Programación shell script

1. Introducción

Los scripts no son más que **ficheros de texto ASCII puro**, que pueden ser creados con **cualquier editor** del que dispongamos.

```
#!/bin/bash
echo "Hola Mundo"
```

La **primera línea sirve** para **indicar qué shell utilizamos** (en nuestro caso **bash**) y dónde se encuentra en nuestro sistema (para saberlo, podemos hacer **locate bash**). La **segunda línea** de nuestro script, simplemente utiliza el **comando para escribir en pantalla** (**echo**) y **escribe** la línea "Hola Mundo".

Para hacer un script, crearemos un archivo de texto y escribiremos el texto del ejemplo anterior. Una vez creado el fichero, debemos darle **permisos de ejecución** (por ejemplo, una opción podría ser **chmod a+x script.sh**), posteriormente para ejecutarlo debemos llamarlo como **bash script.sh**.

Las **comillas dobles** que hemos usado para escribir "Hola Mundo" **no son necesarias**, y se puede comprobar fácilmente como quitándolas el proceso se ejecuta exactamente igual. Sin embargo, es una **buena práctica** encerrar siempre los textos entre comillas dobles, y en caso de que contengan caracteres especiales (como * o $), es mejor usar **comillas simples**, que son **más potentes** que las comillas dobles.

```
echo esto es un asterisco * sin comillas
echo esto es un dólar y tres letras $ABC sin comillas
echo "esto es un asterisco * entre comillas dobles"
echo 'esto es un asterisco * entre comillas simples'
echo "esto es un dólar y tres letras $ABC entre comillas dobles"
echo 'esto es un dólar y tres letras $ABC entre comillas simples'
```

Si tenemos que **ejecutar varias líneas** y queremos escribirlas en una sola, podemos hacerlo usando el **símbolo punto y coma** para indicar que lo siguiente es otra línea, aunque este en la misma:

```
echo Hola ; pwd ; echo Adios
```

En el ejemplo anterior teníamos tres líneas en una sola. También podemos hacer lo contrario, **escribir una sola línea en varias**. Para ello usamos el **símbolo contrabarra** cuando queramos que nuestra línea se "rompa" y continúe en la línea de abajo, con lo que en el ejemplo tenemos un único comando en tres líneas.

```
echo "Esto es un ejemplo \
de una línea escrita realmen\
te en tres"
```

Si un comando no cabe en una línea, la mayoría de los intérpretes continúan en la línea siguiente. Para establecer específicamente que un comando continúa en la línea siguiente, hay dos formas (una es la del ejemplo anterior), mutuamente excluyentes, ya que se usa una u otra, pero no ambas al mismo tiempo.

1. Terminar la primera línea con \:

   ```
   $ echo $LOGNAME \
   > $HOME
   ```

2. Dejar una comilla sin cerrar:

   ```
   $echo "$LOGNAME
   > $HOME"
   ```

Al continuar el comando en la segunda línea, **el indicador de comandos cambia de $ a >**, es decir, del indicador de comando de **primer nivel** (**PS1**) al indicador de comando de **segundo nivel** (**PS2**). Si no se quiere terminar el comando multilínea, puede interrumpirse el ingreso con Ctrl + C, volviendo el indicador de comando a PS1 inmediatamente.

2. El archivo de comandos o script

Es cómodo poder tener una lista de comandos en un archivo, y ejecutarlos todos de una sola vez únicamente invocando el nombre del archivo. Vamos a crear el archivo **misdatos.sh** con las siguientes líneas:

```
# misdatos.sh
# muestra datos relativos al usuario que lo invoca
#
echo "MIS DATOS."
echo "Nombre: "$LOGNAME
echo "Directorio: "$HOME
echo -n "Fecha: "
date
echo
# fin misdatos.sh
```

El símbolo # indica un comentario. Para poder ejecutar los comandos contenidos en este archivo, es preciso dar al mismo los **permisos de ejecución adecuados**.

```
chmod ug+x misdatos.sh
```

La invocación (ejecución) del archivo puede realizarse dando el **nombre de archivo como argumento a bash** o **invocándolo directamente como un comando**. Puede que sea necesario indicar una ruta absoluta o relativa, o referirse al directorio actual, si el directorio actual no está contenido en la variable PATH.

