Informática Teórica y Lenguajes Formales

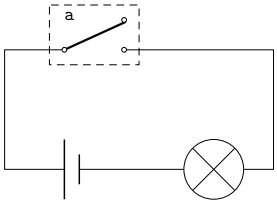
Ejercicios temas 2, 3

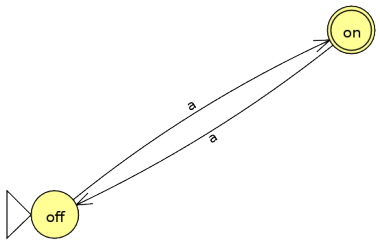
Grado en Inteligencia Artificial

Autómatas finitos

Autómatas deterministas

Para ilustrar el concepto de *autómata finito*, el primer ejemplo que ofrece el libro *Introduction to automata theory, languages, and computation* (J. E. Hopcroft *et al.*) consiste en un interruptor de la luz, que se puede configurar en **dos** posiciones o **estados** distintos.

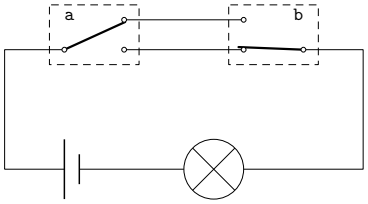




Hemos construido un autómata, tomando como posición inicial el circuito abierto y considerando estado final aceptador el circuito cerrado. Comprueba con JFLAP la evolución del estado de este autómata a lo largo de una sucesión de pulsaciones del interruptor.

1) *Describe formalmente el autómata y su lenguaje aceptado.*

Para poder encender y apagar la luz desde la cama, sustituimos el interruptor por un par de conmutadores, como se muestra en el esquema siguiente.

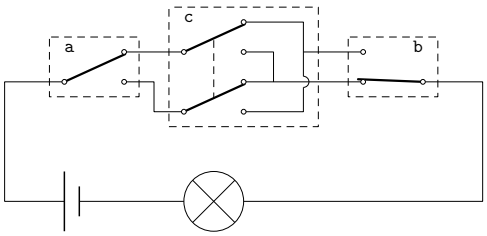


2) *¿De cuántas maneras se pueden combinar las posiciones de los conmutadores?*

3) *Dibuja con JFLAP un autómata asociado, donde los estados se correspondan con las distintas configuraciones (que no con la situación binaria de la bombilla).*

4) *Comprueba con el programa la evolución del estado del autómata a lo largo de una serie de pulsaciones. Deduce cuál es el lenguaje aceptado.*

No contentos con esto, volvemos a cambiar la instalación, incluyendo un tercer punto (en el otro lado de la cama) desde el que encender o apagar. Lo hacemos mediante un conmutador de *cruzamiento*.



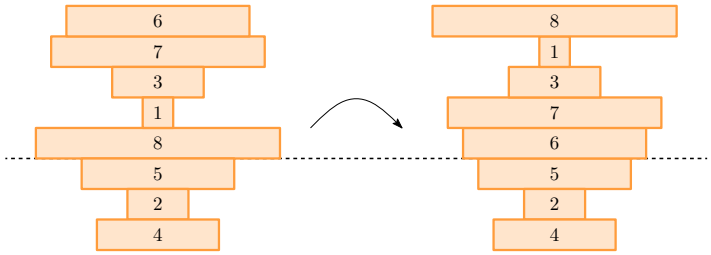
5) *Repite el ejercicio para este circuito.*

6) *Observa el autómata dibujado en el ejemplo 2.4 del libro mencionado arriba (p. 50). También se corresponde con un circuito con un par de conmutadores, ¡aunque mal montados! Dibuja ese circuito.*

7) *Supón que entras en una habitación con una instalación asociada a ese autómata. Desconoces la posición concreta de los dos conmutadores, pero puedes observar si la bombilla está encendida o apagada.*

   Diseña un algoritmo para apagar la luz (suponiendo que está encendida) y otro para encenderla (si está apagada).

Es a los camareros a quienes les toca dar la cara, sirviendo lo que han preparado otros. En 1975, Harry Dweighter[1] propuso un problema relativo a la ordenación de una pila de tortitas. Nunca salía de la cocina una pila con dos tortitas del mismo tamaño. Además, solían estar colocadas de cualquier manera y él quería ordenarlas antes de llevarlas a la mesa. Quería ponerlas por tamaños: cada una encima de otra mayor. Para reorganizarlas en el plato, el único movimiento que podía llevar a cabo era introducir una espátula en la pila (entre dos tortitas cualesquiera o debajo de todas) y voltear la «subpila» superior.



8) *Dibuja con JFLAP un autómata cuyos estados se correspondan con las distintas colocaciones posibles de una pila con* $n = 3$ *tortitas. Cada estado debe admitir tantas transiciones como movimientos pueda efectuar con la espátula el camarero. El autómata debe contar con un único estado final aceptador: ¿cuál? Repite el ejercicio para el caso* $n = 4$.

9) *En los dos casos anteriores (*$n = 3$ *y* $n = 4$*), ¿es posible alcanzar el objetivo a partir de cualquier permutación de las tortitas? ¿Y en el caso general? Implementar el algoritmo en Python, donde la entrada es una lista de enteros y sin utilizar librerías externas.*

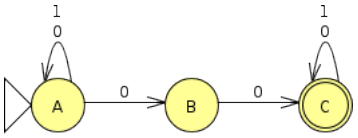
10) *[*Hopcroft et al.*, ex. 2.2.6] Diseña sendos autómatas para estos dos lenguajes sobre* $\Sigma = \{0, 1\}$*:*

- *Representaciones binarias (sin ceros añadidos a la izquierda) de números enteros, positivos y múltiplos de* 5.
- *Palabras que, leídas* **de derecha a izquierda**, *forman la representación binaria (acaso con ceros no significativos —a la derecha— añadidos) de un número natural (incluyendo el cero) y múltiplo de* 5.

