

Práctica 7.

Programación en Shell.

Primeros pasos.



Autor: Francisco de Asís Conde Rodríguez / Lina García Cabrera Copyright:



Índice

Índice.....	1
Introducción.....	2
¿Por qué necesito saber programar shell scripts?.....	2
Formato de un script.....	3
Anatomía de un shell script.....	3
Convenciones de nombramiento de Script o guión.....	4
Agregando el Shebang.....	5
Normas de estilo.....	5
¡Hola mundo! nuestro primer shell script.....	5
¿Cómo llamar a un shell script y dónde colocarlo?.....	7
Script para imprimir los contenidos de una carpeta proporcionada por un usuario.....	8
Depuración de guiones.....	8
Parámetros.....	9
Accediendo al valor de los parámetros.....	10
Estableciendo el valor de variables.....	10
Ejemplo de uso de parámetros posicionales.....	11
¿Para qué se usan los parámetros posicionales?.....	11
Parámetros especiales (\$*, @\$ y \$#).....	12
Ejemplos de uso de parámetros especiales.....	13
Ejemplos de uso de variables.....	14
Ejercicios sobre comportamiento de las variables.....	15
Ejemplos y propuestas de bash scripts sencillos.....	17

Introducción

El sistema operativo Linux ofrece a su administrador cientos de órdenes muy potentes y flexibles listas para ser ejecutadas, con las que podrá realizar la mayoría de sus tareas cotidianas.

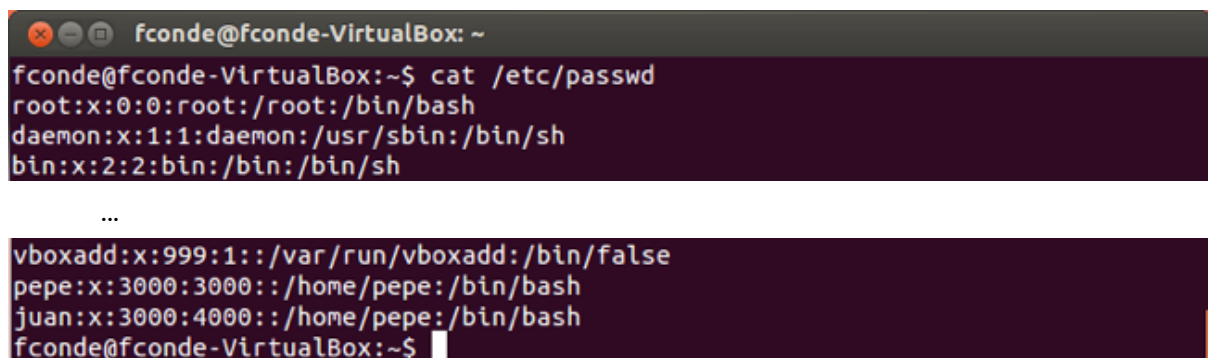
Sin embargo, siempre existen tareas repetitivas, o que incluyen secuencias largas de órdenes, o que incluyen órdenes con sintaxis compleja difícil de recordar, que es mejor tener automatizadas para ahorrar tiempo, y sobre todo, para evitar errores en su realización.

La forma en que se puede automatizar una tarea de administración en Linux es mediante pequeños programas que se pueden ejecutar en la shell, llamados **shell scripts**.

¿Por qué necesito saber programar shell scripts?

Veamos un ejemplo: Una de las tareas de auditoría que un administrador de sistemas Linux debe realizar es comprobar si existen en el sistema dos usuarios con la misma UID, ya que pueden acceder a los ficheros del otro usuario sin que el sistema pueda distinguir quien es quien. Eso incluso aunque tengan nombre de usuario distinto ya que el sistema usa siempre la UID.

Esta situación anómala, se puede comprobar, listando el contenido del archivo `/etc/passwd` y comprobando que no haya dos líneas en las que la tercera columna sea igual.



```
fconde@fconde-VirtualBox: ~  
fconde@fconde-VirtualBox:~$ cat /etc/passwd  
root:x:0:0:root:/root:/bin/bash  
daemon:x:1:1:daemon:/usr/sbin:/bin/sh  
bin:x:2:2:bin:/bin:/bin/sh  
...  
vboxadd:x:999:1::/var/run/vboxadd:/bin/false  
pepe:x:3000:3000::/home/pepe:/bin/bash  
juan:x:3000:4000::/home/pepe:/bin/bash  
fconde@fconde-VirtualBox:~$
```

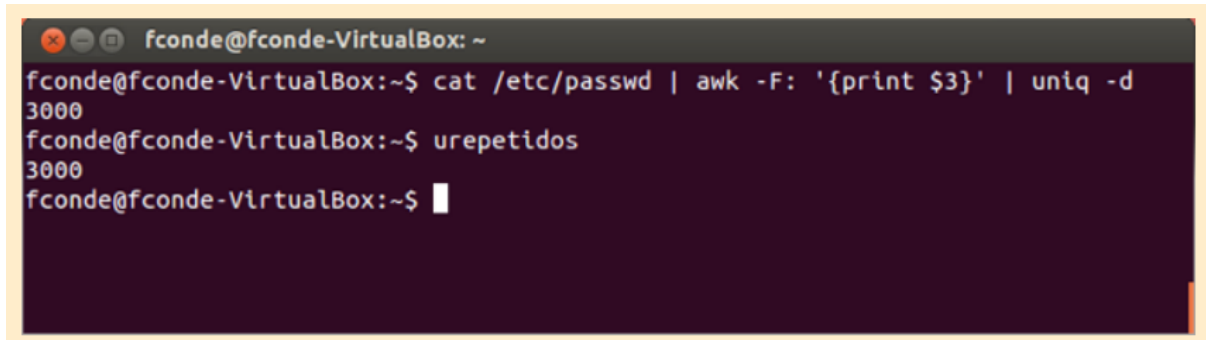
En este ejemplo, los usuarios pepe y juan, ambos tienen UID 3000, lo cual es un error muy grave, que el administrador del sistema debe comprobar y reparar.

