2}}c^n$$

- vwx son bes $u = a^n, v = b^{\frac{n}{2}}, w = \varepsilon, x = b^{\frac{n}{2}}, y = c^n$, no cumple porque al bombear de desbalancean las bes

$$a^n(b^{\frac{n}{2}})^k(b^{\frac{n}{2}})^k c^n$$

- vwx son bes y ces $u = a^n b^{\frac{n}{2}}, v = b^{\frac{n}{2}}, w = \varepsilon, x = c^{\frac{n}{2}}, y = c^{\frac{n}{2}}$, no cumple porque al bombear de desbalancean las bes y ces

$$a^n b^{\frac{n}{2}}(b^{\frac{n}{2}})^k(c^{\frac{n}{2}})^k c^{\frac{n}{2}}$$

- vwx son bes $u = a^n b^n, v = c^{\frac{n}{2}}, w = \varepsilon, x = c^{\frac{n}{2}}, y = \varepsilon$, no cumple porque al bombear de desbalancean las ces

$$(a^n b^n(c^{\frac{n}{2}})^k(c^{\frac{n}{2}})^k$$

Por lo tanto el lenguaje formado de cadenas con la forma $a^n b^n c^n$ es no regular.

7.7.2. El lenguaje de las cadenas *a b c*

A diferencia de los lenguajes que hasta este momento habíamos visto, este lenguaje pide algo que no habíamos visto antes. Dos partes del lenguaje dependen del otra parte. La parte de bes y ces dependen de la parte de aes ya que el número de aes define cuantas bes y ces habrá, y aunque tenemos un mecanismo para pasar información, la pila, una vez que consumimos la información contenida en ella, la perdemos para la siguiente parte, haciendo imposible que este lenguaje sea posible de aceptar por un AP.

7.8. Gramática Dependiente del Contexto

En la sección anterior identificamos al lenguaje formado por cadenas del tipo $a^n b^n c^n$ que no es libre de contexto ¿Entonces qué es? Antes de poder explicar más sobre este lenguaje vamos a presentar el concepto de gramáticas dependientes de contexto, que nos ayudarán a generar cadenas de este lenguaje.

7.8.1. GLC

Definición 8. Una Gramática Dependiente de Contexto (GLDC) es una tupla (V, Σ, P, S) donde:

- V es un alfabeto de símbolos no terminales o símbolos auxiliares
- Σ es un alfabeto de símbolos terminales, que conforman nuestras cadenas
- P es un conjunto de reglas que con la forma $\gamma A\beta \rightarrow \gamma\alpha\beta$ donde $\alpha \in (\Sigma \cup V)^*, \gamma \in (\Sigma \cup V)^*, \beta \in (\Sigma \cup V)^*,$

- $S \in V$ lo denominamos símbolo inicial

7.8.1.1. Ejemplo de GDC

Un ejemplo de Gramática Dependiente del Contexto es:

$(\{S\}, \{a, b\}, P, S)$ donde P:

$$
\begin{array}{llllll}
 & S & \rightarrow & abC & & \\
 & S & \rightarrow & aSBC & & \\
 & C\ B & \rightarrow & W & B & \\
W & B & \rightarrow & W & X & \\
 & W\ X & \rightarrow & B & X & \\
B & X & \rightarrow & B & C & \\
b & B & \rightarrow & b & b & \\
b & C & \rightarrow & b & c & \\
c & C & \rightarrow & c & c &
\end{array}
$$

7.8.2. Derivación

De forma similar que las GLC las GDC siguen un proceso de re-escritura, a un camino de escritura de forma similar se le denomina derivación. A continuación un ejemplo de derivación para la cadena $a^n b^n c^n$.

$$
\begin{aligned}
\underline{S} \Rightarrow\ & a\underline{S}BC \\
\Rightarrow\ & aa\underline{S}BCBC \\
\Rightarrow\ & aaab\underline{CB}CBC \\
\Rightarrow\ & aaab\underline{WB}CBC \\
\Rightarrow\ & aaab\underline{WX}CBC \\
\Rightarrow\ & aaab\underline{BX}CBC \\
\Rightarrow\ & aaabBC\underline{CB}C \\
\Rightarrow\ & aaabBC\underline{WB}C \\
\Rightarrow\ & aaabBC\underline{WX}C \\
\Rightarrow\ & aaabBC\underline{BX}C \\
\Rightarrow\ & aaabB\underline{CB}CC \\
\Rightarrow\ & aaabB\underline{WB}CC \\
\Rightarrow\ & aaabB\underline{WX}CC \\
\Rightarrow\ & aaab\underline{BB}CCC \\
\Rightarrow\ & aaabb\underline{BC}CC \\
\Rightarrow\ & aaabbb\underline{C}CC \\
\Rightarrow\ & aaabbbc\underline{C}C \\
\Rightarrow\ & aaabbbcc\underline{C} \\
\Rightarrow\ & aaabbbccc
\end{aligned}
$$

7.8.3. Lenguajes Dependientes del Contexto

Formalmente un lenguje L generado por una gramática dependiente del contexto G se define como $L = \{w | S \Rightarrow^* w\}$ donde S es el símbolo inicial de la gramática $G = (V, \Sigma, P, S)$. Es decir el lenguaje de asociado a una GDC es el conjunto de todas las cadenas que tienen una derivación partiendo del símbolo inicial de la gramática y a través de aplicar una secuencia finita de re-escrituras siguiendo las reglas de producción.

7.9. Algunas observaciones

Al final de esta sección podemos concentrar lo que sabemos en la siguiente tabla:

Lenguaje	Gramática	Máquina	Ejemplo
Lenguaje Dependientes del Contexto	$\gamma A\beta \rightarrow \gamma\alpha\beta$	??	$a^n b^n c^n$
Lenguaje Independiente del Contexto	$A \rightarrow \alpha$	Autómata de Pila	$a^n b^n$
Lenguaje Regular	$A \rightarrow bC\|d$	AF, AFND y AFND-ε	a^n

Ahora podemos ver porque se denomina a los Lenguajes Independientes del Contexto, como independientes, ya que a diferencia de los dependientes el contexto no está definido (γ y β), o mejor dicho es contexto muy particular donde gamma es epsilon y beta también.

También es importante notar, que hasta este momento si nos fijamos en la tabla los niveles de esta están definidos por la forma de las reglas. La forma de las reglas se vuelven más específicas para los niveles más abajo. Dependientes del contexto define cualquier contexto, independientes del contexto define un sólo contexto, y regulares requiere una forma peculiar de alfa.

8 Revisando la jerarquía de Chomsky

8.1. Gramáticas monotonicas

Definición 9. Una Gramática Monotónica (GLDC) es una tupla (V, Σ, P, S) donde:

- Ves un alfabeto de símbolos no terminales o símbolos auxiliares
- Σ es un alfabeto de símbolos terminales, que conforman nuestras cadenas
- Pes un conjunto de reglas que con la forma $\gamma \to \alpha$ donde $\alpha \in (\Sigma \cup V)^*, \gamma \in (\Sigma \cup V)^*$ y $|\gamma| \leq |\alpha|$ y $S \to \epsilon$
- $S \in V$lo denominamos símbolo inicial

Una GM es equivalente a una GDC, es decir ambos tipos de gramática generan el mismo tipo de lenguajes. Los lenguajes dependientes del contexto.

8.1.1. Ejemplo de GM

Un ejemplo de Gramática Monotónica es:

$(\{S\}, \{a, b\}, P, S)$ donde P:

$$\begin{aligned} S &\to aSBc \\ S &\to abc \\ cB &\to Bc \\ bB &\to bb \end{aligned}$$

8.1.2. Derivación

A continuación un ejemplo de derivación para la cadena $a^n b^n c^n$.

$$\begin{aligned}
\underline{S} &\Rightarrow a\underline{S}Bc \\
&\Rightarrow aa\underline{S}BcBc \\
&\Rightarrow aaabc\underline{B}cBc \\
&\Rightarrow aaabBc\underline{cB}c \\
&\Rightarrow aaabBc\underline{cB}cc \\
&\Rightarrow aaab\underline{bB}Bccc \\
&\Rightarrow aaabb\underline{B}ccc \\
&\Rightarrow aaabbbccc
\end{aligned}$$

Como se puede apreciar las GM son más compactas que las GDC, ya que no tienen que respetar las restricciones del contexto.

8.2. Forma Normal de Chomsky

En alguna ocasiones es posible transformar una Gramática en un otra debilmente equivalente, es decir que genera exactamente el mismo lenguaje pero con una estructura de árbol de derivación diferente. Alguna veces es algo conveniente porque los árboles de derivación de la nueva gramática podrían tener algunas propiedades de las cuales se podría tomar ventaja.