En un script todo lo que venga después del símbolo # y hasta el próximo caracter nueva línea (Intro), se toma como comentario y no se ejecuta. El primer ejemplo sólo imprime "Hola mundo" ya que la segunda lína no ejecuta nada, pues el símbolo # convierte toda la línea en comentario.

```
echo "Hola Mundo" # comentario hasta fin de línea
# cat /etc/passwd
```

Los scripts **suelen encabezarse con comentarios que indican el nombre de archivo y lo que hace**. Se colocan comentarios de documentación en diferentes partes del script para mejorar la comprensión y facilitar el mantenimiento. Un **caso especial** es **el uso de # en la primera línea para indicar el intérprete con que se ejecutará el script**. El script anterior con comentarios quedaría así:

```
#!/bin/bash
# misdatos.sh
# muestra datos propios del usuario que lo invoca
#
echo "MIS DATOS."
echo "Nombre: "$LOGNAME
echo "Directorio: "$HOME
echo -n "Fecha: "
date # muestra fecha y hora
echo # línea en blanco para presentación
# fin misdatos.sh
```

La primera línea indica que el script será ejecutado con el intérprete de comandos bash. Esta indicación debe ser siempre la primera línea del script y **no puede tener espacios en blanco**.

Scripts del sistema

Cuando el sistema arranca, se ejecutan una serie de scripts cuya función es ejecutar comandos iniciales y asignar valores a las variables de entorno necesarias. Al salir del sistema se ejecutan otros shell scripts cuya función es dejar el sistema preparado para la próxima vez que iniciemos una sesión en él.

Scripts de inicio de sesión

Al arrancar el sistema o cada vez que el usuario inicia una sesión comienza a ejecutarse un shell que leerá los archivos **/etc/environment**, **/etc/profile** y **/etc/bash.bashrc**. A continuación, se ejecutará el archivo **~/.bash_profile**, **~/.bash_login** o **~/.profile**, que están en el directorio personal del usuario que inicia la sesión y que sirven para que el usuario pueda configurar ciertos parámetros de su cuenta. Si se cambia algo en este fichero habrá que volver a reiniciar la sesión de usuario. A menos que ejecutemos el comando **source** seguido del **nombre del shell script que queremos que se ejecute**.

```
echo $SHELL           # nos dice qué shell usamos
echo $BASH_VERSION    # si usamos el shell bash nos informa de su versión
```

Los ficheros **/etc/bash.bashrc** y **~/.bashrc** se ejecutan cada vez que el usuario llama al shell, que puede ser porque entre en el sistema o porque inicie una nueva sesión, lo que nos da la posibilidad de ejecutar comandos diferentes para el inicio de sesión de cada usuario.

En vez de .bash_profile, el fichero se puede llamar .bash_login o .profile. Esto se debe a la existencia de distintas shells cada una de las cuales tiene un nombre diferente para este fichero. Pero si usamos el shell bash, el sistema busca primero .bash_profile, si no existe busca .bash_ login y si no existen ninguno de los dos, ejecuta profile. **Solo se ejecuta uno de los tres**.

Scripts de fin de sesión

Cuando salimos del sistema, el fichero que lee el shell bash es **.bash_logout**. Podemos utilizarlo para incluir comandos que queremos que se ejecuten cuando salimos. Si en el fichero no hay ningún comando o no existe (ya que puede no existir), no se ejecutará ningún comando al terminar la sesión.

3. Uso y expansión de variables en los scripts

Las variables de los scripts son muy simples, ya que **no tienen tipo definido ni necesitan ser declaradas antes** de poder ser usadas. Para **introducir valor en una variable simplemente se usa su nombre**, y para **obtener el valor** de una variable **se le antepone un símbolo dólar**.

```
#!/bin/bash
DECIR="Hola Mundo"
echo $DECIR
```

Este script realiza la misma función que el anterior pero usando una variable. Cualquier valor introducido en una variable se considera alfanumérico, así que en el siguiente script obtenemos por pantalla la cadena de caracteres 4+3.

```
NUMERO=4          # No se debe dejar ningún espacio en la asignación.
echo NUMERO+3
```

En Linux, podemos **usar varias expansiones en las líneas de comandos** que son especialmente útiles en los scripts. La primera expansión consiste en usar **$( )**. Esta expansión permite ejecutar lo que se encuentre entre los paréntesis y devuelve su salida.

```
echo pwd          # escribe por pantalla la palabra pwd
echo $(pwd)       # ejecuta la orden pwd y escribe su resultado
```