[1] Léase *hurried waiter*. La propuesta aparece en el *elementary problem* E 2569, *American Mathematical Monthly* **82**(10).

Autómatas indeterministas

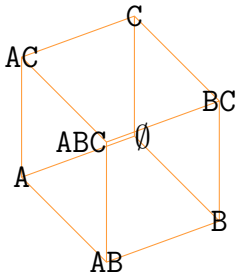
La siguiente ilustración muestra un autómata indeterminista. La novedad con respecto a los ejemplos anteriores es la posibilidad de que, desde algún estado, un mismo estímulo (es decir, un mismo símbolo) conduzca a varios estados distintos.



`aut_01_15.jff`

Podemos interpretar esta situación de la siguiente manera: tras recibir una cadena de estímulos, el autómata podría situarse tal vez en estados diferentes, dependiendo de la decisión adoptada (la bifurcación seguida) en los casos ambiguos. Colocándose si es preciso en varios al mismo tiempo, el autómata indeterminista indica todos los estados a los que puede llegar tras una cadena de símbolos. Una palabra forma parte del lenguaje aceptado si alguno de los estados a los que permite conducir es aceptador.

En el caso del ejemplo anterior, las cadenas que permiten conducir a los distintos estados se pueden describir de la siguiente manera:



A) una palabra cualquiera en $\{0,1\}^*$

B) una terminada en `0`

C) una que incluya la subcadena `00`

11) *Determina el lenguaje aceptado por el autómata.*

Dependiendo de que cada una de las propiedades anteriores se cumpla o no, las palabras se clasifican en ocho categorías. Arriba hemos representado los ocho elementos de $\mathcal{P}\{A,B,C\}$.

12) *¿Cuáles de ellos pueden darse y cuáles son imposibles?*

13) *Utiliza la* subset construction *para encontrar un autómata determinista equivalente. ¿Cuántos estados contiene? Describe el conjunto de palabras que conduce a cada uno de ellos.*

14) *¿Puedes encontrar un autómata determinista que acepte el mismo lenguaje y cuente con solo tres estados? ¿Y con dos?*

15) *Diseña un autómata con el alfabeto de símbolos* $\Sigma = \{0,1\}$ *cuyo lenguaje aceptado sean las palabras terminadas en tres símbolos alternos (*`010` *o* `101`*).*

Si tu solución es indeterminista, utiliza JFLAP para encontrar un autómata determinista equivalente. ¿Resulta más cómodo incluir transiciones indeterministas o restringirse a un autómata determinista?

Autómatas indeterministas con transiciones vacías

16) *[*Hopcroft et al.*, ex. 2.5.3] Diseña autómatas con los siguientes lenguajes aceptados. Aprovecha el recurso del indeterminismo y de las transiciones* λ *para simplificar el diseño.*

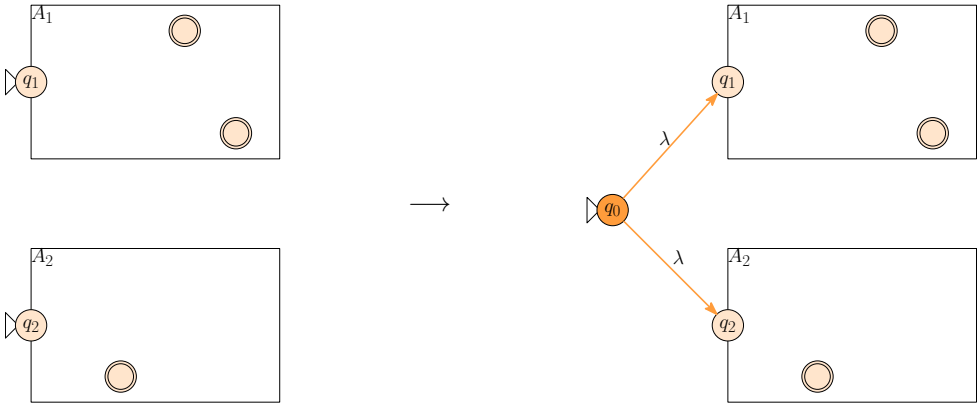
a) Palabras formadas por la concatenación de tres bloques, cada uno de ellos de longitud nula o positiva. El primero incluye únicamente símbolos `a`*; el segundo,* `b`*, y el último,* `c`*.*

b) Palabras compuestas o bien por una o más repeticiones de `01`*, o bien por una o más repeticiones de* `010`*.*

c) Palabras de $\{0,1\}^*$ *entre cuyas diez últimas posiciones hay al menos un* `1`*. Estas palabras no tienen necesariamente longitud mayor o igual que diez.*

Suponemos ahora que contamos con dos autómatas, cada uno de ellos con uno, ninguno o varios

estados aceptadores. Los combinamos en uno nuevo, con un estado inicial añadido que se conecta mediante transiciones vacías con los estados iniciales de los autómatas de partida.