El archivo `/etc/passwd` puede contener muchos registros, por lo que la comprobación manual es muy pesada y propensa a errores si las entradas con UID repetida están muy separadas entre sí. Una forma mejor de comprobar esta situación es mediante la orden¹:

¹ No te preocupes si no entiendes la orden. Ese no es el objetivo por ahora.

```
cat /etc/passwd | awk -F: '{print $3}' | sort | uniq -d
```

Es una orden difícil de recordar y escribir, es mejor automatizarla mediante un shell script al que podemos llamar `urepetidos.sh` que hace el mismo trabajo pero mucho más fácil.

A screenshot of a terminal window titled 'fconde@fconde-VirtualBox: ~'. The terminal shows three lines of command execution. The first line is 'fconde@fconde-VirtualBox:~\$ cat /etc/passwd | awk -F: '{print \$3}' | uniq -d', which outputs '3000'. The second line is 'fconde@fconde-VirtualBox:~\$ urepetidos', which also outputs '3000'. The third line shows the prompt 'fconde@fconde-VirtualBox:~\$' with a cursor, indicating the command has finished.

NOTA: `awk` procesa ficheros de texto, se usa para procesar la salida del comando anterior. El parámetro `-F:` establece el carácter de separación de campos (en este caso, `:`), ya que los campos en el archivo `/etc/passwd` están separados por dos puntos. Luego, `{print $3}` le indica a `awk` que imprima el tercer campo de cada línea, que corresponde al UID.

El comando `uniq -d` se utiliza para mostrar solo las líneas duplicadas (UIDs repetidos) en la entrada. Para que la orden `uniq` funcione correctamente, la entrada debe estar ordenada, lo que no se asegura en este caso a menos que se use `sort`.

Formato de un *script*

Un shell script (o programa shell) es un archivo de texto que contiene una o más órdenes que realizan una tarea concreta, la tarea que se quiere automatizar.

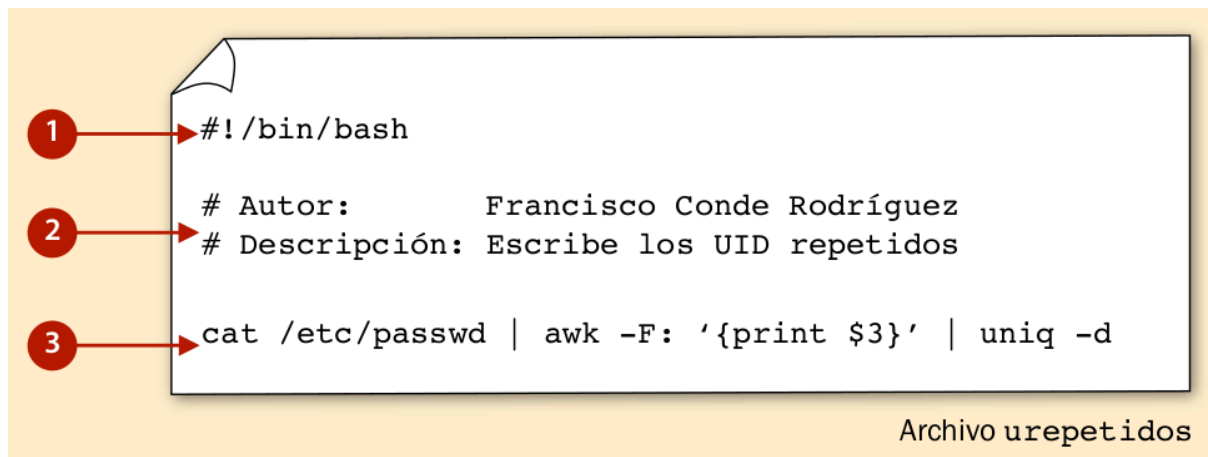
Son las mismas órdenes que escribiríamos si estuviésemos realizando la tarea a mano. La diferencia es que están escritas en un archivo de texto para que el intérprete de órdenes las ejecute cuando se invoque al *shell script*.

Cuando se ejecuta un *shell script*, el intérprete de órdenes, lee las órdenes. una a una, del archivo y las ejecuta por orden.

Los programas de shell no se compilan, sólo hay que escribirlos mediante un editor de textos y darles permisos de ejecución para poder usarlos.

Anatomía de un *shell script*

Un shell script es un archivo de texto que contiene una o varias órdenes para el sistema. Por ejemplo, el archivo `urepetidos.sh` que contiene el programa del ejemplo anterior tiene en su interior:



Todos los shell scripts deberían tener las siguientes partes:

- (1) **Cabecera (hash-bang).** La primera línea de un shell script es un tipo especial de comentario. Comienza por los caracteres `#!` y va seguido del nombre de un intérprete de órdenes o shell. Dice al sistema qué intérprete de órdenes debe usar para ejecutar el archivo. En este ejemplo `/bin/bash`.

IMPORTANTE: Los caracteres `#!` deben aparecer justo al comienzo del archivo, es decir, deben ser los dos primeros bytes del archivo de texto para que el sistema los pueda reconocer.

- (2) **Comentarios.** Se indican con el carácter `#` al comienzo de una línea o palabra. Todos los *shell scripts* deberían incluir como mínimo quién es su autor y una breve descripción de para qué sirve el *script*. El intérprete ignora el comentario desde el signo `#` hasta el final de la línea.

IMPORTANTE: Un carácter `#`, sólo se interpreta como el comienzo de un comentario, si va justo al principio de una línea o de una palabra de un shell script. Si va en medio o al final de una palabra de un shell script, no se interpreta como comentario.

- (3) **Órdenes.** Las órdenes que componen el *shell script*. Se escriben de la misma forma que se escriben en la consola cuando se ejecutan a mano.

Convenciones de nombramiento de Script o guión

Por convención de nomenclatura, los scripts de bash terminan con `.sh`. Sin embargo, los scripts de bash pueden ejecutarse perfectamente sin la extensión `sh`, para que se puedan ejecutar lo único que se debe hacer es activar el permiso de ejecución.

