

Teoría de la Computación

Expresiones regulares

Construcción de Thompson, eliminación de estados y el teorema de Kleene

Gerson Garrido

Semana 3

UNIVERSIDAD NACIONAL DE INGENIERÍA

Hoy

Sesión A

1. Describir en vez de reconocer
2. La definición y la precedencia
3. Leer y escribir expresiones
4. Identidades y trampas
5. De expresión a NFA: Thompson
6. Las abreviaturas de las herramientas

Sesión B

7. La dirección difícil
8. Autómatas con expresiones en las flechas
9. Eliminar un estado
10. El orden de eliminación
11. El teorema de Kleene
12. Cómo busca un motor real

Lectura

Apuntes de la semana 3. Apoyo: Sipser, sec. 1.3; Erickson, *Models of Computation*, notas 2 y 4. Código en `code/week-03`.

SECCIÓN 01

Expresiones regulares

01



Describir en vez de reconocer

Buscar las líneas de un registro que tienen una fecha:

un patrón

```
[0-9][0-9]/[0-9][0-9]/[0-9][0-9][0-9][0-9]
```

No dice qué recordar ni en qué orden leer: dice **qué forma tiene** lo que se busca.

- **grep y los editores**: `g/re/p`, de *global regular expression print*.
- **Validadores**: un código postal, un teléfono.
- **Lexers**: la forma de cada token de un lenguaje.

El plan de la semana

Sesión A: de la **descripción** a la **máquina**. Sesión B: de la máquina a la descripción.

La definición es inductiva

Expresión regular ***(regular expression)*** **sobre** Σ

R es una expresión regular si es de una de estas seis formas:

1. a, para algún $a \in \Sigma$
2. ε
3. $\emptyset$
4. $(R_1 \cup R_2)$
5. $(R_1 R_2)$
6. (R_1^*)

donde R_1 y R_2 son expresiones regulares más pequeñas.

- Tres casos base y tres operaciones: las **operaciones regulares** de la semana 1.
- No es circular: toda expresión se descompone hasta llegar a símbolos, ε y $\emptyset$.
- En la teoría la unión es $\cup$. En el teclado y en el código es `|`: `(0|1)*1`.

El lenguaje que describe una expresión

Expresión R	**Lenguaje** $L(R)$
a	$\{a\}$
ε	$\{\varepsilon\}$
$\emptyset$	$\emptyset$
$(R_1 \cup R_2)$	$L(R_1) \cup L(R_2)$
$(R_1 R_2)$	$L(R_1)L(R_2)$
(R_1^*)	$L(R_1)^*$

- $L(0) = \{0\},\ L(1) = \{1\}$
- $L((0 \cup 1)) = \{0, 1\}$
- $L(((0 \cup 1)^*)) = \{0, 1\}^* = \Sigma^*$
- Una máquina **reconoce**. Una expresión **describe**.
- $R_1 = R_2$ quiere decir $L(R_1) = L(R_2)$.

Dos objetos distintos

R es una fórmula finita. $L(R)$ es un conjunto de cadenas, casi siempre infinito. Dentro de una expresión, ε y $\emptyset$ son **expresiones**: describen $\{\varepsilon\}$ y $\emptyset$.

Precedencia: estrella, concatenación, unión

Como la potencia, el producto y la suma:

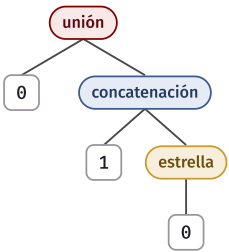
$$a + bc^2 = a + (b(c^2)) \qquad 0 \cup 10^* = 0 \cup (1(0^*))$$

- La estrella se aplica solo a lo que tiene inmediatamente a su izquierda: un símbolo, o un grupo entre paréntesis.
- Con los paréntesis puestos, una expresión es un **árbol**. La última operación que se aplica está en la raíz.
- El **tamaño** $|R|$ es el número de nodos del árbol.

Una expresión es un árbol

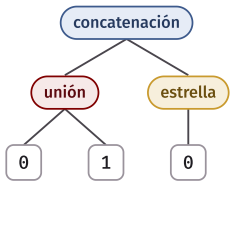
$0 \cup 10^*$

$(0 \cup (1(0^*)))$



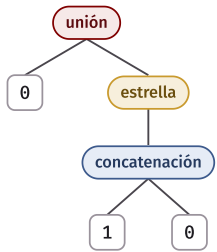
$(0 \cup 1)0^*$

$((0 \cup 1)(0^*))$



$0 \cup (10)^*$

$(0 \cup ((10)^*))$



Los mismos símbolos en el mismo orden. Lo que liga más fuerte queda más abajo. Los tres tienen tamaño 6.