Formalmente, si $A_1 = (Q_1, \Sigma_1, \delta_1, q_1, F_1)$ y $A_2 = (Q_2, \Sigma_2, \delta_2, q_2, F_2)$, con $Q_1 \cap Q_2 = \emptyset$, construimos $A = (Q, \Sigma, \delta, q_0, F)$ del siguiente modo:

- $Q = Q_1 \cup Q_2 \cup \{q_0\}$
- $\Sigma = \Sigma_1 \cup \Sigma_2$
- $\delta|_{Q_1 \times (\Sigma_1 \cup \{\lambda\})} = \delta_1$, $\quad \delta|_{Q_2 \times (\Sigma_2 \cup \{\lambda\})} = \delta_2$, $\quad \delta(q_0, \lambda) = \{q_1, q_2\}$ $\quad$ y $\quad$ δ es constantemente $\emptyset$ sobre $Q_1 \times (\Sigma \backslash \Sigma_1)$, $Q_2 \times (\Sigma \backslash \Sigma_2)$ y $\{q_0\} \times \Sigma$.

  Utilizando la formalización de las funciones de transición como correspondencia (es decir, como subconjunto de $Q \times (\Sigma \cup \{\lambda\}) \times Q$) en vez de como aplicación $Q \times (\Sigma \cup \{\lambda\}) \to \mathcal{P}(Q)$, tendríamos $\delta = \delta_1 \cup \delta_2 \cup \{((q_0, \lambda), q_1), ((q_0, \lambda), q_2)\}$
- $F = F_1 \cup F_2$

17) *¿Cuál es el lenguaje aceptado por ese nuevo autómata?*

Expresiones regulares

Expresión regular ⟶ autómata

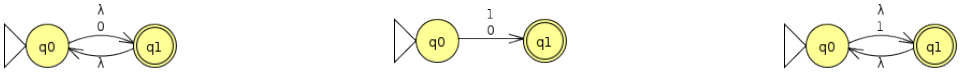
18) *Convierte las dos expresiones regulares siguientes en sendos autómatas, siguiendo el método general. Obtén, para cada una de las respuestas, un autómata determinista equivalente.*

a) $0^*(0+1)1^*$

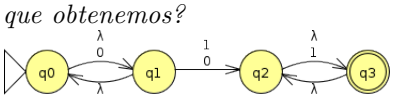
b) $11^* + 00^*1^*$

¿Algunos de los dos lenguajes está contenido en el otro?

19) *Para obtener un autómata cuyo lenguaje aceptado sea el del apartado (a) del ejercicio (18), actuamos del modo siguiente: los tres autómatas que se muestran a continuación aceptan, respectivamente, los lenguajes* 0^*, $0+1$ *y* 1^*.



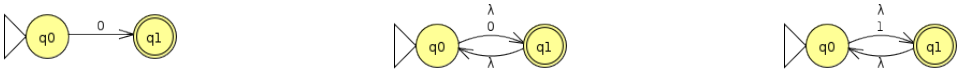
Para «concatenarlos», juntamos el (único) estado final de cada uno con el inicial del siguiente. ¿Es válido el resultado que obtenemos?



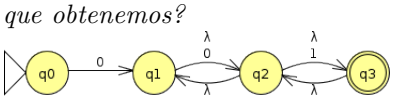
`aut_02_14.jff`

Obtén mediante la subset construction *un autómata determinista equivalente al anterior.*

20) *Para obtener un autómata cuyo lenguaje aceptado sea* 00^*1^*, *actuamos del modo siguiente: los tres autómatas que se muestran a continuación aceptan, respectivamente, los lenguajes* 0^*, 0^* *y* 1^*.



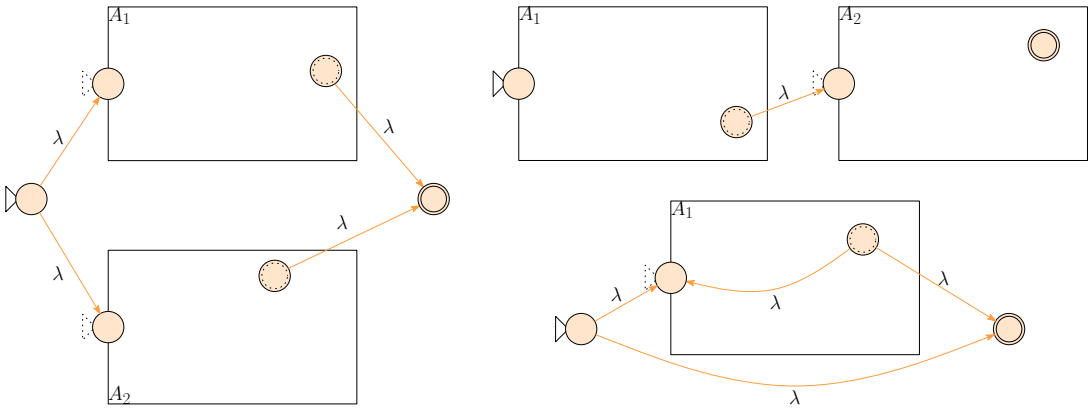
Para «concatenarlos», pegamos el (único) estado final de cada uno con el inicial del siguiente. ¿Es válido el resultado que obtenemos?



