Expresiones regulares

- *Prof. Maureen Murillo*
- *Teoría de la Computación*
- *Escuela de Ciencias de la Computación e Informática*
- *Universidad de Costa Rica*

¿Qué es una expresión regular?

- Es una representación de cadenas de caracteres que se ajustan a determinado patrón.

- Por ejemplo, la siguiente expresión regular reconoce todas las hileras formadas por las sílabas “ma”, “me”, “mi”, “mo”, “mu”:

  (m[aeiou])+

  Reconoce: mima, mu, memima, momomomo

- Sirven para aplicaciones de búsqueda de texto o componentes de compilador.

¿Cómo se construyen?

- Concatenando los símbolos aceptados en el lenguaje que se desea representar.
- Hay caracteres especiales que se pueden usar en las regex.
- Por lo que a veces hay que usar caracteres “escapados”.

Ejemplo de expresión regular que reconoce la suma de dos números enteros tal como “34+567”:

[0-9]+\+[0-9]+

Ejemplos

Pruebe en: regex101.com

- Escriba una expresión regular que reconozca cualquier número binario

  (0|1)+

- Cualquier número binario par

  (0|1)*0

- Combinación de letras de la “a” a la “z”

  [a-z]+

- Oraciones, que inician con mayúscula y pueden tener números y espacios

  [A-Z][a-z0-9]*

- Direcciones IP

  … pensemos …

Operadores y algunos comandos útiles

OPERADORES BÁSICOS

- Ver el resumen de operadores en sitio del curso

OTROS COMANDOS

- \w (word character): [A-Za-z0-9_]
- \d (digit): [0-9]
- \s (whitespace character): [\t\r\n\f]
- \W, \D, \S: Negación de los anteriores [^\w], [^\d], [^\s]

Práctica de expresiones regulares

- Realice individualmente la práctica de expresiones regulares colgada en el entorno del curso

Volviendo a la teoría…
Operaciones sobre expresiones regulares

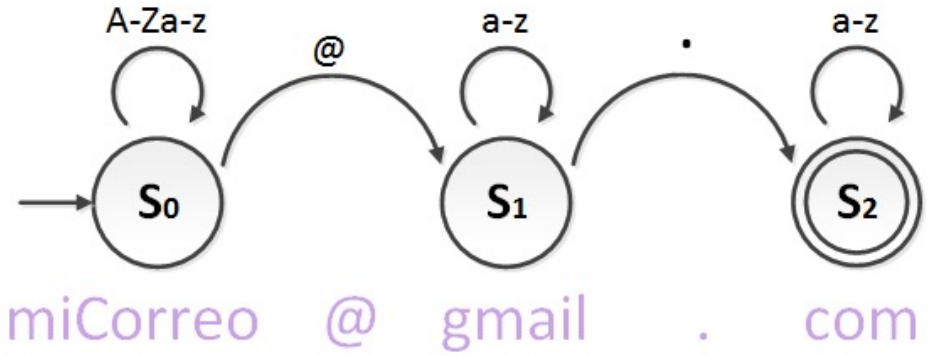
Las regex denotan lenguajes. Por ejemplo, la regex 01* define el lenguaje de todas las cadenas que empiezan con 0 seguido de cualquier cantidad de 1's. Las regex tienen 3 operaciones:

- La *unión* de dos lenguajes *L* y *M*, designada como $L \cup M$. Si $L = \{001,10,111\}$ y $M = \{\varepsilon,001\}$, entonces $L \cup M = \{\varepsilon,10,001,111\}$.
- La *concatenación* de los lenguajes *L* y *M*, designada como L.M o LM. Si $L = \{001, 10, 111\}$ y $M = \{\varepsilon,001\}$, entonces *L.M*, o simplemente *LM*, es $\{001,10,111,001001,10001,111001\}$.
- La *clausura de Kleene* de un lenguaje *L* se designa mediante $L*$ y representa el conjunto de cadenas que se pueden formar tomando cualquier número de cadenas de L, posiblemente con repeticiones y concantenándolas. Si $L = \{0,1\}$, entonces $L*$ es igual a todas las cadenas de 0s y 1s. Si $L = \{0,11\}$, entonces $L*$ constará de aquellas cadenas de 0s y 1s tales que los 1s aparezcan por parejas, como por ejemplo 011, 11110 y ε, pero no 01011 ni 101. Más formalmente, $L*$ es la unión infinita $\cup_{i\geq 0} L^i$, donde $L^0 = \{\varepsilon\}$, $L^1 = L$ y L^i, para $i>1$ es LL···L (la concatenación de *i* copias de L).

Tomado de Hopcroft, Motwani y Ullman

¿Qué tiene que ver una expresión regular (regex) con un autómata finito no determinista (NFA)?

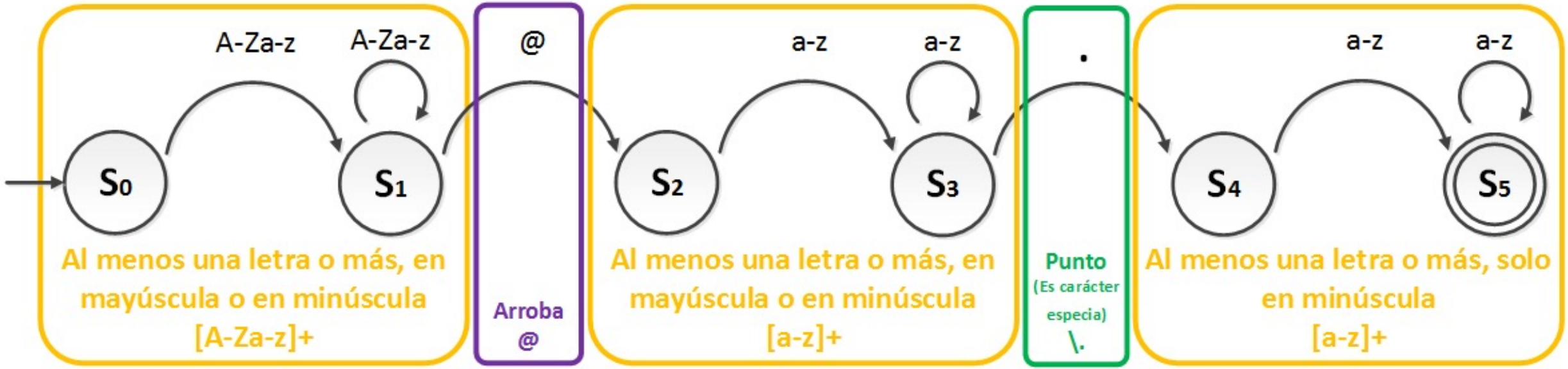
Están estrechamente relacionados. Las regex son una alternativa a los NFA para la definición de lenguajes regulares.



¿Qué problema tiene este autómata?

Tomado de: https://jarroba.com/busqueda-de-patrones-expresiones-regulares/

¿Qué tiene que ver una expresión regular (regex) con un autómata finito no determinista (NFA)? (continuación)



[A-Za-z]+@[a-z]+\.[a-z]+

Tomado de: https://jarroba.com/busqueda-de-patrones-expresiones-regulares/

Ejemplo de vida real útil

- Construya un formulario en Google Forms para solicitar a los estudiantes de la UCR su nombre, número de carnet y dirección de correo institucional.

Expresiones regulares usadas para análisis léxico

El **análisis léxico** es el análisis de un texto de entrada para identificar diferentes agrupaciones de símbolos, conocidos como ***tokens***.

Por ejemplo, un compilador realiza análisis léxico para identificar palabras reservadas (IF, WHILE, …), operadores (==, &&, …), identificadores, etc. Los tokens identificados serán utilizados luego para el análisis sintáctico (estructura de los tokens).

Herramientas de análisis léxico y sintáctico

- Lex: programa para generar analizadores léxicos. Es el estándar en los sistemas UNIX y devuelve un analizador en código C.
- Yacc: programa para generar análisis sintáctico que se usa generalmente con lex. Genera código C.

- Flex: es una versión de software libre de Lex. Se usa comúnmente con GNU Bison.
- Bison: es un programa generador de analizadores sintácticos compatible con Yacc. Genera código C, C++ o Java.

- PLY: es una versión de Lex y Yacc escrita en Python.

Analizador léxico de PLY

1) Importar biblioteca:

```
import ply.lex as lex
```

2) Definir lista de tokens que servirán para el analizador sintáctico:

```
tokens = (
    'NUMBER',
    'PLUS',
    'MINUS',
    'TIMES',
    'DIVIDE',
    'LPAREN',
    'RPAREN',
)
```

4) Crear objeto lexer y asociarlo a entrada de datos:

Sustituir por entrada desde archivo

```
# Build the lexer
lexer = lex.lex()

# Test it out
data = '''
3 + 4 * 10
  + -20 *2
'''

# Give the lexer some input
lexer.input(data)
```

5) Obtener un token:

```
tok = lexer.token()
```

3) Definir expresiones regulares para los tokens:

```
# Regular expression rules for simple tokens
t_PLUS    = r'\+'
t_MINUS   = r'-'
t_TIMES   = r'\*'
t_DIVIDE  = r'/'
t_LPAREN  = r'\('
t_RPAREN  = r'\)'

# A regular expression rule with some action code
def t_NUMBER(t):
    r'\d+'
    t.value = int(t.value)
    return t

 # Define a rule so we can track line numbers
def t_newline(t):
    r'\n+'
    t.lexer.lineno += len(t.value)

# A string containing ignored characters (spaces and tabs)
t_ignore  = ' \t'

# Error handling rule
def t_error(t):
    print("Illegal character '%s'" % t.value[0])
    t.lexer.skip(1)
```

Algunas no interesan para el analizador sintáctico

Objetos de PLY

```
lexer = lex.lex()
lexer.input("3 + 4")
```

`tok = lexer.token()` → Retorna el próximo token. Es una instancia de *LexToken* con atributos: type, value, lineno, lexpos

`tok.type` → Devuelve **NUMBER**

`tok.value` → Devuelve **r'\d+'**, en este caso un **3**

`tok.lineno` → Devuelve el número de línea del token, en este caso un **1**

`tok.lexpos` → Devuelve la posición del token en la línea, en este caso un **0**

Atención… esto le puede ahorrar algunas horas

El lexer de PLY agrega las reglas en el siguiente orden:

1. Se agregan todos los tokens definidos por funciones en el orden en el que aparecen en el archivo.
2. Los tokens definidos por hileras se agregan luego en orden decreciente según el largo de la expresión regular, o sea, las expresiones más largas se agregan primero.

Por ejemplo, si se quiere reconocer los tokens “=“ y “==“, hay que asegurarse que se revisa primero “==“. Aquí no habría problema:

```
t_ASSIGN =    r’=‘
t_EQUAL  =    r’==’
```