Agregando el Shebang

Los scripts de Bash comienzan con un **shebang** (en algunas ocasiones se le denomina también *hash-bang* o *shebang*). Shebang es una combinación de bash # y bang ! seguido de **la ruta de la shell de bash**. Esta es la primera línea del script. Shebang le dice a la shell que lo ejecute por medio de la shell de bash. Shebang es simplemente un path absoluto al intérprete de bash.

```
#!/bin/bash
```

Puedes encontrar tu path de la shell de bash (el cual podría variar del de arriba) usando el comando:

```
lina@lina-PROX15-AMD:~$ which bash
/usr/bin/bash
```

Normas de estilo

Entre las normas que se dan a la hora de escribir un guión o script, se indica que es buena costumbre incluir comentarios para conocer siempre quién lo ha escrito, en qué fecha, qué hace, etc. Para ello, utilizaremos en símbolo "#", bien al inicio de una línea o bien tras una orden. Por ejemplo, nuestros guiones pueden empezar:

```
#!/bin/bash

# Título: prueba
# Fecha: 5/10/2011
# Autor: Profesor de SO
# Versión: 1.0
# Descripción: Guion de prueba para la práctica x
# Opciones: Ninguna
# Uso: prueba directorio
```

¡Hola mundo! nuestro primer shell script

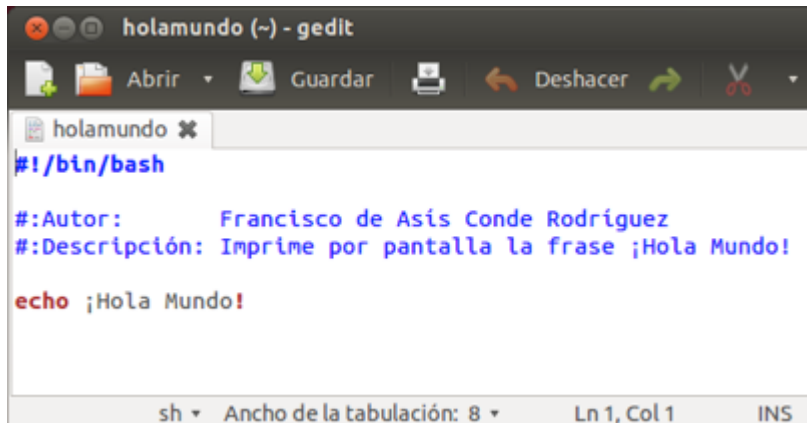
Vamos a escribir nuestro primer shell script, un sencillo programa llamado `holamundo` que simplemente escriba por pantalla la frase: `¡Hola mundo!`.

Si no estamos en nuestro directorio base `~`, tecleamos la orden `cd` sin argumentos para ir hasta él.

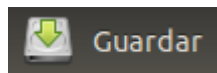
1. Abrir un editor de textos y escribir el programa. Para ello ejecutamos²:

```
gedit holamundo.sh &
```

Escribimos el texto que aparece en la figura.



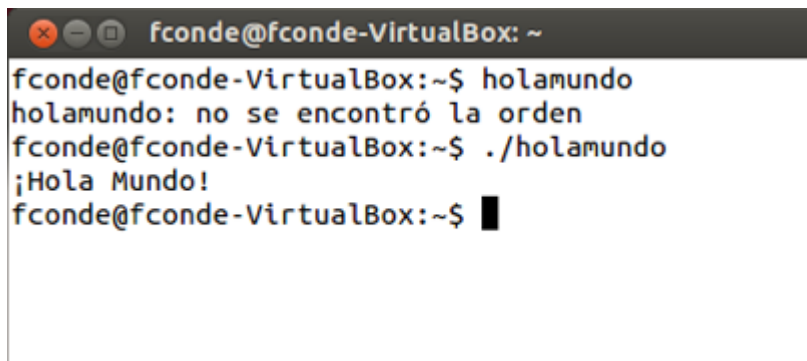
Pulsamos el botón Guardar.



2. Dar permisos de ejecución al archivo. Para ello ejecutamos:

```
chmod +x holamundo.sh
```

3. Ejecutar el archivo escribiendo su nombre: `./holamundo.sh`



IMPORTANTE: si simplemente se escribe `holamundo`, el sistema no encuentra al programa y no lo ejecuta. Por tanto siempre hay que especificar la ruta completa.

NOTA: El signo `&` final de la orden `gedit holamundo &`, indica que el proceso `gedit` se ejecute en *background*. En este ejemplo se hace así para no bloquear la consola y poder seguir ejecutando órdenes en ella.

² **& (ampersand):** Indica que el programa `gedit` se debe ejecutar en segundo plano (*background*), lo que significa que, aunque se inicie el editor, la terminal o consola no quedará bloqueada esperando a que se cierre el editor. Esto te permite seguir ejecutando otros comandos en la misma terminal sin tener que cerrar `gedit`.

¿Cómo llamar a un *shell script* dónde colocarlo?

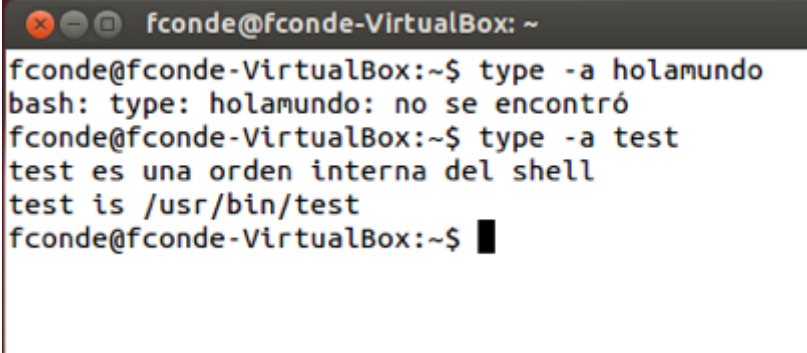
Podemos dar a nuestros *shell script* cualquier nombre que queramos, sin embargo, si ese nombre ya pertenece a alguna orden interna del intérprete de órdenes, o a alguna orden del sistema, puede causar confusiones.

