

Shell scripting

Sistemas Operativos

Enrique Soriano, Gorka Guardiola

GSYC

17 de diciembre de 2025



Universidad
Rey Juan Carlos

Este trabajo se entrega bajo la licencia “Atribución-CompartirIgual 4.0 Internacional”[1] (cc-by-sa).

Usted es libre de:

- ▶ **Compartir:** *Copiar y redistribuir el material en cualquier medio o formato.*
- ▶ **Adaptar:** *Remezclar, transformar y construir a partir del material para cualquier propósito, incluso comercialmente.*
- ▶ **Se pueden dispensar estas restricciones si se obtiene el permiso de los autores.**
- ▶ *Las imágenes de terceros mantienen sus derechos originales.*

©2022 Gorka Guardiola y Enrique Soriano.

[1] Algunos derechos reservados. “Atribución-CompartirIgual 4.0 Internacional” (cc-by-sa). Para obtener la licencia completa, véase <https://creativecommons.org/licenses/by-sa/4.0/deed.es>. También puede solicitarse a Creative Commons, 559 Nathan Abbott Way, Stanford, California 94305, USA.

¿Cuándo hago un script de Shell?

► Pasos para realizar una tarea:

1. Mirar si hay alguna herramienta que haga lo que queremos → buscar en el manual.
2. Si no encontramos, intentar combinar distintas herramientas → **programar un script de Shell**. La primera aproximación debe ser pipelines de filtros, etc.
IDEA: combinar herramientas que hacen bien una única tarea para llevar a cabo tareas más complejas.
3. Si no podemos, hacer una herramienta → programada en C, Python, Java, Ada, Go, ...

¿Qué tipo de cosas ?

- ▶ La shell es especialmente buena
 - ▶ Para realizar tareas con ficheros
 - ▶ Para procesar texto
 - ▶ Para automatizar tareas (con un IDE, Makefile)

Automatizar

HOW LONG CAN YOU WORK ON MAKING A ROUTINE TASK MORE
EFFICIENT BEFORE YOU'RE SPENDING MORE TIME THAN YOU SAVE?
(ACROSS FIVE YEARS)

		HOW OFTEN YOU DO THE TASK					
		50/DAY	5/DAY	DAILY	WEEKLY	MONTHLY	YEARLY
HOW MUCH TIME YOU SHAVE OFF	1 SECOND	1 DAY	2 HOURS	30 MINUTES	4 MINUTES	1 MINUTE	5 SECONDS
	5 SECONDS	5 DAYS	12 HOURS	2 HOURS	21 MINUTES	5 MINUTES	25 SECONDS
	30 SECONDS	4 WEEKS	3 DAYS	12 HOURS	2 HOURS	30 MINUTES	2 MINUTES
	1 MINUTE	8 WEEKS	6 DAYS	1 DAY	4 HOURS	1 HOUR	5 MINUTES
	5 MINUTES	9 MONTHS	4 WEEKS	6 DAYS	21 HOURS	5 HOURS	25 MINUTES
	30 MINUTES		6 MONTHS	5 WEEKS	5 DAYS	1 DAY	2 HOURS
	1 HOUR		10 MONTHS	2 MONTHS	10 DAYS	2 DAYS	5 HOURS
	6 HOURS				2 MONTHS	2 WEEKS	1 DAY
	1 DAY					8 WEEKS	5 DAYS

Shells

- ▶ sh es la shell original de Unix, escrita por Ken Thompson. Fue rescrita por Stephen Bourne en 1979 para Unix Version 7: *bourne shell*.
- ▶ Los sistemas derivados usan distintas shells: sh, ash, bash, dash, ksh, csh, tcsh, zsh, rc, etc.
- ▶ Cada una tiene sus características, pero también tienen mucho en común.
- ▶ En sistemas modernos, /bin/sh suele ser un enlace simbólico a su shell por omisión para ejecutar scripts. En Ubuntu y Debian es dash¹.
- ▶ Política: los scripts que tienen #!/bin/sh deben usar únicamente las características POSIX (IEEE Std 1003.1-2017): el subconjunto común que tienen la mayoría de las shells. Así, los scripts pueden ser portables entre distintos sistemas.

¹ No confundir con el shell por omisión para un terminal, que es bash.