Así, por ejemplo, la siguiente instrucción copia el fichero /etc/network/interfaces en el directorio actual con el nombre red190520.conf (suponiendo que estamos en la fecha 19 de mayo de 2020).

```
NOMBRE_FICHERO="red"$(date +%d%m%y)".conf"
cp /etc/network/interfaces $NOMBRE_FICHERO
```

Es perfectamente posible no usar variables en el ejemplo anterior y hacerlo todo en una línea, pero es una buena práctica no complicar excesivamente cada una de las líneas del script, ya que nos permitirá una modificación mucho más simple y la depuración en caso de que existan errores suele ser más rápida.

El efecto conseguido **con $(orden) se puede conseguir también usando la tilde invertida `orden`**. Otra expansión que podemos usar es **$(( ))** (símbolo dólar pero con dos paréntesis). **Los dobles paréntesis también podemos sustituirlos por corchetes**. Esta expansión va a tratar como una **expresión aritmética** lo que este incluido entre los paréntesis, va a evaluarla y devolvernos su valor con lo que obtenemos el valor 7.

```
NUMERO=4
echo $(($NUMERO+3))   # sería lo mismo poner echo $[$NUMERO+3]
```

El comando let nos permite realizar operaciones aritméticas como la anterior, pero sin tener que usar expansiones ni dólares para las variables. Obtenemos el mismo valor 7 que en ejemplo anterior y no hemos usado ni dólar ni paréntesis ni corchete.

```
NUMERO=4
let SUMA=NUMERO+3
echo $SUMA
```

Valores lógicos

El shell toma la convención inversa de C para cierto y falso: **cierto es 0**, y **falso es distinto de 0**. El shell adopta esta convención porque **los comandos retornan 0 cuando no hubo error**. La variables **true** y **false**, que retornan siempre estos valores se usan en algunas situaciones de programación para **fijar una condición**.

- ✔ **true** → este comando **no realiza ninguna acción**, **sólo devuelve siempre 0**, el valor **verdadero**. La ejecución correcta de un comando cualquiera devuelve 0. La ejecución del siguiente código muestra el valor 0, ya que la **variable $? retiene el valor de retorno del último comando ejecutado**.

  ```
  true
  echo $?
  ```

- ✔ **false** → este comando **tampoco realiza ninguna acción, sólo devuelve siempre 1**; cualquier valor diferente de 0 se toma como **falso**. Las diversas condiciones de error de ejecución de los comandos devuelven valores diferentes de 0. Su significado es propio de cada comando. El siguiente código muestra el valor 1.

  ```
  false
  echo $?
  ```

Variables de entorno

Por defecto, cuando creamos una variable únicamente es visible a nivel de script, a menos que se indique que **se almacene en el área de memoria del entorno**, lo que pasará a ser **de entorno** y puede ser vista por todas las shells que estén abiertas, para lo que habría que escribir:

```
export variable
export VAR="VALOR"
```

Estas variables almacenan valores que servirán para que **ciertos programas** las utilicen **o** bien para **configurar ciertos parámetros del entorno de trabajo**. No es obligatorio, pero las variables de entono se suelen escribir en mayúsculas para diferenciarlas de las variables del usuario que las puede escribir en mayúsculas o minúsculas.

Las **variables de entorno las puede utilizar cualquier usuario y desde cualquier terminal**, ya sea el gráfico o alguno de los terminales virtuales. Las variables de entorno tienen un valor asignado por el sistema operativo en el inicio de la sesión hasta el cierre. Se pueden asignar en alguno de los siguientes archivos:

- **/etc/profile**.
- **/etc/environment**, como **PATH**.
- **/etc/default/locale**, como **LANG**, utilizada por algunos programas para definir el lenguaje. El valor por defecto es **es_ES.UTF-8 para el español**. Para el inglés sería **en_US.UTF-8**. Y variaría el valor para otros idiomas.

Para que los cambios en las variables de entorno realizadas en estos ficheros tengan efecto será necesario volver a **reiniciar el sistema o reiniciar la sesión**.