Por ello, se recomienda que antes de elegir un nombre para nuestro script, comprobemos que no pertenece a ninguna orden del sistema o del intérprete. Para ello existe la orden

```
type -a [nombre script]
```

donde `[nombre script]` se sustituye por el nombre que queramos comprobar.

Por ejemplo, podemos comprobar que el nombre `holamundo.sh` no pertenece a ninguna orden, mientras que el nombre `test` o `ls` sí.



```
fconde@fconde-VirtualBox: ~  
fconde@fconde-VirtualBox:~$ type -a holamundo  
bash: type: holamundo: no se encontró  
fconde@fconde-VirtualBox:~$ type -a test  
test es una orden interna del shell  
test is /usr/bin/test  
fconde@fconde-VirtualBox:~$
```

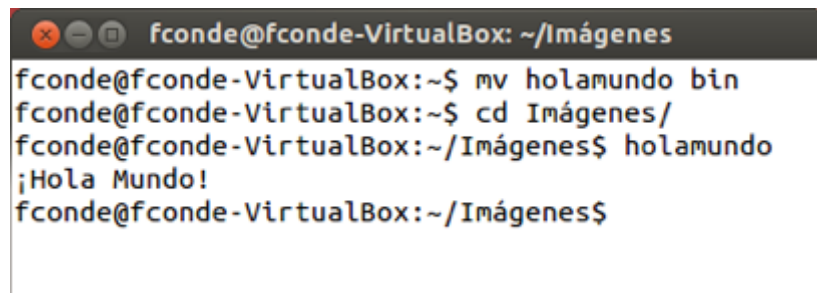
Podemos colocar el *shell script* en cualquier carpeta donde tengamos permisos, sin embargo se recomienda tener una carpeta específica para los ejecutables llamada `bin`, que cuelgue directamente del directorio base del usuario `~/bin`.

En Ubuntu Linux (desde la versión Ubuntu 20.04 LTS), la ruta `~/bin` o `$HOME/bin` se añade automáticamente a la variable de entorno `PATH` (para ver su valor ejecute `echo $PATH`), con lo que los programas que se coloquen en esa carpeta se pueden ejecutar directamente aunque no estemos situados en ese directorio.

Para crear la carpeta `~/bin` escribimos:

```
mkdir ~/bin    o    mkdir ${HOME}/bin
```

Cerramos la sesión y volvemos a entrar, a partir de ese momento, para ejecutar los programas que hayamos copiado en la carpeta `bin` sólo hay que escribir su nombre (sin `./`), y el intérprete los encontrará, da igual el directorio en donde estemos.

A screenshot of a terminal window titled 'fconde@fconde-VirtualBox: ~/Imágenes'. The terminal shows a sequence of commands and their outputs: first, 'mv holamundo bin' is executed; then, 'cd Imágenes/' is executed; finally, 'holamundo' is executed, which outputs '¡Hola Mundo!'. The prompt returns to 'fconde@fconde-VirtualBox: ~/Imágenes\$'.

Script para imprimir los contenidos de una carpeta proporcionada por un usuario

```
#!/bin/bash
# Autora: Lina García Cabrera
# Descripción: Imprime la fecha y el contenido de una
# carpeta que lee de la entrada

1  #!/bin/bash
2  echo "Hoy es " `date`
3
4  echo -e "\nEscribe la ruta al directorio"
5  read la_ruta
6
7  echo -e "\n tu ruta tiene los siguientes archivos y carpetas:"
8  "
   ls $la_ruta
```

Línea #1: El shebang (`#!/bin/bash`) apunta hacia la ruta de la shell de bash.

Línea #2: El comando `echo` muestra la fecha actual y el tiempo en la shell. Nota que `date` está con comillas invertidas.

Línea #4: Queremos que el usuario escriba una ruta válida. La opción `-e` permite que los caracteres de escape dentro de la cadena se interpreten. `\n`: Este es un carácter de escape que representa un salto de línea (nueva línea).

Línea #5: El comando `read` lee la entrada y lo almacena en la variable `la_ruta`.

Línea #8: El comando `ls` toma la variable con la ruta almacenada y muestra los archivos y las carpetas actuales.

Depuración de guiones

La depuración de guiones puede resultar bastante compleja. No existen programas que permitan analizar los errores sintácticos de un guión. No obstante, las siguientes opciones de bash son una gran ayuda a la hora de depurar un guión:

```
$ bash -x guion
```



```
$ bash -v guion
$ bash -n guion
```

Cada opción actúa de una manera diferente

- x** hace que bash ejecute el guión asignando previamente la variable `echo`, esto provoca que se muestren en la pantalla las órdenes que va ejecutando el guión.
- v** hace que bash ejecute el guión asignando previamente la variable `verbose`, esto provoca que se muestren en la pantalla las órdenes que va ejecutando el guión después de las sustituciones históricas.
- n** se comprueba la sintaxis del guión pero no se ejecuta. Pero solo detecta los errores más graves.

Una forma sencilla de depuración, es usar la orden `echo` en puntos críticos del guión para seguir el rastro de las variables más importantes.

También es posible introducir comentarios, como se ha visto en los ejemplos anteriores, en un guión utilizando el carácter `#`, todo lo que aparezca en la línea tras `#` no es ejecutado. Tenga presente que el carácter `#` no se interpreta si se utiliza en un shell interactivo:

```
$ echo a # b Produce sólo a
```

Parámetros.

Para que un *shell script* sea realmente útil, debe usar parámetros. Según la ayuda de manual que acompaña a Ubuntu Linux, un parámetro es “una entidad que almacena valores” y puede ser de tres tipos: parámetros posicionales, parámetros especiales y variables.

1. **Parámetros posicionales.** Son los argumentos presentes en la línea que invoca a la orden (la llamada) y se referencian mediante números que indican su posición.

Por ejemplo, en la orden `ls /bin`, hay un parámetro posicional, `/bin`, que ocupa la posición 1.

2. **Parámetros especiales.** Su contenido lo rellena el intérprete de órdenes para guardar información sobre su estado actual, como por ejemplo el número de parámetros posicionales de la orden que se está ejecutando.

