
*Facultad de Ciencias*

# La transformación de lenguajes formales a autómatas finitos

From formal languages to finite automata

Trabajo de fin de grado
para acceder al

**GRADO EN INGENIERÍA INFORMÁTICA**

Autor: Álvar Tapia Ruiz

Director: Domingo Gómez Pérez

Junio - 2019

## *Agradecimientos*

A Domingo por dejarme trabajar en esta idea,
y a mis padres por aguantarme hablar de esto durante meses.

*La transformación de lenguajes formales a autómatas finitos*


## Resumen



**palabras clave:** expresiones regulares, optimización, Python, RE



Un lenguaje formal se puede definir por el conjunto de palabras, y para saber si una palabra pertenece a un lenguaje, se pueden utilizar autómatas diseñados a partir de la definición del lenguaje.

Una clase de lenguajes muy importantes son los aceptados por autómatas finitos, también conocidos como lenguajes regulares.

Un lenguaje regular arbitrario puede ser aceptado por diferentes autómatas finitos; por ello es interesante buscar aquellos que tengan buenas propiedades, para hacer el proceso lo más sencillo posible. Existe un algoritmo que, dada la descripción del lenguaje mediante una expresión regular, genera un autómata finito. Pero, en ciertas implementaciones, la transformación de una expresión regular en un autómata finito puede tardar más en ejecutarse que otra, aunque proporcionen los mismos resultados.

En este documento nuestro objetivo es diseñar un conjunto de reglas que permitan transformar una expresión regular «problemática» en otra con mejores propiedades, pero manteniendo el mismo resultado de la expresión original.

Nos apoyaremos en autómatas finitos para demostrar por qué hay un ahorro de tiempo de cómputo en una expresión y no en la otra, aun obteniendo los mismos resultados.


---

*From formal languages to finite automata*


## Abstract



**keywords:** regular expressions, optimization, Python, RE



A formal language can be defined by the set of words. To solve the problem wether a word belongs to certain classes of languages, automata based on the language definition can be built.

An important class of languages is the languages that can be defined as the words accepted by finite automata, also known as regular languages. Any regular language can be defined by a regular expresion

A particular regular language can be accepted by different finite automata; so it is interesting to find those with good properties, to make the computation as efficient as possible. There are algorithms that, given a regular expression, can generate a finite automaton. But, in some implementations, a regular expression can be more complex for computing than another, even though they define the same language.

In this document our objective is to design a set of rules that allow as to transform a «problematic» regular expression into another one with better properties, while keeping the results of the original one.

We will use finite automata to discuss why a regular expression need less time than another to achieve the same goal.

# Índice


**1. Introducción** **1**
- 1.1. Motivación . . . 1
  - 1.1.1. La idea . . . 1
  - 1.1.2. Pensando en los ordenadores . . . 1
- 1.2. Ejemplos problemáticos . . . 1
  - 1.2.1. Primer ejemplo: confirmar SVGs «bien formados» . . . 2
  - 1.2.2. Segundo ejemplo: caracteres opcionales . . . 4
- 1.3. Objetivo del documento . . . 5

**2. Gramáticas formales** **7**
- 2.1. Lenguaje de una gramática . . . 7
  - 2.1.1. Operaciones de los lenguajes . . . 7
  - 2.1.2. Propiedades de los lenguajes . . . 7

**3. Expresiones regulares** **9**
- 3.1. Definiciones . . . 9
  - 3.1.1. El lenguaje de las expresiones regulares . . . 9
  - 3.1.2. Relación entre lenguajes y expresiones regulares . . . 10
  - 3.1.3. Propiedades de las expresiones regulares . . . 10
- 3.2. Autómatas finitos . . . 11
  - 3.2.1. Traducción de expresiones regulares a autómatas deterministas . . . 11

**4. Motores de expresiones regulares** **15**
- 4.1. DFA . . . 15
- 4.2. NFA . . . 15
- 4.3. ¿Cuál es la mejor alternativa? . . . 16
- 4.4. ¿Qué utiliza Python? . . . 16
  - 4.4.1. Documentos a inspeccionar . . . 16
  - 4.4.2. Interpretación del código fuente . . . 17
  - 4.4.3. Conclusiones del código fuente . . . 17

**5. Equivalencias** **19**
- 5.1. Equivalencias demostradas con teoría de conjuntos . . . 19
- 5.2. Comparando equivalencias con autómatas finitos . . . 21

**6. Conclusiones** **25**

**A. Ejemplo de código SVG válido** **27**

**Referencias** **29**

# 1 Introducción

## 1.1. Motivación

### 1.1.1. La idea

En la clase de Lenguajes Formales de 3º de Grado de Ingeniería Informática, realicé ejercicios en los que se debía trabajar con expresiones regulares.

Estos ejercicios empezaron siendo sencillos, pero a medida que avanzaban los días, las expresiones regulares eran cada vez más complejas, y los ejercicios obligaban a pasar cada vez más tiempo procesando las expresiones.

Como ejercicio para mejorar en esta parte de la asignatura, decidí modificar enunciados de problemas ya resueltos. Para mi sorpresa, algunas de estas modificaciones resultaban en los mismos conjuntos de palabras que los enunciados originales, pero con menos pasos.

Investigando los enunciados en los que este suceso ocurría, inferí unas reglas que permitían modificar el enunciado del problema, pero manteniendo su resultado y que se resolvían de manera más sencilla, con menos pasos.

### 1.1.2. Pensando en los ordenadores

En primer lugar, pensé en formalizar este método como un proceso más sencillo para hacer ejercicios escritos de expresiones regulares, debido a que se obtienen resultados con menos esfuerzo cuando los enunciados cumplen las reglas que se proponen en este documento.

Pero, cuanto más utilizaba este proceso, más me planteaba si a los motores de expresiones regulares también les facilitaría el trabajo que las expresiones que tenían que procesar seguían estas mismas reglas.

Haciendo un código de prueba en Python, escribiendo yo mismo una expresión regular y la expresión transformada, la eficiencia obtenida en casos concretos era órdenes de magnitud superior a la implementación del módulo RE en ese momento.

Si pudiera aplicar las normas bajo cualquier expresión regular, ampliando un poco la implementación actual del módulo RE, esos «casos especiales» dejarían de ser «especiales». El usuario se podría desentender de las optimizaciones que ocurren «entre bastidores», y los problemas encontrados en este trabajo causarían menos preocupaciones.

## 1.2. Ejemplos problemáticos

Con los ejemplos siguientes se pretende ilustrar la capacidad de optimización que pueden tener las expresiones regulares, y cómo encontrar métodos que las optimicen por nosotros puede ayudarnos a trabajar de manera más «inocente» y con menos quebraderos de cabeza.

### 1.2.1. Primer ejemplo: confirmar SVGs «bien formados»

Imaginemos que queremos comprobar si nuestro archivo SVG está bien formado. Con «bien formado» queremos decir que haya una superficie en la que dibujar, y que cada vez que se abra una etiqueta (con el símbolo $<$) dentro de la superficie de dibujo, se cierre (con el símbolo $>$) antes de que se abra otra (o lo que es lo mismo, sin etiquetas anidadas).
Muchos conceptos a la vez, empecemos por el principio:

- Un archivo SVG permite describir gráficos que no se «pixelan» independientemente de la resolución a la que se representen (debido que son gráficos que se basan en vectores, no en píxeles, y se recalculan al cambiar la resolución).
    - Puede definirse una imagen SVG en un fichero con un lenguaje de etiquetas muy parecido a HTML, por eso se amolda bien a nuestro problema.
- SVG necesita un espacio dedicado para poder dibujar. Lo llamaremos «canvas» a partir de ahora, y tiene una etiqueta dedicada.
- Dentro del canvas se pueden dibujar formas geométricas como líneas, triángulos, cuadrados, etc. Éstas se representan con su propia etiqueta.

Primera idea
Para que una expresión nos confirme que un archivo SVG está «bien formado», podemos utilizar la siguiente expresión:

$$< svg.^* > .^*(< .^* > .^*)^* < /svg > \tag{1}$$

El comportamiento de la expresión regular puede definirse con la siguiente lista de objetivos:

1. Encuentra los caracteres $< svg$, y captúralos.
2. Captura todos los caracteres que encuentres.
3. Si encuentras el carácter $>$, captúralo.
4. Captura todos los caracteres que encuentres.
5. Si encuentras esta estructura, captúrala todas las veces posibles:
    *a*) El carácter $<$.
    *b*) Los caracteres que encuentres.
    *c*) El carácter $>$.
    *d*) Los caracteres que encuentres.
6. Captura los caracteres $< /svg >$
7. En caso de llegar a este punto, acepta la captura y termina.
8. En caso de no llegar a este punto, revisar los grupos capturados y realizar «backtracking».

Si se aplica la expresión regular 1 al archivo A, se puede confirmar que el archivo cumple con la estructura descrita en la expresión regular.

Sin embargo, si se modifica el archivo de tal manera que «no esté bien formado» y se aplica la expresión regular, pueden ocurrir dos situaciones:

- O bien tarda varios segundos en confirmar que la expresión no está bien formada (que para los poco más de 1000 caracteres que tiene el archivo es demasiado tiempo de procesamiento),

- o bien, si se han modificado los caracteres de identificación de etiquetas, puede llegar a capturar todo el documento, indicando que el archivo «estaría bien formado» (lo cual sabemos que no es verdad por la modificación realizada).

Esto sucede porque en el comportamiento de la expresión regular 1 hay objetivos que engloban a objetivos posteriores.

Al utilizar la estructura .∗, que significa «captura todos los caracteres que encuentres», cada vez que no se cumpla la estructura regular en el documento, el programa intentará «ir hacia atrás», a ver si alguno de los caracteres capturados con la estructura .∗ era uno de los necesarios para cumplir otros objetivos más adelante.

Por lo tanto, si el documento «no está bien formado», la expresión regular va a analizar varias veces cada carácter para comprobar todas las posibles combinaciones de captura de .∗ antes de declarar que, ciertamente, el documento «no está bien formado».

Además, en el caso de etiquetas «anidadas» (que antes de que aparezca el símbolo > aparezca el símbolo <) esta expresión regular no se comporta bien.

Si en el archivo de ejemplo se borra uno de los caracteres de inicio de etiqueta en medio del documento, esta estructura regular interpretará que, aunque ha aparecido un carácter de cierre, la etiqueta no se ha terminado, y capturará «dos etiquetas» para completar la estructura 5, de forma que puede aceptar el documento.

Con los problemas de eficiencia y falsos positivos que genera esta expresión regular, debería buscarse una alternativa que cumpla mejor con los requisitos enunciados.

Analizando el enunciado

Sabiendo que no van a aparecer los caracteres < o > a menos que se usen para identificar etiquetas, podemos modificar la anterior expresión para disminuir drásticamente los falsos positivos:

$$< svg[^{\wedge}<>]^* > [^{\wedge}<>]^*(< [^{\wedge}<>]^* > [^{\wedge}<>]^*)^* < /svg > \tag{2}$$

Como se puede comprobar, hemos cambiado todas las construcciones .* por construcciones $[^{\wedge}<>]^*$. El comportamiento de esta estructura regular se puede definir identificando los siguientes objetivos:

1. Encuentra los caracteres $< svg$, y captúralos.
2. Captura todos los caracteres que no sean ni < ni >.
3. Si encuentras el carácter >, captúralo.
4. Captura todos los caracteres que no sean ni < ni >.
5. Si encuentras esta estructura, captúrala todas las veces posibles:
   a) El carácter <.
   b) Los caracteres que no sean ni < ni >.
   c) El carácter >.
   d) Los caracteres que no sean ni < ni >.
6. Captura los caracteres $< /svg >$
7. Al llegar a este punto, acepta la captura y termina.
8. En caso de no llegar a este punto, revisar los grupos capturados y realizar «backtracking».

En esta expresión se han reducido en gran medida los objetivos que engloban a objetivos posteriores. Este fenómeno sólo ocurre entre el objetivo 5 y el objetivo 6, y aunque no es ideal, es una mejora considerable con respecto a la expresión regular 1.

La expresión regular 2 reduce drásticamente el número de veces que se tiene la opción de «ir para atrás», y con ello se reducen las veces que se comprueba un mismo carácter, aumentando la eficiencia.

Además las etiquetas «anidadas» no cumplen con el objetivo 5, y, por lo tanto, cada vez que la expresión regular se encuentre con una etiqueta anidada dentro de otra, concluye que ese documento «no está bien formado».

Conclusión

Se ha encontrado una expresión regular que cumple mejor nuestros objetivos, evitando estructuras demasiado generales y estudiando los caracteres que no poseen modificadores, que son los caracteres de obligado cumplimiento para la expresión regular.

### 1.2.2. Segundo ejemplo: caracteres opcionales

En el siguiente ejemplo evaluaremos las diferencias de estas dos expresiones:

$$a?a?a?a?a?a?a?a?a?a?a?a?a?a?a?aaaaaaaaaaaaaaa \tag{3}$$

$$aaaaaaaaaaaaaaa?a?a?a?a?a?a?a?a?a?a?a?a?a?a? \tag{4}$$

Ambas expresiones se pueden definir como la lista de objetivos siguiente:

1. Captura 15 caracteres $a$.
2. Si encuentras algún carácter $a$ más, puedes capturarlo.
   a) Puedes hacer esto hasta 15 veces.

Ejecutando estas expresiones en documentos compuestos únicamente de caracteres $a$, se puede observar que hay un comportamiento inusual a medida que se aumenta el número de caracteres $a$ en el documento. Si representamos los tiempos de cómputo de este experimento en una gráfica, obtenemos la figura 1:

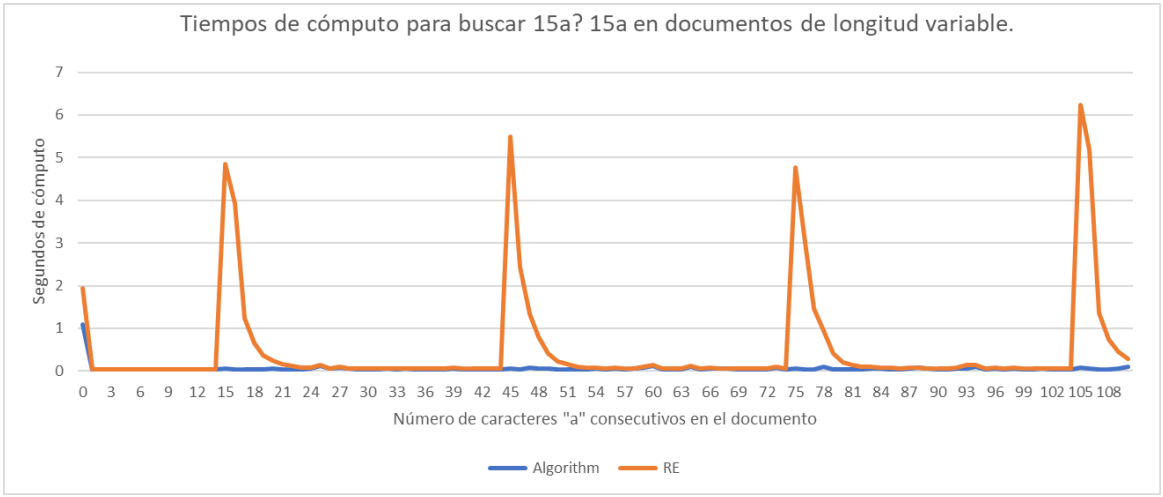


Ilustración 1: Gráfica comparando tiempos de ejecución entre expresiones regulares equivalentes.

Mientras que la expresión 4 mantiene un tiempo de cómputo estable durante todo el experimento, la expresión regular 3 muestra unos «picos de sierra» en su ejecución, con un máximo local muy pronunciado que evoluciona a los tiempos de cómputo de 4 a gran velocidad a medida que se aumentan los caracteres presentes en el documento.

Se puede observar que estos máximos locales siguen un patrón: son equidistantes. Y no solo eso, sino que resulta que los máximos ocurren cuando el documento tiene $15 \cdot (2n-1)$ letras, $\forall n \in \mathbb{N}^*$.

Conclusión
Dos expresiones regulares con los mismos objetivos y que retornan las mismas palabras pueden presentar tiempos de computación radicalmente distintos, y se debería estudiar cómo evitar los casos de baja eficiencia al procesar expresiones regulares.

## 1.3. Objetivo del documento

En este documento se pretende estudiar qué causa estas fluctuaciones de rendimiento entre expresiones, si hay alguna manera de detectar problemas de optimización antes de ejecutar la expresión regular, y las modificaciones que permiten mejorar el rendimiento de dicha expresión manteniendo el funcionamiento esperado.

Para ello, manejaremos expresiones regulares buscando su significado mediante el uso de gramáticas formales, las transformaremos mediante ciertas equivalencias y devolveremos el resultado al usuario.

# 2 Gramáticas formales

En esta sección se adaptará teoría del Capítulo 1 de la asignatura «Lenguajes Formales» del año 2013, impartida en la Universidad de Cantabria lft [2013], que pertenece a la plataforma gratuita «Open Course Ware» Uni [2013].

## 2.1. Lenguaje de una gramática

**Definición 1.** *Sea $G$ una gramática definida como $G := (V, \Sigma, S, P)$. Llamaremos lenguaje generado por la gramática $G$ al lenguaje $\mathcal{L}(G) \subseteq \Sigma^*$ dado por:*

$$\mathcal{L}(G) := \{x \in \Sigma \mid S \to^* x\}, \tag{5}$$

*es decir, a las palabras formadas únicamente con símbolos terminales alcanzables desde el símbolo inicial de la gramática.*

Las gramáticas formales son útiles porque permiten definir el significado de una palabra mediante las producciones usadas para generar la palabra. En el caso de lenguajes generados por gramáticas libres de contexto, todas las palabras de un lenguaje admiten una representación como árboles de derivación.

### 2.1.1. Operaciones de los lenguajes

En este apartado, vamos a dar varias operaciones que se pueden definir para lenguajes. Además, es fácil ver que si los lenguajes están generados por gramáticas, es fácil dar una gramática que genere cada uno de los lenguajes dados por las operaciones siguientes.

**Definición 2.** *Sea $\Sigma$ un alfabeto finito, y $\mathcal{L}$ y $\mathcal{M}$ lenguajes que cumplen $\mathcal{L}, \mathcal{M} \subseteq \Sigma^*$. Se definen las siguientes operaciones para dichos lenguajes:*

1. *Unión de lenguajes:*
$$\mathcal{L} \cup \mathcal{M} := \{x \in \Sigma^* \mid [x \in \mathcal{L}] \vee [x \in \mathcal{M}]\} \tag{6}$$

2. *Concatenación de lenguajes:*
$$\mathcal{L} \cdot \mathcal{M} := \{x \cdot y \in \Sigma^* \mid x \in \mathcal{L}, y \in \mathcal{M}\} \tag{7}$$

3. *Potencia de lenguajes:*
$$\mathcal{L}^n := \begin{cases} \{\lambda\} & \text{si } n = 0, \\ \mathcal{L} \cdot \mathcal{L}^{n-1} & \text{en otro caso.} \end{cases} \tag{8}$$

### 2.1.2. Propiedades de los lenguajes

**Definición 3.** *Sea $\Sigma$ un alfabeto finito, y $\mathcal{L}$, $\mathcal{M}$ y $\mathcal{N}$ lenguajes que cumplen la propiedad $\mathcal{L}, \mathcal{M}, \mathcal{N} \subseteq \Sigma^*$. Se verifican las siguientes propiedades para dichos lenguajes:*

1. *Asociativa:*

$$\begin{aligned} \mathcal{L} \cup (\mathcal{M} \cdot \mathcal{N}) &= (\mathcal{L} \cdot \mathcal{M}) \cdot \mathcal{N} \\ \mathcal{L} \mid (\mathcal{M} \mid \mathcal{N}) &= (\mathcal{L} \mid \mathcal{M}) \mid \mathcal{N} \end{aligned} \tag{9}$$

2. *Conmutativa:*

$$\mathcal{L} \cup \mathcal{M} = \mathcal{M} \cup \mathcal{L} \tag{10}$$

3. *Idempotencia:*

$$\mathcal{L} \cup \mathcal{L} = \mathcal{L} \tag{11}$$

4. *Distributiva:*

$$\begin{aligned} \mathcal{L} \cdot (\mathcal{M} \cup \mathcal{N}) &= \mathcal{L} \cdot \mathcal{M} \cup \mathcal{L} \cdot \mathcal{N} \\ (\mathcal{L} \cup \mathcal{M}) \cdot \mathcal{N} &= \mathcal{L} \cdot \mathcal{N} \cup \mathcal{M} \cdot \mathcal{N} \end{aligned} \tag{12}$$

Estas propiedades serán centrales para los algoritmos desarrollados en los siguientes capítulos. Entre otras cosas, permitirán reducir las expresiones regulares o modificarlas para simplificar las computaciones realizadas para saber si una palabra es capturada por una expresión regular.

