

Programación y Administración de Sistemas

Práctica 2. Expresiones regulares para programación de la *shell*.

Víctor Manuel Vargas

Asignatura "Programación y Administración de Sistemas"
2º Curso Grado en Ingeniería Informática
Escuela Politécnica Superior
(Universidad de Córdoba)
vvargas@uco.es

14 de marzo de 2022



Objetivos del aprendizaje I

- Definir qué es una expresión regular.
- Justificar la necesidad de las expresiones regulares y su importancia en la programación de *scripts* para administración de sistemas.
- Distinguir entre expresiones regulares básicas y expresiones regulares extendidas.
- Entender el significado de los distintos caracteres especiales que se pueden utilizar para expresiones regulares.
- Ser capaz de interpretar una expresión regular.
- Ser capaz de escribir expresiones regulares dada una especificación.
- Utilizar correctamente expresiones regulares para el comando `grep`.
- Utilizar correctamente expresiones regulares para el comando `sed`.

2.1. Expresiones regulares.

- 2.1.1. Concepto.

- 2.1.2. Justificación.

- 2.1.3. Caracteres especiales.

2.2. Comandos.

- 2.2.1. `grep` y `egrep`.

- 2.2.2. `sed`.

- Pruebas de validación de prácticas.

¿Qué son las expresiones regulares?

- Una expresión regular (**regex**) describe un **conjunto de cadenas de texto**.
- Se utilizan:
 - En entornos UNIX, con comandos como **grep, sed, awk...**
 - De manera intensiva, en lenguajes de programación como **perl, python, ruby...**
 - En bases de datos.
- **Ahorran mucho tiempo** y hacen el código más **robusto**.



¿Qué son las expresiones regulares?

- La expresión regular más simple sería la que **busca una secuencia fija de caracteres literales.**
- La cadena **cumple la expresión regular si contiene esa secuencia.**

o	l	a
---	---	---

Ella me dijo hola. ⇒ Empareja.

Ella me dijo mola. ⇒ Empareja.

Ella me dijo adiós. ⇒ No empareja.



¿Qué son las expresiones regulares?

- Puede que la expresión regular empareje a la cadena en más de un punto:

o	l	a
---	---	---

Lola me dijo hola. \Rightarrow Empareja 2 veces.

- El carácter "." empareja cualquier cosa:

o	l	a	.
---	---	---	---

Lola me dijo hola. \Rightarrow Empareja 2 veces.



¿Por qué las necesito?

- ¿Para qué necesito aprender a utilizar las *regex*?
- Historia real¹:
 - Direcciones de calles.
 - Quiero actualizar su formato, de "100 NORTH MAIN ROAD" a "100 NORTH MAIN RD.", sobre un conjunto de muchas carreteras.

```
1 victor@victor-ayrna:~$ echo "100 NORTH MAIN ROAD" | sed -e 's/ROAD/RD\./'
2 100 NORTH MAIN RD.
3 victor@victor-ayrna:~$ cat carreteras.txt
4 100 NORTH MAIN ROAD
5 45 ST JAMES ROAD
6 100 NORTH BROAD ROAD
7 victor@victor-ayrna:~$ cat carreteras.txt | sed -e 's/ROAD/RD\./'
8 100 NORTH MAIN RD.
9 45 ST JAMES RD.
10 100 NORTH BRD. ROAD
```

¹https://linux.die.net/diveintopython/html/regular_expressions/street_addresses.html



¿Por qué las necesito?