Se referencian mediante caracteres no alfanuméricos, como por ejemplo `*` o `#`.

3. **Variables.** Las usa el programador del *shell script* para almacenar cualquier información que necesite. Se referencian mediante un nombre.

El nombre de una variable puede ser cualquier combinación de letras, números o el

guión bajo (`_`) y siempre comienza por una letra o el guión bajo.

Por ejemplo, nombres válidos de variables son: `resultado`, `res_1`, o `_res`, pero no `1res`.

¿Por qué NO debo usar mayúsculas en los nombres de las variables?

Porque los nombres de variables en mayúsculas están reservados para las variables internas del shell, y se corre el riesgo de sobrescribirlas. Los scripts pueden dejar de funcionar o tener un mal comportamiento. Por tanto, **NUNCA uses MAYÚSCULAS en los nombres de tus variables.**

Accediendo al valor de los parámetros

Para acceder al valor de cualquier parámetro, se usa el signo `$` precedido del identificador del parámetro. También se llama sustitución de parámetros.

Por ejemplo, para acceder al contenido del parámetro posicional que ocupa la segunda posición se escribe `$2`, para acceder al contenido del parámetro especial `#`, se escribe `$#`, y para acceder al contenido de la variable `resultado`, se escribe `$resultado`.

También se puede encerrar entre llaves el identificador del parámetro. Esto permite delimitar bien cual es ese identificador y es el método preferible³ aunque sea más difícil de escribir. Así, los ejemplos anteriores también se pueden escribir de la siguiente forma:

```
${2}, ${#} o ${resultado}.
```

Estableciendo el valor de variables

Para crear una variable y asignarle un valor sólo hay que escribir el nombre de la variable, un signo igual (=) y el valor. Por ejemplo:

```
nombre=Francisco
```

IMPORTANTE: no debe haber espacios en esa asignación o el sistema dará un error.

```
nombre = Francisco
```

no es correcto.

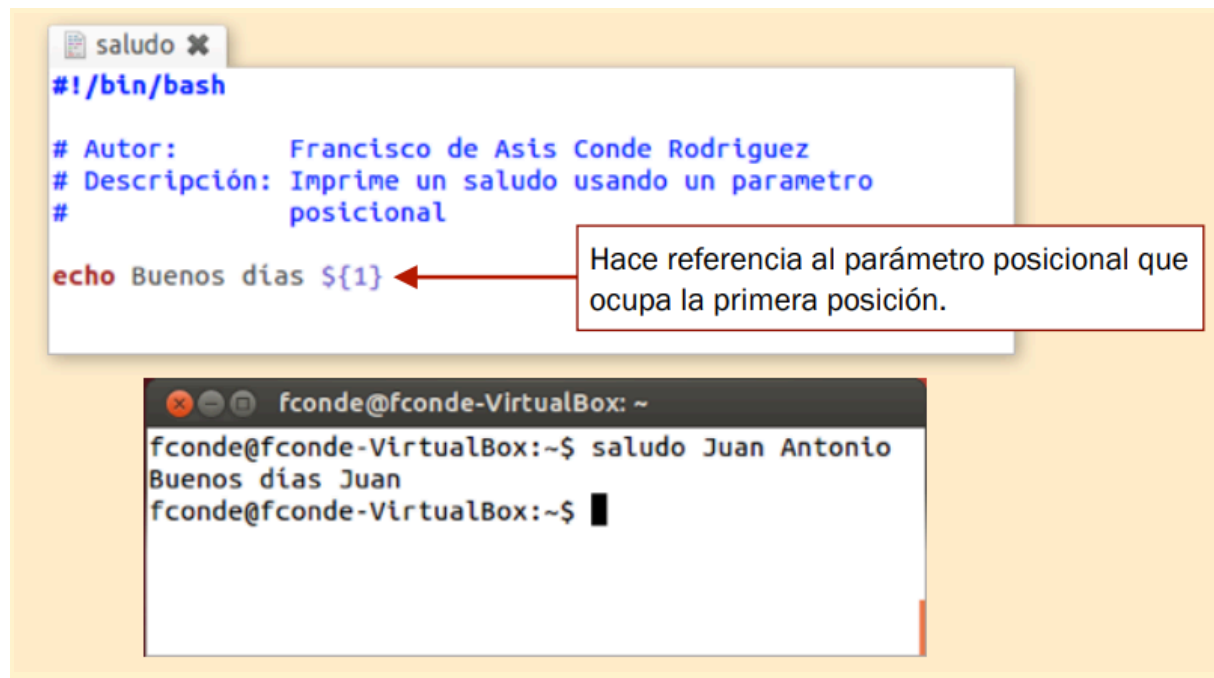
Para eliminar una variable se usa `unset`. Por ejemplo:

³ En algunos casos es obligatorio. Por ejemplo, si se quiere acceder a un parámetro posicional con identificador mayor que 9, debe hacerse obligatoriamente usando llaves. Por ejemplo `${15}`.

```
unset nombre
```

Ejemplo de uso de parámetros posicionales.

Vamos a escribir un shell script que reciba como argumento un parámetro posicional y que imprima un saludo seguido del valor de ese parámetro posicional. Llamaremos al script `saludo`.



Fíjate que en la llamada al script `saludo` del ejemplo anterior, hay dos parámetros posicionales, *juan* y *antonio*, pero sólo se imprime el primero, porque en la orden `echo` sólo se hace referencia al primer de ellos.

¿Para qué se usan los parámetros posicionales?

Para poder pasar argumentos a un *shell script*, como por ejemplo el directorio en el que trabajar, o el archivo que comprobar; y para pasar opciones a un *shell script* (recuerda que las opciones se preceden de un guión).

Aquellos *shell scripts* que necesiten trabajar con argumentos u opciones pueden hacer referencia a ellos mediante el uso de los parámetros posicionales.

Parámetros especiales (\$*, @\$ y \$#)

La variable \$# almacena el número de argumentos recibidos por el script y es de tipo cadena de caracteres.