`aut_02_15.jff`

Obtén mediante la subset construction *un autómata determinista equivalente al anterior.*

21) *[cf.* Hopcroft et al.*, ex. 3.2.7] Sean* A_1 *y* A_2 *dos autómatas sobre el mismo alfabeto con un único estado final aceptador y* E_1 *y* E_2 *expresiones regulares de sus respectivos lenguajes aceptados. Los procesos de los siguientes esquemas construyen autómatas para los lenguajes* $L(E_1 + E_2)$*,* $L(E_1E_2)$ *y* $L(E_1^*)$*, respectivamente.*

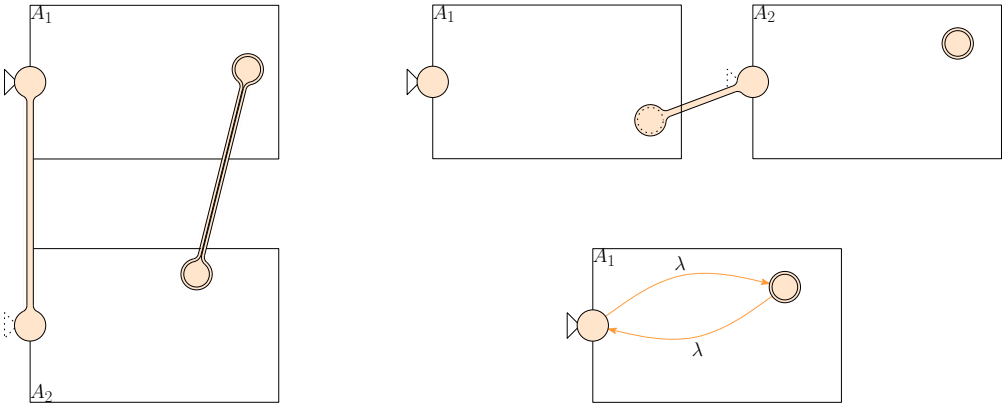


Tratamos de simplificar estas construcciones del modo siguiente:

+) *Reunir en uno solo los estados iniciales de ambos autómatas, por una parte, y los estados finales, por otra*

·) *Fundir el estado final de* A_1 *con el inicial de* A_2

∗) *Ampliar* A_1 *con una transición vacía del estado inicial al final y otra en sentido contrario*



Busca, para cada uno de los tres casos, un contraejemplo que muestre que la simplificación no sirve.

¿Puedes encontrar alguna hipótesis sobre los autómatas de partida que garantice la validez de las tres simplificaciones?

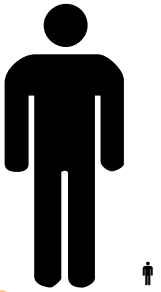
grep

La tarea que desempeña por defecto el programa *grep* consiste en recorrer una a una las líneas de los ficheros de entrada, comprobando si cada una de ellas encaja en cierto *patrón* o modelo e imprimiendo los resultados positivos. Estos patrones agrupan, como las expresiones regulares —de hecho, también se denominan así—, un conjunto (posiblemente infinito) de cadenas de texto.

```
$ wget -O odisea.txt http://www.gutenberg.org/files/58221/58221-0.txt ↲
$ grep -B 1 dedos odisea.txt ↲
$ info grep -n Context ↲
$ grep -E --colour tran?sport odisea.txt ↲
$ wget -O mujercitas.txt http://www.gutenberg.org/cache/epub/514/pg514.txt ↲
$ grep -E --color travell?ed mujercitas.txt ↲
```

Parece que los sabios de la corte del emperador de Lilliput equivocaron el cálculo del cubo de 12: ¿en cuánto?

```
$ wget -O Gulliver.txt http://www.gutenberg.org/files/829/829-0.txt ↲
$ grep -n $((12**3)) Gulliver.txt ↲
$ grep -nE [[:digit:]]{4} Gulliver.txt ↲                    ERE
$ grep -n "[[:digit:]]\{4\}" Gulliver.txt ↲                 BRE
$ for N in $(seq -10 10); do grep $((12**3+$N)) Gulliver.txt; done ↲
$ A=""; for N in $(seq -10 10); do A="$A -e $((12**3+$N))"; done ↲
$ grep $A Gulliver.txt ↲
```



Grep maneja (además de una tercera sintaxis) el estándar POSIX para expresiones regulares, en sus dos variantes: **BRE** (*basic regular expressions*) y **ERE** (*extended regular expressions*).

22) *Escribe, para cada* $n = 0, \ldots, 10$*, un fichero con todas las palabras de longitud* n *sobre el alfabeto* $\{a, b, c\}$ *. Crea otra tanda de ficheros con todas las palabras de* n *símbolos (*$n = 0, \ldots, 20$*) sobre* $\{0, 1\}$*. ¿Cuántas salen en cada una de las dos tandas?*

23) *Encuentra, utilizando grep, todas las palabras del lenguaje asociado a la expresión regular* 0^*11^* *cuya longitud no sea superior a* 20*. ¿Cuántas salen? Si* $n \in \mathbb{N}$*, ¿cuántas palabras de longitud* n *hay en el lenguaje?*

En las expresiones regulares que utiliza grep, no se emplea el carácter '+' para denotar la disyunción (la unión de los lenguajes asociados). En su lugar, contamos (entre otras) con las siguientes opciones:

- El operador '|' combina dos expresiones, formando su unión.

```
$ grep -E "Escila|Caribdis" odisea.txt ↲                    ERE
$ grep -G "Escila\|Caribdis" odisea.txt ↲                   BRE
$ grep -E -A 2 "(larg|small)er end" Gulliver.txt ↲
```

- El «comodín» '.' representa cualquier carácter.

```
$ grep -xE ".{31}" Gulliver.txt ↲                           líneas con 30 caracteres
$ grep -E "^.{31}$" Gulliver.txt ↲
```

