
UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL CÓRDOBA - EXTENSIÓN ÁULICA
BARILOCHE
INGENIERÍA EN SISTEMAS DE INFORMACIÓN
AÑO LECTIVO 2025

---

# Sintáxis y Semántica de los Lenguajes

## Resumen de la Materia

### Parcial 1

---

Profesor: Santiago Oliva
Ayudante:
Fecha: 01/05/2025

Alumno: Ricardo Nicolás Freccero
Número de legajo: 415753

# Índice

# 1. Introducción a la Teoría de Autómatas y Lenguajes Formales

## 1.1. Gramáticas formales

A diferencia de los lenguajes naturales (castellano, ingles, etc.), que van cambiando constantemente y pueden tener diferentes interpretaciones según los pueblos y las épocas, los lenguajes formales desarrollados a partir de gramáticas formales no admiten ninguna forma de excepción o desviación. Estos surgieron de la necesidad de comunicarse con los computadores de manera inequívoca.

## 1.2. Máquinas abstractas

Son sistemas que operan en respuesta a estímulos del exterior, adoptando **estados** y enviando una respuesta al medio exterior. La conducta de estas máquinas depende tanto del estímulo recibido, como del estado en el que se encuentra en ese momento.

## 1.3. Características y formalismos de las máquinas abstractas

- Reconocen que el tiempo avanza de manera discreta
- Para cadad intervalo de tiempo existe un único estado de la máquina.
- El conjunto de los estados posibles es finito y está agrupado en un conjunto o *alfabeto de estados*.
- Recibe información o estímulos del medio exterior por medio de una *cinta de entrada*.
- En cada intervalo de tiempo lee un símbolo de la cinta de entrada. El conjunto de todos los símbolos que reconoce se denomina *alfabeto de entrada*.
- Pueden actuar sobre el medio exterior imprimiendo símbolos sobre una *cinta de salida*. El conjunto de símbolos que es capaz de imprimir se denomina *alfabeto de salida*.
- Las lecturas y grabaciones en cinta son realizadas en cada intervalo de tiempo por *cabezales* apropiados.
- La *función de transición* determina las aptitudes de las máquinas. Esta función define para cada símbolo de entrada y para cada estado posible, el próximo estado a ser adoptado, el funcionamiento del cabezal (si es que el autómata lo puede controlar), y el símbolo que debe ser grabado en la cinta correspondiente.

## 1.4. Jerarquía de máquinas y gramáticas

Los autómatas finitos y la máquina de Turing representan los dos extremos de una jerarquía creciente en complejidad y capacidad. En el nivel mas bajo de la jerarquía se encuentran los autómatas

finitos. A estos se le pueden ir agregando mejoras hasta llegar a la máquina de Turing, que es capaz de resolver todo problema que tenga solución y, por lo tanto, representa el límite natural de lo computable.

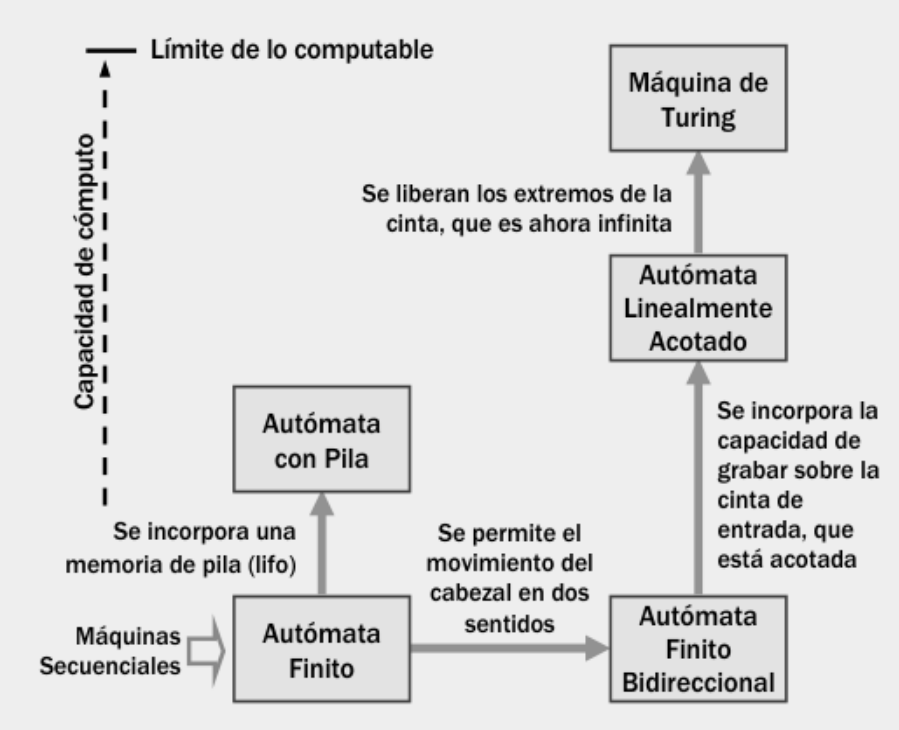



Figura 1: Jerarquía de las máquinas abstractas

### 1.4.1. Isomorfismo

Las gramáticas formales son metalenguajes destinados a la generación de los lenguajes formales. Los autómatas son modelos de entidades reconocedoras de esos lenguajes. El lenguaje es el vínculo que establece un isomorfismo entre ambos. Esto quiere decir que el estudio de las gramáticas puede reducirse al estudio de los autómatas y viceversa. Si entendes uno, entendés el otro.

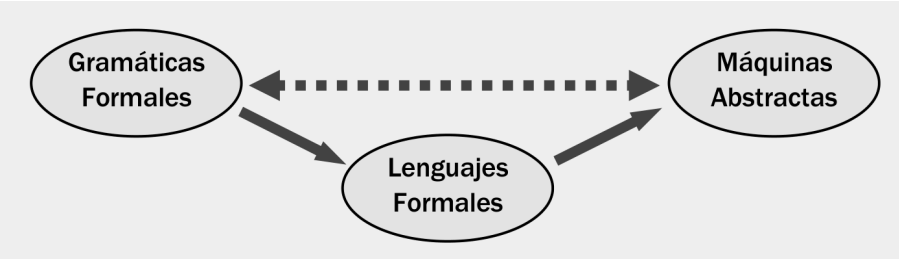



Figura 2: Isomorfismo entre gramáticas formales y máquinas abstractas através del lenguaje formal.

Chomsky clasificó las gramáticas formales en 4 tipos, de las menos restringidas a las mas, y luego a cada autómata se le asignó un nivel dentro de esa clasificación.

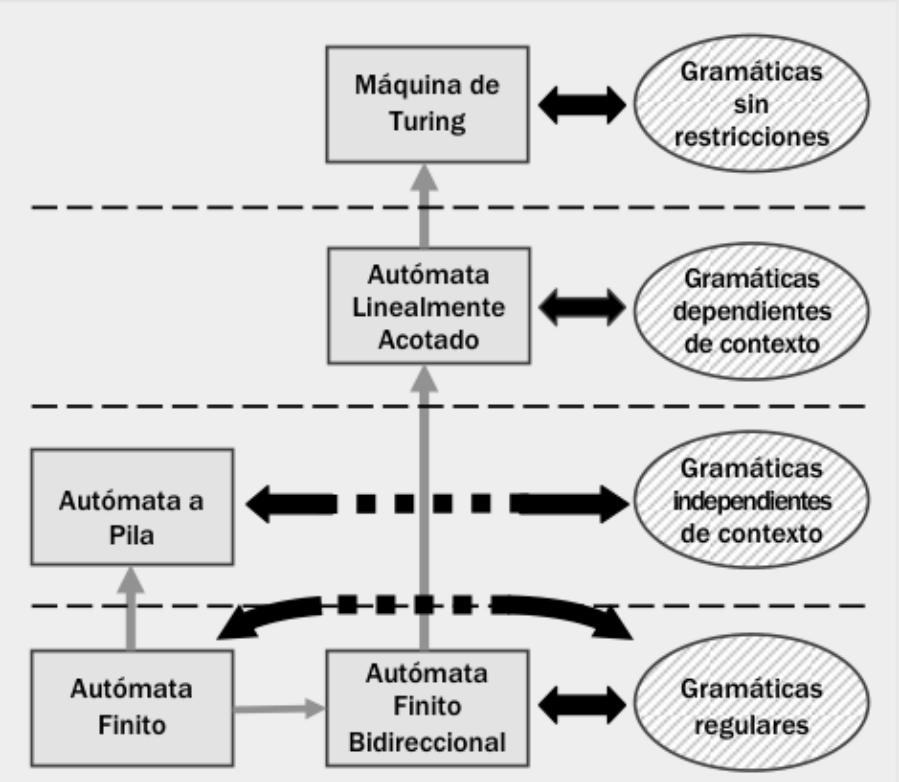



Figura 3: Relaciones entre las áquinas y los tipo de gramáticas según la jerarquía de Chomsky

# 2. Compiladores

Un compilador es una herramienta de desarrollo de software destinada a traducir un programa escrito en un lenguaje de alto nivel (programa fuente), en otro programa escrito en lenguaje máquina (programa objeto).

Los compiladores nacieron porque, cuando recién se habían creado las computadoras, la forma que había para programar en ellas era con códigos binarios que representaban las instrucciones individuales reconocidas por la CPU. Estos eran los ***Lenguajes de primera Generación*** y su utilización era dificil y muy susceptible a errores.

Después aparecieron los ***Lenguajes de Segunda Generación***, que reemplazaron el código binario de máquina por nombres simbólicos. También se los conoce como ***Lenguajes ensambladores***. Estos lenguajes sí eran mucho mas faciles de usar que los anteriores, pero igual presentaban dificultades, como la necesidad de tener conocimientos muy profundos sobre la arquitectura de la máquina.

Para sustituir a los lenguajes ensambladores se crearon los ***Lenguajes de Tercera Generación*** o ***Lenguajes de alto nivel***, que permitieron poder escribir código de programación independizándose de la máquina.

## 2.1. Conceptos relacionados

- **Editor de programas**: Los programas fuente son archivos de texto que contienen el código expresado en un lenguaje de alto nivel.
- **Compilación**: Es el proceso de convertir un programa fuente en un programa objeto, para lo cual el compilador comienza por verificar la integridad y ausencia de errores del primero.
- **Compilación en varias pasadas**: Define la cantidad de veces que un compilador debe leer el programa fuente previo a la generación del programa objeto.
- **Intérprete**: Se denomina así a una herramienta o módulo que interpreta y ejecuta las sentencias de un programa funte una tras otra, sin generar un programa objeto.
- **Conversor Fuente-Fuente**: Tiene por finalidad la conversión de un programa fuente desde un lenguaje de alto nivel, a otro también de alto nivel. Por ejemplo, existen conversores $C \rightarrow Java$ o $C \rightarrow Pascal$.
- **Ensamblador**: Es un compilador cuyo lenguaje fuente es de segunda generación.
- **Depurador**: Llamado *debugger*, es un módulo usado para facilitar las pruebas y eliminar errores de los programas.
- **Enlazador**: Llamado *task builder* o *linkeditor*, su objetivo es construir el programa ejecutable a partir del programa objeto, las librerías que pueda tener la aplicación y la librería del sistema operativo.

## 2.2. Tipos de compiladores

- **Compilador cruzado**: Es un compilador que genera programas objeto que están destinados a ser ejecutados en computadores diferentes de aquél en el que se lo ha compilado. Por ejemplo, si armamos microcontroladores en la PC que están destinados a que los utilice un teclado o un mouse.
- **Autocompilador**: Es un compilador cuyo programa fuente está escrito en el mismo lenguaje que los lenguajes fuente que admite. Por ejemplo, muchos compiladores de lenguaje C están escritos en C.
- **Metacompilador**: Es un compilador que admite programas fuentes de distintos lenguajes.
- **Compilador optimizador**: Al generar el programa objeto, optimizan el programa para mejorar el rendimiento del sistema, manteniendo la funcionalidad original.
- **Compilador gráfico**: Son aquellos que admiten programas objeto representados en forma simbólica.

- **Compilador intérprete**: Son compiladores que generan los programas objeto en un lenguaje intermedio, que luego son interpretados en el momento de la ejecución.
- **Ambiente integrado de desarrollo (IDE)**: Son sistemas interactivos que incorporan al compilador servicios complementarios, tales como un editor de programas fuentes, facilidades para interpretar los programas y ejecutarlos paso paso o en forma parcial, identificar errores y gestionar las librerías de los proyectos. Un ejemplo es VSCode.

## 2.3. Compiladores e Intérpretes

Los intérpretes permiten leer y ejecutar los programas en forma progresiva, sin necesidad de cargar en la memoria al programa fuente completo, ni de generar en memoria el programa objeto. Se puede ejecutar el código del programa sin la necesidad de que esté completo.

Los compiladores necesitan cargar en memoria el programa fuente y generar el programa objeto. Se requiere una presentación completa y rigurosa del códgio fuente.

Todo compilador involucra tres lenguajes de programación:

1. El del programa fuente que se desea compilar
2. El del programa objeto que se desea generar
3. El lenguaje de implantación, que es aquél en el que el propio compilador fue escrito.

## 2.4. Estructura y componentes de un compilador

En la compilación, se distinguen dos etapas principales, que son las de análisis y de síntesis, motradas en la figura 4. La etapa de análisis está dividida en tres fases y se encarga de dividir el programa fuente en sus elementos componentes y crear una representación intermedia del mismo. La etapa de síntesis también está dividida en tres fases y se encarga de reconstruir el programa objeto a partir de la representación intermedia generada.

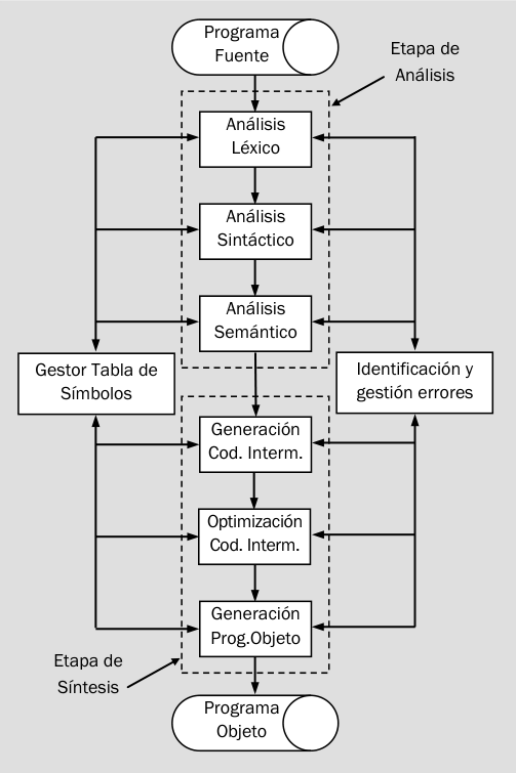



Figura 4: Estructura conceptual de un compilador

Vemos que a lo largo de estas etapas hay **dos módulos** esenciales que están permanentemente activos en el compilador. Uno de ellos es el módulo destinado a gestionar la Tabla de Símbolos, y el otro es el Administrador de Errores.

## 2.5. Análisis léxico

En esta fase, la cadena de caracteerers que constituye el programa fuente es leída carácter a carácter, para identificar y agrupar sus componentes léxicos. Estos componentes léxicos son secuencias de caracteres (cadenas) que tienen un siginificado colectivo y son denominados **tokens**. En otras palabras, lo que se hace en esta fase es agarrar la cadena entera del programa fuente e identificar todas las palabras o **tokens** que hayan en la misma. Por ejemplo, si el programa era `if (x==10) then return` los **tokens** generados son `if`, `(`, `x`, `==`, `10`, `)`, `then`, `return`.

## 2.6. Análisis Sintáctico (Parser)

En esta fase, se reciben las secuancias de componentes léxicos que fueron identificados en la fase anterior, y debido a que las combinaciones de estos componentes dan lugar a sentencias del programa fuente, lo que hay que hacer ahora es verificar que todas las sentencias puedan haber sido generadas por la gramática del lenguaje fuente.

## 2.7. Análisis semántico

Se revisa que la semántica de los programas fuente sea correcta. Es decir, en esta fase se detectan y comunican los errores que corresponden a:

- Comprobación de tipos. Por ejemplo, que no esté sumando un entero con un caracter.
- Comprobación de flujos de control.
- Comprobación de unicidad o coherencia en las denominaciones de identificadores. Por ejemplo que no hayan dos funciones con el mismo nombre.
- Coherencia en los argumentos de funciones.
- Potenciales errores en el tiempo de ejecución. Por ejemplo cuando llamas a una función o una variable que todavía no fue declarada.

## 2.8. Generación de código intermedio

Si ninguno de los análisis anteriores tiró error, el programa fuente es convertirdo en un nuevo programa escrito en un lenguaje elemental que se denomina *lenguaje intermedio*. El lenguaje universal en el que es escrito es UNCOL (Universal Compiler Oriented Language).

## 2.9. Optimización de código

Se realiza una mejora en la calidad y eficiencia del código intemedio siguiendo el siguiente orden:

1. Mejoras que se refieren a la calidad de la implementación del programa desde el punto de vista lógico.
2. Mejoras particulares para el mejor aprovechamiento global de una cierta máquina, como son la selección de las instrucciones mas apropieadas, direccionamientos, etc.

## 2.10. Generación del programa objeto

Se toma como entrada a la representación intermedia y se produce un programa objeto equivalente, que debe ser correcto, eficiente, apropiado a la máquina en la que se va a operar y apto para dar lugar a un ejecutable compatible con el entorno (sistema operativo).

## 2.11. Módulo de Gestión de Tabla de Símbolos

Es un administrador de una tabla que contiene información sobre los identificadores del programa fuente. Esta tabla, que funciona como base de datos de la información que contiene, es definida y completada en las fases de análisis, de manera que pueda consultarse en las fases de síntesis.

En esta tabla se guardan los siguientes atributos de los identificadores:

- Identidad: Si es una variable, una constante, una función, una lista, etc.
- Tipo: Si es un entero, un float, una string, etc.
- Dimensiones: En el caso de las listas.
- Argumentos: En el caso de las funciones.
- Dirección de memoria asignada.

## 2.12. Módulo de Identificación y Gestión de errores

Detecta los errores, los asocia a determinadada línea del programa fuente e intenta recuperarlo para poder seguir compilando. Cuando los compiladores están incorporados en los IDEs los errores son mostrados sobre el editor del programa fuente, facilitando la identificación y corrección por parte del programador.

# 3. Gramáticas y Lenguajes Formales

Una primera definición de lenguaje puede ser que *es un conjunto de signos y de reglas que organizan esos signos*.

## 3.1. Símbolos, alfabétos y palabras

### 3.1.1. Símbolos

Un **símbolo** es un signo creado convecionalmente. Un símbolo puede ser cualquier cosa, la interpretación es lo que determina que el símbolo actúe como tal.

### 3.1.2. Alfabeto

Un **alfabeto** es cualquier conjunto finito y no vacío de símbolos. Se usan letras griegas mayúsculas para denotarlos.

Los símbolos de un alfabeto suelen llamarse también letras o caracteres.

### 3.1.3. Palabra

Una **palabra** definida sobre un alfabeto $\Sigma$ es cualquier secuencia finita de símbolos de $\Sigma$, escritos uno a continuación del otro. Se usan letras griegas minúsculas para denotar a las palabras.

Las palabras formadas con símbolos de algún alfabeto pueden concatenarse entre ellas para formar nuevas palabras y la concatenación se escribe como el producto de las palabras. Por ejemplo, tengo $\alpha = casa$ y $\beta = blanca$, entonces $\alpha\beta = casablanca$. También existen las potencias de palabras: $\alpha^2 = \alpha\alpha = casacasa$.

### 3.1.4. Longitud de una palabra

Es la cantidad de símbolos que conforman una palabra y se denota como el módulo de una palabra. $|\alpha| = 4$

### 3.1.5. Cadena vacía

La palabra o cadena vacía se denota como $\lambda$ y es una palabra que no tiene símbolos, por ende su longitud es cero.

### 3.1.6. Potenciación de las palabras

La potenciación está definida de la siguiente forma:

$$\alpha^n = \begin{cases} \lambda & ,\text{si} \quad n = 0 \\ \alpha.\alpha^{n-1} & ,\text{si} \quad n > 0 \end{cases}$$

Existen tambien la notación $\alpha^{-1}$ para denotar a la palabra reflejada: $\alpha^{-1} = asac$
Aquellas palabras que son iguales a sus reflejas se denominan palíndromos.

### 3.1.7. Prefijos, sufijos y subpalabras

Supongamos que una palabra puede escribirse como una concatenación de otras tres, digamos $\omega = \alpha\gamma\beta$; entonces decimos que $\alpha$ es un **prefijo** de $\omega$, $\beta$ es un **sufijo** de $\omega$ y $\gamma$ es una **subpalabra** de $\omega$. En este caso **cualquiera de las tres palabras puede ser vacía**.

De un sufijo o prefijo de una palabra se dice que es **propio** si no es la misma palabra en cuestión o la palabra vacía.

| Prefijos de ω | | Sufijos de ω | |
|---|---|---|---|
| | λ | λ | |
| Propios | p | a | Propios |
| | pr | ba | |
| | pru | eba | |
| | prue | ueba | |
| | prueb | rueba | |
| | prueba | prueba | |

Figura 5: Prefijos y sufijos

### 3.1.8. Operaciones con alfabetos

Como los alfabetos son conjuntos, podemos realizar todas las operaciones que conocemos para conjuntos con ellos: los podemos unir, intersectar, restar, complementar, etc.

Podríamos operarlos con el producto cartesiano para obtener pares ordenados, pero esto no tiene mucho uso en realidad. Lo que sí podemos hacer es una operación parecida: dados dos alfabetos $\Sigma_1$ y $\Sigma_2$, la **concatenación** de $\Sigma_1$ y $\Sigma_2$, denotada como $\Sigma_1\Sigma_2$ es el conjunto de palabras formadas por la concatenación de cada símbolo de $\Sigma_1$ con cada uno de $\Sigma_2$ en ese orden.

Basicamente, la operación de concatenación de dos alfabetos es lo mismo que el producto cartesiano, pero ahora en vez de generar tuplas, concatenas las letras del alfabeto.

Ejemplo:

Sean los alfabetos $\Sigma_1 = \{0,1\}$ y $\Sigma_2 = \{a,b,c\}$. Entonces:

$$\Sigma_1\Sigma_2 = \{0a, 0b, 0c, 1a, 1b, 1c\}$$
$$\Sigma_2\Sigma_1 = \{a0, a1, b0, b1, c0, c1\}$$

### 3.1.9. Potenciación de un alfabeto

Tomemos el alfabeto $\Sigma_1 = \{0,1\}$; si lo concatenamos consigo mismo obtenemos:

$$\Sigma_1\Sigma_1 = \{0,1\}\{0,1\} = \{00, 01, 10, 11\}$$

que es el conjunto de todas las palabras de largo dos que pueden formarse con los símbolos 0 y 1 del alfabeto.

Se define a la **potenciación de un alfabeto** (que nos permite obtener todas las palabras de larno $n$ del alfabeto), de la siguiente forma:

$$\Sigma^n = \begin{cases} \{\lambda\} & ,\text{si} \quad n = 0 \\ \Sigma\Sigma^{n-1} & ,\text{si} \quad n > 0 \end{cases}$$

Se denomina **universo de discurso de un alfabeto** $\Sigma$, al conjunto de todas las palabras que pueden formarse con sus símbolos, sean del largo que sean. Este conjunto suele denotarse $\Sigma^*$

El universo de disurso de un alfabeto $\Sigma$ entonces está compuesto por todas las cadenas de símbolos de $\Sigma$ de largo cero $\left(\Sigma^0\right)$, todas las de largo uno $\left(\Sigma^1\right)$, todas las de largo 2 $\left(\Sigma^2\right)$, ... y así sucesivamente.

$$\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \cdots \cup \Sigma^i \cup \ldots$$

Una mejor forma de expresarlo es así:

$$\Sigma^* = \bigcup_{i=0}^{\infty} \Sigma^i$$

Si al universo de discurso le quitamos la unión de la potencia cero, obtenemos el **cierre** o **clausura**

del alfabeto y se denota como $\Sigma^+$:

$$\Sigma^+ = \bigcup_{i=1}^{\infty} \Sigma^i$$
$$\Sigma^+ = \bigcup_{i=0}^{\infty} \Sigma^i - \Sigma^0$$
$$\Sigma^+ = \Sigma^* - \{\lambda\}$$

El conjunto clausura de $\Sigma$ es el conjunto de todas las palabras de largo mayor que uno.

## 3.2. Lenguajes y operaciones

Un lenguaje definido sobre un alfabeto es un conjunto de palabras construidas con los símbolos de ese alfabeto. En símbolos, si $L$ es un lenguaje definido sobre $\Sigma$, entonces:

$$L \subseteq \Sigma^*$$

Como los lenguajes son conjuntos de palabras, podemos aplicar toda la teoría de conjuntos sobre ellos.

La única diferencia respecto a las operaciones que hay entre los lenguajes y los alfabetos es que en el caso de los lenguajes no podemos definir la clausura de $L$ como la unión de todas las potencias menos la potencia 0. En símbolos, la siguiente definición es válida:

$$L^+ = \bigcup_{i=1}^{\infty} L^i$$

pero también es cierto lo siguiente:

$$L^+ \neq \bigcup_{i=0}^{\infty} L^i - L^0$$
$$L^+ \neq L^* - \{\lambda\}$$

Esto se debe a que como $L \subseteq \Sigma^*$, $L$ puede contener como elemento a la cadena vacía $\lambda$. Entonces, puede pasar que la potencia 1 de $L$ contenga a la cadena vacía, y si se la restamos al restar la potencia 0, estaríamos perdiendo ese caracter por lo que el conjunto que tendríamos ya no sería el conjunto clausura de $L$, porque que le faltaría un caracter que pertenece al conjunto de palabras de $L^1$.

Sin embargo, siempre vamos a poder decir que:

$$L^* = L^+ \cup \{\lambda\}$$

### 3.2.1. Descripción de los lenguajes

Los podemos describir por **extensión** o por **comprensión**. Ya lo sabemos esto por Lógica (aguante Fabi).

## 3.3. Gramáticas formales

### 3.3.1. Reglas de escritura o producciones

Para un alfabeto $\Sigma$ dado, diremos que una **producción** o **regla de reescritura** es un par ordenado de palabras $(\alpha, \beta)$ definidas sobre $\Sigma$. Las producciones las podemos ver escritas de la siguiente forma también:

$$\alpha := \beta$$

Decimos que $\alpha$ es el lado izquierdo de la producción, $:=$ es el símbolo de producción, y $\beta$ es el lado derecho de la producción. La producción se lee *alpha produce beta*.

Se llama derivación directa a la operación que aplica una sola producción a una palabra obteniendo una nueva palabra y se simboliza:

$$\delta \to \varphi$$

Se dice que $\delta$ deriva directamente en $\varphi$.

Se llama **derivación** a la operación de aplicar una secuencia finita de producciones a una cadena $\delta$ dada para obtener otra cadena $\varphi$, y se la simboliza:

$$\delta \to^* \varphi$$

que se lee $\delta$ deriva en $\varphi$. Esto equivale a decir que existen cadenas $\alpha_0, \alpha_1, \ldots, \alpha_n$ tales que:

$$\delta = \alpha_0 \to \alpha_1 \to \cdots \to \alpha_n = \varphi$$

Si durante el proceso de derivación, cada vez que puede optarse por una producción a aplicar, se efectúa el reemplazo posible mas a la derecha en la cadena, se dice que se hizo una **derivación por derecha**. Si el reemplazo que siempre se elige es el de mas a la izquierda, se dice que se hizo una **derivación por izquierda**. Cuando las opciones se toman mezcladas, se dice que se hizo una **derivación mixta**.

Al paso contrario de la derivación, cuando vamos de una palabra derivada y llegamos a la palabra original, se dice que hicimos una **reducción**. Todo lo mismo que aplica a las derivaciones se cumple para las reducciones.

### 3.3.2. Gramáticas formales

Una gramática formal es una cuádrupla $(\Sigma_T, \Sigma_N, S, P)$ en la cual sus cuatro componentes representan:

- $\Sigma_T$: es el alfabeto de los símbolos que formarán las cadenas del lenguaje que se está describiendo. Es el **alfabeto de los símbolos terminales**.
- $\Sigma_N$: es un conjunto de variables o símbolos auxiliares llamado **alfabeto de símbolos no terminales**.
- $S$: $S \in \Sigma_N$ es un símbolo no terminal distinguido denominado **axioma** de la gramática.
- $P$: es un **conjunto de producciones** donde ambas palabras pueden estar compuestas de símbolos terminales y no terminales, pero el lado izquiero de la producción debe tener al menos un símbolo no terminal

El alfabeto de símbolos no terminales debe ser disjunto con el alfabeto de símbolos terminales:

$$\Sigma_T \cap \Sigma_N = \varnothing$$

### 3.3.3. Lenguaje Generado

Dada una gramática formal $G = (\Sigma_T, \Sigma_N, S, P)$ se llama **lenguaje formal generado por G** al conjunto de todas las cadenas de símbolos terminales que puedan derivarse desde el axioma $S$ usando la reglas de producción de $P$. En símbolos:

$$L(G) = \{\alpha \in \Sigma_T^* / S \rightarrow^* \alpha\}$$

Durante la derivacion de una cadena del lenguaje descripo por $G$, se generarán eventualmente cadenas intermedias. Se denomina **forma sentencial** o **metapalabra** a una cadena de terminales y no terminales $\alpha_i \in (\Sigma_T \cup \Sigma_N)^*$ que puede derivarse desde el axioma de la gramática.

La cadena final de terminales obtenida derivando desde el axioma se denomina **sentencia** o **palabra generada** por la gramática.

### 3.3.4. Equivalencias de gramáticas

Dos gramáticas $G_1$ y $G_2$ so **equivalentes** sí y solo sí generan exactamente el mismo lenguaje.

$$G_1 \equiv G_2 \iff L(G_1) = L(G_2)$$

### 3.3.5. Clasificación de las gramáticas

Dijimos que los lenguajes formales son aquellos lenguajes que pueden ser generados por gramáticas formales. Chomsky estableció que todos los lenguajes formales podían clasificarse en cuatro tipos que solo se distinguen por el formato de las producciones de las gramáticas que lso generan. Mientras mas restricciones se le ponen a las producciones, menos lenguajes se pueden describir.

**Tipo 0: Lenguajes estructurados por frases**

Son los menos restrictivos. Las producciones pueden contener cualquier cadena de terminales y no terminales tanto en lado izquierdo como en el lado derecho, con al menos un símbolo no terminal en el lado izquierdo. En símbolos:

$$\alpha := \gamma \quad \alpha \in (\Sigma_T \cup \Sigma_N)^* \Sigma_N (\Sigma_T \cup \Sigma_N)^*, \quad \gamma \in (\Sigma_T \cup \Sigma_N)^*$$

Donde $(\Sigma_T \cup \Sigma_N)^*$ es cualquier palabra de cualquier longitud que tenga caracteres de cualquier alfabeto, y la concatencación en el medio de $\Sigma_N$ nos obliga a que $\alpha$ tenga al menos un símbolo no terminal.

**Tipo 1: Lenguajes dependientes del contexto**

Son lenguajes sensibles al contexto. Sus producciones tiene la forma:

$$S := \lambda \quad \text{o} \quad \alpha A \beta := \alpha \gamma \beta \qquad \alpha, \beta \in (\Sigma_T \cup \Sigma_N)^*, \quad A \in \Sigma_N, \quad \gamma \in (\Sigma_T \cup \Sigma_N)^+$$

Esto dice que el símbolo no terminal $A$, solo puee ser reemplazado por la cadena $\gamma$ si tiene un $\alpha$ a su izquierda y un $\beta$ a su derecha, es decir, *si se encuentra dentro del contexto alpha-beta*.

Ademas, hay que fijarse que $\gamma$ es una cadena que puede contener tanto caracteres terminales como no terminales, pero su longitud debe ser mayor a cero. Por esta razón, en estos casos siempre la caden del lado izquierdo es de largo igual o menor que la del lado derecho.

Otra cosa importante es que el lenguaje podría contener como palabra a la cadena vacía, así que esta debería poder ser generada por el lenguaje. Por esto, se permite $S := \lambda$

**Tipo 2: Lenguajes independientes del contexto**

Sus producciones pueden adoptar las siguientes formas:

$$S := \lambda \quad \text{o} \quad A := \alpha \qquad A \in \Sigma_N, \quad \alpha \in (\Sigma_T \cup \Sigma_N)^+$$

En este caso el símbolo terminal $A$ puede ser reemplazado por la cadena $\alpha$ de terminales y no terminales en cualquier lugar donde aparezca durante el proceso de derivación. Por eso estos lenguajes son *independientes del contexto*.

Ademas, el lado derecho de la producción no puede ser de menor longitud que el lado izquierdo (no puede existir una *regla compresora*).

Al igual que los lenguajes de tipo 1, se permite $S := \lambda$.

**Tipo 3: Lenguajes regulares o lineales**

Las producciones tienen un solo símbolo no terminal del lado izquierdo, pero su lado derecho está compuesto por un solo símbolo terminal, o por un símbolo terminal y un no terminal, y puede contener la regla lambda.

Su formato de escritura puede ser de dos formas, totalmente equivalentes:

Regular por derecha: $S := \lambda \quad \text{o} \quad A := aB \quad \text{o} \quad A := a \qquad A, B \in \Sigma_N, \quad a \in \Sigma_T$

Regular por izquierda: $S := \lambda \quad \text{o} \quad A := Ba \quad \text{o} \quad A := a \qquad A, B \in \Sigma_N, \quad a \in \Sigma_T$

Estos formatos no pueden mezclarse en una misma gramática y seguir siendo regular (si esto pasa sería una gramática de tipo 2).

**Tips**

Si en una gramática existe al menos una producción $\alpha := \beta$, siendo la cadena del lado izquierdo $\alpha$ de mayor longitud que la del lado derecho $\beta$, entonces la gramática será **tipo 0** (la regla $S := \lambda$ no cuenta ya que siempre es permitida).

Si no se cumple eso, y en todas las reglas siempre un símbolo no terminal del lado izquierdo puede ser reemplazado por una cadena no vacía del lado derecho, entonces la gramática es **tipo 1**.

Si en una gramática, el lado izquierdo de todas las producciones solo tiene un símbolo no terminal, la gramátcia es **tipo 2**.

Finalmente, si todas las producciones de la gramática tienen un no terminal en el lado izquierdo y solo un terminal o un símbolo terminal y un no terminal en el lado derecho, la gramática es de **tipo 3**.

## 3.4. Lenguajes regulares

Como vimos, los lenguajes regulares son los mas restringidos en la jerarquía de Chomsky, pero ampliamente utilizados en informática. Son necesarios durante la etapa de análisis léxico de los compiladores, que tiene como tarea separar al programa fuente en tokens.

Los lenguajes regulares admiten la siguiente **definición recursiva**:

a) Cualquier lenguaje **finito** $L_1$ definido sobre algún alfabeto $\Sigma$ es regular.

b) Si $L_1$ y $L_2$ son lenguajes regulares, entonces también los son su unión $L_1 \cup L_2$ y su concatenación $L_1 L_2$.

c) Si $L_1$ es un lenguaje regular, entonces su estrella de Kleene $L_1^*$ también es regular.

d) Solo son lenguajes regulares los construidos con a), b) y c).

### 3.4.1. Expresiones regulares

Una expresión regular es una forma mas compacta de expresar lenguajes regulares que las gramáticas tipo 3 que vimos.

Podemos definir las expresiones regulares recursivamente como sigue:

Sea $\Sigma$ un alfabeto, entonces:

a) $\varnothing$ es una expresión regular que denota al lenguaje vacío $L(\varnothing) = \varnothing$.

b) $\lambda$ es una expresión regular que denota al lenguaje cuyo único elemento es la cadena vacía $L(\lambda) = \{\lambda\}$.

c) Cualquier símbolo $a$ del alfabeto $\Sigma$ es una expresión regular que denota al lenguaje cuya única palabra es la de largo unitario formada por ese símbolo $L(a) = \{a\}$.

Si $E$ y $F$ son expresiones regulares, entonces:

d) $E + F$ es la unión de los lenguajes denotados por $E$ y por $F$ : $E + F = L(E) \cup L(F)$.

e) $EF$ es la concatenación de los lenguajes $E$ y $F$: $EF = L(E)L(F)$.

f) $E^*$ es el universo de discurso del lenguaje $E$: $E^* = [L(E)]^*$.

g) $(E)$ es el mismo lenguaje $E$: $(E) = E$.

h) Solo son expresiones regulares las construidas con las pasos a) al g).

## 3.5. Lenguajes Independientes del Contexto (LIC)

Las gramáticas independientes del contexto son muy importantes ya que describen la sintaxis de los lenguajes de programación.

### 3.5.1. Gramática limpia

Una gramática limpia es aquella que no tiene reglas o símbolos inútiles. Cuando una gramática no está limpia, hay que limpiarla para conseguir una gramática equivalente que ocupa menos espacio en memoria y su tiempo de procesamiento es menor.

Pueden existir varios tipos de “impurezas”dentro de una GIC, y son las siguientes:

**Regla innecesaria**

Son las reglas de tipo $A := A$ en las que un símbolo no terminal se genera a sí mismo y no aporta nada al lenguaje. Esta regla **puede eliminarse del conjunto P** de producciones de cualquier gramática sin la necesidad de efectuar ningún cambio adicional en la misma.

**Símbolos inaccesibles**

Son los símbolos terminales o no terminales de la gramática que no pueden ser alcanzados desde el axioma por ninguna derivación válida. Estos símbolos pueden ser eliminados junto con todas las producciones que los tengan y no se va a modificar el lenguaje generado.

**Símbolo superfluo**

Son los símbolos no terminales que no permiten generar desde ellos al menos una cadena vacía o de solo símbolos terminales. Los símbolos superfluos junto a todas las producciones que lo contengan pueden eliminarse de la gramática y el lenguaje generado no se verá modificado. (rima :))

Ejemplo: Analicemos la siguiente gramática:

$$G = (\{a,b\}, \{S,A,B\}, S, \{S := aAb, A := aAb \mid ab \mid aB, B := aBb\})$$

El no terminal $B$ resulta superfluo ya que desde $B$ es imposible derivar una cadena de terminales. Así que lo podemos eleminar del conjunto de no terminales y quitar las dos producciones que lo contienen, obteniendo la gramática equivalente:

$$G = (\{a,b\}, \{S,A\}, S, \{S := aAb, A := aAb \mid ab\})$$

**Gramática limpia**
Se dice que una gramática independiente del contexto (GIC) **está limpia** sí y solo sí no tiene reglas innecesarias, ni símbolos inaccesibles, ni símbolos superfluos.

### 3.5.2. Gramática bien formada

Según la jerarquía de Chomsky, en este tipo de gramáticas no pueden existir reglas compresoras (el lado derecho de la producción es de menor longitud que el lado izquierdo).

**Regla no generativa**

Una regla del tipo $A := \lambda$, no siendo $A$ el axioma de la gramática es una regla compresora denominada **regla no generativa**. Muchas veces es mejor eliminar este tipo de reglas de una gramática y para hacer eso hay que proceder de la siguiente forma:

a) Para cada producción $X := \alpha A \beta$ que contenga el no terminal A en lado derecho, agregar regla de reescritura $X := \alpha\beta$ que se obtiene de reemplazar $A$ por la cadena vacía.

b) Luego eliminar del conjunto de producciones $A := \lambda$ ya que todos los efectos que produciría la misma, han sido incluidos explícitamente como producciones en el paso a).

Por ejemplo, si tuviesemos la siguiente gramática:

$$G = (\{a,b\}, \{S,A\}, S, \{S := aAb, A := aAb \mid \lambda\})$$

, y queremos eliminar la regla no generativa $A := \lambda$ no podemos simplemente quitar el $\lambda$ de las producciones de $A$ ya que habrían cadenas que se pueden obtener estando $\lambda$ que sin ella no se podrían obtener.

En cambio, si seguimos los pasos descriptos, la gramática quedaría de la siguiente forma:

$$G = (\{a,b\}, \{S,A\}, S, \{S := aAb \mid ab, A := aAb \mid ab\})$$

, que es una gramática equivalente a la anterior y no tiene ninguna regla no generativa.

**Regla de redenominación**

Si en una gramática existen producciones del tipo $A := B$ donde $A$ y $B$ son símbolos no terminales, esta producción se llama **regla de redenominación**. Esta regla dice que $A$ puede ser reescrito como $B$ en cualquier contexto donde se encuentre. Sin embargo, $A$ puede tener otras derivaciones distintas de las de $B$ y esto puede confundir a las rutinas de análisis sintáctico.

Es necesario entonces eliminar este tipo de reglas de las gramáticas y para eso hay que seguir los pasos siguientes:

a) Por cada regla $B := \alpha$ existente en la gramática, agregar una regla $A := \alpha$ para hacer explícito que a partir de $A$ podemos derivar todas las producciones de $B$ ($A \to B \to \alpha$).

b) Luego puede eliminarse $A := B$ del conjunto de producciones y la gramática obtenida será equivalente a la original.

Ejemplo: Dada

$$G = (\{0,1\}, \{S,T\}, S, \{S := 0S \mid S1 \mid T, T := 01 \mid 0T\})$$

, para quitar la regla de redenominación $S := T$ seguimos los pasos anteriores.

Tenemos que agregar al conjunto de producciones de la gramática $S := 01$ y $S := 0T$, que son todas las producciones de $T$, para poder quitar la regla de redenominación. Entonces la gramática equivalente sin dicha regla queda así:

$$G = (\{0,1\}, \{S,T\}, S, \{S := 0S \mid S1 \mid 01 \mid 0T, T := 01 \mid 0T\})$$

**Gramática bien formada**
Se dice que una gramática independiente del contexto (GIC) está **bien formada**, sí y solo sí, está limpia y no tiene reglas no generativas ni de redenominación.

## 3.6. Análisis sintáctico

El análisis sintáctico es un proceso mediante el cuál podemos determinar si una cadena pertenece a un lenguaje determinado o no. Para lograr esto podemos encontrar una derivación $S \to^* \alpha$, o demostrar que tal derivación no existe. Este procedimiento puede llegar a ser extremadamente largo y tedioso, por lo que se crearon métodos claros y repetibles para realizar el análisis.

### 3.6.1. Árbol de derivación

Un árbol de derivación es un árbol cuyo nodo raíz es el axioma de la gramática, cuyos hijos son todas las posibles derivaciones. Cada nodo a su vez tiene también como hijos a todas sus derivaciones de manera que todos las hojas del árbol son símbolos terminales.

Ejemplo: Supongamos que tenemos la siguiente gramática:

$$G = (\{0,1\}, \{S,P,Q\}, S, \{S := PQ \mid 0S1, P := 0Q \mid 1, Q := 1P \mid 0\})$$

y queremos saber si la cadena 0101 pertence al lenguaje $L(G)$. Para eso podemos partir de $S$ e ir armando el árbol de dervación para ver si podemos conseguir la cadena 0101.

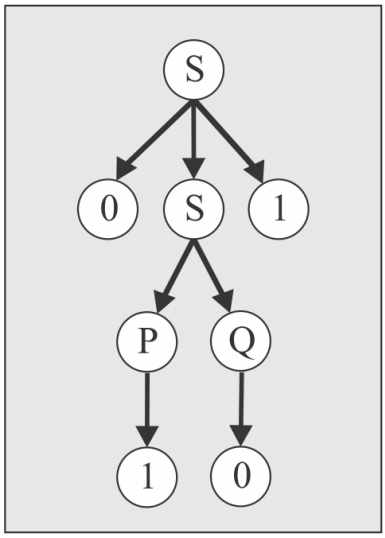



Figura 6: Árbol de derivación de la cadena 0101

Como encontramos un árbol de derivación para la cadena 0101, podemos decir que dicha cadena pertenece al lenguaje generado por la gramática $G$.

### 3.6.2. Ambigüedad

Es cuando una cadena ;$alpha$ de símbolos terminales puede ser generada por distintas derivaciones que generan distintos árboles de derivación.

Esto puede traer problemas ya que una misma cadena podría tener distintos significados según cómo se derive.

### 3.6.3. Recursividad

Una producción de una GIC se dice que es **recursiva** si el no terminal de su lado izquierdo se encuentra también en el lado derecho:

$$A := \alpha A \beta$$

La recursividad permite describir un lenguaje de infinitas cadenas con un número finito de producciones. Si una gramática no tieen recursión solo podrá generar un número finito de cadenas.

Si en una regla recursiva $A := \alpha A \beta$ la cadena $\alpha$ es vacía, osea $A := A\beta$, se dice que la regla es **recursiva por izquierda**. Si en cambio la cadena vacía es $\beta$, osea $A := \alpha A$, se dice que la regla es **recursiva por derecha**.

La recursión por izquierda hay que eliminarla ya que puede provocar recursiones infinitas que llevan a errores de ejecución en el proceso de análisis.

Para eliminar la recursión por izquierda, y obtener una gramática equivalente hay que seguir los siguientes pasos:

Sea $G$ una GIC y $A \in \Sigma_N$ un símbolo que tiene producciones recursivas por izquierda y producciones no recursivas por izquierda:

$$A := A\alpha_1 \mid A\alpha_2 \mid \cdots \mid A\alpha_n \mid \beta_1 \mid \beta_2 \mid \cdots \mid \beta_m$$

con $\alpha_i, \beta_j \in (\Sigma_T \Sigma_N)^+$. Siempre podemos obtener una gramática equivalente sin recursividad por izquierda de la siguiente forma:

a) Creamos un nuevo símbolo no terminal $X$ y lo agregamos al alfabeto de símbolos no terminales $\Sigma_N = \Sigma_N \cup \{X\}$.

b) Eliminamos todas las producciones en $P$ para el no terminal $A$.

c) Agregamos al conjunto $P$ las producciones:

$A := \beta_1 X \mid \beta_2 X \mid \cdots \mid \beta_m X \mid \beta_1 \mid \beta_2 \mid \cdots \mid \beta_m$

$X := \alpha_1 X \mid \alpha_2 X \mid \cdots \mid \alpha_n X \mid \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_n$

La nueva gramática obtenida es equivalente a la original y no tiene recursión por izquierda.

### 3.6.4. Factorización por izquierda

Algunos analizadores sintácticos pueden presentar problemas cuando hay producciones del siguiente tipo:

$$A := \alpha\beta \mid \alpha\gamma$$

En estos casos podemos crear un nuevo no termial $X$ y reemplazar las producciones por:

$$A := \alpha X \qquad \text{y} \qquad X := \beta \mid \gamma$$

## 3.7. Forma Normal de Chomsky (FNC)

Una gramática se dice que está en **Forma Normal de Chomsky** sí y solo sí, todas sus producciones tienen en el lado derecho dos sómbolos no terminales, o un solo símbolo terminal (o la cadena

vacía en el caso del axioma).

$$A := BC \quad \text{o} \quad A := a \quad \text{o} \quad S := \lambda$$

Una gramática en FNC siempre tendrá árboles de derivación binarios.
Cualquier GIC puede ser transformada en FNC mediante el siguiente procedimiento:

a) Transformar $G$ en una gramática bien formada.

b) Para cada símbolo $\alpha \in \Sigma_T$ crear un nuevo no terminal $X_a$ y una nueva producción $X_a := a$.

c) Para cada producción que contenga en su lado derecho tanto símbolos no terminales como símbolos terminales, reemplazarla por una nueva que tenga en lugar del terminal $a$ su correspondiente nuevo no terminal $X_a$.

d) Para cada producción con mas de dos símbolos no terminales en su lado derecho $A := B\Gamma$, donde $\Gamma$ es una cadena de dos o mas no terminales, crear un nuevo símbolo no terminal $X$ y reemplazar la producción por el par $A := BX$ y $X := \Gamma$

Ejemplo: Encontrar una gramática equivalente a la siguiente que esté en FNC:

$$G = (\{a,b,c\}, \{A,B,C\}, A, \{A := CBc \mid bB \mid \lambda, B := BC \mid b, C := c\})$$

a) Lo primero que hacemos es revisar que esté bien formada. En este caso el terminal $a$ es un símbolo inaccesible, no tiene reglas innecesarias, no tiene símbolos superfluos, tampoco reglas de redenominación, ni reglas no generativas. De esta manera, elimiando $a$ la gramática queda **bien formada**.

b) Creamos los no terminales $X_b$ y $X_c$ y las producciones $X_b := b$ y $X_c := c$.

c) Vamos producción a producción modificando las que tengan símbolos no terminales y terminales y reemplazando adecuadamente:

   a. Reemplazamos la producción $A := CBc$ por $A := CBX_c$

   b. Reemplazamos la producción $A := bB$ por $A := X_bB$

   c. $A := \lambda$ es una regla válida por que $A$ es el axioma de la gramática.

   d. Las últimas tres producciones $B := BC$, $B : b$ y $C := c$ no hace falta modificarlas

d) La primer producción $A := CBX_c$ tiene mas de dos símbolos no terminales a su derecha así que creamos un nuevo no terminal $X$ y reemplazamos la producción por $A := CX$ y $X := BX_c$

De esta manera, la nueva gramática queda así:

$$G' = (\{b,c\}, \{A,B,C,C,X_c,X_b\}, A, \{A := CX \mid X_bB \mid \lambda, B := BC \mid b, C := c, X := BX_c, X_c := c\})$$

Podemos ver que la producción $X_c := c$ es redundante en este caso porque ya existía la producción $C := c$, así que podemos reemplazar el $X_c$ por $C$ en todas las producciones y eliminar el símbolo $X_c$ quedando:

$$G'' = (\{b,c\}, \{A,B,C,X,X_b\}, A, \{A := CX \mid X_bB \mid \lambda, B := BC \mid b, C := c, X := BC, X_b := b\})$$

## 3.8. Forma Normal de Greibach (FNG)

Una GIC está en **Forma Normal de Greibach** sí y solo sí, todas sus producciones inician su lado derecho con un símbolo terminal al que le sigue, opcionalmente una cadena símbolos no terminales de caulquier largo. En símbolos, las producciones tiene la siguiente forma:

$$A := a\Gamma \quad \text{o} \quad S := \lambda \qquad A, S \in \Sigma_N, \Gamma \in \Sigma_N^*$$

Cualquier GIC puede ser reescrita de forma equivalente en su forma FNG siguiendo los siguientes pasos:

a) Transformar $G$ en una gramática bien formada.

b) Quitar la recursividad izquierda de la gramática.

c) Asignar un orden numerico cualquiera a los símbolos no terminales de la gramática, digamos $A_1, A_2, \ldots, A_k$.

d) Separar las producciones del conjunto $P$ en tres grupos:

   a. Grupo 1: Todas las producciones que comiencen con un terminal y, si existe en la gramática $G$, la regla lambda $S := \lambda$.

   b. Grupo 2: Producciones $A_i := A_j\alpha$ con $\alpha \in (\Sigma_T \cup \Sigma_N)^*$ y con el símbolo $A_i$ anterior a $A_j$.

   c. Grupo 3: Producciones $A_i := A_j\alpha$ con $\alpha \in (\Sigma_T \cup \Sigma_N)^+$ y con el símbolo $A_i$ posterior a $A_j$.

e) Para cada producción del tercer grupo, iniciando por aquellas con el subíndice $i$ mas pequeño, reemplazarlas por $A_i := \delta_1\alpha \mid \delta_2\alpha \mid \cdots \mid \delta_h\alpha$ donde los $\delta$ son los lados derechos de todas las producciones de $A_j$. Cuando termine este proceso todas las producciones van a pertenecer al grupo 1 o 2.

f) Repetir el proceso anterior para las producciones del segundo grupo. Al terminar, todas las producciones serán del grupo 1, por lo que todas inciarán con un símbolo terminal.

g) Para cada símbolo terminal $a \in \Sigma_T$ que esté en el lado derecho de las producciones, pero no al inicio, crear un unevo símbolo no terminal $X_a$ y una nueva producción $X_a := a$

h) Para cada producción de la gramática que contenga en su lado derecho, luego del primer símbolo terminal, tanto símbolos no terminales como símbolos terminales, reemplazarla por una nueva que tenga en lugar del terminal no inical $a$ su correspondiente nuevo no terminal $X_a$.

Al terminar el procedimiento, todas las producciones estarán en FNG.

Ejemplo: Queremos convertir la gramática inicial del ejemplo anterior a una equivalente pero en FNG, tomemos como inicio la gramática bien formada:

$$G = (\{b,c\},\{A,B,C\},A,\{A := CBc \mid bB \mid \lambda, B := BC \mid b, C := c\})$$

a) Primero revisamos que esté bien formada (ya sabemos que sí).

b) Eliminamos la recursividad por izquierda. Vemos que el símbolo no terminal $B$ tiene recursividad por izquierda así que laeliminamos. Para eso creamos un nuevo no terminal $X$ y reemplazamos la producción de $B$ por $B := bX \mid b$ y $X := CX \mid C$. En este caso se generó una **regla de redenominación** $X := C$ que la tenemos que cambiar por $X := c$ para volver a dejar a la gramática bien formada.

La nueva gramática bien formada es ahora:

$$G' = (\{b,c\},\{A,B,C,X\},A,\{A := CBc \mid bB \mid \lambda, B := bX \mid b, C := c, X := CX \mid c\})$$

c) Asignamos un orden alfabético a los símbolos no terminales.

d) Tomando el orden alfabético para los no terminales se pueden separar las producciones en:

Grupo 1. $A := bB \mid \lambda, B := bX \mid b, C := c, X := c$

Grupo 2. $A := CBc$, ya que $A$ antecede a $C$ en el orden.

Grupo 3. $X := CX$, ya que $X$ es posterior a $C$ en el orden.

e) Se opera sobre la producción del grupo 3:

$$X := CX \quad \text{Es reemplazada por} \quad X := cX$$

Todas las producciones ahora son del grupo 2 o 1.

f) Se opera sobre la producción del grupo 2:

$$A := CBc \quad \text{Es reemplazada por} \quad A := cBc$$

Todas las producciones ahora son del grupo 1:

$$A := cBc \mid bB \mid \lambda, B := bX \mid b, C := c, X := xC \mid c$$

g) Creamos un nuevo no terminal $X_c$ para la letra $c$ que es la única que está del lado derecho de la producción pero no al inicio, y agregamos la producción $X_c := c$. Pero este no terminal sería redundante ya que existe el no terminal $C$ que tiene la misma producción, por lo que podríamos usar al no terminal $C$ directamente.

h) Reemplazamos los símbolos correspondientes por la nueva producción. En este caso reemplazamos la primer producción por $A : -cBC$.

De esta manera, la nueva gramática $G''$ en modo FNG equivalente a $G$ es:

$$G'' = (\{b,c\},\{A,B,C,X\},A,\{A := cBC \mid bB \mid \lambda, B := bX \mid b, C := c, X := cX \mid c\})$$

# 4. Máquinas Secuenciales y Autómatas Finitos Deterministas

Las máquinas secuenciales mas difundidas son las propuestas por George Mealy y Edward Moore. Estas máquinas son esencialmente traductoras, es decir que a partir de una sucesión de símbolos de entrada generan una sucesión de símbolos de salida. Además, estas máquinas operan de forma permanente por lo que no cuentan con un estado de arranque ni un estado de detención.

## 4.1. Máquina de Mealy

La máquina de Mealy tiene cinco componentes y es definida así:

$$ME = (\Sigma_E, \Sigma_S, Q, f, g)$$

donde:

$\Sigma_E$ : Alfabeto de símbolos de entrada

$\Sigma_S$ : Alfabeto de símbolos de salida

$Q$ : Conjunto finito y no vacío de estados posibles

$f$ : Función de transición, $f : Q \times \Sigma_E \to Q$

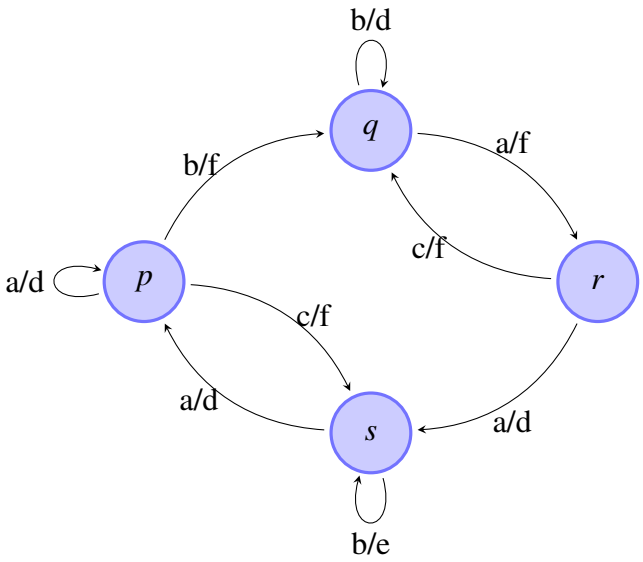
$g$ : Función de salida, $g : Q \times \Sigma_E \to \Sigma_S$

La función de transición $f$ define el próximo estado que adoptará la máquina a partir de su estado actual y cada uno de los posibles símbolos de entrada.

La función de salida $g$ define la salida de la máquina a partir de los mismo argumentos.

Una alternativa para definir una máquina secuencial es la utilización de grafos dirigidos donde los nodos representan los estados y las aristas dirigidos las transiciones.

Para representar los grafos de las máquinas de Mealy, se identifican las aristas con etiquetas de tipo $e/s$, donde en cada $e \in \Sigma_E$ representa un símbolo de entrada y $s \in \Sigma_S$ un símbolo de salida.




La máquina de Mealy es una máquina traductora, lo que significa que establece una relación entre una cadena de entrada y la cadena de salida. Se podría usar por ejemplo para traducir un lenguaje encriptado en el que a cada letra del alfabeto le corresponde otra distinta.

## 4.2. Máquina de Moore

La máquina de Moore tiene los mismos cinco componentes ya indicados para la la máquina de Mealy, solo que ahora la función de salida depende únicamente del estado actual y no de la entrada en ese instante.

$$MO = (\Sigma_E, \Sigma_S, Q, f, g)$$

donde la función de transición $f$ no cambia y en $g$ hay una relación directa entre el estado en cada intervalo de tiempo y el símbolo de salida:

$$f : Q \times \Sigma_E \to Q \qquad g : Q \to \Sigma_S$$

En la máquina de Moore exite un retardo entre la entrada y la salida ya que, cuando ingresa una entrada, la máquina primero debe cambiar de estado para producir una salida.

Puede demostrarse que para toda máquina de Moore hay una máquina de mealy capaz de tener el mismo desempeaño y recíprocamente.

## 4.3. Autómatas Finitos Deterministas (AFD)

Si a la máquina de Mealy le incorporamos un estado inicial y un conjunto de estados de aceptación, tenemos un **Autómata Finito Determinista (AFD)**.

Un AFD entonces, es una séptupla que comienza a operar a partir de un estado inicial, transforma cadenas de entrada en cadenas de salida y completa su operación al terminar de leer su entrada, llegando a un estado que puede o no ser de aceptación. Un AFD puede ser traductor ($AFD_T$) o

reconocedor ($AFD_R$) de cadenas. Se define:

$$AFD_T = (\Sigma_E, \Sigma_S, Q, q_0, A, f, g)$$

donde

| | |
|---|---|
| $\Sigma_E$ : Alfabeto de símbolos de entrada | |
| $\Sigma_S$ : Alfabeto de símbolos de salida | |
| $Q$ : Conjunto finito y no vacío de estados posibles | |
| $q_0$ : Estado inicial de operación | $q_0 \in Q$ |
| $A$ : Conjunto de estados de aceptación, | $A \subseteq Q$ |
| $f$ : Función de transición, | $f : Q \times \Sigma_E \to Q$ |
| $g$ : Función de salida, | $g : Q \times \Sigma_E \to \Sigma_S$ |

Al tener un estado inicial y ser su cadena de entrada finita, el $AFD_T$ siempre completa su operación de la misma forma, en una cantidad finita de tiempo y, por lo tanto, *determina un algoritmo*.

Si el $AFD_T$ se limita a reconocer o validar cadenas, su alfabeto de salida y su función de salida no tienen sentido. En este caso tenemos un Autómata Finito Determinista Reconocedor ($AFD_R$), que queda definido como una quíntupla:

$$AFD_R = (\Sigma_E, Q, q_0, A, f)$$

Un autómata finito es determinista cuando el componente $f$ del mismo es una función.

Para el estudio de los $AFD$ se tiene en cuenta que la máquina es capaz de saber cuándo terminó de leer la cinta de entrada. Además, si se terminó de leer la cadena el autómata puede encontrarse en un estado de aceptación ($q \in A$), y en dicho caso la cadena es aceptada o reconocida; o puede encontrarse en un estado de no aceptación ($q \notin A$), lo que indica que la cadena es desconocida o rechazada.

### 4.3.1. Características de los AFD

Los autómatas finitos leen siempre la cadena de entrada de izquierda a derecha. Esto tiene varias consecuencias:

- Cada símbolo de la cadena es leído una única vez.
- Al completarse la lectura, la cadena es aceptada o no, según se alcance un estado de aceptación.
- La cantidad de intervalos de tiempo necesarios para evaluar la cadea es igual a su largo.
- En todo momento está claramente definida la subcadena pendiente de ser leída.

### 4.3.2. Configuración o descripción instantánea

Se define como configuración o descripción instantánea $K_t$ de un autómata finito en un intervalo de tiempo $t$ al par ordenado:

$$K_t = (q, \beta) \quad , q \in Q, \beta \in \Sigma_E^*$$

donde $q$ representa el estado en el que se encuentra y $\beta$ el sufijo o subacadena de entrada que está pendiente de ser leída. A partir de esta definición, y ante una cadena de entrada $\alpha$ a ser procesada, se puede reconocer la configuración inicial como:

$$K_0 = (q_0, \alpha)$$

De igual forma se define al configuración de aceptación como:

$$K_n = (q_n, \lambda) \quad , q_n \in A, n = |\alpha|$$

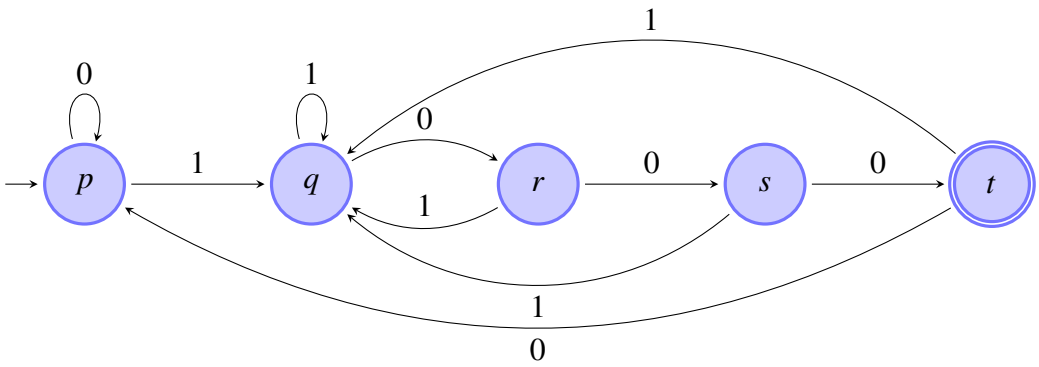
El tránsito de una configuración a otra es denominado movimiento. Si existe la transición $q = f(p,a)$, el movimiento del estado $p$ al $q$ leyendo el símbolo $a$ de la entrada se puede representar así:

$$(p, a\beta) \mapsto (q, \beta)$$

El movimiento desde la configuración inicial a la final es representado:

$$(q_0, \alpha) \mapsto^* (q_n, \lambda)$$

Ejemplo: El siguiente $AFD$ está destinado a reconocer cadenas que respondan a la forma general $\alpha = (0+1)^*1000$:



Podemos identificar la definición formal del siguiente $AFD$ viendo su gráfico:

$$\begin{aligned} AFD &= (\Sigma_E, Q, q_0, A, f) \\ \Sigma_E &= \{0,1\} \\ Q &= \{p,q,r,s,t\} \\ q_0 &= p \quad (\text{indicado con } \rightarrow) \\ A &= \{t\} \quad (\text{indicado con } \circledcirc) \end{aligned}$$

Tomemos las dos siguientes cadenas $\beta = 10110000$ y $\delta = 1011100$. La primera responde a la expresión regular definida $(\alpha)$ y la segunda cadena propuesta no cumple con la condición requerida.

Para estas cadenas podemos ver su árbol de descripción instantánea para comprobar que la cadena $\delta$ no llega a un estado de aceptación y por eso es rechazada. Además, podemos ver que los árboles son lineales por tratarse de un $AFD$, ya que para cada estado y símbolo leído existe una única transición posible.

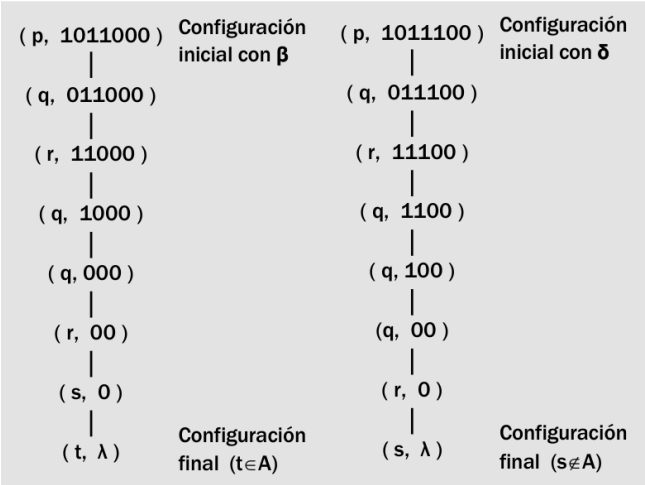



Figura 7: Árboles de descripciones instantáneas

### 4.3.3. Función de transición extendida

La función de transición $f$ define el próximo estado en un $AFD$ para cada estado actual y ante la entrada de cada posible símbolo. La función de transición extendida $f^e : Q \times \Sigma_E^* \rightarrow Q$ recibe una cadena en vez de un símbolo, y devuelve el último estado en el que se encontrará el $AFD$ luego de haber leído y haber pasado por cada estado para cada una de las letras de la cadena. Supongamos que $\alpha = abcde$ pertenece al lenguaje reconocido por el $AFD$, podemos definir un proceso recursivo que finaliza al completarse la lectura de $\alpha$:

$$f^e(q,\alpha) = f(f(f(f(f(q,a),b),c),d),e)$$

### 4.3.4. Aceptación de cadenas y otros conceptos asociados a los AFD

Se dice que un lenguaje $L$ es aceptado por un autómata finito $M$ si todas las palabras que lo componene conducen al autómata a estados de aceptación. En símbolos:

$$L(M) = \{\alpha / \alpha \in \Sigma_E^* \wedge f^e(q_0, \alpha) \in A\}$$

### 4.3.5. Accesibilidad entre estados

En un $AFD$, se dice que un estado $p$ es accesible desde otro estado $q$, y se representa $pAq$, si existe una palabra de símbolos de entrada que haga transitar al autómata desde el estado $q$ hasta el estado $p$. En símbolos:

$$pAq \iff \exists \alpha \in \Sigma_E^* / f^e(q, \alpha) = p \qquad , p, q \in Q$$

Se lee como *p es accesible desde q sí y solo sí existe una cadena* $\alpha$ *formada por símbolos del alfabeto de entrada tal que, partiendo desde q y usando una función de transición extendida sobre* $\alpha$*, obtenga como resultado p, siendo p y q dos estados del conjunto de estados del autómata.*

### 4.3.6. Autómatas conexos

Si todos los estados de un autómata son accesibles desde su estado inicial se dice que el mismo es *conexo*.

### 4.3.7. Equivalencia entre estados

Dos estados $p, q \in Q$ de un $AFD$ son equivalentes, y se representa $pEq$, si desde cualquiera de llos se aceptan y rechazan exactamente las mismas cadenas de símbolos de entrada.

$$pEq \iff [\forall \alpha \in \Sigma_E^* : f^e(p, \alpha) \in A \iff f^e(q, \alpha) \in A]$$

### 4.3.8. Equivalencia de longitud entre estados

Dos estados $p, q \in Q$ de un $AFD$ son equivalentes de longitud $k$, que se representa como $pEkq$, si son equivalentes para todas las cadenas de largo menor o igual $k$.

$$pEq \iff [\forall \alpha \in \Sigma_E^* : |\alpha| \leq k \wedge f^e(p, \alpha) \in A \iff f^e(q, \alpha) \in A]$$

### 4.3.9. Equivalencia entre autómatas

Dos $AFD$ $A_1$ y $A_2$ son equivalentes, y se representa $A_1 E A_2$ si ambos aceptan las mismas palabras o sentencias.

$$A_1 E A_2 \iff L(A_1) = L(A_2)$$

### 4.3.10. Conjunto cociente

Como toda relación de equivalencía, la relación $E$ definida sobre el conjunto de estados posibles $Q$, induce en el mismo una partición. Todos los elementos de $Q$ relacionados con cierto estado $p \in Q$, constituyen un subconjunto de $Q$ que es denominado clase de equivalencia de $p$. El conjunto cociente es esa partición y se denota $Q/E$.

La definición de equivalencia entre estados no sirve para saber si dos estados $p,q \in Q$ son equivalentes, ya que hay infinitas cadenas $\alpha \in \Sigma_E^*$. Sin embargo, con el concepto de equivalencia de longitud $k$ entre estados, es teóricamente posible saber si dos estados $p,q \in Q$ son $k$-equivalentes ya que hay un número finito de cadenas $\alpha \in \Sigma_E^*$ con $|\alpha| \leq k$.

Para encontrar el conjunto cociente podemos usar un procedimiento que consiste en ir buscando equivalencias de longitud $k$ entre estados comenzando con $k = 0$ hasta que las clases de equivalencia de $Q$ se estabilicen.

En el caso de $AFD_R$, el conjunto cociente con $k = 0$ es:

$$Q/E_0 = P_1^0 \cup P_2^0 \qquad , P_1^0 = Q - A, P_2^0 = A$$

Los siguientes conjuntos cocientes $Q/E_{k+1}$ se definen dividiendo las clases $P_i^k$ de $Q/E_k$ las veces que sea necesario hasta que, para todo símbolo $a \in \Sigma_E$ y para todo par de estados $p,q$ de las clases $P_i^k$, queden definidos movimientos que conduzcan a elementos de las mismas clases $Q/E_{k+1}$. Esto se repite hasta obtener $Q/E_{k+1} = Q/E_k$.

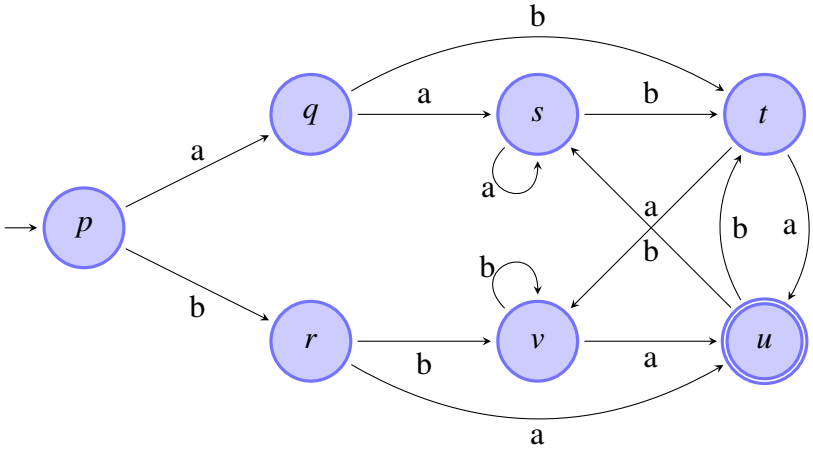
### 4.3.11. Minimización de autómatas

Un autómata mínimo es un autómata conexo que cumple correctamente su función con la menor cantidad posible de estados.

Para minimizar un autómata usamos el concepto del conjunto cociente. El autómata mínimo equivalente a uno dado tendrá como estados a cada una de las clases de estados equivalentes del autómata original.

Ejemplo:

Se desea reconocer cadenas que responden al patrón $(a+b)^*ba$ y para ello se propone el siguiente $AFD$:




La idea es ahora minimizar este autómata.

Vemos que es conexo, así que podemos comenzar con la minimización, buscando las particiones.

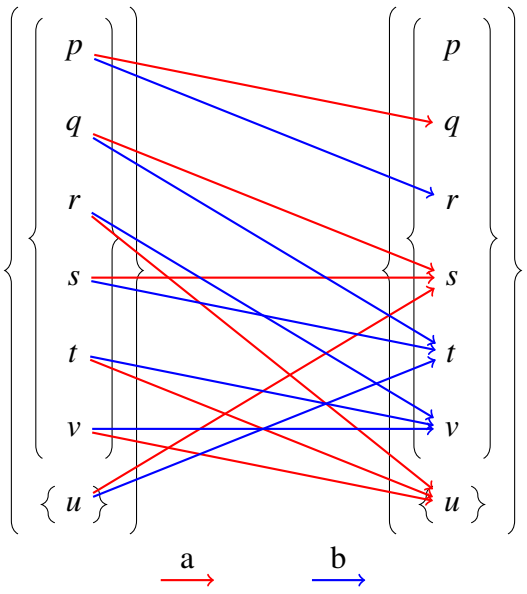
La primera partición del conjunto de estados se obtiene separando a los estados de aceptación del resto, que sería el conjunto cociente para los estados de equivalencia de longitud $k = 0$ (todas las cadenas reconocidas por el autómata llegan a los estados de aceptación, mientras que las no reconocidas no llegan):

$$Q/E_0 = \{\{p,q,r,s,t,v\},\{u\}\}$$

Lo que podemos hacer ahora es ir nombrando a los subconjuntos de las clases de equivalencia de longitud $k_n$ que vayamos obteniendo para facilitar el proceso de busqueda del conjunto cociente. Entonces podemos decir que para $k = 0$ tenemos dos subconjuntos $A = \{p,q,r,s,t,v\}$ y $B = \{u\}$ y juntos forman el conjunto cociente para $k = 0$:

$$Q/E_0 = \{A,B\} = \{\{p,q,r,s,t,v\},\{u\}\}$$

En el siguiente esquema se puede ver gráficamente el proceso de redistribución de los estados equivalentes en los subconjuntos existentes y la incorporación de otros subconjuntos nuevos:



Vemos en el esquema que corresponde a $k = 1$, que los estados $p$, $q$ y $s$ tienen el mismo comportamiento ya que permanecen los 3 en la clase de equivalencia que se encontraban para todos los símbolos del alfabeto de entrada. Es decir, que no importa el símbolo de entrada que reciba el autómata, los estados $p$, $q$ y $s$ siguen perteneciendo a la clase de equivalencia que habíamos nombrado como $A$.

Por el contraio, los estados $r$, $t$ y $v$ tienen una conducta común pero diferente a la de los otros estados: ante la entrada $a$ transitan a la clase de equivalencia $B$, mientras que con la entrada $b$ permanecen en la misma clase $A$. Esto significa que los seis estados de la primera clase no son todos equivalentes con $k = 1$, por lo que deben ser agrupados en dos subonjuntos diferentes:

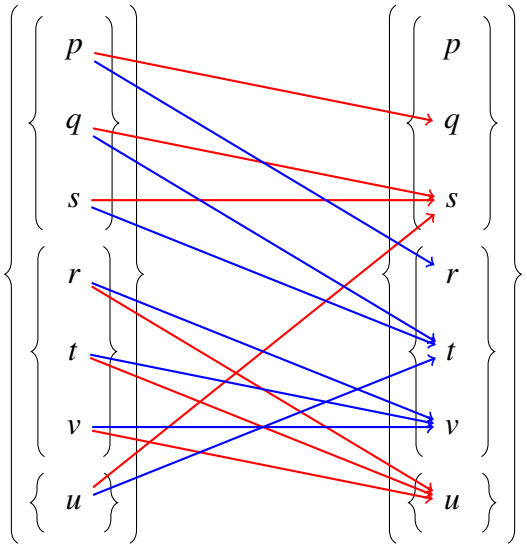
$$Q/E_1 = \{\{p,q,s\},\{r,t,v\},\{u\}\}$$

Otra vez, conviene nombrar a los subconjuntos para que el proceso sea mas facil. De esta forma, podemos nombrar un subconjunto $A = \{p,q,s\}$, otro $B = \{r,t,v\}$ y por último $C = \{u\}$, de manera que:

$$Q/E_1 = \{A,B,C\} = \{\{p,q,s\},\{r,t,v\},\{u\}\}$$

La segunda clase de $Q/E_0$ tiene un solo estado $u$ por lo que no es necesario un análisis.

El siguiente paso es repetir el procedimiento cuando se lee un nuevo símbolo, lo que implica buscar las clases de equivalencia para $k = 2$. El esquema ahora es el siguiente:




Ahora podemos ver que al analizar las clases de equivalencia para $k = 2$ los estados agrupados en cada subconjunto mantienen su equivalencia entre sí:

- Para la entrada $a$, los estados $p$, $q$ y $s$ transitan a la clase $B$. Y para la entrada $b$, se quedan en la clase $A$.
- Para la entrada $a$, los estados $r$, $t$ y $v$ transitan a la clase $C$. Y para la entrada $b$, se quedan en la clase $B$.

- En el conjunto $C$ solo hay un elemento por lo que no es necesario su análisis.

Esto significa que el conjunto cociente se estabilizó y no va a cambiar si seguimos analizando $k$ mayores a 2. De esta manera, el conjunto cociente $Q/E$ es el que acabamos de encontrar:

$$Q/E_2 = \{\{p,q,s\},\{r,t,v\},\{u\}\} = Q/E_1 = Q/E$$

Y si usamos los nombres que le habíamos asignado a cada subconjunto, podemos decir que:

$$Q/E = \{A,B,C\}$$

Acá me confundí: No debería haber usado la letra $A$ para nombrar un subconjunto ya que vamos a ver que al definir al autómata formalmente, este se puede confundir con el estado de aceptación que también tiene por nombre $A$. No lo voy a cambiar porque ya me dió paja, y creo que igual se entiende lo que hicimos y tampoco es tan confusa la definición formal siguiente. Solo hay que tener en cuenta que el $A \in Q$ no es el estado de aceptación y son dos cosas distintas.

El $AFD$ mínimo tiene la siguiente definición formal:

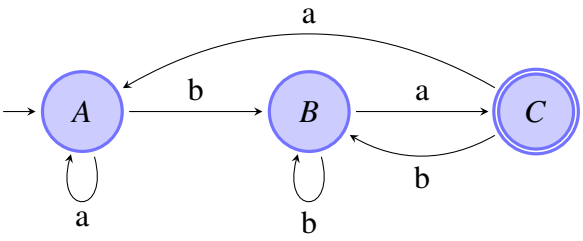
$$AFD = (\Sigma_E, Q, q_0, A, f)$$

donde:

$$\begin{aligned}
\Sigma_E &= \{a,b\} \\
Q &= \{A,B,C\} \\
q_0 &= A \\
A &= C \\
f &= \{((A,a),A),((A,b),B),((B,a),C),((B,b),B),((C,a),A),((C,b),B)\}
\end{aligned}$$

Sabemos que el estado $q_0 = A$ porque el autómata original empezaba en el estado $p \in A$. También sabemos que el estado de aceptación es $A = C$ porque el autómata original terminaba en el estado $u \in C$.

El dígrafo del $AFD$ es el siguiente:



## 4.4. Autómatatas Finitos Bidireccionales

Los autómatas finitos leen siempre la cadena de entrada de izquierda a derecha. Esto tiene varias consecuencias:

- Cada símbolo de la cadena es leído una única vez.
- Al completarse la lectura, la cadena es aceptada o no, según se alcance un estado de aceptación.
- La cantidad de intervalos de tiempo necesarios para evaluar la cadea es igual a su largo.
- En todo momento está claramente definida la subcadena pendiente de ser leída.

Si le otorgamos a un $AFD$ la capacidad de decidir el sentido del movimiento del cabezal en cada intervalo de tiempo tenemos un **Autómata Finito Determinista Bidireccional (AFDB)**. A la definición del autómata entonces, se le debe incorporar una *función de movimiento* que defina el sentido del movimiento del cabezal en cada estado y ante cada símbolo de entrada. También hay que definir el sector de la cinta en el que el autómata debe operar, y para esto se incorporan dos símbolos especiales que indican el inicio y el fin del intervalo de la cinta de entrada a ser analizado.

### 4.4.1. Características del AFDB

- Cada símbolo de entrada puede ser leído varias veces.
- No hay ninguna condición que indique el fin de la lectura de la cadena.
- No es posible anticipar la cantidad de intervalos de tiempo requeridos para evaluar la cadena.
- El concepto de cadena que resta de ser leída desaparece.

### 4.4.2. Definición de un AFDB

El $AFDB$ se define como:

$$AFDB = (\Sigma_E, \Gamma, Q, q_0, A, f)$$

donde:

$\Sigma_E$ : Alfabeto de símbolos de entrada
$\Gamma$ : Alfabeto de cinta, $\Gamma = \Sigma_E \cup \{\vdash, \dashv\}$
$Q$ : Conjunto finito, no vacío, de estados posibles
$q_0$ : Estado inicial de operación, $q_0 \in Q$
$A$ : Conjunto de estados de aceptación, $A \subseteq Q$
$f$ : Función de transición, $f : Q \times \Gamma \to Q \times \{I, N, D\}$

donde $I, N, D$ significan izquierda, neutro y derecha respectivamente.

### 4.4.3. Aceptación de cadenas

Una cadena es aceptada por el $AFDB$ cuando éste ha alcanzado un estado de aceptación y la cadena ha sido leída por lo menos una vez. En otras palabras, la cantidad de intervalos de tiempo requeridos para la aceptación de una cadena es igual o mayor que su largo.

### 4.4.4. Configuración o descripción instantánea

Para definir la condición en la que se encuentra un $AFDB$ en un instante dado se requieren tres componentes:

- El estado actual.
- El contenido de la cinta de entrada.
- La posición del cabezal.

Para la posición del cabezal la convención indica que el símbolo de inicio de cinta $\vdash$ está en la posición 0, y el contador se incrementa hacia la derecha, lo que implica que el primer caracter de la cadena de entrada está en la posición 1.

Según este criterio, la configuración $K_t$ de un $AFDB$ que opera sobre una cierta cadena de entrada $\alpha$, que en el instante $t$ está en el estado $q$ y con el cabezal en la posición $k$ es:

$$K_t = (q, \vdash \alpha \dashv, k)$$

Esto nos permite saber el estado en el que se encuentra el autómata en un tiempo $t$, vemos la cadena que tiene que procesar el autómata, y con la posición del cabezal vemos qué símbolo de esa cadena está procesando el autómata.

La configuración inicial es entonces:

$$K_0 = (q_0, \vdash \alpha \dashv, 0)$$

y la configuración final de aceptación es:

$$K_A = (q_A, \vdash \alpha \dashv, n) \quad , q_A \in A, n = |\alpha| + 1$$

Otra forma de representar la configuración de un $AFDB$ es:

$$K_t = \delta q \beta$$

Donde $q$ es el estado actual del atómata, $\delta$ el prefijo de la cadena de entrada $\alpha$ que antecede al cabezal y $\beta$ el sufijo que sigue al cabezal ($\alpha = \delta\beta$). Según este criterio, la configuración inicial es $K_0 = q_0\alpha$ y la final es $K_A = \alpha q_A$.

Por ejemplo, si $\alpha = abcdef$ y el autómata leyó el caracter $b$, la configuración sería $K_2 = abq_0cdef$.

### 4.4.5. Lenguaje reconocido por el AFDB

Se puede definir el lenguaje $L$ reconocido por el $AFDB$ $M$ como:

$$L(M) = \{\alpha / \alpha \in \Sigma_E^* \wedge q_0\alpha \mapsto^* \alpha q_A \quad , q_a \in A\}$$

Que se lee como *el lenguaje L reconocido por el autómata M son todas las cadenas que se puedan formar con el alfabeto de entrada que, empezando desde el estado inicial, se puedan procesar para llegar al estado de aceptación*.

### 4.4.6. Equivalencia entre AFDB y AFD

El movimiento del caberzal en dos sentidos no brinda ninguna capacidad adicional con respecto al $AFD$, lo que implicax que todo $AFDB$ tiene un $AFD$ equivalente.

# 5. Autómata Finito No Determinista (AFND)

Como ya habíamos dicho, un autómata finito deja de ser determinista cuando su función de transición no es una función. Es decir, si un $AF$ en un cierto estado y ante una cierta señal de entrada admite multiples estados, este es un **Autómata Finito No Determinista (AFND)**. Al igual que en el caso de los $AFD$, los $AFND$ quedan definidos como una quíntupla:

$$AFND = (\Sigma_E, Q, q_0, A, f)$$

solo que en este caso, la función de transición es en realidad una relación de transición y queda definida de la siguiente forma:

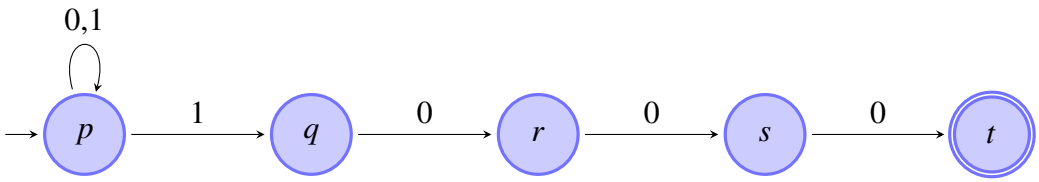
$$f : Q \times \Sigma_E \to \mathscr{P}(Q)$$

donde $\mathscr{P}(Q)$ es el conjunto potencia de $Q$.

En los $AFND$ es díficil saber cuánto tiempo va a llevar el procesamiento de una cadena debido a que para cada estado pueden existir múltiples caminos a tomar. El tiempo demandado para resolver este tipo de problemas es exponencial, lo que implica un esfuerzo muy considerable incluso para casos simples.

Por otro lado, el diseño de estos autómatas es mas sencillo que el de los $AFD$ por lo que el no determinismo de los autómatas resulta de gran ayuda para el diseño de autómatas destinados a reconocer lenguajes complejos, pagando el costo de un mayor esfuerzo de operación.

Ejemplo: Analicemos el siguiente $AFND$ destinado a reconocer cadenas del siguiente tipo $\alpha = (0+1)^*1000$.




Podemos ver que este autómata es no determinista ya que en el estado $p$, para la entrada 1 se puede ir a mas de un estado.

Si le pasamos al $AFND$ la cadena $\beta = 1011000$ puede comprobarse que el éste acepta la cadena al arribar al estado $t$, pero para ello tuvo que explorar varias opciones que podemos ver en las siguientes figuras que corresponden al *árbol de descripciones instantáneas* y al *plano estados-entradas*.

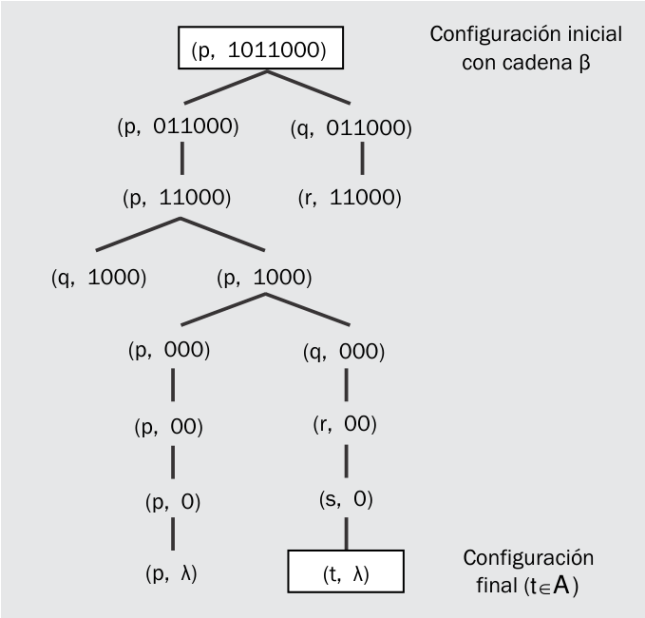



Figura 8: Árbol de descripciones instantáneas.

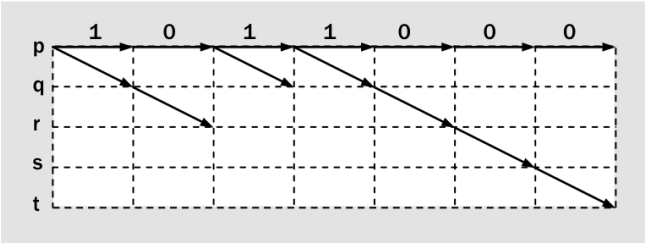



Figura 9: Representación del plano estados-entradas.

Podemos ver que en 2 oportunidades el $AFND$ no pudo continuar operando con la cadena entrada ya que no existían transiciones que lo permitan cuando lee 1 en los estados $q$ o $r$. También vemos que en otras dos oportunidades el autómata sí leyó la cadena completa, pero en un caso llegó al estado $p$ que no es de aceptación, y en el otro llegó al estado $t$ que sí es de aceptación y es esta la razón por la que la cadena fue aceptada.

## 5.1. Transiciónes Lambda

La definición de los *AFND* puede ampliarse para incluir transiciones de un estado a otro que no dependan de ninguna entrada, y son denominadas **transiciones** $\lambda$. Estos autómatas se denominan $AFND-\lambda$.

En estos casos, $\lambda$ pasa a ser un nuevo símbolo del alfabeto de entrada. De esta manera, hay que reescribir la relación de transición $f$ para incluir el símbolo $\lambda$ al alfabeto de entrada y considerar los pares $(q,\lambda)$. Ademas se asume que cada estado tiene su propia transición $\lambda$ que cicla sobre sí mismo:

$$f : Q \times (\Sigma_E \cup \{\lambda\}) \to \mathscr{P}(Q)$$

A todos los pares de estados que están relacionados por una transición $\lambda$ los podemos agrupar en un conjunto $T$ denomindao *relación de transición* $\lambda$.

$$T = \underbrace{\{(p,q)/q \in f(p,\lambda)\}}_{\text{Todos los } p \text{ y } q \text{ relacionados con } \lambda} \cup \underbrace{\{(q,q)/q \in Q\}}_{\text{Todos los } q \text{ tienen su ciclo con } \lambda}$$

Luego se incorporan al conjunto $T$ TODOS los pares ordenados que se originan de la propiedad transitiva de los pares anteriores:

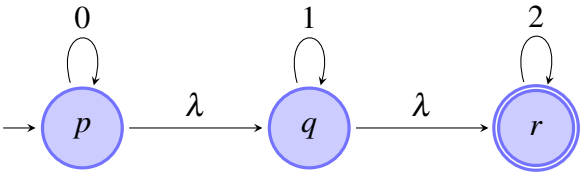
$$T^* = T \cup \{(p,r)/pTq \wedge qTr\}$$

## 5.2. Lambda-clausura

Se denomina $\lambda\text{–}clausura(p)$ al conjunto de todos los estados $q \in Q$ que están relacionados con $p$ por medio del cierre transitivo de de la relación de transición $\lambda$ ($T^*$). En símbolos:

$$\lambda\text{–}clausura(p) = \{q \in Q/(p,q) \in T^*\}$$

Ejemplo: Supongamos que tenemos el siguiente $AFND\text{–}\lambda$ y queremos obtener la $\lambda\text{–}clausura$ de todos sus estados:




Para armar la $\lambda\text{–}clausura$ necesitamos tener $T^*$, pero para eso necesitamos primero obtener $T$, que son todas las tuplas $(p,q)$ relacionadas por $\lambda$ y todas las tuplas $(q,q)$ ya que siempre cada estado se puede relacionar con sigo mismo por medio de $\lambda$.

$$T = \{(p,q),(q,r),(p,p),(q,q),(r,r)\}$$

Ahora, a $T$ le agregamos todos los pares ordenados que originan de la propiedad transitiva de los pares ordenados de $T$:

$$T^* = \{(p,q),(q,r),(p,p),(q,q),(r,r),(p,r)\}$$

Y ahora sí, podemos buscar la $\lambda$–*clausura* para cada uno de los estados del $AFND$–$\lambda$:

$$\lambda\text{–}clausura(p) = \{p,q,r\}$$
$$\lambda\text{–}clausura(q) = \{r,q\}$$
$$\lambda\text{–}clausura(r) = \{r\}$$

## 5.3. Equivalencia con autómatas finitos deterministas

Para la equivalencia entre autómatas fínitos existen dos teoremas.

### 5.3.1. Teorema 1: AFND-lambda a AFND

Sea un $AFND-\lambda$ que es definido como $M = (\Sigma_E, Q, q_0, A, f)$ donde

$$f : Q \times (\Sigma_E \cup \{\lambda\}) \to \mathscr{P}(Q)$$

Este autómata puede siempre ser redefinido como un $AFND$ en el que $M' = (\Sigma_E, Q, q_0, A', f')$ donde $f' : Q \times \Sigma_E \to \mathscr{P}(Q)$ siguiendo los siguientes pasos:

a) Obtenemos la $\lambda$–$clausura(q), \forall q \in Q$ de todos los estados $q$ del autómata.

b) Para cada estado $q \in Q$, armamos un conjunto con $f(p,a)$, siendo $p \in \lambda$–$clausura(q)$ cada estado de la $\lambda$–$clausura(q)$ y $a \in \Sigma_E$ un símbolo del alfabeto de entrada. También unimos a este conjunto, el conjunto $\lambda$–$clausura(p)$ para todo $p$ cuya relación $f(p,a)$ esté definida.

   El conjunto para cada estado $q \in Q$ quedaría así:

$$q_a = \{f(p,a)\} \cup \{f(p,a) \in Q \implies \lambda\text{–}clausura(p)\}$$

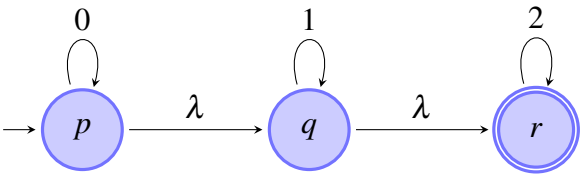
c) Repetimos el paso anterior para cada símbolo $a \in \Sigma_E$ del alfabeto de entrada.

d) Si en $\lambda$–$clausura(q_0)$ se encuentra algún estado final $q_A$, entonces el conjunto de estados finales del nuevo autómata es $A \cup \{q_0\}$, de lo contrario el conjunto de estados finales $A$ no se vé modificado.

e) El estado inicial del nuevo autómata es el mismo que el estado inicial del $AFND$–$\lambda$.

f) Reescribimos el autómata usando los conjuntos que armamos recién como si fuesen las relaciones de transición.

Ejemplo: Queremos convertir el $AFND$–$\lambda$ del ejemplo anterior en un $AFND$ equivalente:




a) Ya habíamos definido los $\lambda-clausura$ de cada estado:

$$\lambda-clausura(p) = \{p,q,r\}$$
$$\lambda-clausura(q) = \{r,q\}$$
$$\lambda-clausura(r) = \{r\}$$

b)

$$p_0 = \{p\} \cup \{p,q,r\} = \{p,q,r\}$$
$$q_0 = \varnothing$$
$$r_0 = \varnothing$$

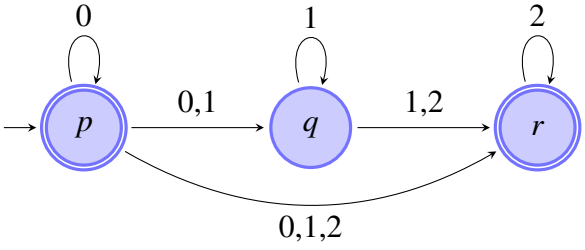
c) Repetimos el punto anterior para el resto de símbolos del afabeto de entrada:

$$p_1 = \{q\} \cup \{q,r\} = \{q,r\}$$
$$q_1 = \{q\} \cup \{q,r\} = \{q,r\}$$
$$r_1 = \varnothing$$
$$p_2 = \{r\} \cup \{r\} = \{r\}$$
$$q_2 = \{r\} \cup \{r\} = \{r\}$$
$$r_2 = \{r\} \cup \{r\} = \{r\}$$

d) $q,r,s \in A$ son los tres un estado de aceptación ya que todos tienen a $r$ en su $\lambda-clausura$.

e) El estado inicial sigue siendo $p$.

f) Reescribimos el autómata como un $AFND$:



### 5.3.2. Teorema 2: AFND a AFD

Sea un $AFND$ definido como $M = (\Sigma_E, Q, q_0, A, f)$ donde

$$f : Q \times \Sigma_E \to \mathscr{P}(Q)$$

Siempre se puede encontrar un $AFD$ $M' = (\Sigma_E, Q', c_0, A', f')$ con $f : Q \times \Sigma_E \to Q'$ que es equivalente siguiendo estos pasos:

a) Reconocemos el estado inicial del $AFND$ y lo nombramos como $c_0$.

b) Por cada símbolo $a \in \Sigma_E$ buscamos un nuevo conjunto $c_{i+1} = f(q,a)$ siendo $q$ todos los estados que pertenecen a $c_i$.

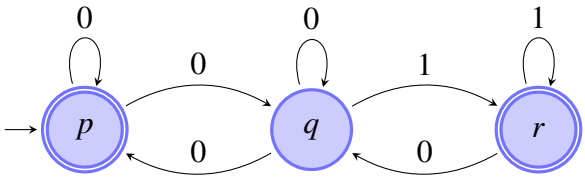
c) Repetimos el paso anterior hasta que no encontremos mas conjuntos nuevos.

d) El estado inicial del $AFD$ equivalente es el mismo que el $AFND$.

e) El conjunto de estados finales del $AFD$ es el conjunto de estados finales $A$ unión todos los $c_i$ que contengan al menos un estado final en ellos. En símbolos: $A' = c_i \in Q'/c_i \cap A \neq \varnothing$.

f) Podemos graficar el $AFD$ equivalente sabiendo que cada $c_i$ encontrado es un estado del mismo.

Ejemplo: Queremos convertir el siguiente $AFND$ en un $AFD$ equivalente:




a) El estado inicial es:

$$c_0 = \{p\}$$

b) Buscamos nuevos conjuntos $c_i$:

| | |
|---|---|
| $f(c_0, 0) = \{p, q\} = c_1$ | Le asignamos un nombre porque encontramos un conjunto nuevo |
| $f(c_0, 1) = \varnothing$ | Como es vacio no hacemos nada |

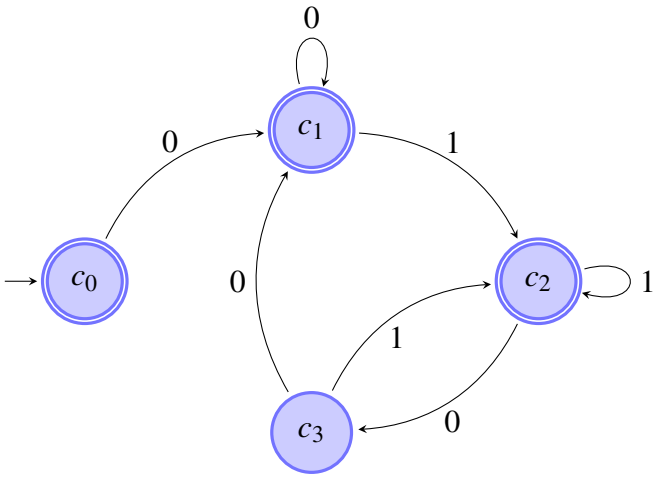
c) En el paso anterior encontramos al menos un conjunto nuevo, así que analizamos ese conjunto:

| | |
|---|---|
| $f(c_1,0) = \{p,q\} \cup \{p,q\}$ | Es igual a $c_1$ así que no hacemos nada |
| $f(c_1,1) = \varnothing \cup \{r\} = c_2$ | Le asignamos un nombre porque es un conjunto nuevo |
| $f(c_2,0) = \{q\} = c_3$ | Le asignamos un nombre porque es un conjunto nuevo |
| $f(c_2,1) = \{r\}$ | Es igual a $c_2$ así que no hacemos nada |
| $f(c_3,0) = \{p,q\}$ | Es igual a $c_1$ así que no hacemos nada |
| $f(c_3,1) = \{r\}$ | Es igual a $c_2$ así que no hacemos nada |

d) Ya no encontramos mas estados nuevos así que no repetimos mas. El estado inicial de nuestro $AFD$ es $c_0$.

e) Los estados finales del $AFND$ eran $p$ y $r$ así que todos los $c_i$ que contengan a $p$ o $r$ van a ser estados finales del $AFD$. En este caso los estados finales son: $c_0$, $c_1$ y $c_2$

f) Ahora podemos graficar nuestro $AFD$:




## 5.4. Gramáticas regulares y autómatas finitos

Hasta ahora veníamos creando los autómatas a base de prueba y error. Pero, en realidad es posible crearlos sistemáticamente a partir de reglas de producción de las gramáticas. A su vez, también es posible definir las reglas de producción capaces de generar lenguajes a ser reconocidos por ciertos autómatas.

### 5.4.1. Definición de gramáticas regulares a partir de autómatas

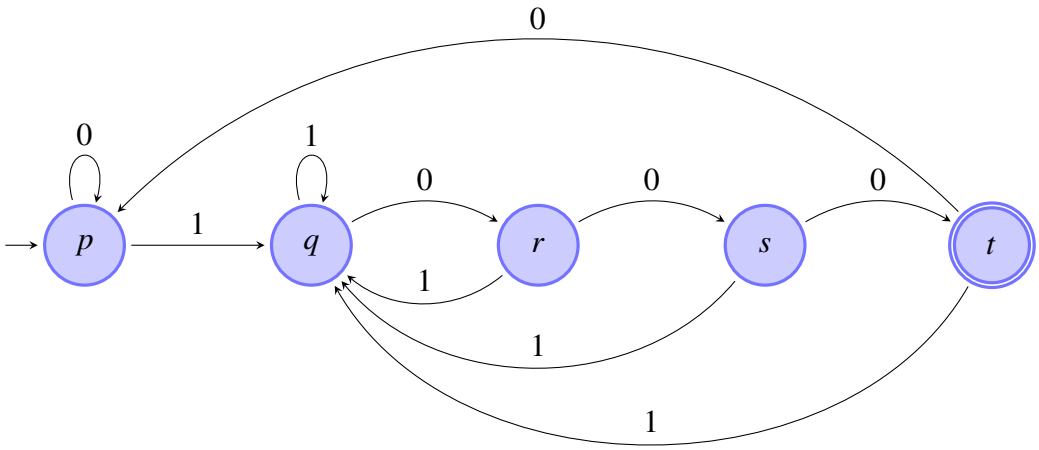
La idea es definir el conjunto de reglas de producción de una gramática regular que generará lenguajes que están destinados a ser reconocidos por un cierto $AFD$. La gramática que vamos a obtener va a estar bien formada y será lineal por derecha.

Dado un $AFD = (\Sigma, Q, q_0, A, f)$, la gramática que genera el mismo lenguaje que acepta el autómata sera $G = (\Sigma, Q, q_0, P)$, donde el conjunto de reglas de reescritura queda definido como:

$$P = \{X := aY / f(X,a) = Y\} \cup \{X := a / f(X,a) = Y, Y \in A\}$$

y se agrega $q_0 := \lambda$ si $q_0 \in A$; $\quad X, Y, q_0 \in Q, a \in \Sigma, A \subseteq Q$

Ejemplo: Determinar la gramática bien formada capaz de generar las sentencias del lenguaje reconocido por el siguiente $AFD$:




Convertimos las transiciones en reglas de producción según lo que se explica en el cuadro anterior:

$$\left.\begin{array}{r} f(p,0) = p \\ f(p,1) = q \\ f(q,0) = r \\ f(q,1) = q \\ f(r,0) = s \\ f(r,1) = q \\ f(s,0) = t \\ f(s,1) = q \\ f(t,0) = p \\ f(t,1) = q \end{array}\right\} \Longrightarrow \left\{\begin{array}{l} p := 0p \\ p := 1q \\ q := 0r \\ q := 1q \\ r := 0s \\ r := 1q \\ s := 0t \mid 0 \\ s := 1q \\ t := 0p \\ t := 1q \end{array}\right.$$

No hay que olvidarse de agregar los $X := a$ cuando $f(X,a) = B, B \in A$, como se hizo con el estado $s$.

El conjunto completo de las reglas de producción de la gramática es entonces:

$$P = \left\{ \begin{array}{l} p := 0p \mid 1q, \\ q := 0r \mid 1q, \\ r := 0s \mid 1q, \\ s := 0t \mid 1q \mid 0, \\ t := 0p \mid 1q \end{array} \right\}$$

Y la gramática que buscabamos es:

$$G = (\{0,1\}, \{p,q,r,s,t\}, p, P)$$

### 5.4.2. Definición de autómatas a partir de gramáticas regulares

En este caso ya conocemos la gramática, que estará bien formada y será lineal por derecha, y lo que queremos hacer es definir el autómata que es capaz de reconocer el lenguaje generado.

Dada la gramática regular $G = (\Sigma_T, \Sigma_N, S, P)$, el autómata finito que acepta el mismo lenguaje que genera la gramática será:

$$AF = (\Sigma_T, \Sigma_N \cup \{A\}, S, \{A\}, f)$$

donde $A$ es un nuevo símbolo que no estaba en $\Sigma_N$, $f(X,a) = Y$ si $X := aY$ está en $P$, y $f(X,a) = A$ si $X := a$ está en $P$, con $X, Y, S \in \Sigma_N, a \in \Sigma_T$.
La regla de producción $S := \lambda$ incropora la transición $f(S, \lambda) = A$.

**Nota**: Si la gramática es lineal por izquierda, es necesario convertirla previamente a lineal por derecha, aplicando el procedimiento adecuado.

Ejemplo: Definir en $AFD$ que reconozca el lenguaje generado por la siguiente gramática lineal por derecha:

$$G = (\{a,b\}, \{p,q,r,s,t\}, p, P)$$

donde

$$P = \left\{ \begin{array}{l} p := 0p \mid 1q, \\ q := 0r \mid 1q, \\ r := 0s \mid 1q, \\ s := 0t \mid 1q \mid 0, \\ t := 0p \mid 1q \end{array} \right\}$$

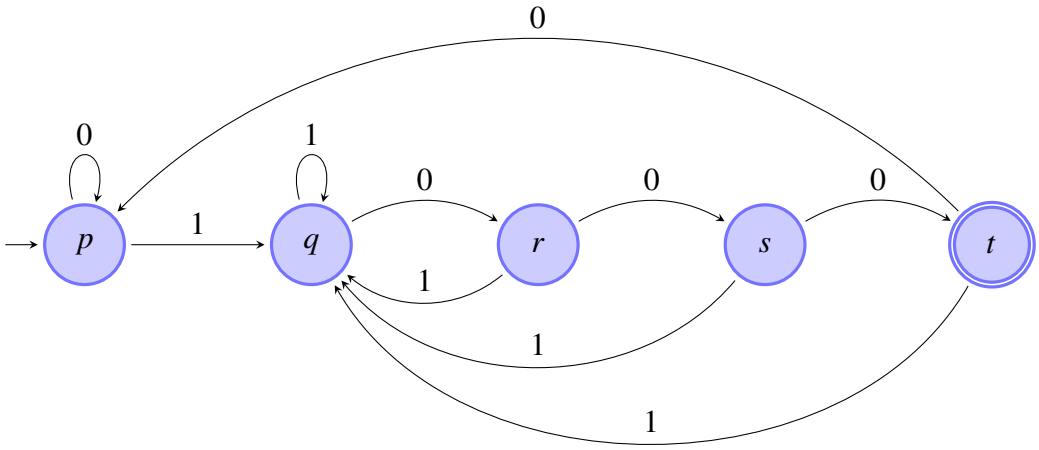
Según el procedimiento explicado para las gramáticas bien formadas, a cada producción $X := aY$

le corresponde una transición $f(X,a) = Y$. Entonces tenemos que:

$$\begin{aligned} f(p,0) &= p \quad f(p,1) = q \\ f(q,0) &= r \quad f(q,1) = q \\ f(r,0) &= s \quad f(r,1) = q \\ f(s,0) &= t \quad f(s,1) = q \\ f(t,0) &= p \quad f(t,1) = q \end{aligned}$$

En este caso, $p$ resulta ser el estado inicial $q_0 = p$ por ser $p$ el axioma. La regla de producción $s := 0$ indica que hay que agregar $f(s,0) = A$ y sabemos que $f(s,0) = t$, por lo tanto $t \in A$.

Con la información que tenemos podemos ya armar el autómata que es el siguiente:




### 5.4.3. Conversión de gramática lineal por izquierda a lineal por derecha

Para cada gramática lineal por izquierda existe una gramática lineal por derecha y viceversa. Para realizar esta conversión hay que seguir los siguientes cuatro pasos:

**Paso 1** : Si la gramática es recursiva en el axioma hay que convertirla a otra equivalente que no sea recursiva en el axioma. Para ello hay que incorporar un nuevo símbolo no terminal $B$ y modificar las reglas de producción de la siguiente manera:

- Por cada regla $S := \alpha$, se crea una nueva regla $B := \alpha$.
- Cada regla de la forma $A := Sa$ debe ser reemplazada por $A := Ba$, donde $S, A, B \in \Sigma_N, a \in \Sigma_T$ y $\alpha$ es una cadena válida.

**Paso 2** : Se debe construir un grafo con las siguientes instrucciones:

- Los nodos del grafo se identifican con los símbolos no terminales de la gramática y se incluye un nodo adicional identificado con $\lambda$.

- Por cada regla de la forma $A := Ba$, se dibuja una arista desde el nodo $A$ al nodo $B$ que es identificado con la etiqueta $a$.
- Por cada regla de la forma $A := a$, se dibuja una arista dese el nodo $A$ al nodo $\lambda$ identificado con $a$.
- Para la regla $S := \lambda$, se incorpora una arista sin etiqueta desde el nodo $S$ al nodo $\lambda$.

**Paso 3** : Se construye un nuevo grafo a partir del grafo anterior según se indica:

- Se intercambian los identificadores de los nodos $S$ y $\lambda$.
- Se cambia el sentido de todas las aristas.

**Paso 4** : A partir del nuevo grafo, se obtiene la gramática lineal por derecha siguiendo los siguientes pasos:

- Los alfabetos de símbolos terminales y no terminales son los mismo de la gramática original lineal por izquierda.
- Por cada arista dirigida etiquetada con $a$ desde el nodo $B$ al nodo $A$, se incorpora a la nueva gramática la regla de producción $B := aA$.
- Por cada arista dirigida etiquetada con $a$ desde el nodo $B$ al nodo $\lambda$, se incorpora la regla de producción $B; = a$.
- En caso de una arista sin etiqueta desde el nodo $S$ al nodo $\lambda$, se incorpora a la gramática la regla $S := \lambda$.

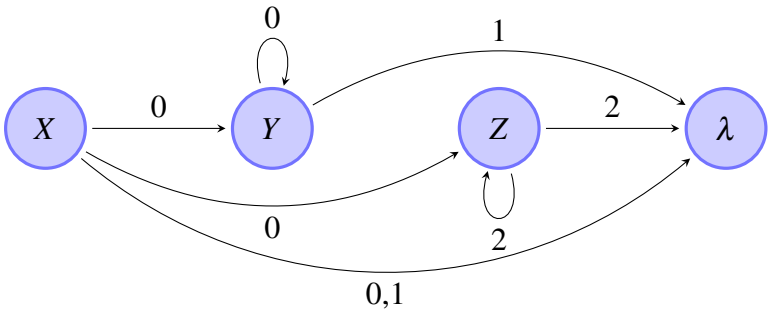
Ejemplo: La siguiente gramática $G$ que es lineal por izquierda debe ser convertida en lineal por derecha para poder definir el $AFD$ que reconoce el lenguaje generado por ella.

$$G = (\{0,1,2\},\{X,Y,Z\},X,P)$$

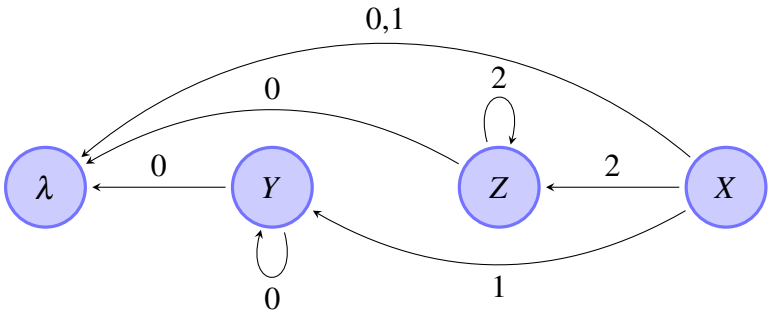
$$P = \{X := Z0 \mid Y0 \mid 0 \mid 1, Y := Y0 \mid 1, Z := Z2 \mid 2\}$$

Seguimos los pasos descriptos anteriormente:

1. La gramática no es recursiva en el axioma así que no tenemos que hacer nada en este paso.
2. Realizamos el grafo:




3. Modificamos el grafo anterior:




4. 
   - Los alfabetos de símbolos terminales y no terminales son los mismos de antes.
   - Las primeras reglas de producción que se crean son las siguientes: $X := 2Z \mid 1Y, Z := 2Z, Y := 0Y$.
   - Las siguientes reglas de producción que se crean son estas: $X := 0 \mid 1, Z := 0, Y := 0$.
   - No hay aristas sin etiquetas.

   De esta forma, la gramática obtenida es la siguiente:

$$G = (\{0,1,2\}, \{X,Y,Z\}, X, P)$$

$$P = \{X := 2Z \mid 1Y \mid 0 \mid 1, Y := 0Y \mid 0, Z := 2Z \mid 0\}$$

## 5.5. Expresiones regulares y autómatas finitos

Lo que se va a presentar ahora es un algorítmo que permite construir para cada expresión regular, un $AFND{-}\lambda$ que reconozca el mismo lenguaje descripto por ella.

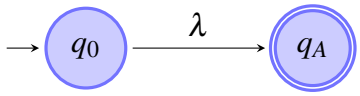
### 5.5.1. Algorítmo de Thompson

El algorítmo de Thompson traduce cada parte de una expresión regular en un atómata que acepta el mismo lenguaje denotado por ella. A continuación vamos item por item de la definición de una expresión regular y graficamos el autómata equivalente.

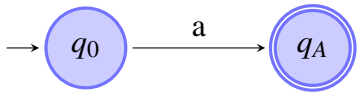
La definición la podemos ver de nuevo en esta sección 3.4.1.

a) Autómata equivalente a $\varnothing$:


b) Autómata equivalente a $\lambda$:

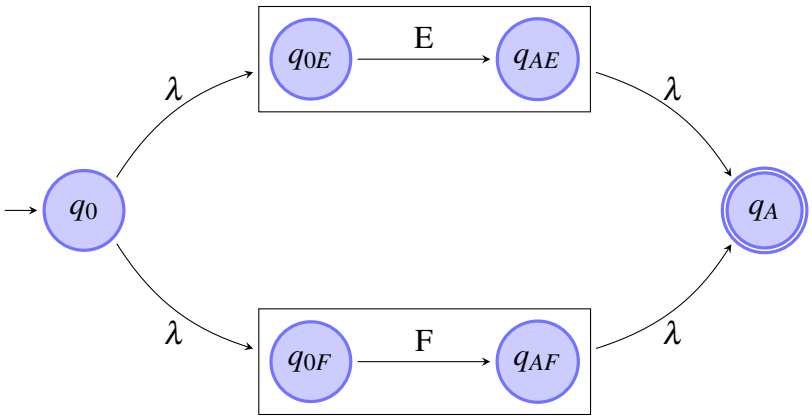



c) Autómata equivalente a la expresión $a$:

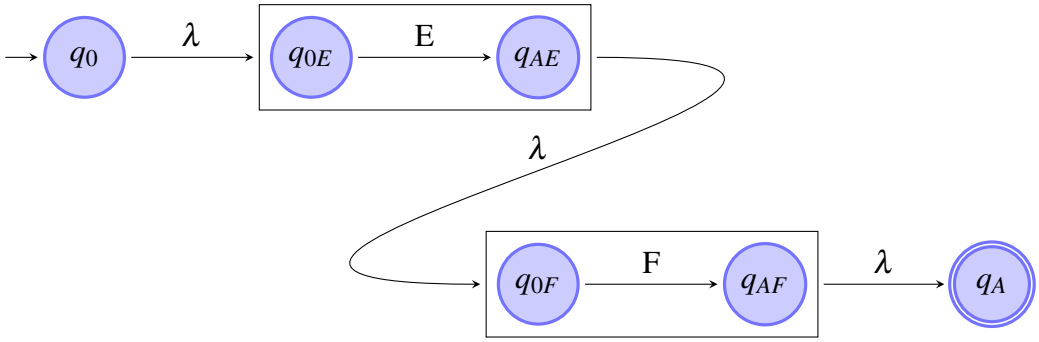



Ahora suponemos los que los autómatas $AF_E$ y $AF_F$ que reconocen las expresiones regulares $E$ y $F$ ya existen.

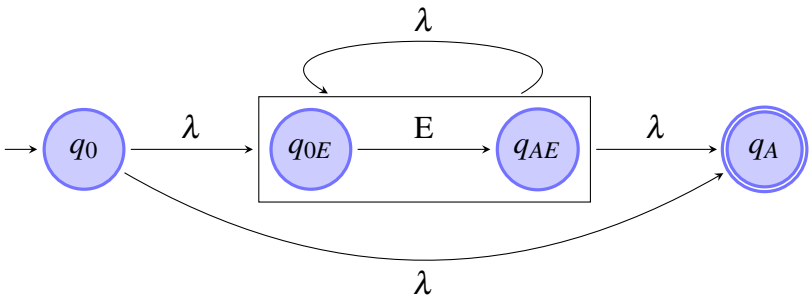
d) Autómata equivalente a $E+F$, notar que se quitan los estados de inicio y aceptacion de los automatas $AF_E$ y $AF_F$:




e) Autómata equivalente a $EF$, se quitan los estados de inicio y aceptación de $AF_E$ y $AF_F$:



f) Autómata equivalente a $E^*$, se quitan los estados de inicio y aceptación de $AF_E$:




El algorítmo de Thompson termina construyendo un $AFND{-}\lambda$ con muchos estados, por lo que debe ser luego transformado en $AFD$ y posiblemente minimizado para que sea eficiente su implementación, pero es muy útil ya que todos estos procesos son automáticos y pueden ser programados (es lo que hace el generador de analizadores léxicos *lex* de UNIX).