Tanto \$* como @\$ nos devuelven los argumentos recibidos por el script. Cuando no se entrecomillan se comportan igual pero al encerrarlos entre comillas débiles "\$*" crea un único token con todos los argumentos recibidos mientras que "\$@" crea un token por cada argumento recibido. Esto es, "\$*" es equivalente a "\$1 \$2 \$3..." pero "\$@" equivale a "\$1" "\$2" "\$3" ...

Diferencia entre comillas dobles y comillas simples en el shell

En Linux, el uso de comillas dobles (" ") y comillas simples (' ') en el shell tiene un impacto significativo en cómo el sistema interpreta los caracteres especiales y variables. La diferencia clave entre ambas es cómo tratan la expansión de variables y el uso de caracteres especiales.

Comillas Dobles (" ")

Las comillas dobles permiten la expansión de variables y la interpretación de algunos caracteres especiales.

- **Expansión de variables:** Si usas comillas dobles, el shell expande las variables (como \$var), los caracteres de escape (\), y los comandos entre comillas inversas (`).
- **Carácter de escape (\):** Dentro de comillas dobles, puedes usar el carácter de escape (\) para evitar que algunos caracteres especiales se expandan.
- **"\$*" todos los argumentos se tratan como una sola cadena, unidos por el primer carácter del IFS (Internal Field Separator, que por defecto es el espacio).**
- **"\$@" cada argumento se trata como un elemento individual, respetando las comillas originales de los argumentos.**

Veamos algunos ejemplos:

```
var="Mundo"
echo "Hola $var"
```

Hola Mundo

Aquí, el contenido de la variable \$var se expandió dentro de las comillas dobles.

```
echo "El precio es \$50"
```

El precio es \$50

El carácter de dólar (\$) se escapó usando \, por lo que no fue interpretado como el

comienzo de una variable.

Comillas Simples (' ')

Las comillas simples deshabilitan toda expansión de variables y la interpretación de caracteres especiales. Todo lo que se coloque entre comillas simples se interpreta literalmente.

- **No hay expansión de variables:** Los valores dentro de las comillas simples se tratan como una cadena literal, es decir, los caracteres especiales y variables no se expanden.
- **No interpretación de caracteres especiales:** Ni el carácter de escape (\), ni los comandos entre comillas inversas (`) tienen efecto dentro de comillas simples.

Veamos unos ejemplos:

```
var="Mundo"  
echo 'Hola $var'
```

```
Hola $var
```

En este caso, `$var` no se expande porque las comillas simples lo tratan como texto literal, por lo que `$var` se imprime tal cual.

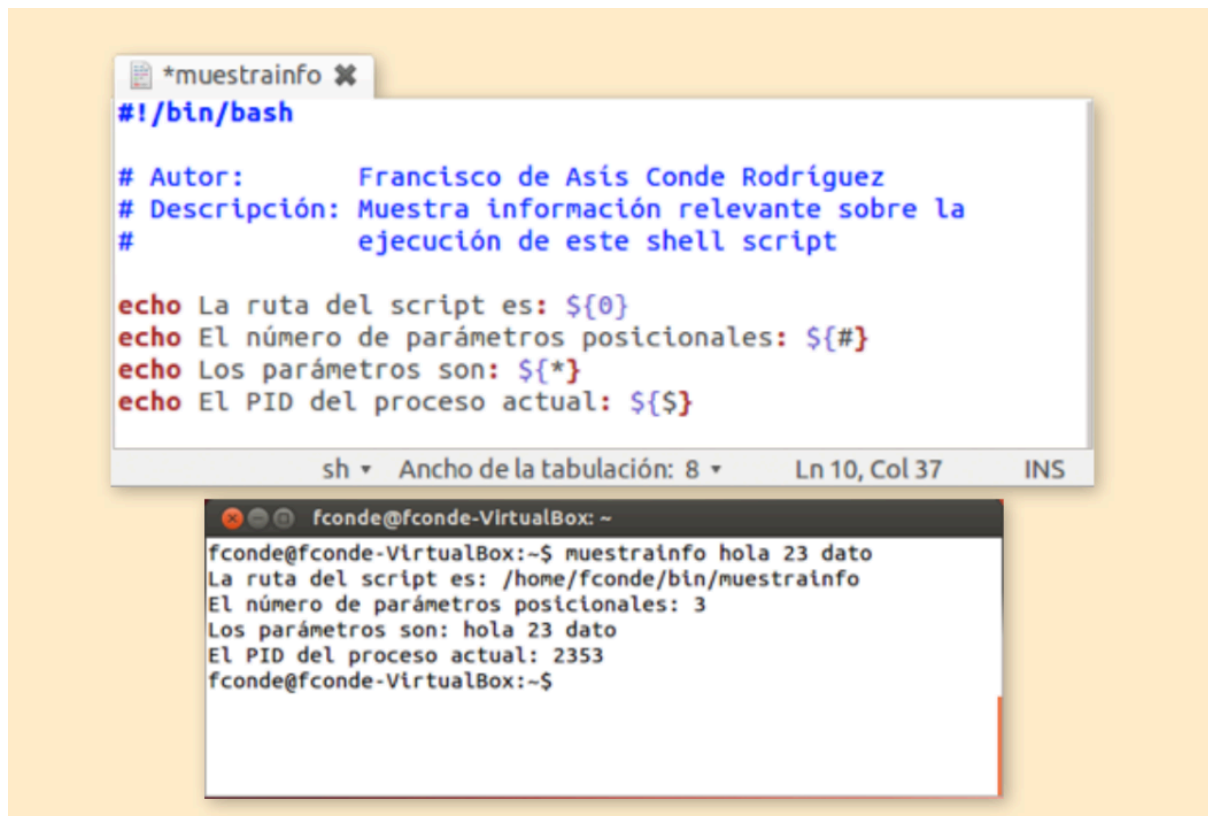
```
echo 'El precio es \$50'
```

```
El precio es \$50
```

Todo lo que está entre comillas simples se interpreta literalmente, por lo que el carácter `\` no tiene ningún efecto.

Ejemplos de uso de parámetros especiales

Vamos a escribir un shell script que haciendo uso de los parámetros especiales, nos muestre información relevante sobre la ejecución de ese *shell script*. Llamaremos a ese shell script `muestrainfo.sh`.



```
#!/bin/bash

# Autor:      Francisco de Asís Conde Rodríguez
# Descripción: Muestra información relevante sobre la
#              ejecución de este shell script

echo La ruta del script es: ${0}
echo El número de parámetros posicionales: ${#}
echo Los parámetros son: ${*}
echo El PID del proceso actual: ${$}
```

```
fconde@fconde-VirtualBox: ~$ muestrainfo hola 23 dato
La ruta del script es: /home/fconde/bin/muestrainfo
El número de parámetros posicionales: 3
Los parámetros son: hola 23 dato
El PID del proceso actual: 2353
fconde@fconde-VirtualBox:~$
```

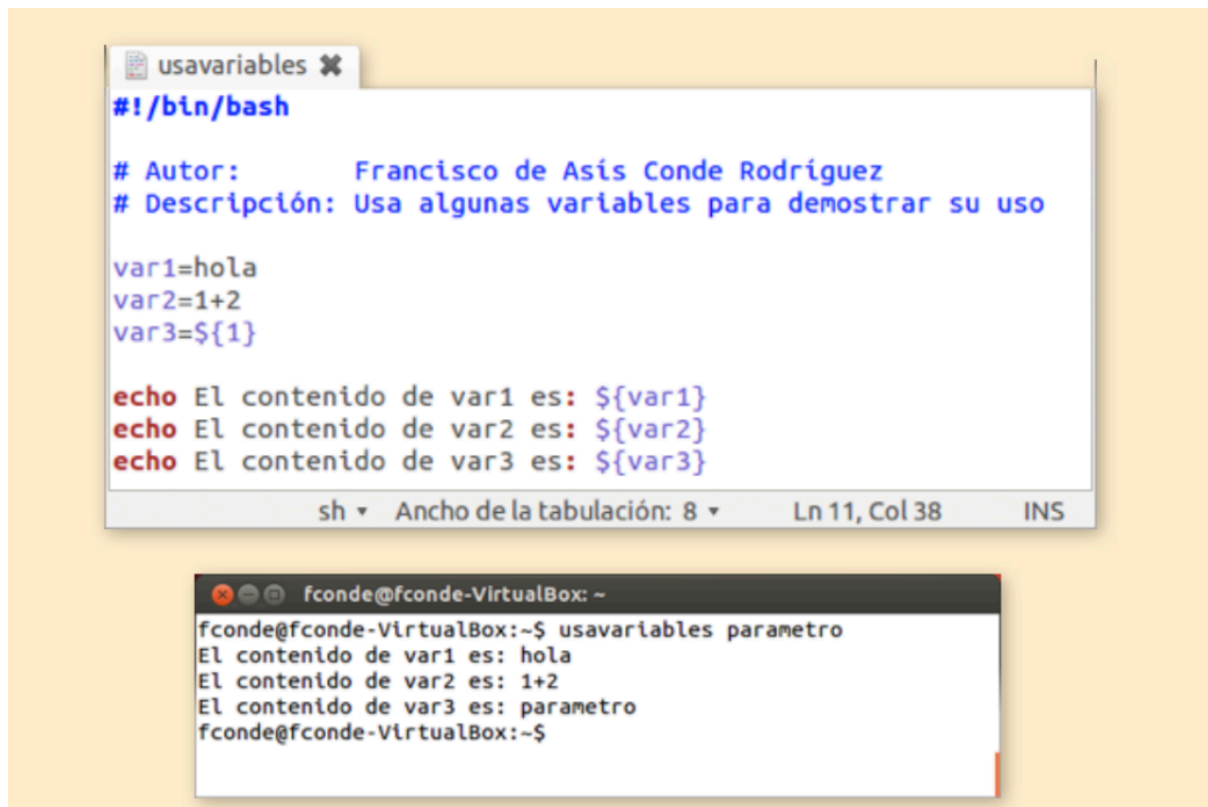
Como podemos ver, en el shell script `muestrainfo.sh`, se usan cuatro parámetros especiales⁴: `0`, `#`, `*` y `$`.

- 0** El parámetro especial `0`, se sustituye por la ruta completa hacia el shell script que se está ejecutando.
- #** El parámetro especial `#`, se sustituye por el número de parámetros posicionales que se hayan pasado al shell script durante la llamada, en el ejemplo 3: `hola`, `23` y `dato`.
- *** El parámetro especial `*`, se sustituye por el valor de todos los parámetros posicionales que se hayan pasado al shell script.
- \$** El parámetro especial `$`, se sustituye por el identificador del proceso actual.

Ejemplos de uso de variables.

Vamos a escribir un *shell script* que ilustre el uso de variables. Llamaremos a ese *shell script* `usavariablen.sh`.

⁴ Hay nueve parámetros especiales en total. El resto: `@`, `?`, `_`, `!` y `-`, se irán viendo en prácticas posteriores, a medida que su uso se vaya haciendo necesario.



The image shows two terminal windows. The top window, titled 'usavariabes', displays the script's content: a shebang line, author and description comments, variable assignments for 'var1', 'var2', and 'var3', and three 'echo' commands. The bottom window, titled 'fconde@fconde-VirtualBox: ~', shows the script being executed with the argument 'parametro', resulting in the output of the 'echo' commands.

```
#!/bin/bash

# Autor:      Francisco de Asís Conde Rodríguez
# Descripción: Usa algunas variables para demostrar su uso

var1=hola
var2=1+2
var3=${1}

echo El contenido de var1 es: ${var1}
echo El contenido de var2 es: ${var2}
echo El contenido de var3 es: ${var3}
```

```
fconde@fconde-VirtualBox: ~$ usavariabes parametro
El contenido de var1 es: hola
El contenido de var2 es: 1+2
El contenido de var3 es: parametro
fconde@fconde-VirtualBox: ~$
```

Aspectos importantes que hay que conocer sobre las variables

En los programas *shell script*, los valores de todas las variables se interpretan como **cadenas de caracteres**. Esa es parte de la potencia de los *shell script*. Permiten un procesamiento muy potente de cadenas de caracteres. Eso se puede comprobar en las líneas 7 y 11 del script `usavariabes.sh`. Como puede verse a la variable `var2` se le asigna el valor `1+2` y el intérprete de órdenes lo asigna como la cadena de caracteres `"1+2"`, no como el valor entero 3.

En la asignación de variables se puede usar el valor de otros parámetros (posicionales, especiales o variables), para ello se usa la sustitución apropiada. En el ejemplo esto se puede ver en la línea 8 del script. Ahí, se le asigna a la variable `var3`, el valor del parámetro posicional 1.

Ejercicios sobre comportamiento de las variables

1. Escribe y ejecuta el siguiente shell script. Llámalo `pruebavariabes01.sh`:

```
#!/bin/bash

# Autor: Francisco de Asís Conde Rodríguez
```

```
# Descripción: Pruebas de uso de variables

var1=hola
var2="hola mundo"
var3=hola mundo

echo El contenido de var1 es: ${var1}
echo El contenido de var2 es: ${var2}
echo El contenido de var3 es: ${var3}
```

¿Qué resultado se obtiene? ¿A qué crees que se debe? ¿Cómo crees que se resuelve?

2. Escribe y ejecuta el siguiente shell script. Llámalo `pruebavariabes02.sh`:

```
#!/bin/bash

# Autor: Francisco de Asís Conde Rodríguez
# Descripción: Pruebas de uso de variables

var1=3
var2=${var1}hola
var3=$var1hola

echo El contenido de var1 es: ${var1}
echo El contenido de var2 es: ${var2}
echo El contenido de var3 es: ${var3}
```

¿Qué resultado se obtiene? ¿A qué crees que se debe? ¿Por qué no se produce un error?

3. Escribe y ejecuta el siguiente shell script. Llámalo `pruebavariabes03.sh`:

```
#!/bin/bash

# Autor: Francisco de Asís Conde Rodríguez
# Descripción: Pruebas de uso de variables

var1=3

echo El contenido de var1 es: ${var1}
unset var1
echo El contenido de var1 es: ${var1}
```

¿Qué resultado se obtiene? ¿Por qué no se produce un error?

Ejemplos y propuestas de bash scripts sencillos

```
#!/bin/bash
# Autora: Lina García-Cabrera
# Descripción: Este guion duerme durante x segundos

# Uso: duerme.sh segundos

echo "Espera $1 segundos"

sleep $1
echo "Completado"
```

```
#!/bin/bash

# Autora: Lina García-Cabrera
# Descripción: Escribe texto con y sin saltos de línea y con
tabuladores

echo -n "Imprimiendo texto sin nueva línea"
echo "Imprimiendo texto con nueva línea"
echo -e "\nInterpretación \t Caracteres \t Tabulador\n"
echo "\nNO Interpretación \t Caracteres \t Tabulador\n"

# mostrar los parámetros de entrada con tabulador
echo -e "\nLos 3 parámetros de entrada son \t$1 \t$2 \t$3\n"
```

1. Escribe un script bash al que se le pasan 2 argumentos al que llamaremos **cpalabra.sh**: el nombre de un fichero de texto y una palabra o frase que debe pasarse entre comillas dobles. El programa cuenta el número de palabras o frases que hay en el fichero de texto que se pasa como argumento. Indicaciones: utiliza las órdenes `grep` (busca la opción que solo muestra las coincidencias, no la línea completa) y `wc`.
2. La orden **uniq** sin argumentos elimina las líneas de texto que están duplicadas sólo si son adyacentes. Escribe una nueva orden que se llamará **uniq_all.sh** que recibe dos parámetros, 2 nombres de ficheros. Lo que hace es quitar las líneas repetidas de un fichero estén o no consecutivas y lo guarda en el otro fichero. A continuación tienes un ejemplo del contenido de un fichero con líneas repetidas.

```
Julio Lorenzo
Pedro Andi3n
Celia Fern3ndez
Celia Fern3ndez
Juan Fern3ndez
Enrique Pea3a
Julio Lorenzo
Pedro Andi3n
Celia Fern3ndez
Juan Fern3ndez
Enrique Pea3a
Julio Lorenzo
```

3. El comando **du** sirve para verificar el uso de espacio en disco en un directorio. Escriba una orden llamada **queocupa.sh** al que se pasan 3 argumentos: un directorio, y una extensi3n (algo como .jpg, .png) y un n3mero **num** y muestra los **num** ficheros que m3s ocupan con esa extensi3n. Indicaciones: Utiliza la opci3n de la orden **du** que muestra el espacio en Byte, Kilobyte, Megabyte, Gigabyte, Terabyte and Petabyte, usa las3rdenes **grep, sort** (buscar las opciones que ordena num3ricamente).
4. Escriba una orden llamada **masenlaces.sh** al que se pasa 2 argumentos: un directorio y un n3mero. Esta orden muestra los **n** ficheros o directorios con m3s enlaces duros de ese directorio. Indicaciones: **tail -n +2**: Omite la primera l3nea de la salida, que suele ser el encabezado en el caso de **ls -l** (normalmente "total XX"). La opci3n **-n** especifica el n3mero de l3neas que quieres mostrar, pero el signo **+** cambia el comportamiento de tail. En lugar de mostrar las3r l3timas **n** l3neas, **+2** indica que se deben mostrar todas las l3neas comenzando desde la segunda l3nea.
5. Escriba una orden llamada **num_duplicados.sh** a la que se pasa 2 argumentos: un n3mero y un fichero. Esta orden busca en un fichero de configuraci3n (por ejemplo, /etc/passwd o /etc/group) si uno de sus campos num3ricos tiene n3meros duplicados. El argumento n3mero hace referencia al campo que debe verificar. Por ejemplo, en el caso de /etc/passwd, en el campo 3 est3n los UIDs, en el caso de /etc/group, en el campo 3 est3n los GIDs. Formas de invocarlos ser3a:
./num_duplicados.sh 3 /etc/group
./num_duplicados.sh 3 /etc/passwd

Se puede simular un fichero para probarlo en el que se repitan n3meros.