- Una lista de caracteres entre corchetes representa uno solo de ellos.

```
$ grep -n roj[oa] odisea.txt ↲
$ grep -nw roj[oa] odisea.txt ↲
$ grep -n "\<roj[oa]\>" odisea.txt ↲
$ grep -iwoE "[^[=a=][=i=]y[=o=][=u=]]+" odisea.txt | ↲      pelebres quen le e
> grep -x "[[:alpha:]]*" | ↲
> grep -iE "([[=e=]].*){3,}" | sort | uniq ↲
$ grep -io "\w*[[=a=]]\w*" 58221-0.txt | ↲                  murciélago
> grep -i "\w*[[=e=]]\w*" | ↲
> grep -i "\w*[[=i=]]\w*" | ↲
> grep -i "\w*[[=o=]]\w*" | ↲
> grep -i "\w*[[=u=]]\w*" | sort | uniq ↲
```

24) *Encuentra, utilizando grep, todas las palabras del lenguaje* $\mathtt{L}(\mathtt{a}^*(\mathtt{b}^* + \mathtt{c}^*))$ *cuya longitud no sea superior a* 10. *¿Cuántas salen? Si* $n \in \mathbb{N}$, *¿cuántas palabras de longitud* n *hay en el lenguaje?*

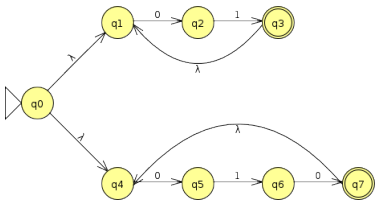
Repite el ejercicio para las expresiones regulares $\mathtt{a}^*\mathtt{cc}^* + \mathtt{b}^*\mathtt{aa}^*$ *y* $\mathtt{aa}^*(\mathtt{b} + \mathtt{c})^*$.

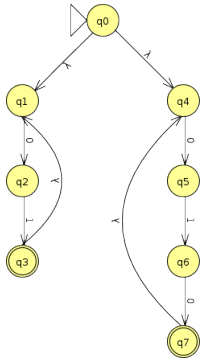
El comando más socorrido del *stream editor* sed es `s` (*substitute*).

```
$ sed 's/ass/\*\*\*/g' Gulliver.txt > excesivo.txt↲
$ diff Gulliver.txt excesivo.txt ↲
$ sed 's/\<ass\>/\*\*\*/g' Gulliver.txt > censurado.txt↲
$ diff Gulliver.txt censurado.txt ↲
```

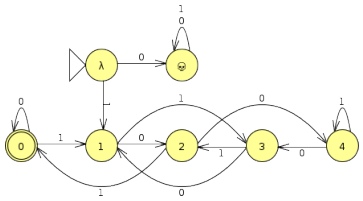
25) *Tenemos varios autómatas en ficheros* `.jff` *y queremos «trasponer» la colocación de sus estados, mediante una simetría con respecto a la diagonal* $x = y$.

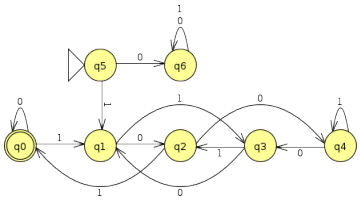
Escribe un script *sed para realizar esta tarea, sin desatender los posibles puntos de control de las transiciones.*





26) *Ahora queremos borrar las etiquetas que les hemos colocado a los nodos, para que JFLAP les asigne la nomenclatura estándar (*`q[i]`*), según su número de orden.*





¿Vagos o voraces?

En las sintaxis habituales para *regexps*, el operador '`*`' es **voraz**, queriendo esto decir que «consume» tantos caracteres como pueda. Así, por ejemplo,

```
$ echo "<from>0</from><to>2</to><from>1</from><to>0</to>" | ↲
> grep -o "<from>.*</from>" ↲
      <from>0</from><to>2</to><from>1</from>
```

Es común también la utilización de variantes *no voraces* o *perezosas* para los agregadores '`?`', '`*`' y '`+`'. Por ejemplo, con la sintaxis de Perl:

```
$ echo "<from>0</from><to>2</to><from>1</from><to>0</to>" | ↲
> grep -oP "<from>.*?</from>" ↲
      <from>0</from>
```

Los operadores no voraces '`??`', '`*?`' y '`+?`' no se pueden emplear con sed.

Gramáticas libres de contexto y autómatas con pila

Gramáticas libres de contexto

27) *Define una gramática libre de contexto con conjunto de terminales* $\mathsf{T} = \{\mathtt{0}, \mathtt{1}\}$ *cuyo lenguaje asociado sea el de las cadenas de longitud impar:*

$$\mathsf{L}(\mathsf{G}) = \{w \in \mathsf{T}^* \ : \ |w| \notin (2)\}.$$

¿Se trata de un lenguaje regular? En caso afirmativo, escribe una expresión regular que lo defina. ¿Puedes localizar con grep las palabras de longitud impar que haya en un fichero?

28) *Consideramos el lenguaje* $\mathcal{D} = \{ww \ : \ w \in \Sigma^*\}$ *de «cuadrados» o «duplicaciones» sobre* $\Sigma = \{\mathtt{0}, \mathtt{1}\}$ *y la gramática libre de contexto* $(\{\mathsf{C}, \mathsf{W}\}, \Sigma, \mathsf{P}, \mathsf{C})$*, donde* P *reúne las producciones siguientes:*

$$\mathsf{C} \ \to \ \mathsf{W}\mathsf{W}; \qquad\qquad \begin{array}{rcl} \mathsf{W} & \to & \lambda \\ & | & \mathsf{W}\mathtt{0} \\ & | & \mathsf{W}\mathtt{1}; \end{array}$$

¿Es $\mathcal{D}$ *es lenguaje asociado a la gramática?*

29) *Diseña una gramática libre de contexto con conjunto de terminales* $\mathsf{T} = \{\mathtt{0}, \mathtt{1}\}$ *y cuyo lenguaje aceptado sea el complementario de los palíndromos:*

$$\mathsf{L} = \{w \in \mathsf{T}^* \ : \ w \neq w^{\mathsf{R}}\}.$$

¿Puedes encontrar una gramática no ambigua que resuelva este ejercicio?

30) *Definimos la gramática libre de contexto* $\mathsf{G} = (\{\mathtt{=}, \mathtt{\neq}, \mathtt{\leftarrow}, \mathtt{\Rightarrow}\}, \{\mathtt{0}, \mathtt{1}\}, \mathsf{P}, \mathtt{\neq})$*, donde* P *está formado por estas ocho producciones:*

$$\begin{array}{rcl} \mathtt{=} & \to & \lambda \\ & | & \mathtt{0=1}; \end{array} \qquad \begin{array}{rcl} \mathtt{\neq} & \to & \mathtt{\leftarrow} \\ & | & \mathtt{\Rightarrow}; \end{array} \qquad \begin{array}{rcl} \mathtt{\leftarrow} & \to & \mathtt{0=} \\ & | & \mathtt{0\leftarrow}; \end{array} \qquad \begin{array}{rcl} \mathtt{\Rightarrow} & \to & \mathtt{=1} \\ & | & \mathtt{\Rightarrow 1}; \end{array}$$

Describe el lenguaje de G*. Repite el ejercicio sustituyendo* P *por el siguiente conjunto:*

$$\begin{array}{rcl} \mathtt{=} & \to & \lambda \\ & | & \mathtt{0=1}; \end{array} \qquad\qquad \begin{array}{rcl} \mathtt{\neq} & \to & \mathtt{0=} \\ & | & \mathtt{=1} \\ & | & \mathtt{0\neq} \\ & | & \mathtt{\neq 1}; \end{array}$$

¿Son regulares los lenguajes que resultan?

Autómatas con pila

31) *Diseña autómatas con pila para los lenguajes que resuelven el ejercicio* **(30)**.

32) *Siempre para el alfabeto* $\Sigma = \{\mathtt{0}, \mathtt{1}\}$*, diseña un autómata con pila cuyo lenguaje aceptado sea el de los palíndromos de longitud impar y otro para el lenguaje de todos los palíndromos (de una paridad u otra):*

$$\mathsf{L}_i = \{w\mathfrak{a}w^{\mathsf{R}} \ : \ w \in \Sigma^*, \ \mathfrak{a} \in \Sigma\}, \qquad \mathsf{L} = \{w \in \Sigma^* \ : \ w = w^{\mathsf{R}}\} = \mathsf{L}_p \,\dot\cup\, \mathsf{L}_i.$$

33) *[cf.* Hopcroft et al.*, ex. 5.1.1.c y ex. 6.2.3.b] Diseña un autómata con pila para cada uno de estos dos lenguajes sobre* $\Sigma = \{0, 1\}$*:*

- *Las palabras de longitud par que no sean «duplicaciones»:*

$$L_1 = \{uv \ : \ u, v \in \Sigma^*, \ |u| = |v|, \ u \neq v\}.$$

- *El complementario del conjunto de las «duplicaciones»:*

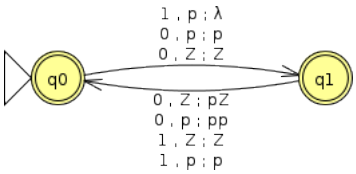
$$L_2 = \Sigma^* \backslash \mathcal{D} = \{w \in \Sigma^* \ : \ \forall u \in \Sigma^*, \ w \neq uu\}.$$

34) *[cf.* Hopcroft et al.*, ex. 5.1.1.d] Diseña un autómata con pila para el siguiente lenguaje sobre* $\Sigma = \{0, 1\}$*:*

$$\{w \in \Sigma^* \ : \ |\{w_i = 0\}| = 2|\{w_i = 1\}|\},$$

es decir, el conjunto de las palabras con el doble de ceros que de unos.

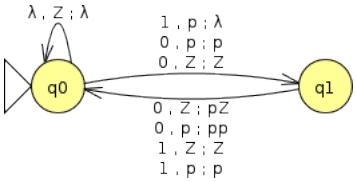
35) *Describe el lenguaje aceptado (atendiendo al criterio de alcance de un estado aceptador) por este autómata con pila:*



`aut_p_03_13.jff`

Busca un autómata más sencillo con el mismo lenguaje aceptado.

36) *Repite el ejercicio* **(35)** *con este autómata, empleando esta vez el criterio de aceptación por agotamiento de pila.*



`aut_p_03_15.jff`

Informática Teórica y Lenguajes Formales

Ejercicios temas 4,5,6

Análisis léxico («lexing»)

El análisis léxico consiste en dividir una cadena de caracteres en objetos llamados «tokens». En el fichero ejercicio1 hay la implementación de un pequeño analizador léxico.

Ejecutando el código, el programa imprime por pantalla:

```
('ID', 'abc')
('SPACE', ' ')
('NUMBER', '123')
('SPACE', ' ')
('ID', 'cde')
('SPACE', ' ')
('NUMBER', '456')
```

1) *Modificar el fichero para que se ignoren los espacios.*

Si se cambia la variable «text» de la siguiente manera

```
text = 'abc 123 ! cde 456'
```

El programa imprime el siguiente error:

```
('ID', 'abc')
('NUMBER', '123')
Traceback (most recent call last):
  File "ex1.py", line 29, in <module>
   for tok in tokenize(text):
  File "ex1.py", line 25, in tokenize
    raise SyntaxError('Bad char \%r' \% text[index])
SyntaxError: Bad char '!'
```

2) *Modificar el fichero para que imprima un mensaje cada vez que encuentra un carácter desconocido.*

El problema de «tokenizar» se resuelve fácilmente utilizando herramientas como SLY. En el fichero analizador hay una base para la implementación de un analizador léxico de expresiones matemáticas. Por ejemplo para la entrada:

```
a = 3 + (4 * 5)
```

la salida debería ser:

```
Token(type='ID', value='a', lineno=1, index=0)
Token(type='ASSIGN', value=' = ', lineno=1, index=2)
Token(type='NUMBER', value='3', lineno=1, index=4)
Token(type='PLUS', value='+', lineno=1, index=6)
Token(type='LPAREN', value='(', lineno=1, index=8)
Token(type='NUMBER', value='4', lineno=1, index=9)
Token(type='TIMES', value='*', lineno=1, index=11)
Token(type='NUMBER', value='5', lineno=1, index=13)
Token(type='RPAREN', value=')', lineno=1, index=14)
```

y la salida de

```
a < b
```

```
a <= b
a > b
a >= b

a != b
```

es

```
Token(type='ID', value='a', lineno=2, index=12)
Token(type='LT', value='<', lineno=2, index=14)
Token(type='ID', value='b', lineno=2, index=16)
Token(type='ID', value='a', lineno=3, index=29)
Token(type='LE', value='<=', lineno=3, index=31)
Token(type='ID', value='b', lineno=3, index=34)
Token(type='ID', value='a', lineno=4, index=47)
Token(type='GT', value='>', lineno=4, index=49)
Token(type='ID', value='b', lineno=4, index=51)
Token(type='ID', value='a', lineno=5, index=64)
Token(type='GE', value='>=', lineno=5, index=66)
Token(type='ID', value='b', lineno=5, index=69)
Token(type='ID', value='a', lineno=6, index=82)
Token(type='ID', value='b', lineno=6, index=87)
Token(type='ID', value='a', lineno=7, index=100)
Token(type='NE', value='!=', lineno=7, index=102)
Token(type='ID', value='b', lineno=7, index=105)
```

3) *Completar la implementación del fichero analizador.*

Análisis sintáctico (Parsing)

Se tiene la siguiente implementación de un Parser:

```
from sly import Lexer, Parser



class EjemploLex(Lexer):
    tokens={UNO, DOS, MAS, POR}
    UNO = '1'
    DOS = '2'
    MAS = '\+'
    POR = '\*'


class EjemploParser(Parser):
    tokens=EjemploLex.tokens
    debugfile="salida.out"
    @_("E OP E")
    def E(self, p):
        pass

    @_("N")
    def E(self, p):
        pass

    @_("N D")
```

```
    def N(self, p):
        pass

    @_("D")
    def N(self, p):
        pass

    @_("UNO")
    def D(self, p):
        pass

    @_("DOS")
    def D(self, p):
        pass

    @_('MAS', 'POR')
    def OP(self, p):
        pass
    def error(self, p):
        pass
```

4) *A partir del archivo salida.out, realice el parsing de la entrada* 1+2. *En la tabla también aparecen varios avisos por conflictos. Explique por qué, y como se han resuelto haciendo el parsing de 1*2 + 2 + 2.*

La gestión de errores en SLY se hace para conseguir detectar todos los errores posibles en una entrada y obtener toda la información posible de la compilación. Para ello, cuando aparce un token inesperado, el parser llama al método de error con el token como argumento. Después, se crea un token llamado *error* y este token entra en juego con la gramática. Ello permite generar reglas de resincronización.

5) *Se pide implementar reglas de resincronización para detectar los dos errores de la entrada 1++2++1 y que se emitan los mensajes de error correspondientes.*

Traducción dirigida por sintaxis con SLY

Mostramos a continuación extractos de los ficheros de entrada que hemos preparado para el generador de parsers con **SLY** . El *parser* resultante no desencadena ninguna acción, salvo reconocer si la cadena de entrada es o no un número capicúa compuesto de ceros y unos. Para ayudarnos en está tarea, ponemos un marcador representado por el número 2 que toca el medio de la palabra.

CapicuaLexer.y

```
from sly import Lexer

class CapicuaLexer(Lexer):
    tokens={UNO,CERO,DOS,FIN}
    UNO="1"
    CERO="0"
    DOS="2"
    FIN="\n"
```

capicuaparser1.py

```
from capicualexer import CapicuaLexer
from sly import Parser

class CapicuaParser(Parser):
    tokens = CapicuaLexer.tokens
    @_("palindromo FIN")
    def palindromo_encajado(self, p):
        pass
    @_("DOS", "CERO palindromo CERO",
       "UNO palindromo UNO")
    def palindromo(self, p):
        pass
Parser = CapicuaParser()
Lexer = CapicuaLexer()
Parser.parse(Lexer.tokenize('020\n'))
```

1) *Haz los cambios oportunos en la especificación de la gramática para que el lenguaje aceptado se reduzca a los palíndromos* **de longitud par** *formados con ceros y unos. ¿Qué sucede en este caso?*

   ¿Qué habría que hacer para detectar cualquier tipo de palíndromos? ¿por qué no se puede substituir el dos por la palabra vacía?.

Mediante una ligera ampliación de la gramática, construimos un programa de manejo más ágil:

capicua2parser.py

```
from capicualexer import CapicuaLexer

from sly import Parser

class CapicuaParser(Parser):
    _cadena = ""
    tokens=CapicuaLexer.tokens
    @_("", "linea retaila")
    def retaila(self, p):
        pass
    @_("palindromo FIN")
    def linea(self, p):
        print("Si")
    @_("CERO palindromo CERO",
       "UNO palindromo UNO","DOS")
    def palindromo(self, p):
        pass
texto = "1112110\n"
Parser = CapicuaParser()
Lexer = CapicuaLexer()
Parser.parse(Lexer.tokenize(texto))
```

```
$ python capicuaparser2.py
11211
Sí
   0112110
Sí
```

2) *Modifica* `capicuaparser1.py` *o* `capicuaparser2.py` *para reconocer las palabras que* **no** *son palíndromos de longitud impar y con un 2 marcando la mitad de la palabra. Modifica el programa para que se imprima la parte derecha, antes del dos.*

Implementando análisis LL mediante recusión

El lenguaje Scheme es un dialecto del lenguaje Lisp que destaca por su diseño minimalista, basado en el cálculo lambda.

Nosotros vamos a considerar un subconjunto, basado en las siguientes reglas que definen a expresiones.

$$
\begin{aligned}
\text{<sexp>} ::= & \ \text{int}^{(1)} \mid \text{bool}^{(2)} \mid \text{float}^{(3)} \\
& \mid \text{id}^{(4)} \\
& \mid (\text{define id <sexp>})^{(5)} \\
& \mid (\text{if <sexp> <sexp> <sexp>})^{(6)} \\
& \mid (\text{lambda (<arg>) <sexp>})^{(7)} \\
& \mid (\text{<call>})^{(8)} \\
\text{<arg>} ::= & \ \text{id <arg>}^{(9)} \mid \epsilon^{(10)} \\
\text{<call>} ::= & \ \text{<sexp> <call>}^{(11)} \mid \text{< sexp >}^{(12)}
\end{aligned}
$$

Éstas son las reglas que definen un programa en Scheme. Todas las reglas están númeradas y permiten definir un algoritmo de parsing descendente basado en tablas. La idea es empezar por la variable inicial y mirar los tokens de la entrada.

Hagamos un ejemplo, si la entrada es (define a 12), una derivación a la izquierda es

$$\text{<sexp>} \mapsto^{(5)} (\text{define}\, \texttt{id}_a\, \text{<sexp>}) \mapsto^{(1)} (\text{define}\, \texttt{id}_a\, \texttt{int}_{12}).$$

	<sexp>	<arg>	<call>
int			
bool			
float			
id			
(define			
(if			
(lambda			
por defecto			

Cuadro 1: Tabla LL

1) *Rellene la tabla de forma que se permita hacer el parsing LL. Utilice la tabla para realizar el parsing de (suma 1 (suma 2 3)).*

La evaluación según las diferentes reglas son:

- Los datos primitivos son enteros (*int*), booleanos (*bool*) y números en punto flotante (*float*) y funciones. Su evaluación es ellos mismo.
- La orden define asocia un tipo primitivo al identificador, su evaluación es guardar en memoria el tipo primitivo asociado al identificador.
- Los identificadores guardan los objetos. La evaluación es el objeto que se ha definido con la orden define.
- La orden if evalua la primera expresión, que debe devolver un booleano. En caso de que el booleano sea verdad, se devolvera la evaluación de la segunda expresión y en otro caso la tercera.
- La función `lambda` crea una función con los argumentos y el cuerpo dado por la expresión.
- En caso de llamada a una función, el primer elemento es la función y el resto los argumentos.

```
fact = ('define', 'fact',
        ('lambda', ('n',), ('if', ('=', 'n', 1),
                            1,
                            ('*', 'n', ('fact', ('-', 'n', 1))))))
def define(label, value):
    global env
    env[label]=value

env = {
    "two": 2,
    "+": lambda x, y: x+y,
    "*": lambda x, y: x*y,
    "-": lambda x, y: x-y,
    "/": lambda x, y: x/y,
    "<": lambda x, y: x<y,
    ">": lambda x, y: x>y,
    "=": lambda x, y: x == y,
}

def substitute(exp, name, value):
    if exp == name:
        return value
    elif isinstance(exp, tuple):
        return tuple(substitute(e, name, value) for e in exp)
    else:
        return exp

def make_procedure(argnames, lexp):
    """Returns a procedure"""
    def call(*values):
        assert len(values) == len(argnames), "Wrong, # args"
        exp = lexp
        # substitute the names
        for name, val in zip(argnames, values):
            exp = substitute(exp, name, val)
        return seval(exp)
    return call

def seval(sexp):
    """Scheme eval"""
    global env
    if isinstance(sexp, (int, bool, float)):
        return sexp
    elif isinstance(sexp, str):
        return env[sexp]
    elif isinstance(sexp, tuple):
        if sexp[0] == "define":
            ##################
        elif sexp[0] == "if":
            ##################
        elif sexp[0] == "lambda":
            ##################

        else:
            evaluated_exp = [seval(e) for e in sexp]
            return evaluated_exp[0](*evaluated_exp[1:])


assert seval(42) == 42
```

```
assert seval("two") == 2
assert seval(('+', 2, 3)) == 5
assert seval(('+', ("*", 33, 44), 3)) is not None
seval(('define', 'n', 5))
assert seval('n') == 5
assert seval(("if", (">", 1, 2), 10, 12)) == 12
assert seval(("lambda", ("x", "y"), ("+", "x", "y"))) is not None, "Lambda isn't working"
seval(fact)
assert seval(('fact', 'n')) == 120
```

2) *Complete el código anterior, de forma que evalue correctamente el código python.*