Variables de entorno más usadas	
HOME	Ruta de nuestro directorio personal.
USER	Nombre de usuario.
SHELL	Ruta al intérprete de órdenes que se está ejecutando.
HOSTNAME	Nombre asignado al equipo.
TERM	Tipo de terminal.
LOGNAME	Nombre del usuario que ejecuta la shell.
PWD	Directorio de trabajo actual.
OLDPWD	Para poder usar cd con la opción -. Guarda el último directorio donde estuvimos.
PATH	Rutas en las que el intérprete busca las órdenes a ejecutar cuando no especificamos dónde están. Un valor de la variable PATH puede ser el siguiente: “/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/home/usuario:”
PS1	**Prompt** o indicador del sistema **primario**. El valor suele ser el siguiente, aunque se puede cambiar si queremos por otros valores: “\u@\h:\w\$” donde: **u** - es el nombre de usuario. **h** - es el nombre de la máquina. **w** - es el directorio donde nos encontramos. **$** - indica si es root que escriba # y si es un usuario normal $.
PS2	**Prompt secundario**. Si escribimos un comando, no lo terminamos y vamos a escribir en otra línea, sale este indicador para que acabemos de escribir el comando. El valor suele ser “>”.
SHELL	Contiene la ruta absoluta de la shell que estamos ejecutando.

Si queremos que la variable esté disponible para todos los usuarios, la deberemos almacenar en uno de los archivos del directorio **/etc donde están los scripts que se ejecutan al inicio del sistema**.

Comando	Función	Sintaxis
set	Muestra las variables locales y las de entorno.	`set`
env	Muestra las variables de entorno. **-U | --unset=NOMBRE** → elimina una variable de entorno, en este caso la variable con nombre NOMBRE.	`env [opción]`
export	Exporta una variable local al entorno.	`export variable`
unset	Elimina una variable.	`unset variable`
echo	Para mostrar una cadena de texto por pantalla, que puede ser el valor de una variable. **-n** → no salta de línea cuando termina de mostrar la cadena. **-e** → con esta opción se interpretan ciertos caracteres dentro de la cadena, entre los que destacan: **\c** - no salta de línea al mostrar la cadena. **\t** - inserta el carácter tabulador. **\n** - inserta el carácter de nueva línea.	`echo [opciones] cadena`

Vamos a mostrar las variables de entorno y las locales además del contenido de las siguientes variables PATH, HOME, SHELL, HOSTNAME Y USER y el valor del prompt secundario.

```
env
set | more
echo –e "$PATH \t $HOME \t $SHELL \t $HOSTNAME \t $USER"
echo $PS2
```

4. Operadores

Los **operadores aritméticos** que podemos usar para realizar operaciones son:

Operadores aritméticos	
+	Suma
-	Resta
*****	Multiplicación
/	División
%	Módulo (resto)

Los **operadores relacionales** que podemos utilizar son:

Operadores de comparación de cadenas alfanuméricas	
Cadena1 = Cadena2	Verdadero si Cadena1 o Variable1 es IGUAL a Cadena2 o Variable2
Cadena1 **!=** Cadena2	Verdadero si Cadena1 o Variable1 NO es IGUAL a Cadena2 o Variable2
Cadena1 < Cadena2	Verdadero si Cadena1 o Variable1 es MENOR a Cadena2 o Variable2
Cadena1 > Cadena2	Verdadero si Cadena1 o Variable1 es MAYOR a Cadena2 o Variable2
-n Variable1	Verdadero si Cadena1 o Variable1 NO ES NULO (tiene algún valor)
-z Variable1	Verdadero si Cadena1 o Variable1 ES NULO (esta vacía o no definida)

Operadores de comparación de valores numéricos	
Numero1 **-eq** Numero2	Verdadero si Numero1 o Variable1 es IGUAL a Numero2 o Variable2
Numero1 **-ne** Numero2	Verdadero si Numero1 o Variable1 NO es IGUAL a Numero2 o Variable2
Numero1 **-lt** Numero2	Verdadero si Numero1 o Variable1 es MENOR a Numero2 o Variable2
Numero1 **-gt** Numero2	Verdadero si Numero1 o Variable1 es MAYOR a Numero2 o Variable2
Numero1 **-le** Numero2	Verdadero si Numero1 o Variable1 es MENOR O IGUAL a Numero2 o Variable2
Numero1 **-ge** Numero2	Verdadero si Numero1 o Variable1 es MAYOR O IGUAL a Numero2 o Variable2

Los **operadores condicionales** que podemos utilizar son los siguientes:

Operaciones condicionales usando test	
-a fichero	Verdadero si fichero existe
-d fichero	Verdadero si fichero existe, y es un fichero de tipo directorio
-f fichero	Verdadero si fichero existe, y es un fichero regular.
-r fichero	Verdadero si fichero existe y se puede leer
-w fichero	Verdadero si fichero existe y se puede escribir
-x fichero	Verdadero si fichero existe y se puede ejecutar
fichero1 **-nt** fichero2	Verdadero si fichero1 es más nuevo que fichero2
fichero1 **-ot** fichero2	Verdadero si fichero1 es más viejo que fichero2

Los **operadores lógicos** que podemos utilizar son los siguientes:

- **!** (**not lógico**, **no** se cumple la condición especificada a continuación del operador).
- **&&** (**y lógico**, se deben cumplir **las dos** condiciones representadas a ambos lados de la expresión).
- || (**o lógico**, se debe cumplir **una** de las condiciones representadas a ambos lados de la expresión).

5. Estructuras condicionales

La cláusula IF

La principal estructura condicional de los scripts en shell es el **if**:

```
if [ expresión ]; then
   comandos a ejecutar si expresión es verdadera
fi
```

La expresión es cualquier **expresión lógica** que produzca un resultado o **verdadero o falso**. La expresión también puede ser **un comando que retorne un valor**. Si el valor retornado es **0** (**cierto**), los comandos se ejecutan; si el valor retornado es **distinto de 0** (**falso**), los comandos no se ejecutan.

```
if [ expresión ]
then
   comandos1
else
   comandos2
fi
```

Si se usa la forma opcional con **else,** si el valor retornado es distinto de 0 (**falso**) se ejecutan los **comandos2**.

```
if true; then echo Cierto; else echo Falso; fi
```
Siempre imprime **Cierto**; no entra nunca en **else**.

```
if false; then echo Cierto; else echo Falso; fi
```
Siempre imprime **Falso**, no entra nunca en **then**.

Construcciones más complejas pueden hacerse usando **elif** para anidar distintas alternativas.

```
if [ expresión1 ]; then
   realizar si expresión1 es verdadera
elif [ expresión2 ]; then
   realizar si expresión1 es falsa, pero expresión2 es verdadera
elif [ expresión3 ]; then
   realizar si exp1 y exp2 son falsas, pero expresión3 es verdadera
else
   realizar si todas las expresiones anteriores son falsas
fi
```

Vamos a ver dos ejemplos de este tipo de construcción **donde pasamos un parámetro** a los scripts:

```
# ciertofalso.sh: escribe cierto o falso según parámetro numérico
#
if [ $1 = "0" ]; then
   echo "Cierto: el parámetro es 0."
else
   echo "Falso: el parámetro no es 0."
fi

# trabajo.sh: dice si se trabaja según el día
# invocar con parámetros:domingo, festivo, u otro nombre cualquiera
#
if [ $1 = "domingo" ]; then
   echo "No se trabaja."
elif [ $1 = "festivo" ]; then
   echo "A veces se trabaja."
else
   echo "Se trabaja."
fi
```

Vamos a er un ejemplo de una **estructura if**, tanto con **valores de cadena** como con **valores numéricos**.

```
NOMBRE="Jose Antonio"
if [ $NOMBRE = "Jose Antonio" ]; then
   echo Bienvenido Jose Antonio
fi

NUMERO=12
if [ $NUMERO -eq 12 ]; then
   echo Efectivamente, el número es 12
fi
```

Si usamos **operadores de comparación numéricos con valores de cadena**, el sistema nos dará un **error**.

Hay que tener cuidado con los espacios en blanco. **Los corchetes llevan espacios en blanco tanto a izquierda como derecha**, que el **punto y coma**, sin embargo, **va pegado al corchete cerrado**, y que SIEMPRE hay que poner espacios en blanco las expresiones. Veamos algunos errores muy comunes que se suelen cometer:

`if [3 -eq 5 ]; then`	bash: [3: command not found. Hemos usado [3 en lugar de [3
`if [ "Jose" -eq "Jose ]; then`	Bash: [: jose: integer expression expected. Debíamos haber usado =
`if [ 3 = 4 ]; then`	Esto no nos devolverá error, y parece que funciona, pero en realidad no es así, hay que usar **-eq**, ya que está comparando cadenas y no números como debería.
`if [ 3 > 4 ]; then`	Esto devuelve verdadero. Es una prueba de que NO hay que usar operadores de cadena para comparar números.
`If [ jose=Antonio ]; then`	No hemos dejado espacios en la condición =. Esta expresión da como valor verdadero. Cuidado con este error, que nos puede volver locos en depuración.