# 3 Expresiones regulares

En esta sección se adaptará la teoría del Capítulo 2 de la asignatura «Lenguajes Formales» del año 2013, impartida en la Universidad de Cantabria lft [2013], que pertenece a la plataforma gratuita «Open Course Ware» Uni [2013].

Las definiciones se han simplificado, y sólo se han introducido aquellas que serán relevantes para el desarrollo de este documento. Para ahondar más en la teoría, recomiendo visitar el Capítulo 2 del libro lft [2013].

## 3.1. Definiciones

### 3.1.1. El lenguaje de las expresiones regulares

**Nota 1.** *Debido a la confusión que puede generar el carácter de alternancia con respecto a la definición de la gramática, todos los caracteres que representen un operador se encerrarán entre comillas simples en la definición 4.*

**Definición 4.** *Sea $\Sigma$ un alfabeto finito. Llamaremos «expresión regular sobre el alfabeto $\Sigma$» a toda palabra generada por la siguiente gramática:*

$$\begin{aligned} &\langle S\rangle \to \emptyset \mid \lambda \mid a, \quad a \in \Sigma \\ &\langle S\rangle \to \langle S\rangle \text{`}*\text{'} \mid \langle S\rangle \text{`}+\text{'} \mid \langle S\rangle \text{`}?\text{'} \mid \text{`}(\text{'}\, \langle S\rangle \text{`})\text{'} \mid \langle S\rangle \text{`}|\text{'}\, \langle S\rangle \mid \langle S\rangle \text{`}\cdot\text{'}\, \langle S\rangle \end{aligned} \tag{13}$$

**Nota 2.** *Por comodidad (y sólo en el caso de que no haya ninguna posibilidad de ambigüedad) se suprimen los paréntesis y los caracteres de concatenación. También, para evitar ambigüedades, el orden de prioridad de las operaciones será el siguiente:*

1. *Los paréntesis, representados con los caracteres '(' ')'.*
2. *Las concatenaciones, representadas con el carácter '·'.*
3. *Las alternancias, representadas con el carácter '|'.*

#### La semántica de las expresiones regulares

El «significado» de una expresión regular está muy relacionado con las producciones de la anterior gramática. A continuación se detallarán los caracteres que tienen un significado concreto independientemente del alfabeto:

Conjuntos simples

- $\emptyset$: Conjunto vacío. Conjunto que no contiene ninguna palabra.
- $\lambda$: Palabra vacía. Este es el conjunto que contiene solo la palabra sin ninguna letra.
  No confundir con el conjunto vacío. La palabra vacía no se puede encontrar en el conjunto vacío.

Operadores

- $\alpha^*$: La expresión regular $\alpha$ debe aparecer 0 o más veces consecutivas para aceptar la palabra.
- $\alpha^+$: La expresión regular $\alpha$ debe aparecer 1 o más veces consecutivas para aceptar la palabra.
- $\alpha?$: La expresión regular $\alpha$ debe aparecer 0 o 1 veces para aceptar la palabra.
- $(\alpha)$: La expresión regular $\alpha$ debe aparecer 1 vez para aceptar la palabra. [1]
- $\alpha_1 \mid \alpha_2$: Debe aparecer o $\alpha_1$ o $\alpha_2$ para aceptar la palabra.
- $\alpha_1 \cdot \alpha_2$: Debe aparecer $\alpha_1$ y, consecutivamente, $\alpha_2$, para aceptar la palabra.

### 3.1.2. Relación entre lenguajes y expresiones regulares

Las expresiones regulares forman conjuntos de palabras, y dada una expresión regular podemos identificar un conjunto de palabras asociado a dicha expresión regular. De manera que diremos que las expresiones regulares representan un lenguaje en particular.

**Definición 5.** *Sea $\Sigma$ un alfabeto finito. A cada expresión regular $\alpha$ sobre $\Sigma$ se le asignará un lenguaje formal $\mathcal{L}(\alpha) \subseteq \Sigma^*$.*

**Definición 6.** *Sea $\Sigma$ un alfabeto finito, y $\alpha$ y $\beta$ expresiones regulares válidas sobre el alfabeto $\Sigma$. Se verifican las siguientes propiedades con respecto a los lenguajes generados por dichas expresiones:*

$$\mathcal{L}(\alpha) = \mathcal{L}(\alpha)^1 \tag{14}$$
$$\mathcal{L}(\lambda) = \mathcal{L}(\alpha)^0 \tag{15}$$
$$\mathcal{L}(\alpha?) = \mathcal{L}(\alpha)^0 \cup \mathcal{L}(\alpha)^1 \tag{16}$$
$$\mathcal{L}(\alpha^*) = \bigcup_{n=0}^{\infty} (\mathcal{L}(\alpha)^n) \tag{17}$$
$$\mathcal{L}(\alpha^+) = \bigcup_{n=1}^{\infty} (\mathcal{L}(\alpha)^n) \tag{18}$$
$$\mathcal{L}(\alpha \cdot \beta) = \mathcal{L}(\alpha)^1 \cdot \mathcal{L}(\beta)^1 \tag{19}$$
$$\mathcal{L}(\alpha \cdot \alpha) = \mathcal{L}(\alpha)^1 \cdot \mathcal{L}(\alpha)^1 = \mathcal{L}(\alpha)^2 \tag{20}$$
$$\mathcal{L}(\alpha \mid \beta) = \mathcal{L}(\alpha)^1 \cup \mathcal{L}(\beta)^1 \tag{21}$$

### 3.1.3. Propiedades de las expresiones regulares

**Definición 7.** *Sea $\Sigma$ un alfabeto finito, y $\alpha$, $\beta$ y $\gamma$ expresiones regulares válidas sobre el alfabeto $\Sigma$. Se verifican las siguientes propiedades para dichas expresiones regulares:*

1. *Asociativa:*
$$\begin{aligned} \alpha \cdot (\beta \cdot \gamma) &\equiv (\alpha \cdot \beta) \cdot \gamma \\ \alpha \mid (\beta \mid \gamma) &\equiv (\alpha \mid \beta) \mid \gamma \end{aligned} \tag{22}$$
2. *Conmutativa:*
$$\alpha \mid \beta \equiv \beta \mid \alpha \tag{23}$$
3. *Elementos neutros:*
$$\lambda \cdot \alpha \equiv \alpha, \quad \emptyset \cdot \alpha \equiv \emptyset, \quad \emptyset \mid \alpha \equiv \alpha \tag{24}$$
4. *Idempotencia:*
$$\alpha \mid \alpha \equiv \alpha \tag{25}$$

[1] Los paréntesis permiten que expresiones regulares complejas se vean afectadas por un mismo operador, diferenciando de esta manera las expresiones regulares entre paréntesis de aquellas sin operadores asociados.

5. *Distributiva:*

$$\begin{aligned} \alpha \cdot (\beta \mid \gamma) &\equiv \alpha \cdot \beta \mid \alpha \cdot \gamma \\ (\alpha \mid \beta) \cdot \gamma &\equiv \alpha \cdot \gamma \mid \beta \cdot \gamma \end{aligned} \tag{26}$$

6. *Invariantes para $^*$:*

$$\lambda^* \equiv \lambda, \quad \emptyset^* \equiv \emptyset \tag{27}$$

7. *Relación de $^*$ con la alternancia:*

$$(\alpha \mid \beta)^* \equiv (\alpha^* \beta^*)^* \tag{28}$$

8. *Notación $\alpha?$:*

$$\alpha? \equiv \lambda \mid \alpha \tag{29}$$

9. *Notación $\alpha^+$:*

$$\begin{aligned} \alpha^+ &\equiv \alpha^* \cdot \alpha \equiv \alpha \cdot \alpha^* \\ \lambda \mid \alpha^+ &\equiv \alpha^* \end{aligned} \tag{30}$$

## 3.2. Autómatas finitos

Los autómatas finitos son el modelo más sencillo de computación, que esta compuesto por estados y transiciones. Utilizaremos la representación gráfica de los autómatas para dar una idea básica del funcionamiento de la expresión regular. Los estados se representan como círculos, y las transiciones como aristas etiquetadas. Cuando la entrada es una palabra, desde el estado inicial siguiendo las transiciones.

Cada estado puede ser final o no final. Los estados finales se representan con dos círculos concéntricos, mientras que los estados no finales sólo tienen un círculo.
Una palabra es aceptada cuando se encuentra en un estado final.

Se puede introducir una etiqueta dentro de los estados para reconocerlos mejor. En estos autómatas, el carácter $\lambda$ representará el estado inicial, el estado desde el que empieza la computación.

Un autómata puede cambiar de estado si consigue leer el carácter que hay marcado en una flecha que «salga» del estado en el que se encuentra en este momento (o lo que es lo mismo, cuya punta de flecha se encuentre en otro estado). Las flechas con el símbolo $\lambda$ representan transiciones que permiten cambiar de estado sin necesidad de leer entradas. Por ello estas flechas sólo están permitidas en autómatas indeterministas (se tiene que tener en cuenta que el autómata puede estar en varios estados a la vez).

### 3.2.1. Traducción de expresiones regulares a autómatas deterministas

A continuación se mostrarán los grafos[2] que representan los autómatas finitos resultantes de procesar estructuras básicas.

Transformación de la expresión regular $\alpha$

**Definición 8.** *El autómata de la expresión regular $\alpha$ sólo llega a un estado final si encuentra la palabra $\alpha$ en el documento que está leyendo.*

Transformación de la expresión regular $\alpha^+$

**Definición 9.** *El autómata de la expresión regular $\alpha^+$ sólo llega a un estado final si encuentra la palabra $\alpha$ 1 o más veces consecutivas en el documento que está leyendo.*

Transformación de la expresión regular $\alpha^*$

**Definición 10.** *El autómata de la expresión regular $\alpha^*$ sólo llega a un estado final si encuentra la palabra $\alpha$ 0 o más veces consecutivas en el documento que está leyendo.*

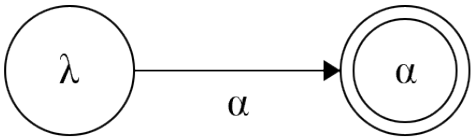


Ilustración 2: Autómata $\alpha$

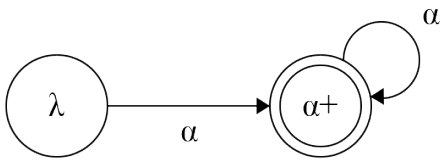


Ilustración 3: Autómata $\alpha^+$

Transformación de la expresión regular $\alpha \mid \beta$

**Definición 11.** *El autómata de la expresión regular $\alpha \mid \beta$ sólo llega a un estado final si encuentra la palabra $\alpha$ o la palabra $\beta$ en el documento que está leyendo.*

Transformación de la expresión regular $\alpha?$

**Definición 12.** *El autómata de la expresión regular $\alpha^?$ sólo llega a un estado final si encuentra la palabra $\alpha$ 0 o 1 veces en el documento que está leyendo.*

Transformación de la expresión regular $\alpha?$

**Definición 13.** *El autómata de la expresión regular $\alpha^?$ sólo llega a un estado final si encuentra la palabra $\alpha$ 0 o 1 veces en el documento que está leyendo.*

Transformación de la expresión regular $\alpha \cdot \beta$

**Definición 14.** *El autómata de la expresión regular $\alpha \cdot \beta$ sólo llega a un estado final si encuentra la palabra $\alpha$ y, de forma consecutiva, la palabra $\beta$.*

Por supuesto, se pueden crear autómatas más complejos combinando estos elementos, estableciendo paréntesis, etc.

[2]Grafos realizados con la herramienta Wallace

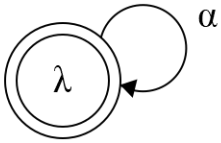


Ilustración 4: Autómata $\alpha^*$

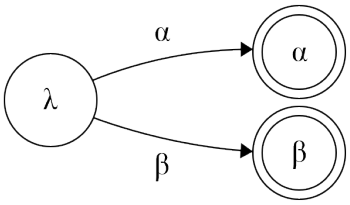


Ilustración 5: Autómata $\alpha|\beta$

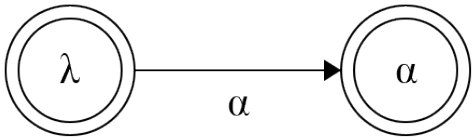


Ilustración 6: Autómata $\alpha?$

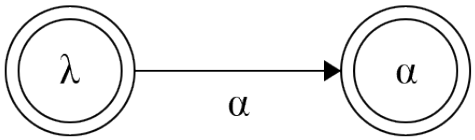


Ilustración 7: Autómata $\alpha?$

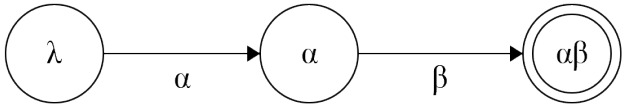


Ilustración 8: Autómata $\alpha \cdot \beta$

# 4 Motores de expresiones regulares

Para poder trabajar con expresiones regulares, los ordenadores necesitan de algoritmos que les permitan «leerlas», «entenderlas», y aplicarlas. Como se ha mencionado anteriormente, entender la expresión regular consiste en hallar que producciones de la gramática que genera el lenguaje de las expresiones regulares. La función de los motores de expresiones regulares es la de proporcionar a los ordenadores las instrucciones necesarias para trabajar con estas producciones.

Como normalmente la forma de trabajar del ordenador resulta confusa al usuario (y a veces incomprensible), los motores de expresiones regulares trabajan en tandem con otros programas, compiladores principalmente, para hacer la tarea más fácil al usuario que quiere utilizar expresiones regulares.

Básicamente, los compiladores se encarga de hallar las producciones, y si procede y es capaz, optimizar las expresiones regulares. No ahondaremos más en este tema dado que el compilador no ejecuta las expresiones regulares, sino que hace de traductor para que el ordenador sepa qué debe computar.

## 4.1. DFA

Motor basado en autómatas finitos deterministas, *Deterministic Finite Automata* en inglés.

Ventajas
Los motores DFA son muy rápidos. Una expresión regular se puede confirmar o denegar en complejidad $O(n)$, siendo $n$ el número de letras del documento.

Inconvenientes
Hallar un autómata finito determinista es complejo, y se conocen expresiones regulares cuyos autómatas finitos determinístas tienen un número exponencial de estados en el tamaño de la expresión regular.

## 4.2. NFA

Motor basado en autómatas finitos indeterministas, *Nondeterministic Finite Automata* en inglés[1].

Ventajas
Proporcionan mucha flexibilidad a la hora de escribir las expresiones regulares. Permiten cierta libertad de expresión a los usuarios al poder declararse el equivalente a «transiciones $\lambda$», movimientos entre nodos del autómata sin necesidad de leer caracteres. Esta característica se simula en DFA con diferentes autómatas, lo que aumenta los recursos necesarios en comparación con NFA.

[1] Técnicamente estos motores no se pueden modelar como NFA, porque opciones como las «referencias hacia atrás» no se pueden implementar en un autómata. Pero su funcionamiento se basó inicialmente en estos autómatas, y el nombre se ha mantenido. mas [2002]

Además, los motores de expresiones regulares basados en NFA permiten «referencias»: leyendo parte del documento, modificar la expresión regular de acuerdo con lo leído.
Esta funcionalidad aumenta en gran medida la potencia de estos motores, lo que hace que sean muy versátiles.

Inconvenientes
La flexibilidad que permiten al usuario estos motores hace que sea difícil optimizarlas a nivel de compilador, y esa responsabilidad recae en el usuario. Una expresión regular poco optimizada puede tardar mucho tiempo en completarse.

## 4.3. ¿Cuál es la mejor alternativa?

Los motores NFA pueden leer las estructuras regulares dedicadas a ellos, como las dedicadas a los DFA. Y, a menos que la expresión tenga referencias, es posible modificar una expresión regular dedicada a un motor NFA para que un motor DFA pueda procesarla.
Si estuviéramos hablando de autómatas finitos, tanto NFA como DFA pueden modelar los mismos lenguajes. Pero en el caso de los motores de expresiones regulares, los motores basados en NFA tienen más potencia que los motores basados en DFA.

Mientras que la ejecución de expresiones DFA es muy veloz, la flexibilidad y legibilidad de las expresiones NFA aportan un valor añadido muy importante.

Por lo tanto, bajo la premisa de hacer lo más fácil posible el trabajo tanto al programador como al ordenador, deberíamos centrarnos en aunar los puntos fuertes de ambos modelos.

Nos conviene utilizar un motor NFA por la flexibilidad y potencia que aporta, pero necesitamos que las expresiones regulares se parezcan lo más posible a las expresiones que aceptaría un DFA, para minimizar el backtracking que dichas expresiones pueden generar y optimizar así su cálculo.

## 4.4. ¿Qué utiliza Python?

Según el libro Friedl [2006], Python utiliza un motor «Traditional NFA». Por los ejemplos que se han realizado en este documento, parece ser así, pero queremos asegurarnos de que Python trabaja de esa manera con las expresiones regulares.

### 4.4.1. Documentos a inspeccionar

Para leer el código fuente de los módulos de Python, primero debemos saber dónde están.

Conocemos el módulo RE, así que con dos comandos en la consola de Python descubrimos dónde se encuentra el archivo con el código de dicho módulo:

Localizando el directorio del módulo RE

```
1 import re
2 re.__file__
```

Al abrir este archivo descubrimos que es una interfaz, y que la información que nos interesa está en otros documentos.
Con el mismo método vamos profundizando en la implementación, investigando los módulos que importan los módulos ya analizados.

Llega un momento en el que este procedimiento no es suficiente. Ya sea por una mejor gestión de recursos, ya sea porque Python se escribió inicialmente en C y esta parte no se ha traducido a Python, hay funciones que están compiladas y no se pueden analizar de esta manera.

Por suerte, Python es un lenguaje de código abierto, y podemos encontrar su código fuente en GitHub pyt (confirmado en Junio del 2019).

Para la siguiente interpretación se han tenido que estudiar los archivos «re.py», «sre_compile.py», «sre_parse.py», «sre_constants.py» y «_sre.pym», módulos que se instalan junto con Python, y los archivos «_sre.c», «sre.h» y «sre_lib.h», módulos que se encuentran compilados en la instalación de Python y que sólo he podido encontrar accediendo al código fuente de Python en GitHub pyt.

### 4.4.2. Interpretación del código fuente

Se realiza una «recursividad» cada vez que el motor se encuentra con varias opciones para completar la expresión regular.
La palabra recursividad se presenta entre comillas porque, aún sin aparecer llamadas recursivas en el código, es la palabra que mejor define el procesamiento de las expresiones regulares.

El motor de expresiones regulares cuenta con una **pila de estados**, en la que puede almacenar las opciones de las que dispone para completar la expresión regular que está procesando. Cada vez que encuentre una alternancia o una repetición, el motor de expresiones guardará las capturas que haya realizado hasta el momento, y distribuirá una copia a cada una de las opciones que acaba de leer, generando así nuevos estados.
El orden de introducción de los estados dependerá, en el caso de la alternancia del orden de las opciones en la estructura regular, y en el caso de la repetición de si el operador es *greedy* (captura lo máximo que puedas para aceptar la estructura) o *lazy* (captura lo mínimo que puedas para aceptar la estructura).

Si el motor, en algún momento del cómputo, descubre que las capturas realizadas no van a poder completar la estructura regular, inspecciona la pila de estados para comprobar si existe alguna opción guardada. Si es así, realiza «backtracking» y vuelve a intentarlo desde ese estado.

Por esta razón, en los ejemplos dados en el capítulo de introducción, los mayores tiempos de cómputo ocurrían cuando la expresión regular no era aceptada 1.2.1, o cuando tenía que desechar la mayoría de las opciones 1.2.2.

### 4.4.3. Conclusiones del código fuente

Para mejorar la eficiencia de las expresiones regulares se debe evitar dar demasiadas opciones a la expresión regular, porque cada vez que no se cumpla, va a recorrerlas todas en una búsqueda infructuosa.
Además debemos aprovechar que las capturas realizadas antes de que el motor se encuentre con una alternancia o con una repetición se quedan guardadas, para que el algoritmo repita cálculos que estaban bien realizados en un principio.

# 5 Equivalencias

Las equivalencias se demostrarán utilizando tanto las operaciones de los lenguajes 2.1.1 como las propiedades de expresiones regulares 6

## 5.1. Equivalencias demostradas con teoría de conjuntos

Caso $\alpha^* \cdot \alpha^* \equiv \alpha^*$

$$\mathcal{L}(\alpha^* \cdot \alpha^*) = \mathcal{L}(\alpha^*) \cdot \mathcal{L}(\alpha^*) = \bigcup_{n=0}^{\infty}(\mathcal{L}(\alpha^n)) \cdot \bigcup_{n=0}^{\infty}(\mathcal{L}(\alpha^n)) = \bigcup_{n=0}^{\infty}(\mathcal{L}(\alpha^n)) = \mathcal{L}(\alpha^*) \tag{31}$$

Como ambas expresiones dan lugar al mismo lenguaje, podemos concluir que son equivalentes.

Caso $\alpha \cdot \alpha^* \equiv \alpha^+$

Por definición, $\alpha^+$ se interpreta como «la expresión se repite una o más veces», y $\alpha^*$ se interpreta como «la expresión se repite cero o más veces». Como en la parte derecha de la equivalencia estamos forzando a que aparezca $\alpha$ al menos una vez al escribirla antes de $\alpha^*$, obtenemos que la concatenación $\alpha \cdot \alpha^*$ significa «la expresión se repite una o más veces».

Usando teoría de conjuntos:

$$\mathcal{L}(\alpha \cdot \alpha^*) = \mathcal{L}(\alpha^1) \cdot \bigcup_{n=0}^{\infty}(\mathcal{L}(\alpha^n)) = \bigcup_{n=0}^{\infty}(\mathcal{L}(\alpha^{n+1})) = \bigcup_{n=1}^{\infty}(\mathcal{L}(\alpha^n)) = \mathcal{L}(\alpha^+) \tag{32}$$

Como ambas expresiones dan lugar al mismo lenguaje, podemos concluir que son equivalentes.

Caso $\alpha^* \cdot \alpha \equiv \alpha \cdot \alpha^*$

Una parte de la equivalencia ya se ha demostrado en la fórmula 32.

$$\mathcal{L}(\alpha^* \cdot \alpha) = \bigcup_{n=0}^{\infty}(\mathcal{L}(\alpha^n)) \cdot \mathcal{L}(\alpha^1) = \bigcup_{n=0}^{\infty}(\mathcal{L}(\alpha^{n+1})) = \bigcup_{n=1}^{\infty}(\mathcal{L}(\alpha^n)) = \mathcal{L}(\alpha^+) \tag{33}$$

Como ambas expresiones dan lugar al mismo lenguaje, podemos concluir que son equivalentes.

Caso $\alpha? \cdot \alpha \equiv \alpha \cdot \alpha?$

$$\mathcal{L}(\alpha? \cdot \alpha) = [\mathcal{L}(\alpha^0) \cup \mathcal{L}(\alpha^1)] \cdot \mathcal{L}(\alpha^1) = \mathcal{L}(\alpha^0) \cdot \mathcal{L}(\alpha^1) \cup \mathcal{L}(\alpha^1) \cdot \mathcal{L}(\alpha^1) = \mathcal{L}(\alpha^1) \cup \mathcal{L}(\alpha^2)$$

$$\mathcal{L}(\alpha \cdot \alpha?) = \mathcal{L}(\alpha^1) \cdot [\mathcal{L}(\alpha^0) \cup \mathcal{L}(\alpha^1)] = \mathcal{L}(\alpha^1) \cdot \mathcal{L}(\alpha^0) \cup \mathcal{L}(\alpha^1) \cdot \mathcal{L}(\alpha^1)] = \mathcal{L}(\alpha^1) \cup \mathcal{L}(\alpha^2) \tag{34}$$

Como ambas expresiones dan lugar al mismo lenguaje, podemos concluir que son equivalentes.

Caso $\alpha?\alpha^* \equiv \alpha^*\alpha? \equiv \alpha^*$

$$\mathcal{L}(\alpha?\alpha^*) = [\mathcal{L}(\alpha^0) \cup \mathcal{L}(\alpha^1)] \cdot \bigcup_{n=0}^{\infty}(\mathcal{L}(\alpha^n)) = [\mathcal{L}(\alpha^0) \cdot \bigcup_{n=0}^{\infty}(\mathcal{L}(\alpha^n))] \cup [\mathcal{L}(\alpha^1) \cdot \bigcup_{n=0}^{\infty}(\mathcal{L}(\alpha^n))] =$$
$$= \bigcup_{n=0}^{\infty}(\mathcal{L}(\alpha^n)) \cup \bigcup_{n=1}^{\infty}(\mathcal{L}(\alpha^n)) = \bigcup_{n=0}^{\infty}(\mathcal{L}(\alpha^n)) = \mathcal{L}(\alpha^*)$$

$$\mathcal{L}(\alpha^*\alpha?) = \bigcup_{n=0}^{\infty}(\mathcal{L}(\alpha^n)) \cdot [\mathcal{L}(\alpha^0) \cup \mathcal{L}(\alpha^1)] = [\bigcup_{n=0}^{\infty}(\mathcal{L}(\alpha^n)) \cdot \mathcal{L}(\alpha^0)] \cup [\bigcup_{n=0}^{\infty}(\mathcal{L}(\alpha^n)) \cdot \mathcal{L}(\alpha^1)] =$$
$$= \bigcup_{n=0}^{\infty}(\mathcal{L}(\alpha^n)) \cup \bigcup_{n=1}^{\infty}(\mathcal{L}(\alpha^n)) = \bigcup_{n=0}^{\infty}(\mathcal{L}(\alpha^n)) = \mathcal{L}(\alpha^*) \tag{35}$$

Como las expresiones dan lugar al mismo lenguaje, podemos concluir que son equivalentes.

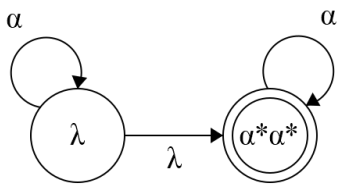


Ilustración 9: Autómata $\alpha^*\alpha^*$

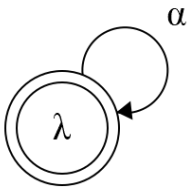


Ilustración 10: Autómata $\alpha^*$

## 5.2. Comparando equivalencias con autómatas finitos

Ya hemos demostrado algunas equivalencias entre expresiones regulares, pero no sabemos qué expresión se puede procesar mejor que otra.

A continuación modelaremos los autómatas finitos que se derivan directamente de las expresiones regulares con las que estamos trabajando, para comparar la complejidad de cada grafo.

Preferiremos las expresiones regulares que se representen directamente como autómatas deterministas (sin realizar ninguna optimización), y dentro de los autómatas deterministas, preferiremos aquellos que necesiten de los menores estados, y después los que necesiten menores transiciones.

Caso $\alpha^* \cdot \alpha^* \equiv \alpha^*$

Comparando los autómatas 9 y 17, podemos observar que el autómata 17 es el único de los dos que se puede representar con un autómata determinista, además de contar con menos estados y menos transiciones.
Por lo tanto, preferiremos usar el autómata 17 siempre que sea posible.

