Guiones del intérprete de órdenes Bash

versión 1 - 12-9-2016

Contenido: shebang, variables, variables de entorno, variable PATH, obtención de las partes del nombre de un fichero por separado (mediante expansión de parámetros), control del flujo de un programa (condicionales if, bucles for y bucles while), sustitución de órdenes, argumentos al invocar al guión, documentos-aquí, expansión de llaves, aplicar una orden recursivamente a un directorio mediante un guión, orden sed para sustituciones, du, df, wc, head, tail, nice, ionice, rsync y guión de copia de seguridad.

1 Para trabajar más cómodo con el intérprete de órdenes y no teclear tantas órdenes podemos juntarlas en un guión de órdenes (shell script o bash script). Para ello escribimos un fichero con las órdenes que queramos, le damos permiso de ejecución y ya lo podemos ejecutar. Por ejemplo, podemos escribir en un fichero:

```
#!/bin/bash
# Cada línea que empiece por almohadilla (#) es un comentario y no
# se interpreta.
# Pero si empieza por #! no es un comentario, se llama shebang, y
# le indica a bash quién va a intérpretar las órdenes, en este caso
# también bash.
#
# Mostramos por la pantalla un saludo:
echo 'Hola mundo'
#
# Le decimos a bash que todo salió bien mediante el valor 0,
# para indicar error usamos cualquier otro valor
exit 0
```

lo llamamos `hola.sh` (no hace falta ponerle extensión pero así lo diferenciamos de un programa binario) y lo guardamos, por ejemplo en `~/fulanito_bin`. Ahora le damos permiso de ejecución y lo ejecutamos tecleando `~/fulanito_bin/hola.sh` desde cualquier directorio o bien nos vamos al directorio `~/fulanito_bin` y tecleamos `./hola.sh`.

2 El término “command” se ha traducido como un falso amigo por “comando” cuando la traducción correcta es “orden”. Por ello es habitual encontrar en libros e internet: “el comando `ls`” en vez de “la orden `ls`”.

3 Una variable es un espacio de la memoria, con un nombre, donde se guarda información. Para crear una variable llamada `nombre` y guardar el contenido `Fulanito` en ella teclea en la terminal:

```
nombre=Fulanito
```

y muestra su valor tecleando:

```
echo $nombre
```

Pero no funciona si el dato contiene espacios, en ese caso:

```
nombre="Fulanito de Tal"
```

y muestra su valor tecleando:

```
echo "$nombre"
```

y evitamos muchísimos problemas ENVOLVIENDO SIEMPRE las variables entre comillas.

Esta variable se borra al cerrar la terminal.

4 Las variables en bash, al contrario que en otros lenguajes como C, no hace falta definirlas, las usamos cuando queramos.

5 Prueba y analiza el guión `io.sh` el cual usa el la orden `read` para leer por la entrada estándar (teclado). En este ejemplo la variable `j` solo existe mientras se ejecuta el guión.

6 Modifica el guión `hola.sh` anterior, llamándolo `hola_interactivo.sh`, para que haga un saludo personalizado. El guión preguntará el nombre del usuario, el usuario lo tecleará y posteriormente se mostrará en la pantalla "`Hola nombre tecleado`".

7 Los nombres de las variables del sistema, también llamadas variables de entorno, están escritas en mayúsculas. Comprueba el valor de estas variables en tu sistema: `LOGNAME`, `SHELL`, `LANG`, `HISTFILE` y `PATH`.

8 La variable `PATH` le indica a `bash` en qué directorio tiene que buscar cuando queremos ejecutar un programa. Por ejemplo `gedit` se encuentra en `/usr/bin`, y tecleando simplemente `gedit` se abre. Esto es porque la ruta `/usr/bin` está en la variable `PATH`. Si no estuviera deberíamos escribir `/usr/bin/gedit` para ejecutarlo. Vamos a guardar todos los guiones que hagamos en la carpeta `~/fulanito_bin`. Podemos modificar la variable `PATH` para incluir esa ruta para la sesión en curso tecleando:

```
export PATH=$PATH:~/fulanito_bin
```

pero vamos a hacer el cambio permanente escribiendo esa línea en el fichero de configuración de nuestra sesión `~/.bashrc`. Cierra la terminal y al volver a abrirla ya tenemos nuestra ruta en la variable.

9 No es conveniente incluir el directorio en curso (`.`) en la variable `PATH` para así teclear `orden.sh` en vez de `./orden.sh`:

`export PATH=$PATH:.`

puesto que si hay un guión malicioso con el mismo nombre de una orden habitual podemos ejecutarlo sin darnos cuenta.

Peor todavía sería así:

`export PATH=.:$PATH`

puesto que ahora conseguiríamos suplantar incluso a la usadísima orden `ls`, puesto que primero buscaría en el directorio en curso y luego en `/usr/bin`.

10 A la hora de trabajar con ficheros es muy útil acceder por separado al nombre completo con la ruta, a nombre completo sin la ruta, a la ruta, a la extensión y al nombre sin la extensión. Comprueba lo siguiente escribiéndolo en el prompt.

Primero definimos la variable `f`:

```
f="/home/usuario/ejemplo.tar.gz"
```

Ahora vamos obteniendo diferentes partes del nombre:

`echo "${f%%.*}"`

obtendrás: `/home/usuario/ejemplo`

`echo "${f%.*}"`

obtendrás: `/home/usuario/ejemplo.tar`

`echo "${f#*.}"`

obtendrás: `tar.gz`

`echo "${f##*.}"`

obtendrás: `gz`

`echo "${f%/*}"`

obtendrás: `/home/usuario`

`dirname "$f"`

obtendrás: `/home/usuario`

`echo "${f##*/}"`

obtendrás: `example.tar.gz`

`basename "$f"`

obtendrás: `example.tar.gz`

`echo "${f%/*}"/"${f##*/}"`

obtendrás: `/home/usuario/example.tar.gz`

`basename` y `dirname` son funciones de bash que usan `${f%/*}` y `${f##*/}`. Por lo tanto lo más seguro es usar `${f%/*}` y `${f##*/}` por si no están implementadas aunque ya vemos que son muy poco amigables.

11 Condicionales. Al evaluar una expresión lógica cero significa cierto y uno falso. Por otro lado el código que devuelve una orden al ejecutarse vale cero si ha sido un éxito y cualquier otro valor si no.

Recuerda que en C, por el contrario, en una expresión lógica cero es falso y cualquier otro valor es cierto. Pero el valor devuelto por un programa es igual que en `bash`, cero significa éxito y cualquier otro número error .

Cualquier expresión entre corchetes es una forma de invocar a la orden `test` que la evalúa y devuelve, como acabamos de indicar, un cero si es cierta o uno si es falsa (ver `man test`). Se puede evaluar (más opciones en la página de manual de `test`):

`-f fichero` cierto si `fichero` existe y es un fichero normal

`-d fichero` cierto si `fichero` existe y es un directorio

`-e fichero` cierto si `fichero` existe, independientemente del tipo que sea

`-s fichero` cierto si `fichero` existe y tiene tamaño mayor que cero

`n1 -eq n2` cierto si los números enteros `n1` y `n2` son iguales

`n1 -ne n2` cierto si los números enteros `n1` y `n2` no son iguales

`n1 -gt n2` cierto si el número entero `n1` es mayor que `n2`

`n1 -ge n2` cierto si el número entero `n1` es mayor o igual que `n2`

`n1 -lt n2` cierto si el número entero `n1` es menor que `n2`

`n1 -le n2` cierto si el número entero `n1` es menor o igual que `n2`

`-z s1` cierto si la longitud de la cadena de texto `s1` es cero

`-n s1` cierto si la longitud de la cadena de texto `s1` es mayor que cero

`s1 = s2` cierto si las cadenas de texto `s1` y `s2` son idénticas

`s1 != s2` cierto si las cadenas de texto `s1` y `s2` no son idénticas

`s1 < s2` cierto si la cadena de texto `s1` se ordena primero alfabéticamente que `s2`

`s1 > s2` cierto si la cadena de texto `s1` se ordena después alfabéticamente que `s2`

Por ejemplo:

```
x="1"
y="1"
[ "$x" -eq "$y" ]
echo $?
```

es evaluado como cierto y por tanto devuelve un cero, que comprobamos mostrando

el valor de la variable `?`, la cual guarda el resultado de la ejecución de la última orden.

12 La construcción del condicional `if` es:

```
if órdenes o tuberías
then
  echo "La expresión es cierta"
fi
```

y relacionado con el ejemplo anterior:

```
if [ "$x" -eq "$y" ]
then
  echo "x e y son iguales"
fi
```

recuerda que los corchetes son una manera de invocar a la orden `test`, luego la sintaxis es correcta.

Veamos tres maneras de hacer lo mismo, la primera:

```
mkdir -p pepe && cd pepe
```

la segunda:

```
if  mkdir -p pepe
then
  cd pepe
fi
```

y la tercera:

```
mkdir -p pepe
if [ $? -eq 0 ]
# La variable ? contiene el resultado de la ejecución de la última
# orden
then
  cd pepe
fi
```

Y la construcción `if` completa es:

```
if órdenes o tuberías
then
  echo "Haz esto si es cierto"
else
  echo "Haz esto otro si es falso"
fi
```

13 Escribe un guión, llamado `mayor_igual_3.sh`, que pida un número e indique si es mayor o igual que 3.

14 El bucle `while` se define así (lo usaremos más adelante):

```
while órdenes o tuberías
do
  echo "Haz esto mientras es cierto"
done
```

15 En las órdenes y tuberías se pueden combinar condiciones usando los operadores lógicos:

`&&` (y lógico)

`||` (o lógico)

`!` (negación lógica)

Por ejemplo:

```
if ! [ "$x" -ne "$y" ]
then
  echo "x e y son iguales"
fi
```

y

```
z=1
if [ "$x" -eq "$y" ] && [ "$y" -eq "$z" ]
then
  echo "x y z son iguales"
fi
```

16 La orden `seq` imprime una secuencia de números. ¿Cómo puedes imprimir los diez primeros números? ¿Y los diez primeros impares?

17 Cuando una orden está envuelta entre `$() bash`, además de ejecutarla, la sustituye por su salida estándar y de error. Por ejemplo prueba:

```
f=$(ls)
echo $f
```

y mira la diferencia con:

```
f="$(ls)"
echo "$f"
```

Una sintaxis anticuada es envolver la orden entre acentos graves (`), por ejemplo la orden anterior:

```
f=`ls`
```

aunque no es muy recomendable porque es fácil equivocarse al ser unos signos ortográficos tan pequeños y parecidos a la comilla simple mecanográfica (`'`) pero está bien conocerla porque se encuentra en muchos guiones.

18 Bucle `for`. Se construye así (difiere bastante de otros lenguajes de programación):

```
for i in item1 item2 item3 etc.
do
  echo $i
done
```

Analiza y prueba el guión `bucle_for.sh`

Como ves no es muy práctico y suele usarse esta construcción:

```
for i in $(seq 1 10)
do
  echo $i
done
```

19 Trabajando con ficheros suele ser muy útil un bucle que recorra todos los ficheros del directorio:

```
for i in $(ls)
do
  echo $i
done
```

Pero este método da problemas cuando los nombres tienen espacios por eso es muy habitual esta construcción del bucle `while`:

```
ls | while read i
do
  echo "$i"
done
```

donde redireccionamos la salida de `ls` a la entrada de `read` (recuerda el guión `io.sh`). Cada ítem de `ls` es una llamada a la orden `read` la cual devuelve un valor cierto que permite realizar una nueva iteración en el bucle `while` hasta que se acaben los datos y ya no entra en el bucle y continúa la ejecución del guión por debajo del bucle.

20 Ahora creamos un guión para dejar los permisos de los ficheros y directorios, recursivamente, en unos valores bastante habituales:

```
#!/bin/bash
# Cambiamos recursivamente los permisos de los directorios a 755 y
# de los ficheros a 644
find . -depth -type d -exec chmod -v 755 '{}' \;
```

```
find . -depth -type f -exec chmod -v 644 '{}' \;
exit 0
```

lo llamamos `permisos.sh` y le damos permiso de ejecución. Nos vamos al directorio en el que queremos modificar los permisos de todos sus ficheros y directorios recursivamente y lo ejecutamos.

21 Lo podemos mejorar para que acepte el directorio de trabajo como argumento en vez de tener que estar en el directorio sobre el que queremos que actúe:

```
#!/bin/bash
# Cambiamos recursivamente los permisos de los directorios a 755 y
# de los ficheros a 644
# Comprobamos que el guión se invoca indicando el directorio como
# argumento
if [ $# -ne 1 ]; then
 echo "Uso: "${0##*/}" ruta"
 exit 1
fi
find "$1" -depth -type d -exec chmod -v 755 '{}' \;
find "$1" -depth -type f -exec chmod -v 644 '{}' \;
exit 0
```

y lo llamamos `permisos_argumento.sh`

Donde:

`$0` contiene el nombre de nuestro guión

`$#` el número de argumentos con los que se ha invocado al guión

`$n` los argumentos, con `n` de `1` a `$#`

`"${0##*/}"` muestra el nombre del guión sin la ruta absoluta.

El condicional `if` verifica si el número de argumentos es diferente de uno y en ese caso muestra un mensaje en la pantalla y aborta el guión enviando un código de error.

22 Modifica el guión `hola_interactivo.sh` para que lea el nombre del usuario como argumento de la orden y llámalo `hola_lotes.sh`

23 Cuando queremos aplicar una orden a todos los ficheros de un directorio habitualmente usamos los comodines como en este ejemplo: `rm *`. Pero con otros órdenes no podemos proceder de ese modo, por ejemplo si queremos añadir a los nombres de los ficheros el sufijo `_prueba`. E incluso aunque se pueda se controla mejor la operación con un guión que recorra el directorio con un bucle. Para generar el fichero vamos a usar un método automático, los “documentos aquí” o “documentos empotrados” (Here Documents) de `bash`. Simplemente copia el

código de abajo y ejecútalo en la terminal. Una vez lo hayas hecho mueve el fichero `renombra.sh`, si quieres, a tu directorio `.bin` y pruébalo ejecutándolo desde el directorio `renombra`.

```
cat > renombra.sh << "EOF"
#!/bin/bash
#
#================================================================
# Añade a todos los ficheros del directorio en curso el sufijo
# _prueba
#================================================================
#
ls * | while read f
do
  mv -v "${f}" "${f}"_prueba
done
EOF
chmod +x renombra.sh
```

Los here documents funcionan así: `cat` lee por la entrada estándar hasta que se encuentra con los caracteres `EOF` (se suele usar End Of File por costumbre) y lo que lee lo envía al fichero `renombra.sh`.

24 El guión anterior no es recursivo. Modifícalo para que lo sea (que renombre todos los ficheros recursivamente pero no los directorios), nómbralo `renombra_recursivo.sh` y pruébalo con el directorio `renombra_recursivo`

25 Igual de práctico que los "documentos aquí" es la expansión de llaves, prueba este orden:

```
mkdir 1_ESO_{A,B,C,D,E,F}
```

Como ves, bash expande el contenido de la llave y `mkdir` es llamado con seis argumentos:

```
mkdir 1_ESO_A 1_ESO_B 1_ESO_C 1_ESO_D 1_ESO_E 1_ESO_F
```

Así podemos hacer guiones o instrucciones más compactos aunque evidentemente más crípticos.

26 El fichero `plantilla.sh` es un buen punto de partida para realizar cualquier guión.

27 En el directorio `fotos` hay imágenes de tamaño 1024x768 píxeles. La orden `convert` (perteneciente al paquete `Imagemagick`) redimensiona las imágenes a 640x480 píxeles:

```
convert -verbose -sample 640x480 img000.jpg img000_640x480.jpg
```

Escribe un guión que reciba como argumento el directorio de trabajo, llamado

`re_640x480_mod.sh`, que guarde las imágenes modificadas añadiendo a los nombres, tras la extensión, el sufijo `_640x480`, quedando por tanto así: `img000.jpg_640x480`

Escribe otro guión que reciba como argumento el directorio de trabajo, llamado `re_640x480_modificadas.sh`, que guarde las imágenes modificadas en una carpeta dentro de `fotos` llamada `modificadas` y le añada a los nombres, tras la extensión, el sufijo `_640x480`, quedando por tanto así: `img000.jpg_640x480`

Escribe otro guión que reciba como argumento el directorio de trabajo, llamado `re_640x480_sob.sh`, que redimensione todas las imágenes sobrescribiendo los originales.

28 Modifica el guión anterior, llamándolo `calidad.sh`, para que reduzca la calidad de la imágenes al tanto por ciento que le indiquemos al invocarlo. Una vez que te funcione puedes añadirle la mejora de que compruebe que has introducido un número en el porcentaje (necesitarás haber hecho la práctica "Búsqueda y tratamiento de datos mediante grep, expresiones regulares, awk y sed") que está comprendido entre 1 y 99, ambos incluidos.

29 Chequear el código html de muchos ficheros con la herramienta de W3C (https://validator.w3.org/) es muy ineficiente. Por ello escribe un guión llamado `html_chequeo.sh` que chequee recursivamente todos los ficheros del directorio `html`.

Para ello usarás la orden `tidy`, que corrige los fallos de sintaxis (etiquetas sin cerrar, etiquetas de cierre sin la barra, etc.) Primero prueba con un fichero individual su funcionamiento probando estas opciones:

```
tidy -utf8 a.html
```

Por la salida estándar te muestra el fichero modificado (si ha sido necesario) y por la salida de error te informa de los errores. Como ambas salidas se muestran por la pantalla es un poco lío por eso prueba:

```
tidy -utf8 a.html >a_modificado.html 2>error
```

es igual que:

```
tidy -utf8 -f error a.html > a_modificado.html
```

Y si quieres modificar el fichero original usamos la opción `m`:

```
tidy -utf8 -f error -m a.html
```

30 Escribe un guión llamado `html_identar.sh` que idente con `tidy` el código html. Identar es formatear el código con sangrías y saltos de línea para que se lea mejor.

31 La orden `sed` permite, entre otras muchas opciones, sustituir texto en un fichero de texto simple. Sed significa editor de flujo (stream editor) y en contraposición con una edición interactiva edita en modo no interactivo por lo que es muy útil en guiones. Prueba estas órdenes.

En cada línea del fichero sustituye todas las apariciones de `cortito` por `corto` mostrando el resultado por la pantalla:

```
sed 's/cortito/corto/g' a.html
```

ídem que antes pero envía el resultado a un fichero:

```
sed 's/cortito/corto/g' a.html > a_modificado_por_sed.html
```

y ahora sobrescribe el fichero original:

```
sed -i 's/cortito/corto/g' a.html
```

Ahora vamos a resolver el siguiente problema: todos los ficheros del directorio `html` tienen un enlace a una dirección web que ha cambiado. Para no usar el ordenador como una máquina de escribir y modificar los ficheros uno a uno a mano usaremos el siguiente guión para modificarlos:

```
cat > modifica_contenido.sh << "EOF"
#!/bin/bash
#
#================================================================
# Modifica recursivamente el contenido de archivos de texto
#================================================================
#
if [ $# -ne 1 ]
then
  echo "Uso: "${0##*/}" ruta"
  exit 1
fi


find $1 -iname "*.html" | while read f
do
  sed -i 's/sabina.pntic.mec.es\/lmuf0005/www.netcom.es\/vildeu/g'
"$f"
done
EOF
chmod +x modifica_contenido.sh
```

Observa en la orden `sed` que como la barra (`/`) separa opciones, para poder escribir la barra de la dirección de internet tenemos que escaparla con la contrabarra (`\`) para indicarle a `sed` que en ese caso no es el separador.

32 Escribe un guión, llamado `cambio_autor.sh`, que modifique el propietario del copyleft, de `Fulanito de Tal y Cual` a tu nombre, en todos los archivos de la carpeta `latex`.

33 En el lenguaje html, para evitar problemas al usar diferentes codificaciones, es bastante común sustituir las tildes y la eñe por sus entidades html (por ejemplo la á por `á`), dejando que el navegador las interprete inequivocamente. Escribe un guión, llamado `html_tildes_enes.sh`, que sustituya las tildes y la eñe, tanto minúsculas como mayúsculas, de los ficheros del directorio `html`. En la orden hay que escapar el signo et (`&`) porque para `sed` es un carácter especial y hay que indicarle que en este caso no lo es: `sed -i 's/á/\á/g' ficheros.html`

34 En el directorio `nombres_complicados` hay nombres con espacios. Prueba el guión `espacios.sh` que sustituye los espacios por guiones bajos. Observarás la siguiente línea:

```
echo "$nombre" | sed 's/ /_/g'
```

En este caso los datos que usa `sed` no vienen de un fichero sino de una tubería y como toda la orden está entre acentos graves `bash` sustituye dicha orden por la salida de su ejecución y ese es el valor que finalmente se guarda en la variable `nuevonombre`:

```
nuevonombre=`echo "$nombre" | sed 's/ /_/g'`
```

Recuerda que usar acentos graves es un método anticuado y que ahora se envuelve entre `$()` puesto que se lee mejor pero es conveniente ver el método antiguo porque se usa en muchos guiones.

35 Modifica el guión anterior, llamándolo `tildes_enes.sh`, para sustituir los nombres con tildes y eñes, tanto minúsculas como mayúsculas, por las letras sin tildes ni virgulillas en el directorio `nombres_complicados`.

36 Modifica el guión `renombra.sh`, llamándalo `renombra_numerando.sh`, para que renombre los ficheros del directorio `renombra` con un patrón introducido como argumento y una numeración que comience por uno, por ejemplo lo invocamos así:

```
renombra_numerando.sh nombre_tonto
```

Y nos genera los nombres: `nombre_tonto-1`, `nombre_tonto-2`, etc. Consulta el guión `contador.sh` para saber cómo manejar una variable contador.

37 Aún faltan algunos órdenes imprescindibles para completar la navaja suiza o caja de herramientas de `bash`.

37.1 Prueba:

```
du
du -h
du -h plantilla.sh
```

37.2 Prueba:

```
df
df -h
```

37.3 Prueba:

```
wc plantilla.sh
cat plantilla.sh | wc
ls | wc
```

37.4 Prueba:

```
head plantilla.sh
head -n 1 plantilla.sh
ls | head
ls | head -n 1
```

37.5 Prueba:

```
tail plantilla.sh
tail -n 1 plantilla.sh
ls | tail
ls | tail -n 1
```

37.6 Qué función tienen los órdenes `tr` y `cut` en estas órdenes:

```
df datos
df datos/ | tr -s ' ' | cut -d ' ' -f4 | tail -n1
```

37.7 Averigua qué hace este orden: `date +%Y-%m-%d-%H-%M-%S`. ¿Qué utilidad puede tener?

37.8 ¿Qué hacen los órdenes `nice` e `ionice`?. ¿Cómo puedes usarlos cuando ejecutas un orden, por ejemplo `cp -r fotos fotos_copia`?

37.9 Una línea larga la podemos cortar tecleando la contrabarra (`\`) justo antes del retorno de carro, así `bash` ignora el retorno de carro (o sea, escapamos el carácter de retorno de carro). Por ejemplo la orden:

```
find . -iname "*.sh" -exec chmod -v +x "{}" \;
```

Podemos cortarla en dos líneas:

```
find . -iname "*.sh" -exec \
chmod -v +x "{}" \;
```

Se suelen cortar para facilitar la lectura de la orden o si no queremos tener líneas más largas de 80 caracteres puesto que pueden resultar incómodas en algunos editores y suponer pérdida de información al enviar el guión en el cuerpo de un correo.

37.10 Averigua qué hace esta orden: `rsync -av  datos copia_seguridad`

Borra, modifica y crea ficheros en el directorio `datos`, vuelve a ejecutarlo y comprueba el resultado.

37.11 Averigua qué hace esta orden:

```
rsync -avb --backup-dir=incr/$(date +%Y-%m-%d-%H-%M-%S) \
```

`--delete datos copia_seguridad`

Borra, modifica y crea ficheros en `datos`, vuelve a ejecutarla y comprueba el resultado.

Cuando se indican los directorios con rutas relativas el directorio configurado en `--backup-dir` es relativo al directorio de destino, en este caso `copia_seguridad`.

38 Ahora puedes entender casi por completo el fichero `copia_seguridad.sh`, el cual puede resultarte muy útil para hacer copias de seguridad de tus datos. Incluso puedes incluirlo en `anacron` para que en cada arranque se ejecute solo. Tan solo tienes que configurar las rutas a tus directorios.

39 Como vemos `bash` tiene herramientas de programación (variables, bucles, condicionales, etc.) por lo que se puede resolver prácticamente cualquier necesidad que surja al trabajar con el sistema operativo. De hecho muchas veces buscamos programas que realicen una función específica cuando sabiendo escribir guiones lo podemos resolver nosotros mismos. Por ejemplo un renombrador masivo (como el programa gráfico Thunar) lo podemos sustituir, como hemos visto, por un guión escrito por nosotros. Pero un gran poder lleva asociado una gran responsabilidad y debemos ser muy precavidos haciendo siempre copias de seguridad antes de hacer un cambio masivo no sea que no obtengamos el resultado esperado.

40 Un guión de `bash` ejecutándose es indistinguible del código ejecutable de un programa compilado. Obviamente se diferencian en que uno está en ASCII y el otro en binario por lo que siempre podemos leer el código fuente de un guión `bash`.

41 Más información en la página de manual, en

http://www.gnu.org/software/bash/manual/ y en http://tldp.org/LDP/abs/html/