Otro **error muy común** es el siguiente:

```
#!/bin/bash
PROFESOR="Juana"
if [ $PROFSOR = "Juana" ]; then
   echo "Hola Juana"
fi
```

Este programa nos devuelve por pantalla el siguiente error:

```
bash: [: unary operador expected
```

Que traducido resulta: he encontrado un [(corchete abierto) y luego un operador = sin nada en medio. Revisando el programa, vemos que **nos hemos equivocado en el nombre de la variable**, por lo que $PROFSOR no tiene valor y, por tanto, al no valer nada, el programa no entiende qué estamos comparando.

Si lo necesitamos, **podemos anidar expresiones usando tanto AND** (y - &&) **como OR** (o - ||).

```
if [ expresión1 ] && [ expresión2 ]; then
   se ejecuta si expresión1 Y expresión2 son verdaderas
fi

if [ expresión1 ] || [ expresión2 ]; then
   se ejecuta si expresión1 O expresión2 son verdaderas
fi
```

También podemos usar el **operador NOT** (!) para indicar una negación.

```
if ! [ expresión1 ]; then
   se ejecuta si expresión1 NO es verdadera
fi
```

Vamos a pedir al usuario un número de tres cifras y vamos a indicar si es capicúa o no (con la función **read**).

```
#!/bin/bash
# capicua.sh
# script que pide un número de tres cifras e indica si es capicúa o no.

clear
echo "Escribe un número entre 100 y 999 (y pulsa INTRO): "
read NUMERO
echo  # este echo sirve para introducir un salto de linea
```

```
if [ $NUMERO -lt 100 ] || [ $NUMERO -gt 999 ]; then
   echo "Lo siento, has introducido un número inválido."
else
   PRIMERA_CIFRA=$(echo $NUMERO | cut -c 1)
   TERCERA_CIFRA=$(echo $NUMERO | cut -c 3)
   if [ $PRIMERA_CIFRA = $TERCERA_CIFRA ]; then
      echo "El número $NUMERO es capicúa."
   else
      echo "El número $NUMERO no es capicúa."
   fi
fi
```

Podríamos haber hecho este último script mucho más corto, por ejemplo, usando una línea como:

```
if [ $(echo $NUMERO | cut -c 1) = $(echo $NUMERO | cut -c 3) ]; then
```

El resultado es idéntico, depende del gusto del programador realizar el script de una forma u otra. Cuando hacemos un script de varias líneas como el anterior, es posible que cometamos algún fallo.

Una opción que podemos usar para **depurar** los scripts y encontrar rápidamente los errores **es añadir un -x en la llamada al bash de la primera línea**. Esto hará que cada línea antes de ejecutarse sea mostrada por pantalla tal y como la está interpretando bash (como si no escribiéramos @ECHO OFF en MS-DOS).

```
#!/bin/bash –x
```

La cláusula CASE

La principal estructura condicional es if, pero tenemos otra a nuestra disposición, como es **case**. Este **comando implementa alternativas o "casos"** y **elige** entre **múltiples secuencias de comandos** la secuencia a ejecutar. La elección se realiza encontrando el **primer patrón** con el que aparea una cadena de caracteres.

```
case $CADENA in
patrón1)
   comandos1;;
patrón2)
    comandos2;;
...
*)
   comandosN;;
esac
```

El patrón *** se coloca al final y concuerda con cualquier cadena que no haya tenido relación con un patrón** hasta el momento. Es el caso por defecto (default). Permite ejecutar comandosN cuando ninguna de las opciones anteriores fue satisfecha. Un ejemplo con un parámetro sería el siguiente:

```
# diasemana.sh: nombres de los días de la semana
# invocar con número del 0 al 6 siendo el 0 domingo
#
case $1 in
   0) echo Domingo;;
   1) echo Lunes;;
   2) echo Martes;;
   3) echo Miércoles;;
   4) echo Jueves;;
   5) echo Viernes;;
```

```
   6) echo Sábado;;
   *) echo Debe indicar un número de 0 a 6;;
esac
```