8.2.1. FNC

Toda GLC se puede transformar en una gramática en la Forma Normal de Chomsky que requiere que las reglas tomen la siguiente forma:

$$\begin{aligned}
A &\rightarrow BC \\
A &\rightarrow a \\
A &\rightarrow \varepsilon
\end{aligned}$$

8.2.2. Proceso de transformación

Para transformar una gramática (V, Σ, P, S) en su FNC se siguen los siguientes pasos:

0. Agregar un nuevo símbolo inicial

$$S_0 \rightarrow \quad S$$

1. Quitar las transiciones ε

   - Identificar las producción $A \rightarrow \varepsilon$
   - Producir nuevas producciones suponiendo que dónde aparece A es ε; por ejemplo si $P \rightarrow AxB; A \rightarrow \varepsilon; B \rightarrow \varepsilon$ entonces se producen las nuevas reglas para P:

$$P \rightarrow \quad xB|Ax|x$$

   - Eliminar de la gramática las transiciones ε

2. Quitar las transiciones unitarias

   - Identificar las producción $A \rightarrow B$
   - Por cada $B \rightarrow \alpha$ agregar las reglas:

$$A \rightarrow \quad \alpha$$

   - Eliminar de la gramática las unitarias

3. Quitar las transiciones largas

   - Identificar las producción $A \rightarrow A_0 A_1 A_2 ... A_m$
   - Cortarlas usando símbolos auxiliares para encadenar la regla

$$\begin{aligned} A &\rightarrow \quad A_0 T_1 \\ T_1 &\rightarrow \quad A_1 T_2 \\ T_2 &\rightarrow \quad A_2 T_3 \\ \vdots &\rightarrow \quad \vdots \\ T_{m-1} &\rightarrow \quad A_{m-1} A_m \end{aligned}$$

4. Quitar los terminales binarios

   - Identificar las producción $A \rightarrow aB$
   - Agregar una regla con el no terminal (si es necesario) y remplazar en la regla original la aparición del terminal con ese no terminal

$$\begin{aligned} A &\rightarrow \quad N_a B \\ N_a &\rightarrow \quad a \end{aligned}$$

8.2.3. Ejemplo

Reducir la siguiente gramática en su Forma Normal de Chomsky:

$$\begin{aligned} S &\rightarrow aSA|BSB|D \\ A &\rightarrow C \\ C &\rightarrow a \\ B &\rightarrow b \\ D &\rightarrow \varepsilon \end{aligned}$$

0. Agregar un nuevo símbolo inicial

$R_0 \rightarrow$	S	Nuevo símbolo
$S \rightarrow$	$aSA\|BSB\|D$	
$A \rightarrow$	C	
$C \rightarrow$	a	
$B \rightarrow$	b	
$D \rightarrow$	ε	

1. Quitar las transiciones ε

$R_0 \rightarrow$	$S\|\varepsilon$	Se agregó por $S \rightarrow \varepsilon$
$S \rightarrow$	$aSA\|BSB\|aA\|BB$	Se agrego por $B \rightarrow \varepsilon$
$A \rightarrow$	C	
$C \rightarrow$	a	
$B \rightarrow$	b	

2. Quitar las transiciones unitarias

$R_0 \rightarrow$	S	
$S \rightarrow$	$aSA \vert BSB \vert aA \vert BB$	
$A \rightarrow$	a	Se agrego por unitaria
$C \rightarrow$	a	
$B \rightarrow$	b	

3. Quitar las transiciones largas

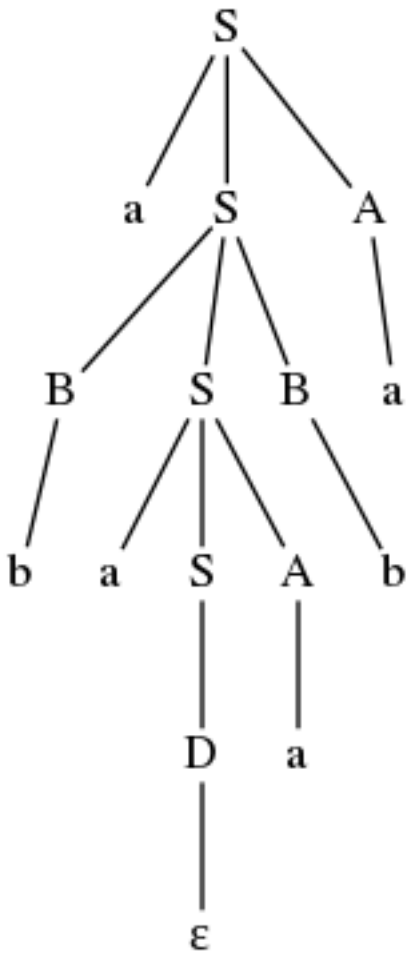
$R_0 \rightarrow$	S	
$S \rightarrow$	$aT_1 \vert BV_1 \vert aA \vert BB$	Se redujeron por regla larga
$T_1 \rightarrow$	SA	Se creo por regla larga
$V_1 \rightarrow$	SB	Se creo por regla larga
$A \rightarrow$	a	
$C \rightarrow$	a	
$B \rightarrow$	b	

4. Quitar los terminales binarios

$R_0 \rightarrow$	S	
$S \rightarrow$	$N_aT_1 \vert BV_1 \vert N_aA \vert BB$	Se modifico por terminales binarios
$T_1 \rightarrow$	SA	
$V_1 \rightarrow$	SB	
$N_a \rightarrow$	a	Se creo por terminales binarios
$A \rightarrow$	a	
$C \rightarrow$	a	
$B \rightarrow$	b	

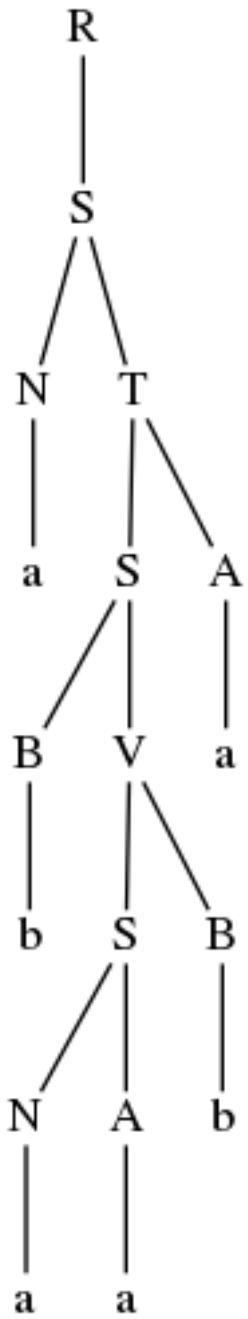
8.2.4. Comparación de árboles de derivación

La gramática original produce el siguiente árbol para la cadena abaaba



{#original, height=250px }

Mientras que la gramática resultante en FNC produce



{#fnc, height=250px }

Como se puede observar las dos cadenas son aceptadas, pero producen diferentes árboles de derivación. Una propiedad importante de la FNC es que cada rama del árbol sólo tiene dos hijos posibles.

8.3. Otras formas normales

8.3.1. FNG

Toda GLC se puede transformar en una gramática en la Forma Normal de Greibach que requiere que las reglas tomen la siguiente forma:

$$\begin{aligned} A \rightarrow &\quad aA_1 \dots A_m \\ S \rightarrow &\quad \varepsilon \end{aligned}$$

8.3.2. FNK

Toda GDC se puede transformar en una gramática en la Forma Normal de Kudura que requiere que las reglas tomen la siguiente forma de gramáticas monotónicas:

$$\begin{aligned} AB \rightarrow &\quad CD \\ A \rightarrow &\quad BC \\ A \rightarrow &\quad B \\ A \rightarrow &\quad a \end{aligned}$$

8.4. Automata Lineal con Frontera

La máquina que puede reconocer lenguajes dependiente del contexto se denomina Autómata Lineal con Frontera (ALF); sin embargo, no se ahondará con ejemplo ya que se optará con el Autómata de Doble Pila (ADP) para reconocer a estos lenguajes. Lo anterior es por una razón didáctica, ya que el funcionamiento de los ALF están más cercanos a la MT que se verá en las siguientes secciones; mientras que ADP resultará más poderoso que un ALF pero su funcionamiento relaciona mejor con AP.

8.4.1. ALF

Definición 10. Un Autómata de Lineal con Frontera (ALF) es una tupla $(Q, \Sigma, \Gamma, q_0, B, A, \delta)$ donde:

- Q es un conjunto de estados finitos
- Σ es un alfabeto de símbolos terminales

- Γ es un alfabeto de la cinta tal que $\Sigma \subset \Gamma$

- q_0 es un estado que denominaremos inicial donde $q_0 \in Q$
- B es un símbolo de espacio en blanco
- A es un conjunto de estados que denominaremos finales donde $A \subset Q$
- δ es una función de transición que cumple con:

{{< katex display>}}δ:Q $\times$($\Gamma\cup${<, >}) $\rightarrow$Q $\times$($\Gamma\cup${<, >}) $\times${left,right} {{< /katex >}}

Donde < y > son marcas sobre la cinta que marcan el final de esta; y left y right son acciones de moverse una celda a la izquierda y a la derecha respectivamente.

8.4.2. Lo que sabemos

Con lo dicho para ALF esto es lo que sabemos:

Lenguaje	Gramática	Máquina	Ejemplo
??	??	Autómata de Doble Pila	??
Lenguaje Dependientes del Contexto	$\gamma A\beta \rightarrow \gamma\alpha\beta$	Autómata Lineal con Frontera	$a^n b^n c^n$
Lenguaje Independiente del Contexto	$A \rightarrow \alpha$	Autómata de Pila	$a^n b^n$
Lenguaje Regular	$A \rightarrow bC\|d$	AF, AFND y AFND-ε	a^n

8.5. Autómata de Doble Pila

Una forma natural de aumentar el poder computacional de nuestra AP es aumentar el número de pilas, en particular podemos agregar una más en el denominado Autómata de Doble Pila

8.5.1. ADP

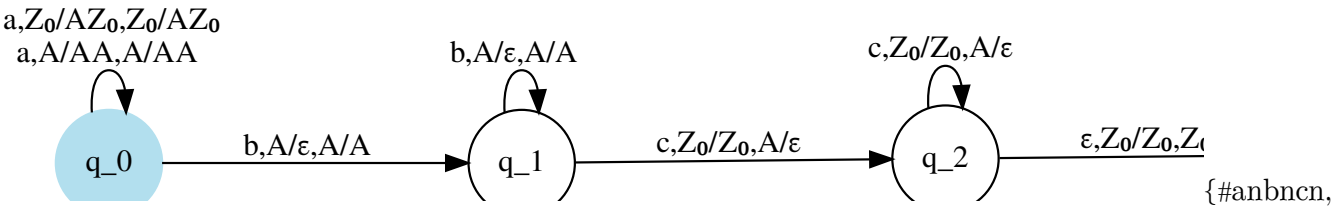
Definición 11. Un Autómata de Doble Pila (ADP) es una tupla $(Q, \Sigma, \Gamma, q_0, Z_0, A, \delta)$ donde:

- Q es un conjunto de estados finitos
- Σ es un alfabeto de símbolos terminales
- Γ es un alfabeto de la pila
- q_0 es un estado que denominaremos inicial donde $q_0 \in Q$
- Z_0 es un símbolo de la pila que denominaremos inicial de la pila donde $Z_0 \in \Gamma$
- A es un conjunto de estados que denominaremos finales donde $A \subset Q$
- δ es una función de transición que cumple con:

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

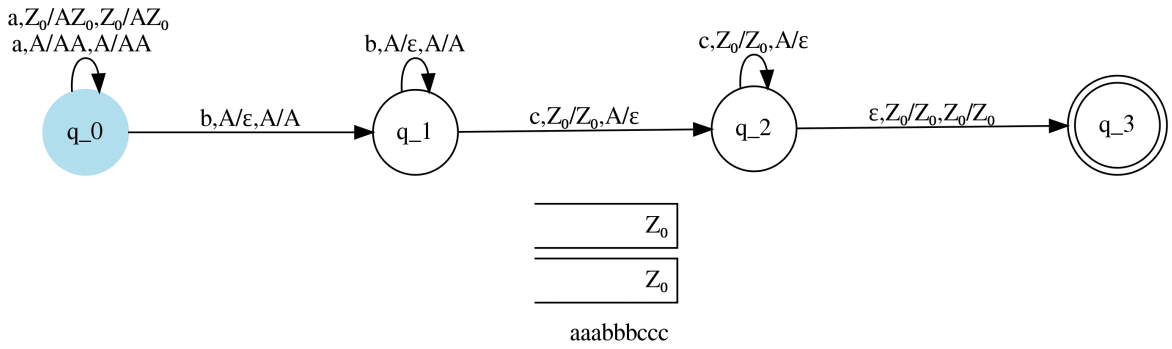
8.5.1.1. Ejemplo de ADP

El siguiente autómata reconoce al lenguaje formado de cadenas $a^n b^n c^n$ para cuando n>0:



{#anbncn, height=250px }

Al igual que con los AF, al comenzar el análisis se comienza en el estado inicial y de ahí se tienen que poner atención a cuatro aspectos, el estado en el que se está, el símbolo en la cadena que está siendo analizado, el símbolo que se encuentra hasta arriba de la primer pila y el símbolo que se encuentra hasta arriba de la segunda pila.



{#anbncn, height=250px }

9 La máquina con cinta

9.1. La máquina de Turing

En este momento presentaremos la Máquina de Turing, esta es una máquina diferente a las que habíamos visto en la forma como trabaja ya que la cadena que acepta está contenida en elemento de memoria. Hasta ahora no habíamos visto nada así, esta situación se puede apreciar en el hecho que el alfabeto de las cadenas terminales están contenidas en el alfabeto de la cinta. También se puede apreciar que es una generalización del ALF ya que el alfabeto de la cinta no especifica la existencia de los marcadores de inicio y fin de cinta. O

9.1.1. MT

Definición 12. Una Máquina de Turing (MT) es una tupla $(Q, \Sigma, \Gamma, q_0, B, A, \delta)$ donde:

- Q es un conjunto de estados finitos
- Σ es un alfabeto de símbolos terminales
- Γ es un alfabeto de la cinta tal que $\Sigma \subset \Gamma$

- q_0 es un estado que denominaremos inicial donde $q_0 \in Q$
- B es un símbolo de espacio en blanco
- A es un conjunto de estados que denominaremos finales donde $A \subset Q$
- δ es una función de transición que cumple con:

$$\delta : Q \times \Gamma \to Q \times \Gamma \times \{left, right\}$$

9.1.2. La cinta

Una situación que genera confusión es la naturaleza de la cinta, ya que pareciera que necesitaríamos una cantidad infinita de esta y que toda máquina de Turing sería imposible de construir físicamente. Sin embargo, a la memoria la debemos considerar como un elemento del cual podemos obtener más si es necesario para la computación específica que se hace, no como un recurso

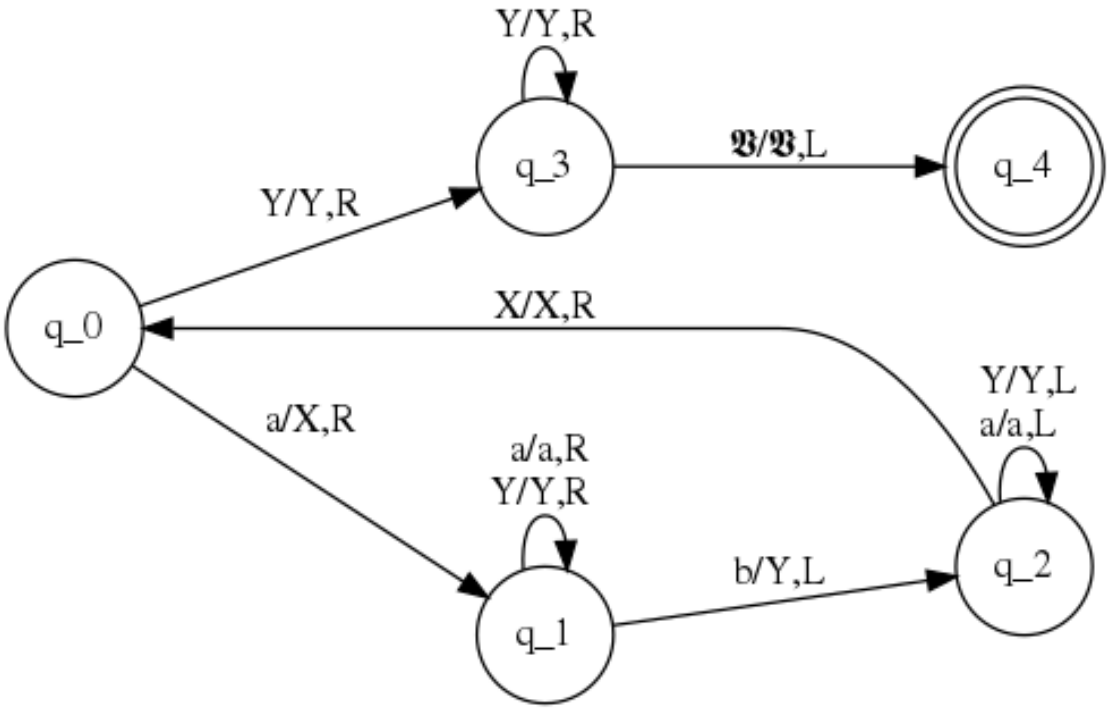
indispensable del diseño. Recordemos que el AP también ya levantaba este dilema, ya que la pila también es un elemento de memoria potencialmente infinita.

9.1.3. Ejemplo de MT

La siguiente tabla de transición especifica una Máquina de Turing

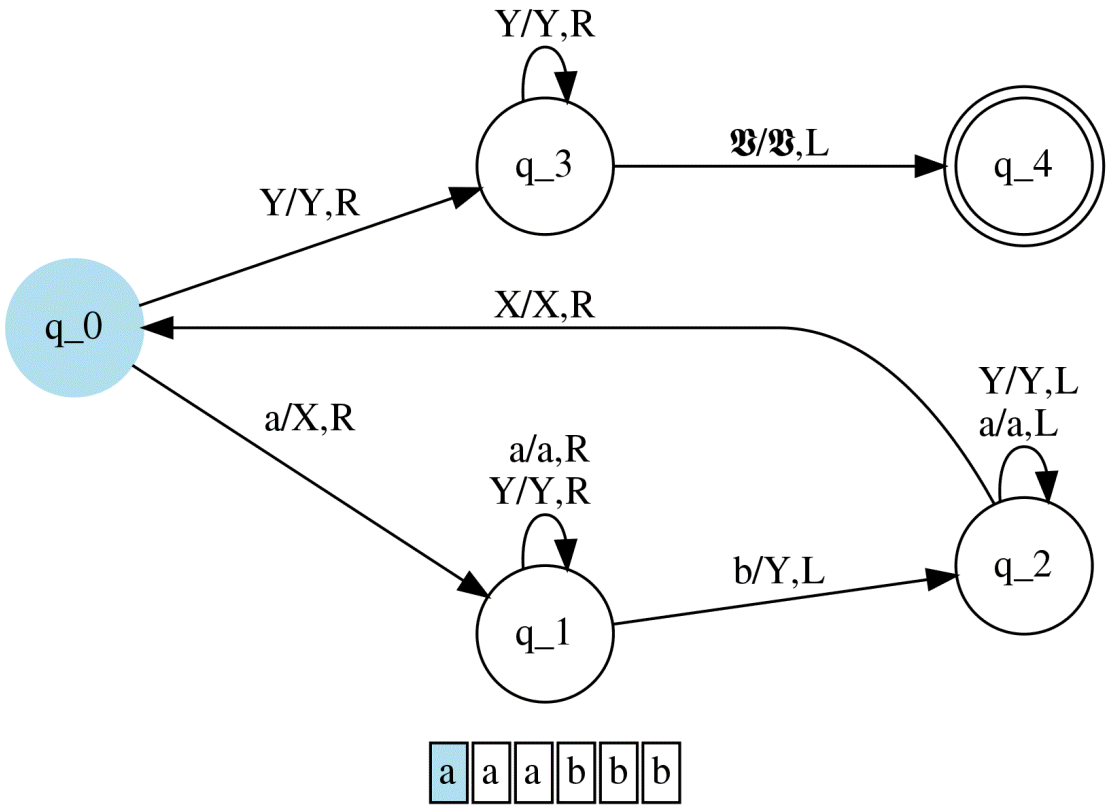
Q	X	Y	a	b	$\mathfrak{B}$
$\longrightarrow$q_0	$\emptyset$	/Y$\rightarrow$q_3,R	/X$\rightarrow$q_1,R	$\emptyset$	$\emptyset$
q_1	$\emptyset$	/Y$\rightarrow$q_1,R	/a$\rightarrow$q_1,R	/Y$\rightarrow$q_2,L	$\emptyset$
q_2	/X$\rightarrow$q_0,R	/Y$\rightarrow$q_2,L	/a$\rightarrow$q_2,L	$\emptyset$	$\emptyset$
q_3	$\emptyset$	/Y$\rightarrow$q_3,R	$\emptyset$	$\emptyset$	/$\mathfrak{B}\rightarrow$q_4,L
q_4	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

De forma gráfica la MT luce de esta forma:



{#anbn, height=250px }

A continuación se puede apreciar como una MT analiza una cadena. En particular, hay que observar que se termina el análisis porque se alcanza el estado final.