Caso $\alpha^* \cdot \alpha \equiv \alpha \cdot \alpha^*$

Comparando los autómatas 11 y 12, podemos observar que el autómata 12 es el único que representa un autómata determinista. Al poder llegarse a dos estados diferentes con una misma palabra, el autómata 11 debe elegir, y por lo que tiene que representarse con un autómata indeterminista.
Por lo tanto, preferiremos usar el autómata 12 siempre que sea posible.

Caso $\alpha? \cdot \alpha \equiv \alpha \cdot \alpha?$

Comparando los autómatas 13 y 14, podemos observar que el autómata 14 es el único que representa un autómata determinista. Además, el backtracking que causa equivocarse en el autómata 13 puede

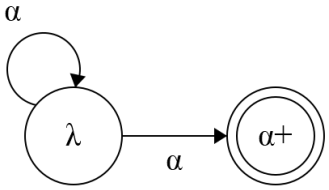


Ilustración 11: Autómata $\alpha^*\alpha$

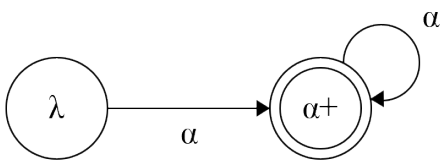


Ilustración 12: Autómata $\alpha\alpha^*$

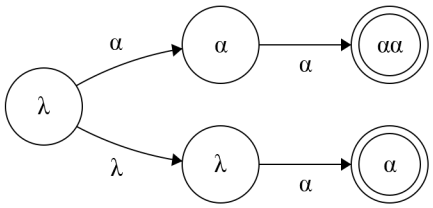


Ilustración 13: Autómata $\alpha?\alpha$

llegar a ser muy costoso a medida que se aumentan los caracteres $\alpha?$. Fijarse en el ejemplo 1.2.2.
Por lo tanto, preferiremos usar el autómata 14 siempre que sea posible.

Caso $\alpha?\alpha^* \equiv \alpha^*\alpha? \equiv \alpha^*$
Comparando los autómatas 15, 16 y 17, podemos observar que el autómata 17 es el único que representa un autómata determinista, además de contar con menos estados y menos transiciones.
Por lo tanto, preferiremos usar el autómata 17 siempre que sea posible.

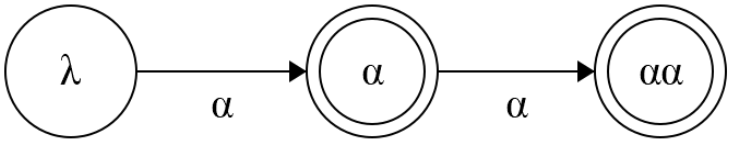


Ilustración 14: Autómata $\alpha\alpha?$

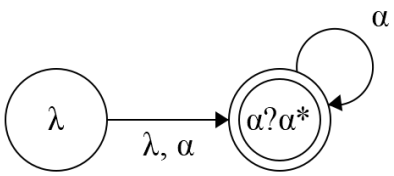


Ilustración 15: Autómata $\alpha?\alpha^*$

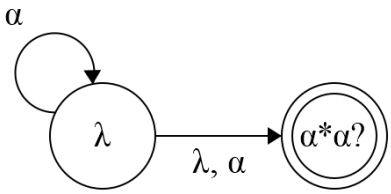


Ilustración 16: Autómata $\alpha^*\alpha?$

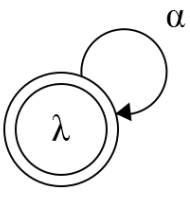


Ilustración 17: Autómata $\alpha^*$

# 6 Conclusiones

De este trabajo, se derivan las siguientes recomendaciones al trabajar con expresiones regulares:

1. Que la expresión regular cumpla con los requerimientos. Una vez confirmemos que es correcta, podemos plantearnos optimizarla.
2. Hacer la expresión regular lo más clara posible. Expresar adecuadamente lo que se necesita, y lo que no se necesita en la expresión regular.
    *a*) No suele significar lo mismo que hacer la expresión lo más corta posible. El ejemplo 1.2.1 es significativo, en ese caso la expresión más precisa es la más larga de las dos.
    *b*) Estudiar la expresión y analizar si hay operadores innecesarios (fijarse en estructuras de tipo 35).
3. Si hay estructuras obligatorias y estructuras opcionales en la expresión regular, intentar que las obligatorias estén más a la izquierda que las opcionales para intentar capturar caracteres lo antes posible.
    *a*) De esta manera, si hay que realizar «backtracking», la profundidad de la búsqueda es menor.
4. Si es necesario, intentar que en las alternancias se aproveche cada caso de la mejor manera posible.
    *a*) Fijarse en la propiedad distributiva de las expresiones regulares 26.
    *b*) Puede dificultad la comprensión de la estructura regular, y por tanto su mantenibilidad. Utilizar esta regla con cautela.

# A Ejemplo de código SVG válido

Este código se puede escribir en un archivo .svg para comprobar que codifica una imagen y que cumple la especificación del lenguaje SVG.

Código SVG de ejemplo

```
<svg width="300" height="200" version="1.1"
     xmlns="http://www.w3.org/2000/svg">
<ellipse stroke="black" stroke-width="1" fill="none" cx="54.5"
         cy="118.5" rx="30"  ry="30"/>
<text x="49.5" y="124.5" font-family="Times New Roman"
      font-size="20">&#955;
</text>
<ellipse stroke="black" stroke-width="1" fill="none" cx="210.5"
         cy="118.5" rx="30"  ry="30"/>
<text x="199.5" y="124.5" font-family="Times New Roman"
      font-size="20">&#945;+</text>
<ellipse stroke="black"
      stroke-width="1"  fill="none" cx="210.5" cy="118.5" rx="24"
      ry="24"/>
<polygon stroke="black" stroke-width="1" points="84.5,118.5
                                                  180.5,118.5"/>
<polygon fill="black" stroke-width="1" points="180.5,118.5 172.5,113.5
                                               172.5,123.5"/>
<text x="127.5" y="139.5" font-family="Times New Roman"
      font-size="20">&#945;
</text>
<path stroke="black" stroke-width="1" fill="none" d="M 224.003,91.842
                                                     A 22.5,22.5 0 1 1
                                                     239.873,113.002"/>
<text x="270.5" y="70.5" font-family="Times New Roman"
      font-size="20">&#945;</text>
<polygon fill="black" stroke-width="1" points="239.873,113.002
                                               247.948,117.88
                                               247.796,107.881"/>
</svg>
```

# Referencias


Curso: Lenguajes formales, 2013.
https://ocw.unican.es/course/view.php?id=116

Mastering regular expressions, 2002.
https://se.ifmo.ru/~ad/Documentation/Mastering_RegExp/main.html

Código fuente del compilador de python.
https://github.com/python/cpython

J. E. F. Friedl. *Mastering Regular Expressions*. O'Reilly, 3rd ed. edition, 2006.

*"Lenguajes Formales" (para Ingenieros Informáticos)*. Universidad de Cantabria, 2013.
https://ocw.unican.es/pluginfile.php/1038/course/section/1209/material_teorico_del_curso_lenguajes_formales.pdf

E. Wallace. Finite state machine designer.
http://madebyevan.com/fsm/