- ¿Para qué necesito aprender a utilizar las *regex*?
 - A veces necesito **hacer operaciones con cadenas con expresiones relativamente complejas**.
 - P.Ej.: reemplazar “ROAD” por “RD.” siempre que esté al final de la línea (carácter especial \$).

```
1 victor@victor-ayrna:~$ cat carreteras.txt | sed -e 's/ROAD$/RD\./'  
2 100 NORTH MAIN RD.  
3 45 ST JAMES RD.  
4 100 NORTH BROAD RD.
```



Caracteres especiales

- Las expresiones regulares se componen de **caracteres normales** (literales) y de **caracteres especiales** (o metacaracteres).
- “**[...]**”: sirve para indicar una **lista caracteres posibles**:

b	[iur]	e
---	-------	---

Octubre me dijo bueno bien. ⇒ **Empareja 3 veces.**

- “**[^...]**”: sirve **para negar** la ocurrencia de uno o más caracteres:

b	[^ur]	e
---	-------	---

Octubre me dijo bueno bien. ⇒ **Empareja 1 vez.**



Caracteres especiales

- “^”: empareja con el principio de una línea:

^	0
---	---

0ctubre me dijo bueno \Rightarrow Empareja 1 vez.

- “\$”: empareja con el final de una línea:

e	\$
---	----

Bueno, me dijo octubree \Rightarrow Empareja 1 vez.



Caracteres especiales

- “*”: empareja con cero, una o más ocurrencias del carácter anterior:

o	l	a	*	s
---	---	---	---	---

Holaaaaaas ⇒ Empareja 1 vez.

Hols ⇒ Empareja 1 vez.

- En caso de duda, el emparejamiento siempre es el de mayor longitud:

a	.	*	e
---	---	---	---

01as emocionantes.



Caracteres especiales

- Los **paréntesis** `()` (o `\(\\)`) **permiten agrupar caracteres** a la hora de aplicar los metacaracteres:
 - `a*` empareja `a`, `aa`, `aaa...`
 - `abc*` empareja `ab`, `abc`, `abcc`, `abccc...`
 - `(abc)*` empareja `abc`, `abcabc`, `abcabcabc...`
- Dos tipos de expresiones regulares:
 - *Basic Regular Expressions* (BRE): propuesta inicial en el estándar POSIX.
 - *Extended Regular Expressions* (ERE): ampliación con nuevos metacaracteres.
- Cada aplicación utiliza una u otra.



Caracteres especiales

Carácter	BRE	ERE	Significado
\	✓	✓	Interpreta de forma literal el siguiente carácter
.	✓	✓	Selecciona un carácter cualquiera
*	✓	✓	Selecciona ninguna, una o varias veces lo anterior
(✓	✓	Principio de línea
\$	✓	✓	Final de línea
[...]	✓	✓	Cualquiera de los caracteres que hay entre corchetes
\n	✓	✓	Utilizar la n-ésima selección almacenada
{n,m}	X	✓	Selecciona lo anterior entre n y m veces
+	X	✓	Selecciona una o varias veces lo anterior
?	X	✓	Selecciona una o ninguna vez lo anterior
	X	✓	Selecciona lo anterior o lo posterior
(...)	X	✓	Selecciona la secuencia que hay entre paréntesis ²
\{n,m\}	✓	X	Selecciona lo anterior entre n y m veces
\(...\)	✓	X	Selecciona la secuencia que hay entre paréntesis ²
	✓	X	Selecciona lo anterior o lo posterior

²Se almacena la selección



Rangos de caracteres

- `[aeiou]`: empareja con las letras `a`, `e`, `i`, `o` y `u`.
- `[1-9]` es equivalente a `[123456789]`.
- `[a-e]` es equivalente a `[abcde]`.
- `[1-9a-e]` es equivalente a `[123456789abcde]`.
- Los rangos típicos se pueden especificar de la siguiente forma:
 - `[[[:alpha:]]]` → `[a-zA-Z]`.
 - `[[[:alnum:]]]` → `[a-zA-Z0-9]`.
 - `[[[:lower:]]]` → `[a-z]`.
 - `[[[:upper:]]]` → `[A-Z]`.
 - `[R[:lower:]]` → `[Ra-z]`.
 - Otros³: *digit*, *punct*, *cntrl*, *blank*...

³man wctype



Comando grep

- **grep** proviene del editor **ed** (editor de texto Unix), y en concreto, de su comando de búsqueda de expresiones regulares “**global regular expression print**”.
- Se utiliza cuando sabes que **un fichero contiene una determinada expresión** y quieres saber qué fichero es.
- **grep** utiliza las BRE, **egrep** utiliza las ERE (no obstante, podemos usar **grep -E** para que considere ERE).
- **Consejo**: antes de incluirlas en el *script*, probar las expresiones regulares en la consola con **grep**.
- Si usamos **grep --color** se resaltan **en rojo** los emparejamientos:
 - Si no lo tenéis activo, es buena idea incluir un alias en **.bashrc: alias grep='grep --color'**.



Comando grep

- Como muchos de los caracteres especiales de las *regex* son también especiales en *bash*, es una buena costumbre rodear la *regex* con comillas simples ('*expr*') cuando estemos escribiendo un *script* → Siempre que la *regex* no contenga variables de *bash*.
- **-i**: hace que considere igual mayúsculas y minúsculas.
- **-o**: en lugar de imprimir las líneas completas que cumplen el patrón, solo muestra el emparejamiento del patrón.
- **-v**: mostrar las líneas que no cumplen el patrón.



Comando grep

```
1 victor@victor-ayrna:~$ cat ejemplo.txt
2 Este es otro ejemplo de expresiones regulares
3 La segunda parte ya la veremos
4 ,,,adios,hola
5 victor@victor-ayrna:~$ cat ejemplo.txt | grep '^E'
6 Este es otro ejemplo de expresiones regulares
7 victor@victor-ayrna:~$ cat ejemplo.txt | grep -E '^(E|L)'
8 Este es otro ejemplo de expresiones regulares
9 La segunda parte ya la veremos
10 victor@victor-ayrna:~$ cat ejemplo.txt | grep -E ',* '
11 Este es otro ejemplo de expresiones regulares
12 La segunda parte ya la veremos
13 ,,,adios,hola
14 victor@victor-ayrna:~$ cat ejemplo.txt | grep -E ',+'
15 ,,,adios,hola
16 victor@victor-ayrna:~$ cat ejemplo.txt | grep -E ',+' -o
17 ,,,
18 ,
19 victor@victor-ayrna:~$ cat ejemplo.txt | grep -E 'L(..)*\1'
20 La segunda parte ya la veremos
```



Comando grep

- Encontrar todos los números con signo (con posibilidad o no de decimales):

`[-+] [0-9]+(\. [0-9]+)?`

```
1 victor@victor-ayrna:~$ grep -E '[-+] [0-9]+(\. [0-9]+)?' $(find -name "*.c")
2 ./svorex/loadfile.c:
3 ./gpor/lgam1.c:          strcat (buf, pstr+4) ;
                          -0.0002109075,0.0742379071,0.0815782188,
```

- 5 cifras decimales o más (sin signo):

`[0-9]+\.[0-9]{5,}`

```
1 victor@victor-ayrna:~$ grep -E '[0-9]+\.[0-9]{5,}' $(find -name "*.c")
2 ./gpor/lgam1.c:          -0.0002109075,0.0742379071,0.0815782188,
```



Comando `sed`

- Es parecido a `grep` pero **permite cambiar las líneas que encuentra** (en lugar de solo mostrarlas).
- En realidad, es un editor de textos no interactivo, que recibe sus comandos como si fuesen un *script*.
- Los comandos que utiliza son los mismos que los de `ed`.
- Solo vamos a estudiar algunos de los comandos posibles.
- Por defecto, todas las líneas se imprimen tras aplicar el comando.



Comando sed

- `sed [-r] [-n] -e 'comando' [archivo]:`
 - `-r`: uso de EREs en lugar de BREs.
 - `-n`: modo silencioso → para imprimir una línea tienes que indicarlo explícitamente mediante el comando `p` (*print*).
 - `-e 'comando'`: ejecutar el comando o comandos especificados.
 - Sintaxis de comandos:
`[direccionInicio[, direccionFin]][!]comando`
`[argumentos]:`
 - Si la dirección es adecuada, entonces se ejecutan los comandos (con sus argumentos).
 - Las direcciones pueden ser expresiones regulares (`/regex/`) o números de línea (`1`).
 - Si no hay `direccionFin` solo se aplica sobre `direccionInicio`.
 - `!` emparejaría todas las direcciones distintas que la indicada.



Comando sed

- **d**: borrar líneas direccionadas.
- **p**: imprimir líneas direccionadas.
- **s**: sustituir una expresión por otra sobre las líneas seleccionadas. Sintaxis:

s/patron/reemplazo/[banderas]

- **patron**: expresión regular BRE.
- **reemplazo**: cadena con qué reemplazarla.
- **Bandera n**, siendo *n* un número entero: reemplazar sólo la ocurrencia *n*-ésima.
- **Bandera g**: reemplazar todas las ocurrencias.
- **Bandera p**: forzar a imprimir la línea (solo tiene sentido si hemos utilizado **-n**).



Comando sed

```
1 victor@victor-ayrna:~$ cat ejemplo.txt
2 Este es otro ejemplo de expresiones regulares
3 La segunda parte ya la veremos
4 ,,,adios,hola
5 victor@victor-ayrna:~$ cat ejemplo.txt | sed -e '3p'
6 Este es otro ejemplo de expresiones regulares
7 La segunda parte ya la veremos
8 ,,,adios,hola
9 ,,,adios,hola
10 victor@victor-ayrna:~$ cat ejemplo.txt | sed -n -e '3p'
11 ,,,adios,hola
12 victor@victor-ayrna:~$ cat ejemplo.txt | sed -n -e '1,2p'
13 Este es otro ejemplo de expresiones regulares
14 La segunda parte ya la veremos
15 victor@victor-ayrna:~$ cat ejemplo.txt | sed -n -e '1,2!p'
16 ,,,adios,hola
17 victor@victor-ayrna:~$ cat ejemplo.txt | sed -e '/~L/d'
18 Este es otro ejemplo de expresiones regulares
19 ,,,adios,hola
20 victor@victor-ayrna:~$ cat ejemplo.txt | sed -e '2,$d'
21 Este es otro ejemplo de expresiones regulares
22 victor@victor-ayrna:~$ cat ejemplo.txt | sed -e '1,/s$/d'
23 ,,,adios,hola
```



Comando sed

```
1 victor@victor-ayrna:~$ cat ejemplo.txt
2 Este es otro ejemplo de expresiones regulares
3 La segunda parte ya la veremos
4 ,,,adidos,hola
5 victor@victor-ayrna:~$ cat ejemplo.txt | sed -r -e 's/La/El/'
6 Este es otro ejemplo de expresiones regulares
7 El segunda parte ya la veremos
8 ,,,adidos,hola
9 victor@victor-ayrna:~$ cat ejemplo.txt | sed -r -e 's/[Ll]a/El/'
10 Este es otro ejemplo de expresiones reguElres
11 El segunda parte ya la veremos
12 ,,,adidos,hoEl
13 victor@victor-ayrna:~$ cat ejemplo.txt | sed -r -e 's/([Ll])a/era\1/'
14 Este es otro ejemplo de expresiones regueralres
15 eraL segunda parte ya la veremos
16 ,,,adidos,hoeraL
17 victor@victor-ayrna:~$ cat ejemplo.txt | sed -r -n -e 's/(d[ea])/"\1"/p'
18 Este es otro ejemplo "de" expresiones regulares
19 La segun"da" parte ya la veremos
20 victor@victor-ayrna:~$ cat ejemplo2.txt
21 Grado:Informatica
22 Informatica2:Grado2
23 victor@victor-ayrna:~$ cat ejemplo2.txt | sed -r -n -e 's/(.*):(.*)/\2:\1/p'
24 Informatica:Grado
25 Grado2:Informatica2
```



Comando sed

- Ejercicio: Utilizar expresiones regulares con sed, para transformar la salida del comando df al formato indicado abajo.

```
1 victor@victor-ayrna:~$ ./espacioLibre.sh
2 El fichero de bloques udev, montado en /dev, tiene usados 0 bloques de un total
  de 8072372 (porcentaje de 0%).
3 El fichero de bloques tmpfs, montado en /run, tiene usados 1684 bloques de un
  total de 1627732 (porcentaje de 1%).
4 El fichero de bloques /dev/nvme0n1p6, montado en /, tiene usados 26204344
  bloques de un total de 60213124 (porcentaje de 46%).
5 El fichero de bloques tmpfs, montado en /dev/shm, tiene usados 200400 bloques de
  un total de 8138640 (porcentaje de 3%).
6 El fichero de bloques tmpfs, montado en /run/lock, tiene usados 4 bloques de un
  total de 5120 (porcentaje de 1%).
7 El fichero de bloques tmpfs, montado en /sys/fs/cgroup, tiene usados 0 bloques
  de un total de 8138640 (porcentaje de 0%).
8 El fichero de bloques /dev/nvme0n1p5, montado en /home, tiene usados 110324328
  bloques de un total de 328804660 (porcentaje de 36%).
9 El fichero de bloques /dev/nvme0n1p1, montado en /boot/efi, tiene usados 32952
  bloques de un total de 262144 (porcentaje de 13%).
10 El fichero de bloques tmpfs, montado en /run/user/1000, tiene usados 92 bloques
    de un total de 1627728 (porcentaje de 1%).
```



Inciso: problemas con espacios en blanco y arrays

- Cuando intentamos **construir un array a partir de una cadena**, **bash utiliza determinados caracteres para separar** cada uno de **los elementos del array**.
- Estos caracteres están en la variable de entorno IFS y por defecto son el **espacio, el tabulador y el salto de línea**.

```
1 victor@victor-ayrna:~$ array=($(echo "1 2 3"))
2 victor@victor-ayrna:~$ echo ${array[0]}
3 1
4 victor@victor-ayrna:~$ echo ${array[1]}
5 2
6 victor@victor-ayrna:~$ echo ${array[2]}
7 3
8 victor@victor-ayrna:~$ array=($(echo -e "1\t2\n3"))
9 victor@victor-ayrna:~$ echo ${array[0]}
10 1
11 victor@victor-ayrna:~$ echo ${array[1]}
12 2
13 victor@victor-ayrna:~$ echo ${array[2]}
14 3
```



Inciso: problemas con espacios en blanco y arrays

- Esto **nos puede producir problemas** si estamos procesando elementos con espacios (por ejemplo, nombres de ficheros con espacios):




```
1 victor@victor-ayrna:~$ array=($(echo -e "El uno\nEl dos\nEl tres"))
2 victor@victor-ayrna:~$ echo ${array[0]}
3 El
4 victor@victor-ayrna:~$ echo ${array[1]}
5 uno
```

- **Solución:** cambiar el IFS para que solo se utilice el `\n`:

```
1 victor@victor-ayrna:~$ OLDIFS=$IFS
2 victor@victor-ayrna:~$ IFS=$'\n'
3 victor@victor-ayrna:~$ array=($(echo -e "El uno\nEl dos\nEl tres"))
4 victor@victor-ayrna:~$ echo ${array[0]}
5 El uno
6 victor@victor-ayrna:~$ echo ${array[1]}
7 El dos
8 victor@victor-ayrna:~$ IFS=$OLDIFS
```



Referencias

-  [Stephen G. Kochan y Patrick Wood](#)
Unix shell programming.
Sams Publishing. Tercera Edición. 2003.
-  [Evi Nemeth, Garth Snyder, Trent R. Hein y Ben Whaley](#)
Unix and Linux system administration handbook.
Capítulo 2. *Scripting and the shell*.
Prentice Hall. Cuarta edición. 2010.
-  [Aleen Frisch](#).
Essential system administration.
Apéndice. *Administrative Shell Programming*.
O'Reilly and Associates. Tercera edición. 2002.



Programación y Administración de Sistemas

Práctica 2. Expresiones regulares para programación de la *shell*.

Víctor Manuel Vargas

Asignatura "Programación y Administración de Sistemas"
2º Curso Grado en Ingeniería Informática
Escuela Politécnica Superior
(Universidad de Córdoba)
vvargas@uco.es

14 de marzo de 2022