Otro ejemplo nos devuelve la estación del año aproximada al crear el archivo estacion.sh que aparece a continuación e introduciendo el mes como parámetro:

```
# estacion.sh
# indica la estación del año aproximada según el mes
#
case $1 in
   diciembre|enero|febrero)
      echo Invierno;;
   marzo|abril|mayo)
      echo Primavera;;
   junio|julio|agosto)
      echo Verano;;
   septiembre|octubre|noviembre)
      echo Otoño;;
   *)
      echo estacion.sh: debe invocarse como
      echo estacion.sh <mes>
      echo con el nombre del mes en minúscula;;
esac
# fin estacion.sh
```

La **opción *)** al concordar con cualquier cadena, actúa como "**en otro caso**" y es útil para dar instrucciones sobre el uso del comando. En las opciones, **| actúa como OR** y es posible usar también los comodines * y ?. Sería además posible modificar el script anterior para que aceptara el mes en cualquier combinación de mayúsculas y minúsculas.

El comando TEST

El **comando test prueba condiciones y devuelve valor cierto** (0) **o falso** (distinto de 0) según el criterio de cierto y falso del shell; esto lo hace apto para usar en la condición de if. Tiene **dos formas equivalentes**, una de las cuales ya hemos visto anteriormente en la cláusula if, pero ambas pueden usarse indistintamente.

```
test expresión
[ expresión ]
```

Los **espacios en blanco** entre la expresión y los corchetes **son necesarios**. Devuelve cierto ante una cadena no vacía, y falso ante una cadena vacía:

```
if test "cadena" ; then echo Cierto ; else echo Falso; fi
if test "" ; then echo Cierto ; else echo Falso ; fi
```

Es **equivalente a escribirlo con los corchetes de forma abreviada**.

```
if [ cadena ] ; then echo Cierto ; else echo Falso; fi
if [ ] ; then echo Cierto ; else echo Falso ; fi
```

Las condiciones siguientes comparan cadenas de caracteres: = para igualdad y **!=** para desigualdad.

```
[ $DIR = $HOME ]
[ $LOGNAME = "usuario1" ]
```

```
[ $RESULTADO != "error" ]
```

La condición que aparece a continuación devuelve **falso si la variable no está definida**. Las comillas devuelven la cadena nula cuando VAR1 no está definida; **sin comillas no habría cadena y aparecería un error de sintaxis**.

```
[ "$VAR1" ]
```

El comando test se usa mucho para **determinar si un comando se completó con éxito**, en cuyo caso el **valor de retorno es 0**. Este valor de retorno se almacena en la variable $?.

```
# nvoarch.sh
# recibe un nombre y crea un archivo con ese nombre
# si ya existe el fichero, muestra un mensaje
#
if [ -f $1 ]; then
   echo El archivo $1 ya existe.
else
   touch $1
   echo Fue creado el archivo $1.
fi
echo
# fin nvoarch.sh
```

Otros operadores aceptados **por test son -a** (**AND**) y **-o** (**OR**).

```
# rwsi.sh
# indica si un archivo tiene permiso de lectura y escritura
#
ARCH=$1
if [ -r $ARCH -a -w $ARCH ]
then
   echo El archivo $ARCH se puede leer y escribir.
else
   echo Al archivo $ARCH le falta algún permiso.
fi
ls -l $ARCH
# fin rwsi.sh
```

Otra **alternativa** en vez de usar -a (AND) en la **misma expresión** es anidar **dos expresiones** de la forma:

```
if [ -r $ARCH ] && [ -w $ARCH ]
```

El comando EXIT

Este comando se utiliza en programación de shell para **terminar inmediatamente un script** y volver al shell original. **Seguido de un número**, **termina el script devolviendo el número** indicado, lo que puede usarse para **determinar condiciones de error**. Si el número devuelto es 0, indica la finalización exitosa de tareas. Escribir sólo exit también devuelve código de error 0.

```
#!/bin/bash
# exitar.sh: prueba valores de retorno de exit
#
clear
echo "Prueba de valores de retorno"
```

```
echo "Invocar con parámetros: "
echo "   bien, error1, error2, cualquier cosa o nada"
echo "Verificar posteriormente el valor de retorno con"
echo "la orden echo $?"
echo
VALOR=$1
case $VALOR in
   bien)
      echo " -> