Expresiones regulares en STklos

Pedro González

Mayo 2005

1. Introducción

Una **expresión regular** es una cadena de caracteres que contiene un patrón de ajuste.

Dada una cadena, queremos comprobar si ésta se adapta a aquella. Los verbos **ajustar**, **encajar**, **aparear**, **adaptar** y **casar** se utilizarán indistintamente y en este contexto significan lo mismo.

Como es evidente, el patrón debe seguir unas reglas. En este escrito se exponen las más utilizadas con ejemplos aclaratorios, para que Vd. adquiera un nivel de conocimiento medio. Una vez que las haya comprendido, puede profundizar leyendo los manuales correspondientes.

2. Reglas

2.1. Crear una expresión regular

Con la siguiente orden:

```
(string->regexp cadena)
```

donde `cadena` es la cadena que contiene el patrón. Para simplificar, es mejor asignarla, por ejemplo:

```
(define er (string->regexp cadena))
```

2.2. Usar una expresión regular

Para comprobar si una cadena se ajusta a la expresión regular, debemos utilizar la orden:

```
(regexp-match er cadena)
```

donde `er` es la expresión regular creada anteriormente y `cadena` es la que queremos comprobar que se adapte a `er`.

La orden anterior retorna o bien **lista**, lo cual indica que hubo ajuste o el valor booleano `#f`, para indicar que no.

2.3. Caracteres especiales

Hay un conjunto de caracteres especiales, llamados **metacaracteres** que modifican el comportamiento de la expresión regular. Son los siguientes:

```
. ^ $ * + ? { [ ] \ | ( )
```

Más adelante explicaremos su significado.

Si una expresión regular no contiene metacaracteres, entonces la cadena se adapta a la expresión regular cuando ésta es parte de aquella. Por ejemplo:

```
stklos> (define er (string->regexp "jugar con"))
;; er
stklos> (define es (string->regexp "juega con"))
;; es
stklos> (regexp-match er "el niño juega con la pelota")
#f
stklos> (regexp-match es "el niño juega con la pelota")
("juega con")
stklos> (regexp-match es "el niño Juega con la pelota")
#f
```

3. Metacaracteres

3.1. Corchetes([ ])

Se utilizan para crear un conjunto de caracteres, el cual puede especificarse enumerando sus elementos o bien mediante un rango, en cuyo caso debemos detallarlo escribiendo el primer carácter, un guión `-` y a continuación el último. Por ejemplo:

- `[abcdef]`, conjunto formado por las seis primeras letras.
- `[0-9]`, conjunto formado por todos los dígitos decimales.
- Pueden combinarse ambos tipos. Por ejemplo, `[0-9abcdef]`, o más sencillamente `[0-9a-f]`.

Un conjunto anterior dentro de una expresión regular significa cualquier carácter que pertenezca a dicho conjunto.

Dentro de un conjunto, los únicos metacaracteres válidos son:

\	Escape
^	Negación
-	Rango

El carácter Escape (\) anula cualquier interpretación que tuviera un carácter de especial, por lo tanto, ante la duda, debe utilizarse. Veamos algunos ejemplos:

- `[^abc]`, casa con cualquier carácter que no sea `a`, `b` o `c`.

- `[^0-9]`, casa con cualquier carácter que no sea un dígito.
- `[\^abc]`, casa con `a`, `b`, `c` o `^`.

En el último caso:

```
stklos> (define p (string->regexp "[\\^abc]"))
;; p
stklos> (regexp-match p "c")
("c")
stklos> (regexp-match p "d")
#f
stklos> (regexp-match p "^")
("^")
stklos> (regexp-match p "z")
#f
stklos> (regexp-match p "zdef")
#f
stklos> (regexp-match p "zdefc")
("c")
```

Hay algunos conjuntos que por ser muy utilizados, están predefinidos. Son los siguientes:

`\d` Cualquier dígito decimal, equivalente por tanto a `[0-9]`

`\D` Cualquier carácter que no sea un dígito decimal, equivalente por tanto a `[^0-9]`

`\s` Cualquier blanco, entendiendo por tal uno de los siguientes:

- □, espacio
- \t, tabulador.
- \n, nueva línea.
- \r, retorno.

`\S` Cualquier no blanco.

`\w` Cualquier carácter constituyente de una palabra, es decir, letras, dígitos o el subrayado (_).

`\W` negación del anterior.

Por ejemplo:

```
stklos> (define q (string->regexp "\\w"))
;; q
stklos> (regexp-match q "12a33")
("1")
stklos> (regexp-match q ".:,;-_")
("_")
stklos> (regexp-match q ".:,;-")
#f
```

3.2. Significado de los demás metacaracteres

Fuera de los corchetes, el significado de los metacaracteres, en forma resumida, es:

Caracteres	Significado
`\\x`	Concuerda con el carácter `x`.
`^`	Casa con la cadena vacía al comienzo de la cadena de entrada.
`$`	Casa con la cadena vacía al final de la cadena de entrada.
`.`	Concuerda con cualquier carácter simple excepto el nueva línea.
`(`	Comienzo de *sub-expresión regular*.
`)`	Fin de *sub-expresión regular*.
`*`	Cuantificador 0 o más.
`+`	Cuantificador 1 o más.
`?`	Cuantificador 0 o 1.
`{`	Comienzo de cuantificador general de repetición.
`}`	Fin de cuantificador general de repetición.
`|`	Alternativa. Por ejmplo, `regexp1 | regexp2` ajusta con `regexp1` o `regexp2`

Para comprender la tabla anterior, comencemos con un problema que iremos complicando progresivamente. Vamos a crear una expresión regular para reconocer un número entero sin signo. Es decir, entradas como las siguientes:

25, 3573, 82921342

son válidas, mientras que, estas otras:

25a, pq35b73, 8292cd1342

no lo son. Inicialmente, empezaríamos así:

```
stklos> (define ne (string->regexp "[0-9]+"))
;; ne
```

lo que significa, reconocer una sucesión de dígitos con al menos 1 (metacarácter +). Veamos:

```
stklos> (regexp-match ne "25")
("25")
stklos> (regexp-match ne "3573")
("3573")
stklos> (regexp-match ne "82921342")
("82921342")
```

¡Magnífico!. Ahora bien:

```
stklos> (regexp-match ne "25a")
("25")
stklos> (regexp-match ne "pq35b73")
("35")
```

¡Horror, esto no es lo que esperábamos!, pero debíamos haberlo imaginado, pues la función `regexp-match` ha detectado una sucesión de dígitos tanto en el primer como en el segundo caso. Sin embargo:

```
stklos> (regexp-match ne "pqrs")
#f
```

En este caso, no se ha encontrado dicha sucesión. Para evitar entradas como "25a" o "pq35b73" debemos cambiar nuestra definición original. Así pues,redefinimos:

```
(define ne (string->regexp "^[0-9]+$"))
```

cuyo significado es idéntico al anterior, aunque los metacaracteres ^ y $ indican que la entrada desde el principio (^) hasta el final ($) solamente deben estar constituidas de dígitos. Veamos:

```
stklos> (regexp-match ne "25a")
#f
stklos> (regexp-match ne "pq35b73")
#f
stklos> (regexp-match ne "pqrs")
#f
stklos> (regexp-match ne "25")
("25")
stklos> (regexp-match ne "3573")
("3573")
stklos> (regexp-match ne "82921342")
("82921342")
```

¡Esto va mucho mejor!. Veamos un pequeño problema:

```
stklos> (regexp-match ne "032")
("032")
stklos> (regexp-match ne "000025")
("000025")
```

Esto no está bien. Nadie escribe números enteros con ceros de cabecera, aunque también es verdad que ningún lenguaje de programación protestaría por ello. **El problema que planteamos ahora es cambiar la definición de `ne` para que no admita ceros de cabecera**, es decir, entradas como las siguientes:

25, 3573, 82921342

son válidas, mientras que, estas otras:

025, 003573, 000082921342

no.

En fin, redefinimos:

```
(define ne (string->regexp "^[1-9]+[0-9]*$"))
```

lo cual significa que la cadena debe comenzar por al menos un dígito entre 1 y 9 y debe ir seguida por 0 o más dígitos (metacaracter `*`) entre 0 y 9. En efecto:

```
stklos> (regexp-match ne "25")
("25")
stklos> (regexp-match ne "025")
#f
stklos> (regexp-match ne "3573")
("3573")
stklos> (regexp-match ne "003573")
#f
stklos> (regexp-match ne "82921342")
("82921342")
stklos> (regexp-match ne "000082921342")
#f
```

3.3. Operadores de repetición

El operador general de repetición adopta la forma

$$\{m, n\}$$

donde m y n son enteros ≥ 0 y $m \leq n$. El entero m representa el mínimo número de caracteres y n el máximo. Por ejemplo, la expresión regular

```
[a-z]{1,5}
```

casa con cualquier cadena formada por letras minúsculas entre 1 y 5 caracteres de longitud. Las siguientes variantes son válidas:

- `{m,}` Al menos m repeticiones.
- `{,n}` Como máximo n repeticiones.
- `{n}` Exactamente n repeticiones.

Una vez comprendido esto debe resultar evidente que:

- `*` es equivalente a `{0,}`
- `+` es equivalente a `{1,}`
- `?` es equivalente a `{0,1}`

Así pues, la expresión regular anterior podía haberse escrito como

```
(define ne (string->regexp "^[1-9]{1}[0-9]*$"))
```

Sabemos que un número entero puede ir precedido por un signo. Si es `+`, lo habitual es omitirlo, pero si es `-`, es obligatorio ponerlo. Así pues, **nuestra nueva expresión regular debe contemplar el signo**, es decir, entradas como las siguientes:

```
+22, -1528, 22, 1528
```

son correctas. En fin, redefinimos:

```
stklos> (define ne (string->regexp "^(\\+|\\-)?[1-9]{1}[0-9]*$"))
;; ne
```

La parte `(\\+|\\-)?` significa 0 o 1 ocurrencia de la sub-expresión regular `\\+|\\-`, que al ser una alternativa, significa introducir el signo `+` o el `-`. Juntándolo todo, la expresión `(\\+|\\-)?` indica finalmente introducir o no los signos `+` o `-`. El resto es ya conocido de antes. Veamos algunos ejemplos:

```
stklos> (regexp-match ne "+22")
("+22" "+")
stklos> (regexp-match ne "22")
("22" #f)
stklos> (regexp-match ne "-1528")
("-1528" "-")
stklos> (regexp-match ne "1528")
("1528" #f)
stklos> (regexp-match ne "*1528")
#f
```

Para acabar con nuestro ejemplo de números, ampliemos la expresión a números decimales, con el punto (`.`) como separador de la parte entera de la parte decimal. Es decir, la nueva expresión regular tendrá de nombre `nd` (por número decimal), de forma tal que entradas como las siguientes:

```
+22, -13, 15.28, -15.28, +334.22
```

son correctas. La definición es:

```
stklos> (define nd (string->regexp "^(\\+|\\-)?[1-9]{1}[0-9]*(\\.[0-9]+)?$"))
;; nd
```

No damos explicaciones, pues, con lo que sabe, ya debe Vd. comprender lo anterior. Veamos algunos ejemplos:

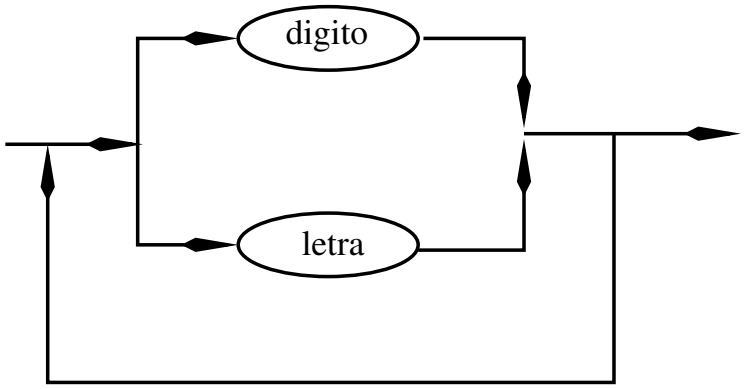
```
stklos> (regexp-match nd "-+22")
#f
stklos> (regexp-match nd "+22")
("+22" "+" #f)
stklos> (regexp-match nd "-13")
("-13" "-" #f)
stklos> (regexp-match nd "15.28")
("15.28" #f ".28")
stklos> (regexp-match nd "-15.28")
("-15.28" "-" ".28")
stklos> (regexp-match nd "334.22")
("334.22" #f ".22")
stklos> (regexp-match nd "334..22")
#f
```

4. Otro ejemplo

4.1. Grupos de un profesor

Para un programa de gestión que tenemos en mente, **queremos crear un analizador sintáctico para los grupos de un mismo curso de un profesor**. Para entendernos, comencemos con el concepto de **curso**.

Un curso es un conjunto de caracteres alfanuméricos, es decir, sólo de letras y dígitos. No vamos a limitar el número de caracteres de que puede constar un curso. El grafo sintáctico es:



Por ejemplo, las siguientes cadenas representan entradas correctas:

`1eso, Cou, tercerodeBUP, 2bup, 1Bachillerato`

mientras que las siguientes no lo son

`tercero-BUP` ;; ¡el guión no debe estar ahí!
`1°Bachillerato` ;; ¡el carácter ° es incorrecto!

En fin, el problema es ya evidente, **crear una expresión regular que reconozca si la cadena de entrada es un curso o no, según las reglas especificadas**. Nuestra primera definición es:

```
stklos> (define curso (string->regexp "^[a-zA-Z0-9]+$"))
;; curso
```

y veamos algunas comprobaciones:

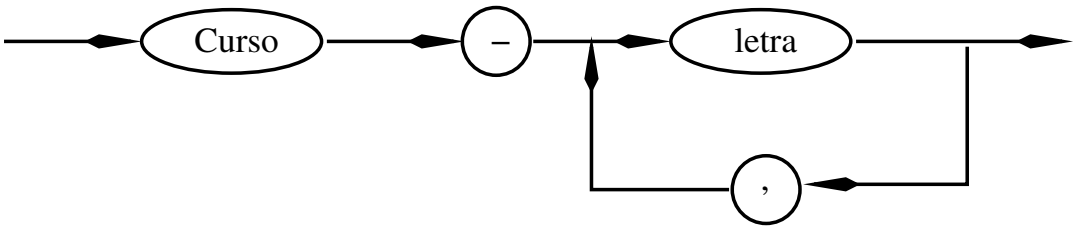
```
stklos> (regexp-match curso "1eso")
("1eso")
stklos> (regexp-match curso "Cou")
("Cou")
stklos> (regexp-match curso "tercerodeBUP")
("tercerodeBUP")
stklos> (regexp-match curso "tercero de BUP")
#f
stklos> (regexp-match curso "2bup")
("2bup")
```

```
stklos> (regexp-match curso "1Bac")
("1Bac")
stklos> (regexp-match curso "tercero-Bup")
#f
```

Parece que la cosa va bien. Podíamos haber simplificado la expresión regular introduciendo la opción `(?i)`, cuyo significado es **no distinguir entre mayúsculas y minúsculas**, con lo cual, modificamos y probamos:

```
stklos> (define curso (string->regexp "^(?i)[a-z0-9]+$"))
;; curso
stklos> (regexp-match curso "1eso")
("1eso")
stklos> (regexp-match curso "1EsO")
("1EsO")
stklos> (regexp-match curso "1bachillerato")
("1bachillerato")
stklos> (regexp-match curso "1BaChIlleRaTo")
("1BaChIlleRaTo")
stklos> (regexp-match curso "tercero-Bup")
#f
```

Un grupo es un conjunto de alumnos de un determinado curso. Es tradicional distinguir los grupos con una letra, por ejemplo, **a**, **b**, **c**, etc. Separamos el curso de las letras de los grupos con el guión (`-`) y las letras con la coma (`,`). El diagrama sintáctico para **los grupos de un determinado curso de un profesor** es:



Por ejemplo, la siguiente entrada

```
1bac-a,b,d
```

es sintácticamente correcta y significa que el profesor da clase en los grupos **a**, **b** y **d** del curso **1bac**. Análogamente,

```
Cou-c,e,f,a
```

significa que el profesor da clase en los grupos **a**, **c**, **e** y **f** del curso **Cou**.

Pues bien, lo que queremos ahora es **crear una expresión regular que reconozca si la cadena de entrada se ajusta al diagrama sintáctico anterior**.

Definimos:

```
stklos> (define grupos (string->regexp "^(?i)[a-z0-9]+\\-[a-z](\\,[a-z])*$"))
;; grupos
```

Comprobamos:

```
stklos> (regexp-match grupos "4eso-a,b,c")
("4eso-a,b,c" ",c")
stklos> (regexp-match grupos "4eSo-A,b,C")
("4eSo-A,b,C" ",C")
stklos> (regexp-match grupos "4eSo-A,B,C")
("4eSo-A,B,C" ",C")
stklos> (regexp-match grupos "4eso-aa,b,c")
#f
stklos> (regexp-match grupos "2bac-b,c,d")
("2bac-b,c,d" ",d")
stklos> (regexp-match grupos "2bac-b,c,d,d")
("2bac-b,c,d,d" ",d")
stklos> (regexp-match grupos "1bac-c")
("1bac-c" #f)
stklos> (regexp-match grupos "1bac<c")
#f
stklos> (regexp-match grupos "1bac_c")
#f
```

Observe bien que la entrada

```
2bac-b,c,d,d
```

es sintácticamente correcta, aunque el grupo cuya letra es `d` está repetido. Este tipo de acontecimientos deben ser controlados en el programa, y lo que parece más sensato en este caso es convertir, sin intervención del usuario la entrada anterior a la siguiente:

```
2bac-b,c,d
```

eliminando la repetición.

Finalmente, si los cursos del centro no van a tener más de 26 grupos (uno por cada letra), parece más interesante simplificar la sintaxis eliminado las comas (`,`) de forma tal que la entrada

```
1bac-abd
```

significa ahora lo mismo que la antigua `1bac-a,b,d`. Análogamente, la antigua

```
Cou-c,e,f,a
```

quedaría convertida a `Cou-cefa`. Para ello, la nueva definición es:

```
stklos> (define grupos-simplificados (string->regexp "^(?i)[a-z0-9]+\\-[a-z]+$"))
;; grupos-simplificados
```

Comprobamos:

```
stklos> (regexp-match grupos-simplificados "1bac-abd")
("1bac-abd")
stklos> (regexp-match grupos-simplificados "Cou-c,e,f,a")
#f
stklos> (regexp-match grupos-simplificados "Cou-cefa")
("Cou-cefa")
stklos> (regexp-match grupos-simplificados "1bac_cde")
#f
```

Para acabar, imagine que el usuario está acostumbrado al sistema antiguo con comas (`,`) y ahora va Vd. y le propone el nuevo. La gente es muy reacia a cambiar algo que le va bien, razón por la cual, lo que le propongo ahora es que **modifique la expresión regular de forma tal que acepte tanto el sistema antiguo como el nuevo**.

5. Nota final

Este tutorial ha utilizado la versión

```
stklos> (version)
"0.61"
```

de STklos.