{#anbn, height=250px }

9.2. Descripciones instantáneas

9.2.1. Descripción instantánea (DI (DI))

Un aspecto interesante de las MT es que es posible representar la configuración en la que se encuentra la máquina con la cinta, lo único que no está presente en la cinta es el estado q de la MT que acaba de producir ese estado y que parte de la cadena se está analizando. Para lograr esto, se produce una representación de la cinta en dónde la posición que está por ser analizada por la MT se marca con el estado q.

Una descripción instantánea luce de la siguiente forma:

$$X_1 ... qX_i ... X_m$$

En dónde $X_1 ... X_m$ es una cadena compuesta por símbolos del alfabeto de la cinta, y cuyo símbolo X_i es el siguiente a ser procesado por la MT y desde el estado q.

9.2.2. DI para moverse a la derecha

Si una transición de la MT se mueve hacia la derecha como la siguiente genera la siguiente descripción instantánea:

$$\delta(q, X_i) = (p, Y_i, R)$$

$$X_1 ... qX_i ... X_m \vdash X_1 ... Y_i pX_{i+1} ... X_m$$

Solo existe un caso especial cuando se está analizando el último símbolo de la cadena en la cinta.

$$X_1 ... qX_m \vdash X_1 ... X_m qB$$

9.2.3. DI para moverse a la izquierda

Si una transición de la MT se mueve hacia la izquierda como la siguiente genera la siguiente descripción instantánea:

$$\delta(q, X_i) = (p, Y_i, L)$$

$$X_1 ... qX_i ... X_m \vdash X_1 ... pX_{i-1} Y_i ... X_m$$

Solo existe un caso especial cuando se está analizando el primer símbolo de la cadena en la cinta.

$$qX_1 ... X_m \vdash pBY_1 ... X_m$$

9.2.3.1. Ejemplo de descripción instantaneas

La siguiente es un ejemplo de una secuencia de descripción isntantánea para la cadena aaabbb

$$
\begin{aligned}
q_0aaabbb &\vdash Xq_1aabbb \\
&\vdash Xaq_1abbb \\
&\vdash Xaaq_1bbb \\
&\vdash Xaq_2aYbb \\
&\vdash Xq_2aaYbb \\
&\vdash q_2XaaYbb \\
&\vdash Xq_0aaYbb \\
&\vdash XXq_1aYbb \\
&\vdash XXaq_1Ybb \\
&\vdash XXaYq_1bb \\
&\vdash XXaq_2YYb \\
&\vdash XXq_2aYYb \\
&\vdash Xq_2XaYYb \\
&\vdash XXq_0aYYb \\
&\vdash XXXq_1YYb \\
&\vdash XXXYq_1Yb \\
&\vdash XXXYYq_1b \\
&\vdash XXXYq_2YY \\
&\vdash XXXq_2YYY \\
&\vdash XXq_2XYYY \\
&\vdash XXXq_0YYY \\
&\vdash XXXYq_3YY \\
&\vdash XXXYYq_3Y \\
&\vdash XXXYYYq_3B \\
&\vdash XXXYYq_4YB
\end{aligned}
$$

9.3. El lenguaje aceptado por una MT

El lenguaje aceptado por una MT es el siguiente

$$L_{MT} = \{w \in \Sigma^* | q_0 w \vdash^* \alpha q_f \beta, p \in A\}$$

Es decir, el lenguaje aceptado por una MT está compuesto por todas las cadenas w tal que partiendo de una descripción con el estado inicial q_0 y de una posición inicial de la cadena pasan por otra descripción instantánea dónde el estado es uno final q_f.

9.3.1. Diferencias con otras máquinas

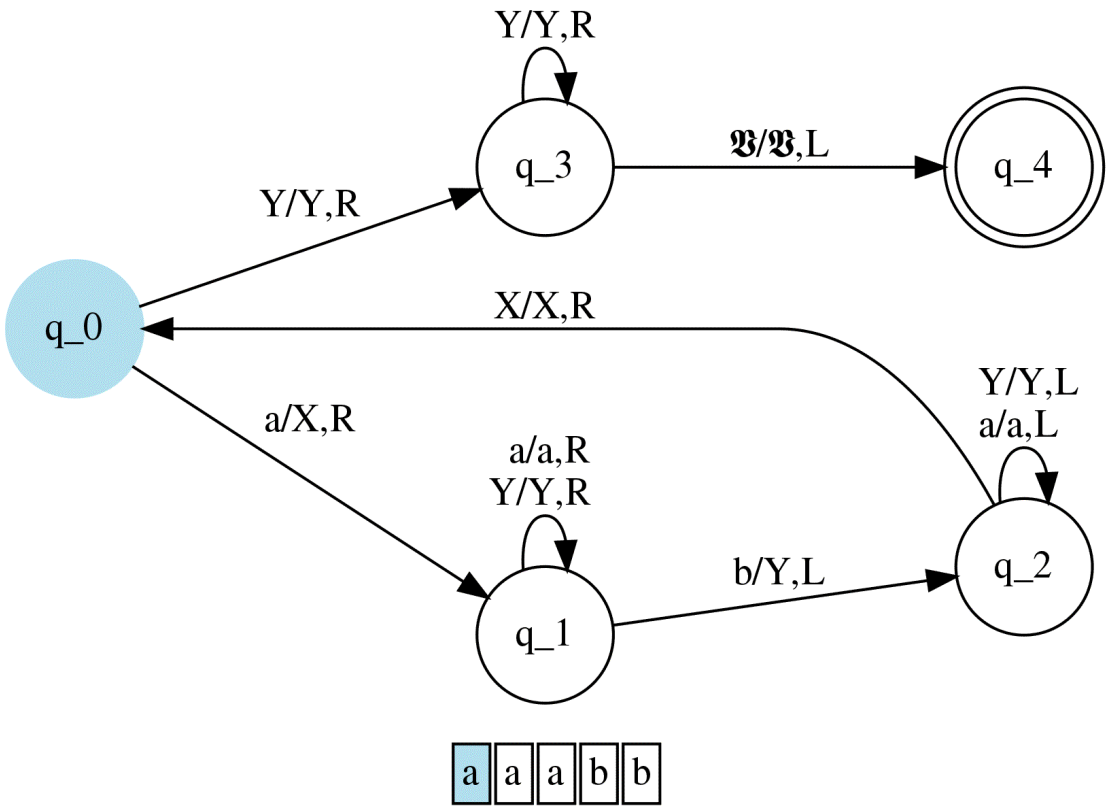
La siguiente tabla resumen las situaciones en que se acepta o rechaza una cadena

Máquina	Acepta	Rechaza
AF	Termina de analizar la cadena y termina en un estado final	Termina de analizar la cadena y termina en un estado no final o en algún momento no encontró una transición
AP	Termina de analizar la cadena y termina en un estado final	Termina de analizar la cadena y termina en un estado no final o en algún momento no encontró una transición
ALF	??	??
MT	Pasa por un estado final	En algún momento no encontró una transición

Como podemos ver en el caso de la MT no es tan claro como en las otras máquinas, ya que la cadena la trabajamos directamente en el elemento de memoria (cinta), entonces no tenemos control sobre si ya terminamos o no de procesar la cadena, entonces la única pista para saber si se acepta la cadena es si eventualmente pasamos por un estado aceptor (final).

9.3.1.1. Ejemplo de cadena rechazada en MT

El siguiente es un ejemplo de una cadena no aceptada, ya que no hubo una transición esperada para nuestro ejemplo de MT visto aquí.



{#anbn_reject, height=250px }

9.4. Relación con otras máquinas

Las MT son equivalentes a varias otras máquinas y a otros formalismos.

9.4.1. MT y ADP

Un Autóma de Doble Pila es equivalente a un MT, ya que un ADP puede simular a una MT y una MT puede simular a un ADP.

- ADP → MT Un ADP puede simular la cinta con sus dos pilas, la idea principal es simular la posición en la cinta de MT con respecto a las dos pilas, lo que está a la derecha del cabezal de la MT en una pila y lo que esté a la izquierda en otra pila. Inicialmente una cadena w a ser analizada por un APD está localizada en un lugar diferente que a los elementos de la memoria (dos pilas), así que lo primero que tendría que hacer este simulador es poner en

la pila derecha toda la cadena, a partir de ahí se usa no-determinismo para programar los movimientos de la MT.

- MT $\rightarrow$ ADP Para simular un ADP en una MT, basta con definir un lado de la cinta como una pila, y el otro lado de la cinta para la segunda pila. Ojo, hay que considerar que en la cinta contendrá la cadena w, por lo que estas pilas simuladas tendrán que evitar tocar la cadena.

Dado que estas simulaciones son posibles podemos considerar que ambas máquinas son equivalentes. Todo lo que concluyamos para MT es cierto para ADP.

9.4.2. MT Múltiples cintas

Cuando estábamos presentando AP fue posible aumentar el poder de la máquina agregando una nueva pila, pero también cabe recordar que agregar otra tercera pila no aumenta el poder de la de dos pilas. Una pregunta que surge, qué pasa si aumento una nueva cinta al diseño de la MT. A esta máquina se le conoce como una Máquina de Turing con k cintas, dependiendo de cuantas se agrega. Sin embargo una MT son equivalentes MT_k.

Si pensamos por un segundo suena lógico del hecho que tres pilas son tan poderosas como dos, y como vimos era posible simular en una cinta dos pilas; si agregamos una cinta, es como si agregáramos dos pilas a lo que es equivalente a un ADP, por lo tanto tendríamos cuatro pilas que sabemos que son tan poderosas como dos pilas.

Por lo tanto todo lo que concluyamos para MT es cierto para MT_k.

9.4.3. MT semi-finita

¿Qué pasa si en lugar de aumentar cinta, disminuimos la cinta, cortándola a la mitad? por ejemplo, dejando sólo el lado derecho. Resulta en una MT semifinita que también es equivalente a la MT, esta máquina tendrá que tener cuidado con ese límite y tendrá que programarse para siempre poner datos a la derecha.

Por lo tanto todo lo que concluyamos para MT es cierto para MT semi-finíta.

9.4.4. Una computadora

¿Qué hay de computadoras físicas? También son equivalentes. Al principio del curso se hizo un esfuerzo por reducir su complejidad para que fueran una caja negra. Sin embargo, podríamos considerar que una computadora es una conjunto de cintas independientes, tanto para dispositivos

de entrada como de salida, y la MT tendría que controlar el comportamiento de todas estas cintas. Por lo tanto, una computadora es equivalente a una MT.

Por lo tanto todo lo que concluyamos para MT es cierto para nuestras computadoras.

9.4.5. Gramáticas de Frase

Definición 13. Una Gramática de Frase es una tupla (V, Σ, P, S) donde:

- Ves un alfabeto de símbolos no terminales o símbolos auxiliares
- Σ es un alfabeto de símbolos terminales, que conforman nuestras cadenas
- Pes un conjunto de reglas que con la forma $\alpha \to \beta$ donde $\alpha \in (\Sigma \cup V)^*$, $\beta \in (\Sigma \cup V)^*$,
- $S \in V$lo denominamos símbolo inicial

Una gramática de frase es equivalente a una MT. Como vemos la gramática de frase representa el proceso de re-escritura con menos restricciones posibles en sus reglas, ya que las reglas pueden re-escribir cualquier patrón a cualquier patrón, sin importar el contexto, la longitud o si es no terminal.

Por lo tanto todo lo que concluyamos para MT es cierto para las gramáticas de frase.

9.4.6. Otras equivalencias

Adicionalmente se ha probado que las MT son equivalentes a los siguientes formalismos

- Cálculo lambda
- Funciones recursivas
- Algoritmos de Markov
- Autómata de cola

Por lo tanto todo lo que concluyamos para MT es cierto para cálculo lambda, funciones recursivas, algoritmos de Markov y Autómata de cola.

9.5. Tipo de lenguaje aceptado

Hasta este momento hemos visto el formalismo de las MT, sus equivalencias y el lenguaje que acepta una MT esecífica. Sin embargo, no hemos visto el tipo de lenguaje que acepta, al tipo de lenguaje que acepta una MT se le conoce como recursivo. Por el momento lo que sabemos es

este tipo de lenguajes contiene a los lenguajes sensibles dependientes del contexto, pero contiene lenguajes que no lo son.

Lenguaje	Gramática	Máquina	Ejemplo
Recursivos	$\alpha \rightarrow \beta$	MT, ADP, MT_k, ...	??
Lenguaje Dependientes del Contexto	$\gamma A \beta \rightarrow \gamma \alpha \beta$	Autómata Lineal con Frontera	$a^n b^n c^n$
Lenguaje Independiente del Contexto	$A \rightarrow \alpha$	Autómata de Pila	$a^n b^n$
Lenguaje Regular	$A \rightarrow bC \vert d$	AF, AFND y AFND-ε	a^n

9.6. Reflexión sobre el complemento

Si recordamos de nuestro conceptualización de máquina computacional tal y como se como se conceptualizó el lenguaje de cadenas aceptadas tiene un complemento que son las cadenas rechazadas. En particular por cada w se puede decidir a través del análisis de la cadena si esta se acepta o no, y por lo tanto para todo el lenguaje aceptado y para todo el complemento de cadenas rechazadas.

{#../02la-máquinasinmemoria/cajanegra2, height=250px }

Si revisamos los casos por tipo de máquina:

- Autómatas finitos para las tres variantes de autómatas que vimos sabemos los lenguajes regulares son decidibles es decir para cualquier lenguaje regular un AF va a decir si sus cadenas pertenecen o no.

- Automatas de pila también sabemos que los lenguajes libres de contexto son decidibles, el AP va a determinar si las cadenas de estos lenguajes pertenecen o no.

9.6.1. ¿Por qué decidibilidad es importante?

El determinar que una máquina es decidible y por lo tanto el tipo de lenguaje también es importante porque nos garantiza que no importa lo grande o complejo que el cálculo que se hace sea, la máquina eventualmente nos dirá si la cadena siendo analizada es aceptada o no, entonces si un análisis está tardando mucho solo es cuestión de esperar a que acabe.

9.6.2. El caso de ALF

Para los Autómatas Lineales con Frontera no hemos visto como funcionan, sin embargo se puede observar que son similares a las MT con una cinta limitada por los símbolos $<$ y $>$. Esto pudiera ser similar a nuestra experiencia con nuestras computadoras que tienen un espacio limitado en RAM o disco duro. Sin embargo, la definición es más general los límites de la cinta en ALF se definen en término del tamaño de la entrada, generalmente proporcional. En algunos casos los límites se encontrarán alrededor de w, y en otros podrá haber un espacio mayor, esto es dependiendo de la configuración que se determine. Formalmente podemos escribir

$$|\langle ... \rangle| = kw + 2$$

Es decir la longitud en la cinta entre las marcas de límites es proporcional a la cadena w más dos posiciones por la marca.

9.6.3. El problema con MT y ALF

El problema con la definición de MT es que para determinar que una cadena pertenece al lenguaje la única opción que tenemos es que eventualmente se pase por un estado final, esto es diferente al caso de Autómatas Finitos y Autómatas de Pila, ya que al visitar la cadena de izquierda a derecha es suficiente para determinar si la cadena se acepta o se rechaza (dependiendo al estado al que se llegue), es decir es fácil determinar que son máquinas decidibles; para MT no es tan sencillo porque internamente la MT podría estar haciendo cálculos en la cinta y no necesariamente progresando para resolver la pertenencia de ésta en el lenguaje, es decir esos cálculo nunca la llevarán a pasar por un estado final.

La pregunta para MT y ALF, se transforma en si hay un mecanismo para determinar para una cadena w para la cual la máquina lleva mucho tiempo analizándola que no pertenece al lenguaje

ya que nunca pasarán por un estado final y por lo tanto deberían ser rechazadas. Por supuesto, se antoja a revisar el comportamiento de la máquina y basado en esta determinar si efectivamente nunca pasará por un estado final. Esto se puede hacer por caso, pero de forma general veremos que sólo es posible para máquinas tipo ALF, y MT no.

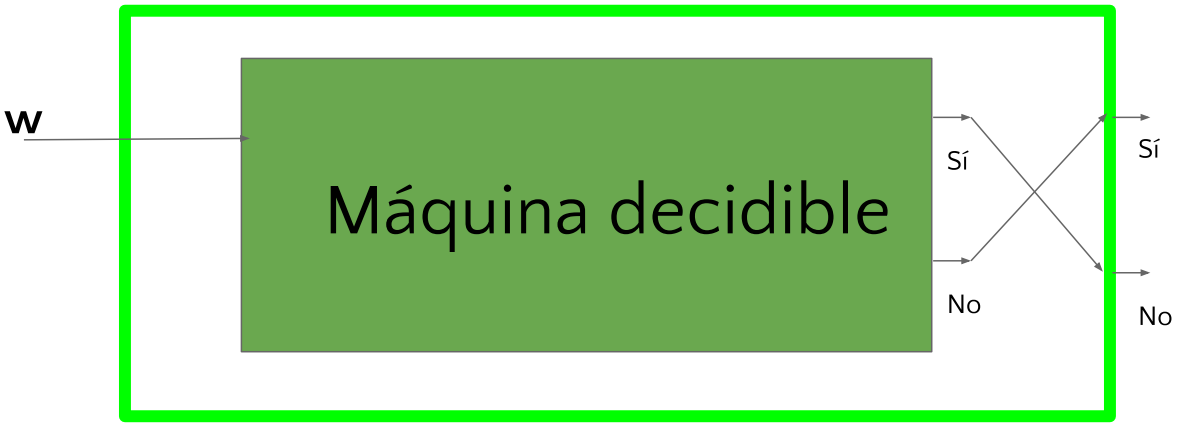
En el caso de ALF podemos agregar un contador de configuraciones de la cinta, por cada transición acumulamos el contador ¿De qué nos sirve esto? Como la cinta está limitada, sabemos que sólo puede guardar un número de configuraciones finitas. Si revasamos ese número y no hemos aceptado a la cadena, quiere decir que no la aceptaremos porque si hubiera sido aceptada ya hubieramos visto un estado final, pero como no lo vimos los siguiente análisis sólo podrán repetir las configuraciones conducentes a un estado no final y por lo tanto nunca aparecerá, y por lo tanto la cadena no forma parte del lenguaje.

9.6.4. Conclusión sobre el caso de ALF

Toda esta argumentación, nos permite establecer que los ALF son decidibles. Es decir que para todas las cadenas del tipo de lenguaje que aceptan eventualmente la máquinas nos dirá que la cadena pertenece, y para las que no pertenecen al complemento.

9.7. El complemento de un lenguaje decidible es decidile

Es fácil ver que una máquina decidible produce un lenguaje decidible, y su complemento también será decidible, solo tenemos que invertir la respuesta de la máquina (que ya es decidible) y tenemos una máquina que acepta al complemento.



{#Decidible, height=250px }

10 Máquinas que comen otras máquinas

10.1. Codificación de una máquina de Turing

Hasta ahora hemos visto como crear máquinas de Turing siguiendo los lineamientos de la definición propuesta aquí, es decir definimos los elementos de la tupla y establecemos la función de transición, también hemos visto que ese objeto abstracto lo podemos representar de forma gráfica o en forma de una tabla siguiendo las convenciones que hemos establecido. Sin embargo, no son las únicas formas de representar a una MT, en particular vamos a mostrar que la función de transición de una máquina de Turing también es posible representarla como una cadena del alfabeto $\Sigma = \{0, 1\}$.

10.1.1. Elementos como números enteros

Lo primero que tenemos que hacer es asignar un entero a cada elemento de la Máquina de Turing. Tenemos tres elementos: los estados Q, los símbolos de Γ y la dirección en la que se mueve la cinta.

$$\begin{aligned} M_Q(q) &\to \mathbb{N}^+ \\ M_\Gamma(a) &\to \mathbb{N}^+ \\ M_{\{left,right\}}(d) &\to \mathbb{N}^+ \end{aligned}$$

Estos tres mapeos tienen que ser inyectivos, es decir para cada elementos de dominio se le asignará sólo un número único (codiminio), el número a asignar es a partir de uno. Por ejemplo, los siguientes son mapeos posibles para la máquina ejemplo

$$\begin{aligned} M_Q(q) &= \{q_0 : 1, q_1 : 2, q_2 : 3, q_3 : 4, q_4 : 5\} \\ M_\Gamma(a) &= \{\mathfrak{B} : 1, a : 2, b : 3, X : 4, Y : 5\} \\ M_{\{left,right\}}(d) &= \{left : 1, right : 2\} \end{aligned}$$

10.1.2. Codificación de transiciones

Con esto en mente es posible codificar una transición de MT:

$$\delta(q_i, X_j) = (q_k, X_l, D_m)$$

siguiendo la siguiente especificiación:

$$0^{M_q(q_i)}10^{M_\Gamma(X_j)}10^{M_q(q_k)}10^{M_\Gamma(X_l)}10^{M_{\{left,right\}}(D_m)}$$

Es decir los elementos de la transición se codifican en una cadena donde cada elemento está dividido por un uno y la cantidad de ceros indica el indice dentro del mapeo correspondiente al elemento original. Si los indices de los elementos de los alfabetos y la dirección los asociamos directamente a un mapeo podemos escribir la codificación como:

$$0^i10^j10^k10^l0^m$$

Por ejemplo, para la transición $\delta(q_0, a) = (q_1, X, R)$ de la máquina ejemplo y el mapeo presentado anteriormente su codificación es:

$$010010010000100$$

10.1.3. Codificación de la función de transición

Ya que podemos codificar una transición, podemos codificar como una cadena toda la función de transición, cada transición se separa por dos 11, de tal forma que la tabla de transición para la máquina ejemplo y el mapeo presentado anteriormente:

0100100100001001100101001010011001000001001000000100110010001000001011000101000101011000100000

A la codificiación resultante se le conoce como descripción estándar de una MT.

10.1.4. Reflexión

Debe ser intuitivo que la codificación anterior se puede hacer para cualquier MT. Con esto en mente hemos podido crear una cadena que representa el comportamiento de dicha MT. Como cadena, nada evita que la pongamos en la cita de otra MT.

10.2. Máquina de Turing Universal

Una MT M puede codificarse en cadena, y nada evita que esa cadena sea analizada por otra MT M', por ejemplo que determine cuantos estados hay la máquina M.

10.2.1. Máquina de Turing Universal

Un caso interesante de MT que consume una cadena descripción estándar de una MT y simula su ejecución. En 1937, Alan Turing en su artículo "On Computable Numbers, with an Application to the Entscheidungsproblem" escribió:

> It is possible to invent a single machine which can be used to compute any computable sequence. If this machine U is supplied with a tape on the beginning of which is written the S.D ["standard description" of an action table] of some computing machine M, then U will compute the same sequence as M.

Que se puede traducir a:

> Es posible inventar una máquina que pueda ser usada para computar cualquier secuencia computable. Si esta máquina U se le provee con una cinta en la que al principio se le escribe la descripción estándar de una tabla de acción de alguna máquina M, entonces U computará la misma secuencia que M.

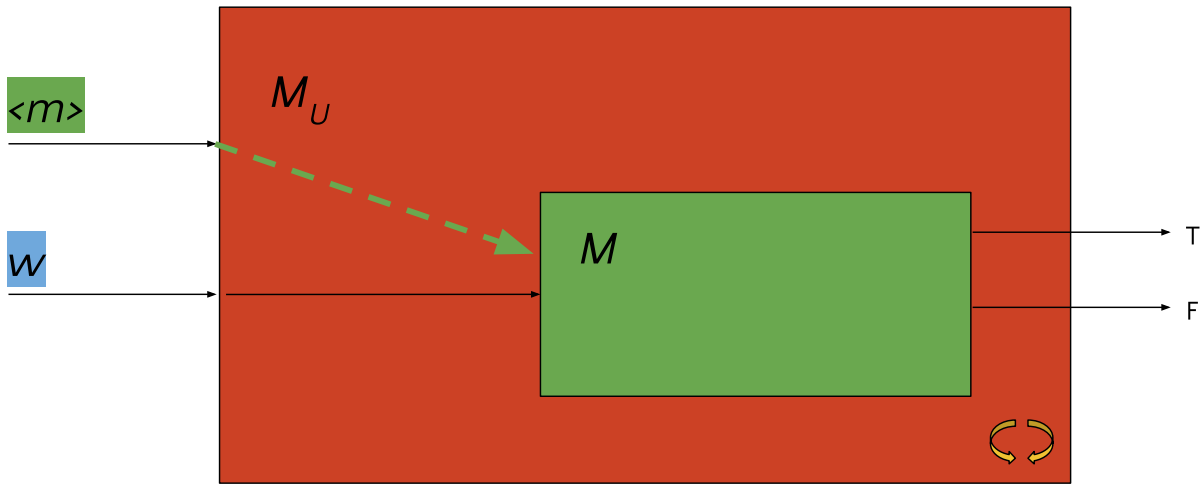
A esta máquina U se le denomina Máquina de Turing Universal (M_U).

10.2.2. El lenguaje aceptado por MTU

La M_U recibe como entrada una cadena que representa a un MT M seguida de una cadena w. El lenguaje aceptado por la M_U se define de la siguiente forma:

$$A_{MT} = \{\langle \mathrm{m} \rangle w | w \in L(M)\}$$

Es decir el A_{MT} es el lenguaje conformado por cadenas con dos partes, la primera define la función de transición de una MT y la segunda es una cadena aceptada por esa MT.



{#mtu, height=250px }

10.3. Máquinas para máquinas

10.3.1. Problemas para las MT con MTs

La codificación de funciones de MT hace posible que una MT reciba como parte de su entrada otra MT ¿Por qué querríamos eso?

- Simular una M
- Procesan la codificación de una M
- Analizan el comportamiento de la máquina codificada M

El el caso de la simulación sabemos que se puede, la MTU no es tan sencilla de construir pero es posible. Por otro lado para procesar aspectos de la codificación de la MT hay cosas interesantes que se pueden hacer:

- Verificar el formato de la cadena
- Contar el número de estados
- Verificar si la cadena tiene un loop hacia un mismo estado de longitud de un símbolo
- Étcetera

Para el caso del comportamiento vamos a esperar un poco para poder sacar conclusiones.

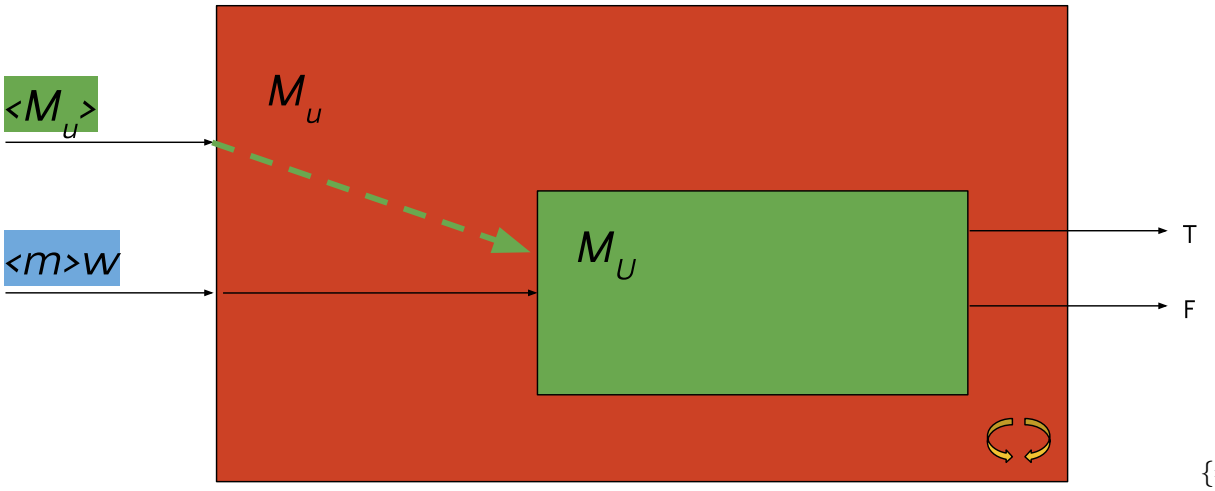
10.3.2. El truco de presentarse a si misma

Dado como hemos definido las MT que aceptan otras MT, nada evita que una MT M que recibe en su entrada una codificación de una MT que esa entrada sea la codificación de si misma; esta

situación va a levantar aspectos interesantes.

10.3.2.1. Para máquinas universales

Nada evita que en la MTU la máquina a simular sea la misma MTU. La entrada w en este caso tendría dos partes, una que sería una MT M y una cadena que se probará en esa máquina.



{#mtum-tu, height=250px }

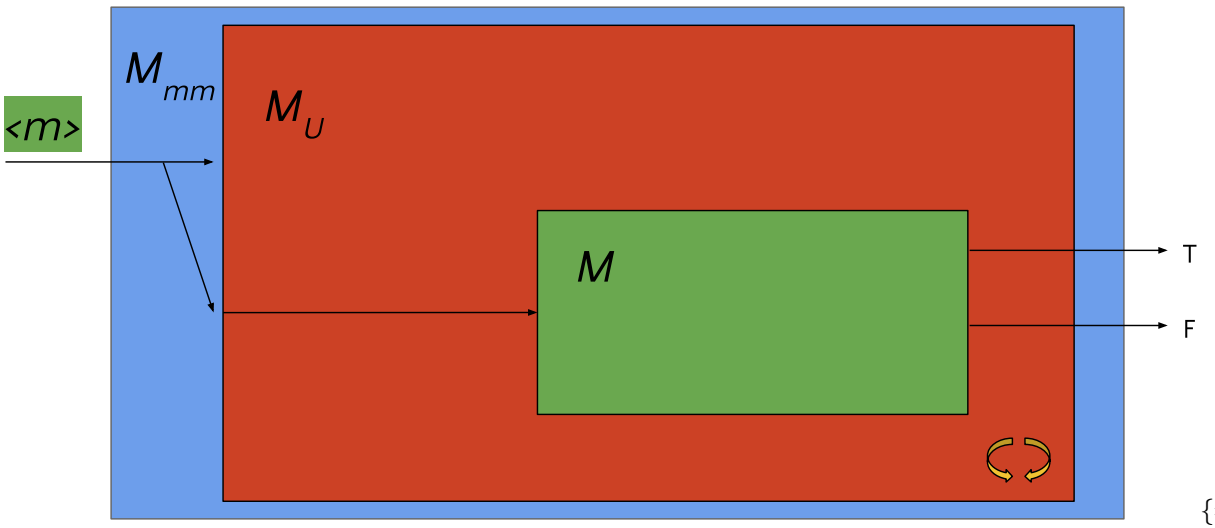
Este escenario se parece a la situación cuando una máquina virtual en un sistema operativo, corre el mismo sistema operativo. El escenario es factible y nada previene que no sea posible. Al contrario, la teoría predice que es posible hacer esa simulación.

10.3.2.2. Para máquinas que procesan la codificación

El mismo caso se puede hacer para aquellas máquinas que identifican propiedades basadas en las cadenas, por ejemplo: si una máquina M#q cuenta el número de estados presentes en la máquina que recibe como entrada, entonce si alimento a esa máquina con su propia codificación <M#q> lo que arrojará la máquina es el número de estados que utiliza la máquina. Como vemos este escenario es totalmente factible y hace sentido, de nuevo la teoría predice que es posible.

10.3.3. El lenguaje de las máquinas que se aceptan a si mismas

Usando MU es posible construir una máquina Mmm_que acepte aquellas descripciones de máquinas que tienen la capacidad de aceptarse a si mismas. Como esta existe esta MT, quiere decir que existe un lenguaje asociado Lmm. Este lenguaje está conformado por codificaciones de máquinas que se aceptan a si mismas. Este lenguaje se refiere a máquinas que cumplen con cierto comportamiento.



{#mm, height=250px }

Si nos damos cuenta la Mmm usa la MU para identificar aquellas máquinas que se acepan a si mismas, al copiar la descripción tanto como maquina a simular como entrada a esa máquina a simular.

10.4. El problema del paro

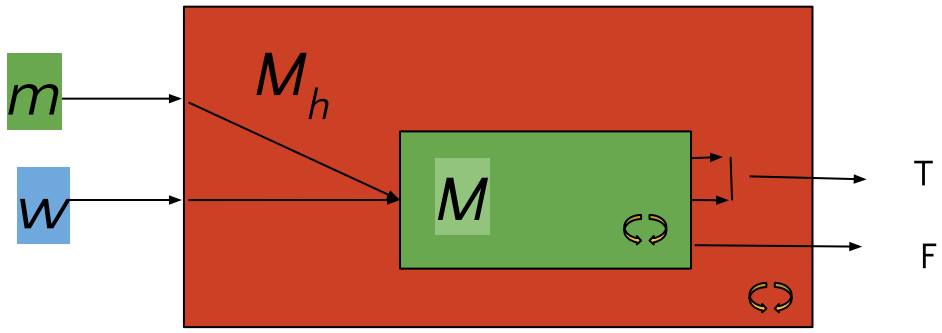
Vimos que la máquina Mmm puede construirse basándose en el comportamiento de la MTU y podemos visualizar algún aspecto del comportamiento de la máquinas de Turing, en este caso se puden separar en dos grupos todas la MT, las que se aceptan a si mismas y las que no. Por supuesto este comportamiento no es tan relevante, y no hay muchas más formas en que podamos alambrar la MU para saber propiedades basadas en ellas.

10.4.1. El problema del paro

Una propiedad que sería interesante averiguar para las MT es si dada una descripción de una MT M y una cadena w la máquina M en algún momento para su ejecución y determina si acepta o no a la cadena. A este problema se le conoce como el problema del paro, y oor supuesto queremos construir una máquina Mh que averigüe específicamente este comportamiento de otras MTi (h por halt que en inglés significa paro).

Para el problema del paro no podemos usar directamente MTU porque si se queda la máquina representada m no va a parar, entonces MTU tampoco lo hará y no podremos saber si esta máquina sigue trabajando porque necesita más tiempo o porque está trabada, por eso la urgencia de diseñar a Mh.

Usando nuestro esquema de cajas negras la máquina luciría de esta forma:

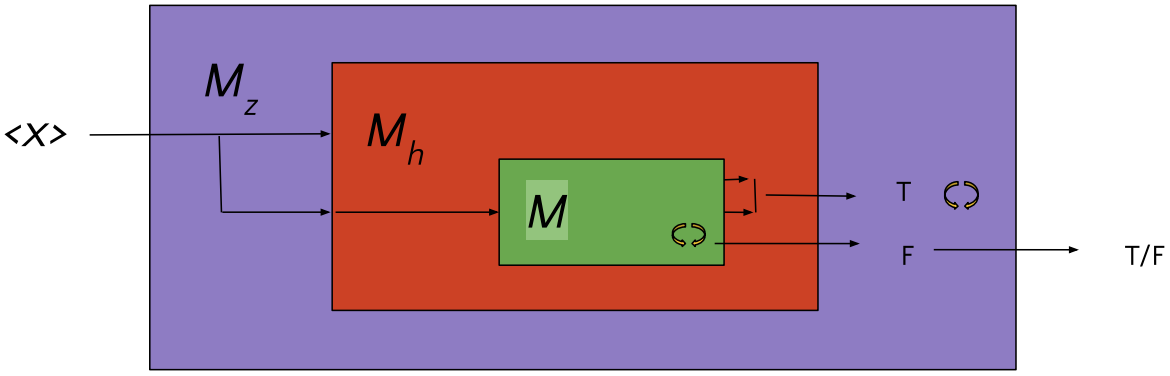


{#mh, height=250px
}

10.4.2. Preguntas sobre Mh

Supongamos que la MT que determina si otra MT para para una w especifica existe, entonce la siguiente pregunta es cómo luce dicha máquina. Un aspecto importante es que nos gustaría que dicha máquina fuera decidible, es decir para toda combinación m y w la máquina pudiera decir si para o no para, este es tipo de máquina que sería útil para determinar la propiedad de paro de todas la MT.

Antes de ponernos a diseñarla, tenemos que ver algunas consecuencias de que Mh sea decidible. Una consecuencia es que si Mh es decidirle es posible construir una máquina Z basada en Mh que es evidentemente no decidible de la siguiente forma:



{#z,
height=250px }

Como se puede apreciar Z sólo utiliza lógica para reacomodar las entradas en Mh y dada la salida de la máquina decidible, decide ciclarse o no no, muy parecido a como definimos la máquina Mmm que identifica máquinas que se aceptan a si mismas.

La secuencia lógica hasta este punto es si Mh es decidible entonces Z existe.

10.4.3. El problema con *Z*

El problema con Z es determinar si es una máquina que se acepta a si misma o no, es decir si pertenece a Lmm o no. Pareciera trivial, pero habíamos establecido que este lenguaje existe y sólo existen dos opciones.

	Z pertenece a Lmm	Z no pertenece a Lmm
Comportamiento de Z	se cicla consigo mismo	para consigo mismo
Decisión de Mh	Falso	Verdadero
Decisión de MZ	Verdadero	Se cicla
Interpretación	Paro consigo mismo	Se cicló consigo mismo

En la tabla podemos observar que la condición de entrada es diferente a la de salida. Cuando suponemos que MZ se cicla consigo mismo resulta que para, y cuando suponemos que se para se cicla. Estamos ante una paradoja.

10.4.4. Dónde está el error

Hay dos puntos de error:

- Tratar de asignar <Mz> como parte del lenguaje Lmm
- Definir Mh como decidible

En el primer caso no hay nada contradictorio, la decisión de si acepta o no a si misma depende de la máquina Z no depende de ningún aspecto, entonces si Z existe debería estar o no. El hecho de que no podamos determinar apunta un problema más intrinsico con Z. En este caso suponerlo decidible.

10.5. Consecuencia de ser no decidible

Sabemos que Mh es no decidible, porque de serlo Z tendría que existir y no podríamos saber si se acepta o no. Mh es nuestra primera máquina que no es decidible, es decir que no puede determinar para todas sus entradas si las acepta o rechaza. Si este es el caso, entonces que hace una que intente resolver el problema, lo que pasará es que además de la opción de parar existe la opción de que se cicle, es decir que para cierta entrada nunca de respuesta.

Pero por otro lado sabemos que existen MT que dada cualquier entrada van a parar, es decir que son decidibles. Esto significa que habrá dos tipos de lenguajes aceptados por las MT, los decidibles y los no decidibles.

10.5.1. Revisando la MTU

Si una MTU simula a una MT, que pasa si la MT que revisa es no decidible, entonces la ejecución de la MT será no decidible. Como el comportamiento de la MTU depende de la máquina que simula, la MTU es no decidible también.

10.5.2. Dentro de malas noticias un poco de buenas noticias

Mh no es imposible de construir, podemos usar una MTU para simularla, lo único que tenemos que hacer pasar la entrada de Mh una MU y esperar si para. Por supuesto, esta estrategia hace al problema de paro no decidible, es decir para algunas MT se podrá decir y para otras no, ya que están cicladas.

Por otro lado hay que notar que dado el funcionamiento de MT si la cadena de entrada se va aceptar, eventualmente la MT parará no importa lo complicado del cómputo, eventualmente parará. Sin embargo si la entrada se debe rechazar, es cuando el caso de ciclado se puede dar. El problema práctico que para una entrada w muchas veces no sabemos si debe ser aceptada o rechazada, y además no sabemos diferenciar si mientras está siendo procesada por nuestra MT se trata de un caso de que requiere más tiempo para procesar o simplemente ya se cicló. Por supuesto se antoja construir un mecanismo que decida durante ejecución si la máquina ya está ciclada, pero es exactamente lo que demostramos ese mecanismo también se podría quedar ciclado haciendo todo el proceso no decidible.

10.5.3. Lenguajes Recursivos

Los Lenguajes Recusivos (R) son aquellos asociados a una MT decidible.

10.5.4. Lenguajes Recursivamente Enumerables

Los Lenguajes Recursivamente Enumerables (RE) son aquellos para los cuales se puede construir una MT que podrá verificar una cadena aceptada, pero no podrá verificar una cadena rechazada ya que podrá ciclarse. Cabe aclarar que los Lenguajes Recusivamente Eneumerables contienen a los recursivos.

10.5.5. La Jerarquía de Chomsky

Lenguaje	Gramática	Máquina	Ejemplo
RE/R	$\alpha \rightarrow \beta$	MT,ADP,MT_k,...	AMT, Lh/??
Lenguaje Dependientes del Contexto	$\gamma A \beta \rightarrow \gamma \alpha \beta$	Autómata Lineal con Frontera	$a^n b^n c^n$
Lenguaje Independiente del Contexto	$A \rightarrow \alpha$	Autómata de Pila	$a^n b^n$
Lenguaje Regular	$A \rightarrow bC \vert d$	AF, AFND y AFND-ε	a^n

10.6. El complemento de los no decidibles

Anteriormente habíamos establecido que si un lenguaje era decidible entonces su complemento también lo era. Sin embargo, con el descubrimiento le lenguajes y máquinas no decidibles, surge la duda de cómo lucen sus complementos.

10.6.1. Fuera de lo computable

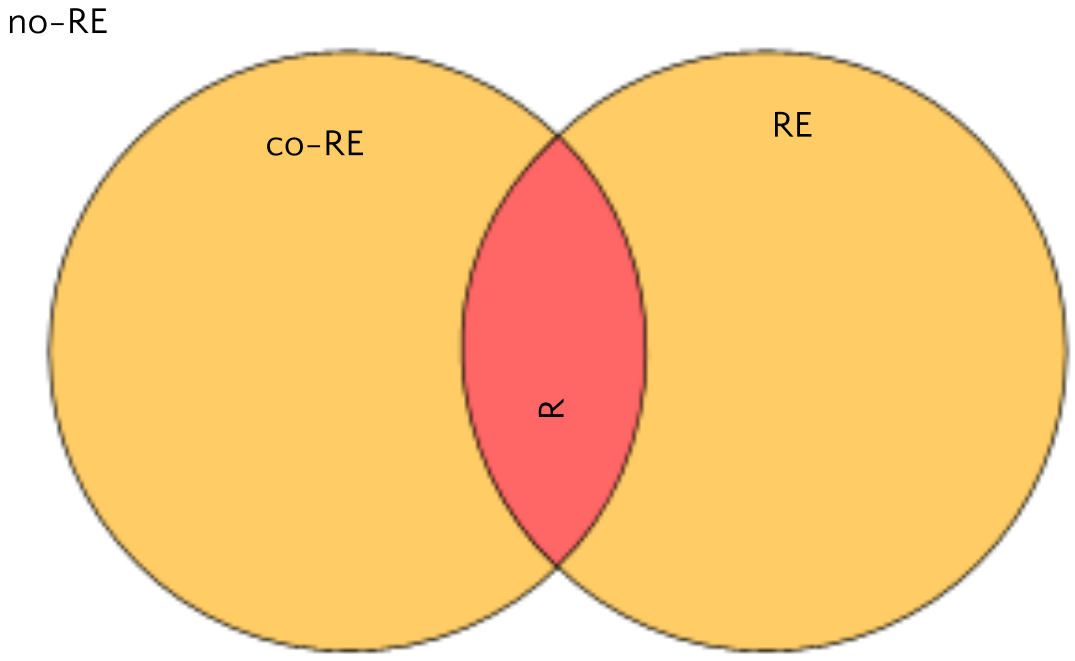
El complemento de un lenguaje no decidible debe tener una naturaleza diferente hasta lo que hemos visto ahora. Un lenguaje no decidible tendrá una MT que lo acepte sin embargo no podemos garantizar que la máquina termine.

Como establecimos antes. Una MT eventualmente va a poder parar si la cadena que analiza pertenece al lenguaje, quiere decir para todas las cadenas del lenguaje las MT eventualmente pararán aceptando a las cadenas. Para las cadenas no aceptadas, algunas veces terminará o algunas veces se ciclará. En el complemento del lenguaje ocurrirá lo contrario, para todas las cadenas que no pertenecen al lenguaje la MT parará (ya que son las cadenas que en el complemento la MT aceptó y paró) y para las cadenas que debe aceptar se quedará ciclada o determinará aceptar.

Con esto en mente se determina que el complemento de los lenguajes RE conforman a los lenguajes co-RE, es decir aquellos lenguajes para los cuales no podemos determinar para una cadena su pertenencia al lenguaje, pero si su no pertenencia.

10.6.2. Visualizando

La relación entre estos tipos de lenguajes se puede apreciar en la siguiente figura.



{#viztlang, height=250px }

Vemos que en el caso de los lenguajes Recursivos (R)/decidibles se trata de la intersección entre los lenguajes RE y co-RE.

10.6.3. ¿Qué hay afuera de RE y co-RE?

Resulta que hay problemas que no se pueden verificar la pertenencia de las cadenas ya sea aceptadas o rechazadas. Por ejemplo el problema:

{{< katex display>}} REGULAR = { <m> | L(M) es un lenguaje regular} {{< /katex >}}

Es decir $REGULAR$ contiene todas las descripciones de máquinas de Turing que representan a un lenguaje regular. Resulta que este lenguaje no es ni RE ni co-RE cae fuera de lo computable. Y por lo tanto el complemento también $\overline{REGULAR}$

10.6.4. Recursivo pero no Depediente del Contexto

Hasta ahora los ejemplos de MT o han sido Dependientes del Contexto, es decir se podrían implementar con un ALF o son no decidibles (es decir RE pero no Recursivos). Un ejemplo trivial

de una MT que es decidible es una MT universal MUt que decide si otra MT acabará antes de cierto tiempo o número de pasos. Como vemos esta MT debe ser en su naturaleza una MT porque tiene que simular a todas las MT, pero además sabemos que es decidible porque eventualmente para, una vez pasado el tiempo o alcanzado el número de pasos.

10.6.5. La Jerarquía de Chomsky

Lenguaje	Gramática	Máquina	Ejemplo
NCNR	–	–	$REGULAR, \overline{REGULAR}$
co-RE/R	$\alpha \rightarrow \beta$	MT,ADP,MT_k,...	$\overline{\text{AMT}}$, $\overline{\text{Lh}}/\overline{\text{L}}$ $\overline{\text{Ut}}$
RE/R	$\alpha \rightarrow \beta$	MT,ADP,MT_k,...	AMT, Lh/LUt
Lenguaje Dependientes del Contexto	$\gamma A \beta \rightarrow \gamma \alpha \beta$	Autómata Lineal con Frontera	$a^n b^n c^n$
Lenguaje Independiente del Contexto	$A \rightarrow \alpha$	Autómata de Pila	$a^n b^n$
Lenguaje Regular	$A \rightarrow bC \vert d$	AF, AFND y AFND-ε	a^n