Los mismos símbolos, tres lenguajes

Expresión	Con sus paréntesis	Describe
$0 \cup 10^*$	$(0 \cup (1(0^*)))$	`0`, o un `1` seguido de ceros
$(0 \cup 1)0^*$	$((0 \cup 1)(0^*))$	un símbolo cualquiera seguido de ceros
$0 \cup (10)^*$	$(0 \cup ((10)^*))$	`0`, o repeticiones de `10`

Error común

Leer 01^* como "repeticiones de `01`". Es $0(1^*)$: un cero seguido de unos. Las repeticiones de `01` son $(01)^*$. Y $0 \cup 1^*$ no es $(0 \cup 1)^*$. Si dudan, pongan el paréntesis.

En el código: `|`, `%` y `@`

Teoría	Código
$R_1 \cup R_2$	`R1|R2`
$R_1 R_2$	`R1R2`
R^*	`R*`
ε	`%`
$\emptyset$	`@`

Un símbolo es una letra o un dígito. Al imprimir, los programas escriben ε y $\emptyset$.

terminal

```
$ build/regex_tree '(0|1)0*'
expresión:           (0|1)0*
con sus paréntesis:  ((0|1)(0*))
nodos del árbol:     6
concatenación
    unión
        0
        1
    estrella
        0
```

- El parser tiene una función por nivel de precedencia: `parse_union` → `parse_concat` → `parse_star` → `parse_atom`.
- Lo que liga más fuerte queda más abajo en las llamadas, y más abajo en el árbol.

SECCIÓN 02

Leer y escribir expresiones

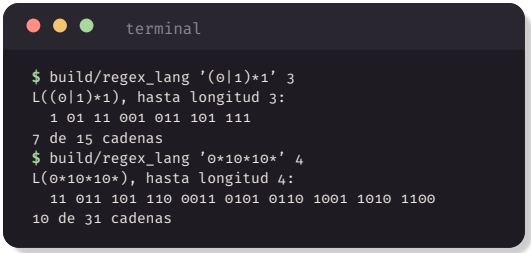
02



Leer: de adentro hacia afuera

- $(0 \cup 1)^*1$
  cualquier cadena y un 1: **termina en 1**
- $(\Sigma\Sigma)^*$
  trozos de dos símbolos: **longitud par**
- $0^*10^*10^*$
  dos unos y ceros alrededor: **exactamente dos unos**

$0110 = 0 \cdot 1 \cdot \varepsilon \cdot 1 \cdot 0$: una parte por factor.
Basta que exista **alguna** manera de partir la cadena.



Escribir: ¿en qué trozos se parte una cadena?

Número par de ceros

- Los unos, donde sea. Los ceros, de dos en dos.
- Unos, y bloques con dos ceros: $1^*(01^*01^*)^*$
- O trozos sueltos, más corta: $(1 \cup 01^*0)^*$

Sin 11

- No hay "no": se dice en positivo. Todo `1` va seguido de un `0`, salvo el último símbolo.
- Bloques `0` y `10`, y puede sobrar un `1`: $(0 \cup 10)^*(\varepsilon \cup 1)$

Diseñar

Un autómata: **qué recordar**. Una expresión: **en qué trozos se parte** una cadena.

Demostrar que una expresión es correcta

Afirmación. $L((0 \cup 10)^*(\varepsilon \cup 1))$ es el lenguaje de las cadenas sin `11`. Por doble inclusión.

- **No describe de más** ($\subseteq$). En $(0 \cup 10)^*$, cada `1` abre un bloque `10`: le sigue un `0`. El `1` opcional del final va después de un `0`, o es toda la cadena. Nunca aparece `11`.
- **No describe de menos** ($\supseteq$). Sea w sin `11`. De izquierda a derecha:
  - sigue un `0`: bloque `0`;
  - sigue un `1` que no es el último: el siguiente es `0`, bloque `10`;
  - sigue un `1` y es el último: la parte final.

Dos inclusiones

Una por cada manera de equivocarse.

Probar con cadenas de los dos lados

Primer intento: $(0 \cup 10)^*$. **Describe de menos.**

terminal

```
$ build/regex_diff '(0|10)*' '(0|10)*(%|1)' 6
R1 = (0|10)*
R2 = (0|10)*(ε|1)
127 cadenas de longitud a lo sumo 6
solo en R1: 0
solo en R2: 20. Las más cortas: 1 01 001 101 0001 0101
```

Segundo: $(0 \cup 10)^*1^*$. **Describe de más.**

terminal

```
$ build/regex_diff '(0|10)*1*' '(0|10)*(%|1)' 6
R1 = (0|10)*1*
R2 = (0|10)*(ε|1)
127 cadenas de longitud a lo sumo 6
solo en R1: 26. Las más cortas: 11 011 111 0011 0111 1011
solo en R2: 0
```

Error común

Comprobar la expresión solo con ejemplos del lenguaje. Esa prueba pasa siempre: Σ^* también describe todas las cadenas sin `11`. Hay que probar con cadenas que **deben estar** y con cadenas que **no**. Y siempre con ε.

Identidades y trampas

Valen para toda R

$R \cup \emptyset = R$	$\emptyset$ hace de cero
$R\varepsilon = \varepsilon R = R$	ε hace de uno
$R\emptyset = \emptyset R = \emptyset$	$r \cdot 0 = 0$
$\emptyset^* = \varepsilon$	cero trozos
$\varepsilon^* = \varepsilon$	
$R \cup R = R$	
$(R^*)^* = R^*$	

Son falsas

- $R \cup \varepsilon = R$
  con $R = 0$: $\{\varepsilon, 0\} \neq \{0\}$
- $R\emptyset = R$
  con $R = 0$: $\emptyset \neq \{0\}$

Error común

En la unión no hace nada $\emptyset$. En la concatenación no hace nada ε. Cruzados, uno agrega una cadena y el otro borra el lenguaje.

SECCIÓN 03

De expresión a NFA: Thompson

03

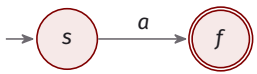
La idea: una pieza por cada caso

- La definición tiene seis casos. Para los tres casos base, el autómata se escribe a mano.
- Para la unión, la concatenación y la estrella: las construcciones de cierre de la semana pasada, con NFA y transiciones ε.
- Se recorre el árbol de las hojas a la raíz. Cada nodo arma su autómata con los de sus hijos.

> **La disciplina de Thompson (1968)**
>
> Cada pieza tiene **un solo estado inicial** s y **un solo estado de aceptación** f, distintos. Al combinar piezas solo se agregan estados nuevos y transiciones ε: nunca se toca lo que hay dentro.

Una pieza es una caja: para conectarla no hace falta saber qué tiene adentro.

Los casos base



un símbolo a



ε

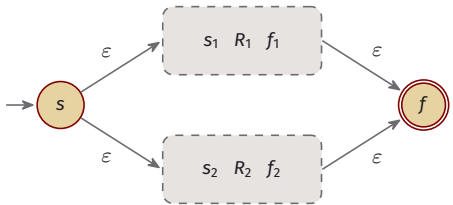


$\emptyset$: sin flechas

- Para a: la única manera de ir de s a f es leyendo a.
- Para ε: sin leer nada.
- Para $\emptyset$: no hay ninguna manera. El autómata no acepta nada.

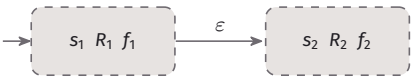
Unión y concatenación

$R_1 \cup R_2$: dos estados nuevos



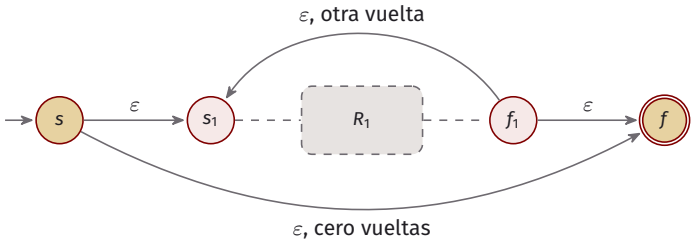
Desde s se adivina cuál de las dos piezas recorrer.

$R_1 R_2$: ningún estado nuevo



La salida de la primera se une con la entrada de la segunda. Entrada: s_1. Salida: f_2.

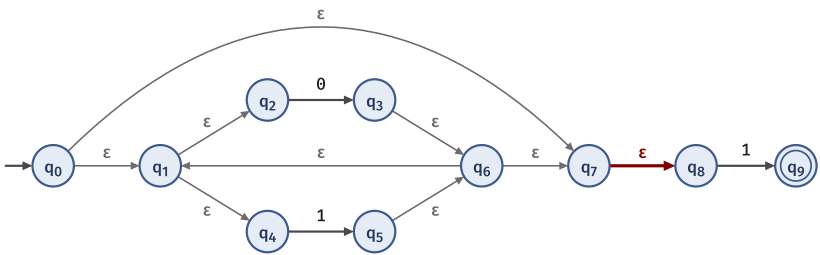
Estrella



- Desde s: ir directo a f, o entrar a la pieza.
- Al llegar a f_1: volver a s_1 para dar otra vuelta, o salir a f.
- La semana pasada: marcar el estado inicial de la pieza como de aceptación puede aceptar cadenas de más. Aquí s y f son nuevos y no forman parte de la pieza.