Un script:

- ▶ Tiene que tener permisos de ejecución.
- ▶ Hay comandos que se implementan dentro del shell (no se ejecuta un fichero externo al shell, es una parte del propio shell). Se llaman *built-in*. Si se quiere ver si es builtin *type* o *whatis*.
- ▶ El comando *built-in* *exit* sale del script con el status indicado en su argumento.
- ▶ Si un script no sale con *exit*, deja el status que tiene \$?.

```
#!/bin/sh

# este es un hola mundo en sh
# esto es un comentario

echo hello world
exit 0
```

Un script:

- ▶ Una ventaja de la shell, es que puedo probar de forma interactiva
- ▶ No escribo el script directamente, voy probando los comandos
- ▶ O ni siquiera escribo el script (escribo los comandos directamente)

Ya conocemos:

- ▶ | es un pipe
- ▶ & ejecuta un comando en background
- ▶ \$ se usa para las variables. \$var es lo mismo que \${var}
- ▶ " y ' se usan para escapar cadenas (las dobles expanden algunas cosas)
- ▶ > , < y >> son redirecciones
 - ▶ 5> para redir del fd 5
 - ▶ 5>&3 para hacer dup del 5 al 3
 - ▶ Cuando hay varios el orden importa ej: 2>&1 >/dev/null
- ▶ \ se usa para escapar caracteres
- ▶ && , || para ejecución condicional
- ▶ Globbing (wildcards): ? * [a – z] etc.

Parámetros posicionales

- ▶ Se pueden acceder a los parámetros que se han pasado al script con \$1, \$2, \$3 ...
- ▶ \$0 expande al nombre con el que se ha invocado el script.
- ▶ \$# expande al número de parámetros (sin contar el 0).
- ▶ \$* expande a los parámetros posicionales.
- ▶ "\$*" expande a "\$1 \$2 ..."
- ▶ \$@ expande a los parámetros posicionales (igual que \$* pero separados)
- ▶ "\$@" expande a "\$1" "\$2" ...
- ▶ shift desplaza los parámetros (p. ej. \$4 pasará a ser \$3). Se actualiza el valor de #.
 - ▶ Útil para parámetros optativos (pongo lo que sea, o hago shift, el resto igual)

Agrupaciones

- ▶ Si queremos ejecutar comandos en un subshell:

(comando; comando; ...)

- ▶ Si queremos ejecutar una agrupación de comandos en el shell actual:

{ comando; comando; ... }

Ejemplo:

```
$> { echo uno; echo dos; } | tr o O
unO
dOs
$> { echo los ficheros de /tmp son; ls /tmp; } > ficheros
```

Agrupaciones

- ▶ Ejecutar en un subshell útil
- ▶ Para no cambiar el entorno en el shell actual (cd, export)

Ejemplo:

```
#sigo en tmp al final:  
$> pwd; (cd /etc; ls apt;); pwd;  
/tmp  
apt.conf.d      sources.list  
preferences.d   sources.list.d  trusted.gpg    trusted.gpg.d  
/tmp  
#BLA no existe al final:  
$> echo z$BLA; (export BLA=bla; echo $BLA;); echo z$BLA;  
z  
bla  
z  
$>
```

Sustitución de comando

- ▶ Se sustituye el comando por su salida.
- ▶ Se puede escribir de dos formas:

`$(comando)`

`‘comando’`

```
$> l=$(wc -l /tmp/a | cut -d' ' -f1)
$> echo $l
31
$>
```

if

Las condiciones depende del status de salida del comando: éxito es verdadero, fallo es falso.

```
if comando
then
    comandos
elif comando
then
    comandos
else
    comandos
fi
```

case

Los casos pueden contener patrones de globbing.

```
case palabra  in
patrón1)
    comandos
    ;;
patrón2 | patrón3)
    comandos
    ;;
*)  # este es el default
    comandos
    ;;
esac
```

Bucles

```
while comando
```

```
do
```

```
    comandos
```

```
done
```

```
for variable in palabra1 palabra2 palabraN
```

```
do
```

```
    comandos
```

```
done
```

Read

- ▶ El comando `read` lee una línea de su entrada estándar y la guarda en la variable que se le pasa como argumento.
- ▶ Se puede usar para procesar la entrada línea a línea en un bucle.
- ▶ Solo debemos hacer eso cuando no tenemos ningún filtro o pipeline que nos sirva para hacer lo que queremos.