Un ejemplo completo: $(0 \cup 1)^*1$

EXPRESIÓN $(0 \cup 1)^*1$



Concatenación: ningún estado nuevo, solo la flecha ε de q_7 a q_8.

El autómata completo: 10 estados, entra por q_0 y sale por q_9.

El mismo autómata, como archivo y en marcha

terminal

```
$ build/thompson '(0|1)*1'
# NFA de Thompson de (0|1)*1
states: q0 q1 q2 q3 q4 q5 q6 q7 q8 q9
alphabet: 0 1
start: q0
accept: q9
delta:
q0 eps q1 q7
q1 eps q2 q4
q2 0 q3
q3 eps q6
q4 1 q5
q5 eps q6
q6 eps q1 q7
q7 eps q8
q8 1 q9
```

terminal

```
$ build/run_nfa machines/ends_in_1.nfa 011 010
w = 011
  inicio  {q0, q1, q2, q4, q7, q8}
  lee 0   {q1, q2, q3, q4, q6, q7, q8}
  lee 1   {q1, q2, q4, q5, q6, q7, q8, q9}
  lee 1   {q1, q2, q4, q5, q6, q7, q8, q9}
  hay un estado de F entre los activos: acepta
w = 010
  inicio  {q0, q1, q2, q4, q7, q8}
  lee 0   {q1, q2, q3, q4, q6, q7, q8}
  lee 1   {q1, q2, q4, q5, q6, q7, q8, q9}
  lee 0   {q1, q2, q3, q4, q6, q7, q8}
  ningún estado activo está en F: rechaza
```

Antes de leer nada ya está en seis estados. No sabe cuál `1` será el último: sigue las dos posibilidades.

Por qué funciona: inducción sobre la estructura

Afirmación

En la pieza de R: hay un camino de s a f que deletrea w si y solo si $w \in L(R)$.

- **Casos base.** Un solo camino, que deletrea a o ε. Para $\emptyset$, ninguno.
- **Unión.** De s solo salen dos ε. De una pieza solo se sale por su f_i, hacia f.
- **Concatenación.** Una sola flecha une las piezas, sin vuelta: antes, $u \in L(R_1)$; después, $v \in L(R_2)$.
- **Estrella.** El camino directo deletrea ε. Cualquier otro se corta en cada uso de la flecha de vuelta: cada trozo va de s_1 a f_1 y deletrea una cadena de $L(R_1)$.

Seis casos, tres veces: la definición, la demostración y el `switch` de `build_fragment` en `thompson.h`.

Cuántos estados tiene

- Cada caso base crea 2 estados. La unión y la estrella, 2 cada una. La concatenación, ninguno.

$$\text{estados} = 2 \cdot (\text{hojas} + \text{uniones} + \text{estrellas}) \leq 2\,|R|$$

- Para $(0 \cup 1)^*1$: 3 hojas, 1 unión, 1 estrella. $2 \cdot 5 = 10$ estados.
- De cada estado salen **a lo sumo dos** flechas: a lo sumo $4\,|R|$ flechas.
- A mano bastan 2 estados para “termina en 1”. Los otros 8 son el precio de no pensar.

Idea clave

El NFA de Thompson es **lineal** en el tamaño de la expresión. Pasar de la descripción a una máquina no determinista es barato. Lo caro, cuando hace falta, es la construcción de subconjuntos.

Las abreviaturas de las herramientas

Herramienta	Tres operaciones
`R+`	RR^*
`R?`	$R \cup \varepsilon$
`[abc]`	$a \cup b \cup c$
`[a-z]`	$a \cup b \cup \cdots \cup z$
`.`	Σ
`R{2,3}`	$RR(R \cup \varepsilon)$

- Se traducen operador por operador: **no agregan poder**.
- Sí ahorran escritura: `[a-z]{1000}` son once caracteres y 1000 copias.
- `grep R` busca **dentro** de la línea: $\Sigma^* R \Sigma^*$. Con `-x`, la línea completa.

terminal

```
$ printf '1\n0110\n\n0111\n' | grep -xE '(0|1)*1'
1
0111
```

SECCIÓN 04

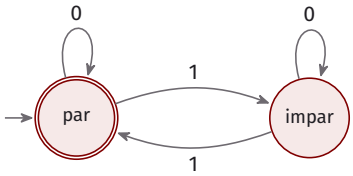
De DFA a expresión regular

04



La dirección difícil

- Toda expresión tiene su autómata. ¿Todo lenguaje regular tiene su expresión?
- No es evidente: las expresiones no tienen complemento ni intersección.
- La paridad: una cadena aceptada es una secuencia de **vueltas** desde *par* hasta *par*.
- Vuelta corta: 0. Vuelta larga: 10^*1.
- Entonces: $(0 \cup 10^*1)^*$



La idea de la sesión

Todo lo que pasa en *impar* quedó resumido en 10^*1: el estado desapareció y una flecha pasó a llevar una **expresión**.

Autómatas con expresiones en las flechas

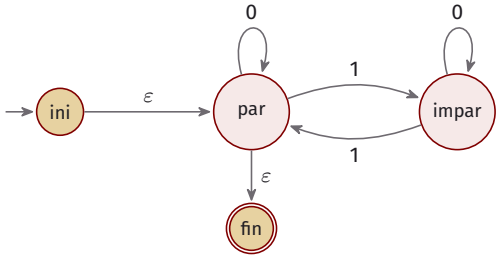
GNFA *(generalized nondeterministic finite automaton)*

Un autómata en el que cada flecha lleva una expresión regular, y:

1. al estado inicial q_{ini} **no entra** ninguna flecha;
2. hay un único estado de aceptación $q_{\text{fin}} \neq q_{\text{ini}}$, del que **no sale** ninguna flecha;
3. entre cada par de estados hay una flecha en cada sentido, y cada estado tiene su ciclo, salvo lo que prohíben 1 y 2.

- Una flecha que no está dibujada lleva $\emptyset$: no se puede usar nunca.
- Lee **por trozos**: para pasar de q_i a q_j consume un trozo de $L(R_{ij})$.
- Acepta w si $w = w_1 w_2 \cdots w_k$ y hay estados $q_{\text{ini}} = p_0, p_1, \ldots, p_k = q_{\text{fin}}$ con cada w_i en el lenguaje de la flecha de p_{i-1} a p_i.

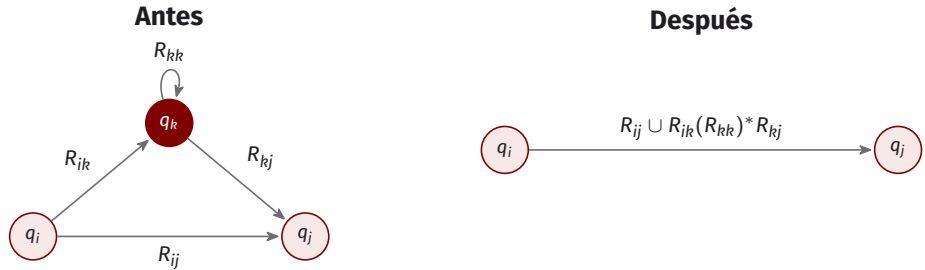
Normalizar un DFA



1. Un estado inicial nuevo, con una flecha ε al inicial del DFA.
2. Un estado de aceptación nuevo, con una flecha ε desde cada estado de aceptación del DFA.
3. Varias flechas entre los mismos dos estados se juntan en una, con la unión de sus símbolos.

Acepta lo mismo que el DFA: el primer y el último trozo son ε, y cada trozo intermedio es un símbolo.

Eliminar un estado



- Un camino por q_k: entra, da cero o más vueltas en su ciclo, y sale.
- La regla se aplica a **cada pareja** (q_i, q_j) de los estados que quedan. También con $i = j$.
- Solo cambian las parejas en las que q_i entra a q_k y q_k sale a q_j: con $R_{ik} = \emptyset$, el desvío es $\emptyset$.
- Sin ciclo, $R_{kk} = \emptyset$ y el factor del medio es $\emptyset^* = \varepsilon$.

El método, a mano

1. Listar los que **entran** a q_k.
2. Listar los que **salen** de q_k.
3. Anotar el **ciclo** R_{kk}, si lo hay.
4. Por cada combinación de uno que entra y uno que sale: unir $R_{ik}(R_{kk})^* R_{kj}$ a la flecha de q_i a q_j.

- Con m que entran y n que salen: $m \cdot n$ parejas.
- Un estado puede estar en las dos listas: la pareja (q_i, q_i) repara su ciclo.

Error común

Olvidar el ciclo de q_k o alguna de las parejas. Sin el ciclo se pierden las cadenas que dan vueltas en q_k. Cada pareja que falta quita cadenas del lenguaje. Hagan las dos listas antes de calcular, y cuenten.

La paridad, paso a paso

Se elimina *impar*

- Entra: *par*, con `1`. Sale: *par*, con `1`.
- Ciclo: `0`. Una pareja, (*par*, *par*):

$$0 \cup 1\,(0)^*\,1$$

Se elimina *par*

- Entra q_{ini} y sale q_{fin}, los dos con ε.

$$\emptyset \cup \varepsilon\,(0 \cup 10^*1)^*\,\varepsilon = (0 \cup 10^*1)^*$$

```
eliminate even_ones.dfa impar par

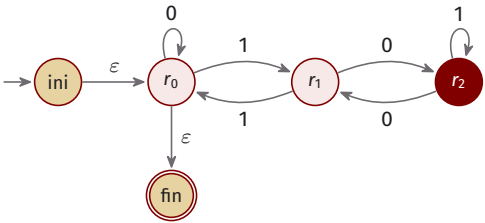
Se elimina impar
  estados: ini par fin
  ini -> par :  ε
  par -> par :  0|10*1
  par -> fin :  ε

Se elimina par
  estados: ini fin
  ini -> fin :  (0|10*1)*

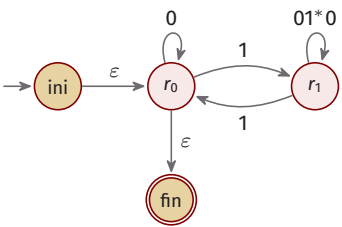
Expresión: (0|10*1)*
```

Un ejemplo completo: los múltiplos de 3

El GNFA inicial



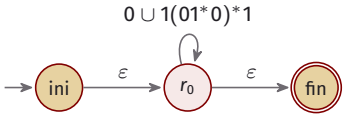
Se elimina r_2



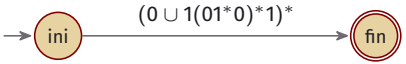
- r_i: el valor de lo leído deja resto i al dividir entre 3.
- A r_2 entra r_1 con 0, y sale a r_1 con 0. Ciclo: 1. Una pareja, (r_1, r_1): $\emptyset \cup 0\,(1)^*\,0 = 01^*0$.
- Las otras cinco flechas no cambian: sus parejas no tenían desvío por r_2.

Los múltiplos de 3: se eliminan r_1 y r_0

Se elimina r_1



Se elimina r_0



- A r_1 entra r_0 con 1, y sale a r_0 con 1. Ciclo: 01*0. La pareja (r_0, r_0) ya tenía el 0.
- A r_0 entra q_{ini} y sale q_{fin}, con ε: $\emptyset \cup \varepsilon\,(\cdots)^*\,\varepsilon$.
- Quedan dos estados y una flecha. Su expresión es la respuesta.

Los múltiplos de 3 en binario

$(0 \cup 1(01^*0)^*1)^*$: vueltas desde el resto 0. La corta es un 0. La larga sale con un 1, hace excursiones 01*0 al resto 2 y vuelve con un 1.

Por qué funciona la eliminación

Afirmación

Si G' resulta de eliminar q_k en G con la regla, G y G' aceptan las mismas cadenas.

- **De G a G'.** Cada racha $q_i, q_k, \ldots, q_k, q_j$ lee un trozo de R_{ik}, cero o más de R_{kk} y uno de R_{kj}: se reemplaza por un solo paso de q_i a q_j.
- **De G' a G.** Cada paso lee un trozo de R_{ij}, y existe en G, o uno de $R_{ik}(R_{kk})^* R_{kj}$, y en G se hace el desvío por q_k.

Teorema

Para todo DFA M hay una expresión R con $L(R) = L(M)$: normalizar, eliminar los estados en cualquier orden, leer la flecha que queda.

El orden cambia la expresión, no el lenguaje

terminal

```
$ build/orders machines/multiple_of_3.dfa
r0 r1 r2  (65 nodos)
    0*|0*1(10*1)*10*|0*1(10*1)*0(1|0(10*1)*0)*0(10*1)*10*
r0 r2 r1  (27 nodos)
    0*|0*1(10*1|01*0)*10*
r1 r0 r2  (50 nodos)
    (0|11)*|(0|11)*10(1|00|01(0|11)*10)*01(0|11)*
r1 r2 r0  (21 nodos)
    (0|11|10(1|00)*01)*
r2 r0 r1  (27 nodos)
    0*|0*1(01*0|10*1)*10*
r2 r1 r0  (14 nodos)
    (0|1(01*0)*1)*
```

- Seis expresiones para el mismo lenguaje. En las dos más cortas, r_0 se elimina **al final**.
- Regla práctica: primero los estados con pocas flechas; los más conectados, al final.

Las expresiones crecen rápido

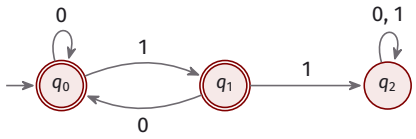
- La flecha nueva contiene hasta cuatro anteriores: R_{ij}, R_{ik}, R_{kk}, R_{kj}.
- Si todas tienen tamaño a lo sumo s, la nueva tiene hasta $4s + 4$.
- Después de n eliminaciones: del orden de 4^n.
- Para algunos autómatas, **toda** expresión equivalente es exponencialmente más grande.

terminal

```
$ build/growth
 k   estados   nodos (ro primero)   nodos (ro al final)
 2          2                   22                     9
 3          3                   65                    14
 4          4                  110                    27
 5          5                  314                    52
 6          6                  384                    67
 7          7                 1341                   209
 8          8                 1636                   267
```

El DFA de k estados de los múltiplos de k en binario.

Lo que a mano costó, sale solo



"Sin 11", con su estado trampa q_2.

- De q_2 no sale ninguna flecha: no hay parejas que reparar.
- Sale la expresión de la sesión A.

```
eliminate no_11.dfa q2 q1 q0

Se elimina q1
  estados: ini q0 fin
  ini -> q0 :  ε
  q0 -> q0 :  0|10
  q0 -> fin :  ε|1

Se elimina q0
  estados: ini fin
  ini -> fin :  (0|10)*(ε|1)
```

SECCIÓN 05

El teorema de Kleene

05

El teorema de Kleene

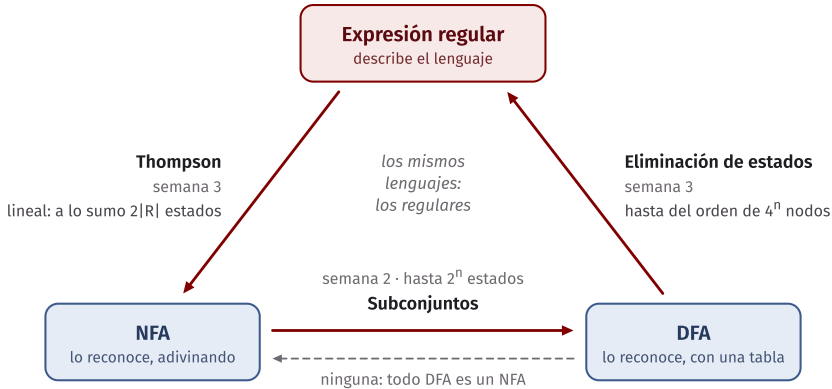
Teorema (Kleene, 1956)

Un lenguaje es regular si y solo si alguna expresión regular lo describe.

Las dos direcciones ya están hechas:

- De la expresión a la máquina: Thompson da un NFA, y la construcción de subconjuntos lo convierte en un DFA.
- De la máquina a la expresión: la eliminación de estados.
- El mismo poder, no la misma economía: se elige el modelo según el problema.
- ¿Son equivalentes dos expresiones? Se convierten en DFA y se comparan los autómatas.

Tres modelos, una sola clase de lenguajes



SECCIÓN 06

Cómo busca un motor real

06

Simular el NFA

pseudocódigo

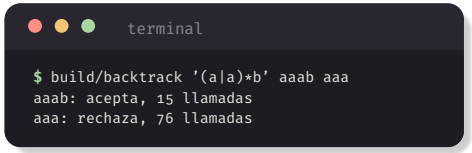
```
algoritmo simular(N, w)
    activos ← clausura_ε({q0})
    para cada símbolo a de w, en orden
        siguientes ← ∅
        para cada estado q de activos
            agregar a siguientes los estados de δ(q, a)
        fin para
        activos ← clausura_ε(siguientes)
    fin para
    devolver si activos contiene un estado de F
fin algoritmo
```

- Los activos nunca son más de m, el número de estados.
- Una vuelta: cada estado y cada flecha, una vez. $O(m)$.
- Con $n = |w|$: $O(n \cdot m)$, y $m \leq 2\,|R|$.

Siempre $O(|w| \cdot |R|)$, para toda expresión y toda cadena. La memoria, $O(m)$.

Backtracking: probar los caminos uno por uno

- En cada elección se prueba una opción. Si falla más adelante, se vuelve y se prueba la otra.
- Fácil de programar, y casi siempre rápido.
- $(a \cup a)^* b$ contra a^n: cada `a` se lee con una rama o con la otra. Son 2^n combinaciones, y todas fallan al final: falta la `b`.



$S(n)$: llamadas de la estrella con n letras.
$S(0) = 5$, $S(n) = 5 + 2\,S(n-1)$
$S(n) = 10 \cdot 2^n - 5$
Con la llamada inicial, $10 \cdot 2^n - 4$: para $n = 3$, 76.

Cada letra más duplica el trabajo. Para rechazar, tiene que convencerse de que **ninguna** combinación funciona.

La medición

terminal

```
$ make bench
build/matchers_bench
expresión: (a|a)*b    cadena: a^n    NFA de Thompson: 10 estados

      n   backtracking: llamadas   backtracking: tiempo   NFA: tiempo
      5                       316              0.0000 s      0.0000 s
     10                     10236              0.0000 s      0.0000 s
     15                    327676              0.0013 s      0.0000 s
     20                  10485756              0.0434 s      0.0000 s
     25                 335544316              1.3842 s      0.0000 s
   1000                         -                     -      0.0009 s
 100000                         -                     -      0.0871 s
```

- Con `-O2`, en el contenedor del curso. Los tiempos cambian de máquina a máquina; las llamadas no.
- Con $n = 35$ serían 2^{10} veces más: más de veinte minutos.

Caminos frente a estados

- Las dos búsquedas exploran el mismo autómata.
- Después de leer k letras hay 2^k caminos, y todos están en el mismo lugar de la expresión.
- El backtracking los visita uno por uno. La simulación guarda un **conjunto**: los 2^k caminos ocupan un solo lugar.

La idea de la semana 1, otra vez

Lo único que importa de lo leído es el estado al que se llegó.

En la práctica: las dos familias existen

- **Con autómatas:** `grep` y las bibliotecas de Go y de Rust. Tiempo lineal en el texto.
- **Con backtracking:** las de Perl, Python, Java y JavaScript.
- En el contenedor del curso, con $(a \cup a)^* b$ contra a^n:
  - `re.fullmatch` de Python 3.14: 0,05 s con $n = 20$ y 1,47 s con $n = 25$;
  - `grep -xE`: 0,001 s con $n = 100\,000$.
- Un motor con autómatas suele guardar cada conjunto de activos la primera vez que lo calcula: la construcción de subconjuntos, **sobre la marcha**.

Cuando el patrón o el texto vienen de un usuario

Un patrón así es un ataque: unas decenas de caracteres ocupan un procesador durante horas.

Lo que no es regular: las referencias hacia atrás

un patrón de grep

```
((a|b)*)\1
```

`\1`: “aquí va otra vez el texto que coincidió con el primer paréntesis”.

- Coincide con `abab` y con `bbabba`. No con `abba`.
- Describe $\{ ww : w \in \{a, b\}^* \}$.
- No es una abreviatura: ese lenguaje **no es regular**. Hay que recordar la primera mitad entera.

Por ahora, solo el enunciado

La demostración llega la semana próxima, con las herramientas para probar que un lenguaje no es regular. Las *regex* con referencias hacia atrás describen más que los lenguajes regulares: ningún autómata finito las ejecuta, y de ahí el backtracking.

Laboratorio: un `grep` pequeño

1. Leer una expresión con `|`, concatenación, `*` y paréntesis, y una lista de líneas. Responder `acepta` o `rechaza` por cada línea **completa**.
2. Construir el árbol, respetando la precedencia.
3. Convertirlo en un NFA con la construcción de Thompson.
4. Decidir cada línea con el conjunto de estados activos, sin backtracking.
5. Pasar el caso grande: `(a|a)*b` contra una línea larga de `a`.

Dónde: `code/week-03/lab/mini_grep`. `make test` corre el juez. Cuidado con los ciclos de transiciones ε: `(a*)*`.

casos/01.in

```
(a|b)*abb
strings: 5
abb
babb

ab
abba
```

casos/01.out

```
acepta
acepta
rechaza
rechaza
rechaza
```

Resumen

- Una expresión regular **describe** un lenguaje con unión, concatenación y estrella. Precedencia: estrella, concatenación, unión.
- Se escribe pensando en cómo se parte una cadena, y se prueba con cadenas de los dos lados.
- **Thompson**: una pieza por nodo del árbol, con una entrada y una salida. A lo sumo $2\,|R|$ estados.
- **Eliminación de estados**: cada pareja pasa a $R_{ij} \cup R_{ik}(R_{kk})^* R_{kj}$. El orden cambia la expresión, no el lenguaje.
- **Kleene**: DFA, NFA y expresiones regulares describen los mismos lenguajes.
- Simular el NFA cuesta $O(|w| \cdot |R|)$ siempre. El backtracking puede ser exponencial.

Para la próxima clase

- Leer los apuntes de la semana 3 y terminar el laboratorio.
- Preguntas de repaso de la guía: qué describen $(0 \cup \varepsilon)1^*$ y $(01)^* \cup (10)^*$; convertir $(ab \cup a)^*$ en un NFA; eliminar los estados de un DFA de dos estados en los dos órdenes.
- Próxima semana: los límites de lo regular. Cómo se demuestra que un lenguaje **no** es regular.

Pregunta

Toda la semana construimos autómatas y expresiones para lenguajes que sí son regulares. ¿Cómo se demostraría que para $\{ 0^n 1^n : n \geq 0 \}$ no existe **ninguno**, si no se pueden probar todos?