Read

- ▶ Por ejemplo, esto itera 2 veces:

```
echo 'a b  
c d' > /tmp/e

while read line
do
    echo $line
done < /tmp/e
```

- ▶ Esto itera 4 veces:

```
for x in `cat /tmp/e`
do
    echo $x
done
```

Variable IFS

- ▶ Esta variable contiene los caracteres que se usan como separadores entre campos.
- ▶ Por omisión contiene el tabulador, espacio y el salto de línea.
- ▶ Hay que tener cuidado: cambiar el valor de esta variable rompe las cosas.
- ▶ Mejor en subshell.

```
$> for i in $(echo uno dos tres) ; do echo $i ; done
uno
dos
tres
$> export IFS=-
$> for i in $(echo uno dos tres) ; do echo $i ; done
uno dos tres
$
```

Funciones

- ▶ Se pueden definir funciones. Sus parámetros se acceden como los parámetros posicionales. P. ej.:

```
hello () {  
    echo hola $1  
    shift  
    echo adios $1  
}
```

...

```
# ejecutamos la función  
hello uno dos
```

El comando test sirve para comprobar condiciones de distinto tipo.

Ficheros:

- ▶ -f fichero
si existe el fichero
- ▶ -d dir
si existe el directorio

Cadenas:

- ▶ -n String1
si la longitud de la string no es cero
- ▶ -z String1
si la longitud de la string es cero
- ▶ String1 = String2
si son iguales
- ▶ String1 != String2
String1 and String2 variables no son idénticas
- ▶ String1
si la string no es nula

Test

Enteros:

- ▶ `Integer1 -eq Integer2`
si los enteros `Integer1` e `Integer2` son iguales.

Otros operadores:

- ▶ `-ne`: not equal
- ▶ `-gt`: greater than
- ▶ `-ge`: greater or equal
- ▶ `-lt`: less than
- ▶ `-le`: less or equal

Test

Test también se puede usar así:

- ▶ Esto:

```
[ $a -eq $b ]
```

- ▶ es lo mismo que esto:

```
test $a -eq $b
```

Operaciones aritméticas

Para operaciones básicas con enteros podemos usar el propio shell.
También podemos usar el comando bc.

- ▶ Esto:

```
$((5 + 7))
```

- ▶ se reemplaza por

12

Filtros útiles

- ▶ sort
ordena las líneas de varias formas.
- ▶ uniq
elimina líneas contiguas repetidas.
- ▶ tail
muestra las últimas líneas.

P. ej:

```
$> ps | tail +3 # a partir de la 3a
$> ps | tail -3 # las 3 ultimas
$> seq 1 1000 | sort
$> seq 1 1000 | sort -n
```

Sort

- ▶ Puede recibir una lista de columnas (empezando por la 1)
- ▶ Y un separador
- ▶ Y ordena por esos campos como clave (es un intervalo de campos)
- ▶ Ojo con estabilidad (-s): una ordenación es estable si no te cambia de orden, en el resultado, entradas que son iguales según el criterio de ordenación.

```
$> cat x.txt
1-2-4
2-3-3
2-2-1
2-1-4
$> sort -k2 -t\| - x.txt
2-1-4
2-2-1
1-2-4
2-3-3
$> sort -k1,2 -t\| - x.txt
1-2-4
2-1-4
2-2-1
2-3-3
```

Comandos útiles

- ▶ **diff**
compara ficheros de texto línea a línea
- ▶ **cmp**
compara ficheros binarios byte a byte

P. ej:

```
$> diff -n fich1 fich2
```

```
$> cmp fich1 fich2
```

- ▶ Traduce caracteres. El primer argumento es el conjunto de caracteres a traducir. El segundo es el conjunto al que se traducen. El enésimo carácter del primer conjunto se traduce por el enésimo carácter del segundo.
- ▶ -d
Borra los caracteres del único conjunto que se le pasa como argumento.
- ▶ Se le pueden dar rangos, p. ej.
\$> cat fichero | tr a-z A-Z

Expresiones regulares (*regexp*)

- ▶ Es un lenguaje formal para describir/buscar cadenas de caracteres.
- ▶ Parecidas a los patrones de la Shell o de globbing, pero más potentes.
- ▶ Veremos las que se llaman *extended regular expressions*. Es un estándar de POSIX.
- ▶ Una string encaja con sí misma, por ejemplo 'a' con 'a'.

Expresiones regulares (*regexp*)

- ▶ .
encaja con cualquier carácter, por ejemplo 'a'.
- ▶ [conjunto]
encaja con cualquier carácter en el conjunto, por ejemplo
[abc] encaja con 'a'. Se pueden especificar rangos, p. ej.
[a-zA-Z].
- ▶ [^conjunto]
encaja con cualquier carácter que **no esté** en el conjunto, por
ejemplo [^abc] NO encaja con 'a', sin embargo sí encaja con
'z'.

Expresiones regulares (*regexp*)

- ▶ \wedge
encaja con *principio de línea*.
- ▶ $\$$
encaja con *final de línea*.
- ▶ Una regexp e_1 concatenada a otra regexp e_2 , e_1e_2 , encaja con una string si una parte p_1 de la string encaja con e_1 y otra parte contigua, p_2 , encaja con e_2 .

P. ej:

'az' encaja con la regexp [a-d]z

Expresiones regulares (*regexp*)

- ▶ **exp***

encaja si aparece **cero o más veces** la regexp que lo precede.

- ▶ **exp⁺**

encaja si aparece **una o más veces** la regexp que lo precede.

P. ej:

'aaa' encaja con la regexp a*

'baaa' encaja con la regexp ba⁺

'bb' encaja con la regexp ba*

'bb' no encaja con la regexp ba⁺

Expresiones regulares (*regexp*)

- ▶ **exp?**
encaja si aparece **cero o una vez** la regexp que lo precede. Se utiliza para partes opcionales.
- ▶ **(exp)**
agrupa expresiones regulares.

P. ej:

'az', 'av', 'a' encajan con la regexp az?

'abab' encaja con la regexp (ab)+

'abab', 'ababab', 'ababababa' encajan con la regexp (ab)+

Expresiones regulares (*regexp*)

- ▶ $\text{exp} \mid \text{exp}$
si encaja con alguna de las regexp que están separadas por la barras
- ▶ \ carácter de escape: hace que el símbolo pierda su significado especial.

P. ej:

'aass' encaja con la regexp (aass|booo)
'hola*' encaja con la regexp a*

Egrep

- ▶ Filtra líneas usando expresiones regulares.
- ▶ -v
Realiza lo inverso: imprime las líneas que no encajan.
- ▶ -n
Indica el número de línea.
- ▶ -e
indica que el siguiente argumento es una expresión.
- ▶ -q
silencioso, no saca nada por la salida (cuando solo nos interesa el status de salida).

Sed

- ▶ Stream Editor
- ▶ Editor de flujos de texto con comandos.
- ▶ Basado en Ed (editor con comandos, tatarabuelo de vi).
- ▶ Muchas de las cosas de sed, igual en ed.

Sed

- ▶ Es un editor: aplica el comando de sed a cada línea que lee y escribe el resultado por su salida. Sin el modificador `-n`, escribe todas las líneas después de procesarlas.
- ▶ Si queremos usar expresiones regulares extendidas, hay que usar la opción `-E`.
- ▶ Comandos:
 - `q` → Sale del programa.
 - `d` → Borra la línea.
 - `p` → Imprime la línea. (correr con `-n`)
 - `r` → Lee e inserta un fichero.
 - `s` → Sustituye. ← la que más se usa!!!

- ▶ Direcciones:

número → actúa sobre esa línea.

/regexp/ → líneas que encajan con la regexp.

\$ → la última línea.

- ▶ Se pueden usar intervalos:

número , número → actúa en ese intervalo.

número , \$ → desde la línea *número* hasta la última.

número , /regexp/ → desde la línea *número* hasta la primera que encaje con regexp.

Sed

Ejemplos:

sed '3,6d' → borra las líneas de la 3 a la 6

sed -E -n '/BEGIN|begin/,/END|end/p' → imprime las líneas entre esas regexp

sed '3q' → imprime las 3 primeras líneas.

sed -n '13,\$p' → imprime desde la línea 13 hasta la última.

sed -E '/[Hh]ola/d' → borra las líneas que contienen 'Hola' u 'hola'.

Sustitución

- ▶ `sed 's/regexp/sustitución/'` → sustituye la primera subcadena que encaja con la exp. por la cadena *sustitución*.
- ▶ `sed 's/regexp/sustitución/g'` → sustituye todas las subcadenas de la línea que encajan con la exp. por la cadena *sustitución*.
- ▶ `sed 's/(regexp)regexp.../\1 sustitución/g'` → usa las subcadenas que encajaron con las agrupaciones en la cadena de sustitución.

Sed

Ejemplos

`sed 's/[0-9]/X/'` → el primer dígito de la línea se sustituye por una X.

`sed 's/[0-9]/X/g'` → todos los dígitos de la línea se sustituyen por una X.

`sed 's/^([A-Za-z]+) ([A-Z]+)/NOMBRE:\1 NOTA:\2/g'`

Imprimir:

- ▶ Lenguaje completo de programación de texto.
- ▶ Útil, veremos sólo la superficie.

Imprimir:

- ▶ **print**
Sentencia que imprime los operandos. Si se separan con comas, inserta un espacio. Al final imprime un salto de línea.
- ▶ **printf()**
Función que imprime, ofrece control sobre el formato de forma similar a la función de libc para C:

```
$> ls -l | awk '{ printf("Size: %08d KBytes\n", $6) }'
```

Variables:

- ▶ \$0
La línea que está procesando.
- ▶ \$1, \$2 ...
El primer, segundo... campo de la línea.
- ▶ NR
Número del registro (línea) que se está procesando.
- ▶ Ejemplo
para imprimir la tercera y segunda columna de un csv:
`$> cat a.txt|awk -F, '{printf("%d\t%d", $3, $2)}'`

Variables:

- ▶ NF

Número del campos del registro que se está procesando.

- ▶ nombrevar=contenido

Se pueden declarar variables dentro del programa. Con el modificador -v se pueden pasar variables al programa.

```
$> ls -l | awk '  
{  
size=$6 ; printf("Size: %08d KBytes\n", size)  
}',
```

patrón { programa }

Actuando sólo en unas líneas, que se ajustan a un patrón, que puede ser:

- ▶ Expresión regular

Se procesan las líneas que encajen con la regexp.

```
$> ls -l | awk '/[Dd]esktop/{ print $1 }'
```

```
$> ls -l | awk '$1 ~ /[Dd]esktop/ { print $1 }'
```

- ▶ Expresión de relación

Se comparan valores y se evalúa la expresión.

```
$> ls -l | awk 'NR >= 5 && NR <= 10 { print $1 }'
```

Inicialización y finalización:

```
BEGIN{
```

```
...
```

```
}
```

```
patrón{
```

```
...
```

```
}
```

```
END{
```

```
...
```

```
}
```

Recorrer un árbol

- ▶ Para recorrese un árbol de ficheros
 - ▶ du -a
 - ▶ find
 - ▶ tree
- ▶ Ojo: los nombres con espacios nos pueden dar problemas.
- ▶ Construcción habitual:

```
find . | while read line ; do echo la linea es: "$line" ; done
```

- ▶ Ojo: todos los comandos que se ejecutan en un pipeline se ejecutan en un proceso hijo, por tanto el while se ejecutará en una subshell (con lo que eso conlleva: variables, status de salida, etc.).

Join

- ▶ join
- ▶ Extremadamente útil
- ▶ Hace un *join* relacional de dos columnas (tienen que estar ordenadas)

```
$> echo '  
a bla  
b ble  
c blo' > a.txt  
$> echo '  
a ta  
b te  
c to' > b.txt  
$> join a.txt b.txt  
a bla ta  
b ble te  
c blo to
```

Join

- ▶ join quita las que no están en alguno de los dos (inner join)
- ▶ Tienen que estar ordenadas, usar sort antes
- ▶ Igual que sort puede usar diferentes campos

Peligro

- ▶ Ojo con las redirecciones
- ▶ Esto crea un fichero vacío
- ▶ ¿Por qué?

```
$> echo '
> a bla
> b ble
> c blo' > a.txt
$> cat a.txt | tail > a.txt
$> cat a.txt
```

Otros comandos

- ▶ Para ejecutar comandos sobre una lista
 - ▶ echo a b c |xargs ls -l
- ▶ Para crear ficheros o directorios temporales (el comando crea el fichero/directorio y escribe su ruta por su salida estándar):
 - ▶ mktemp
 - ▶ mktemp -d

Comandos texto

- ▶ cut y paste
- ▶ Mejor dominar awk (puede hacer lo mismo que estos comandos)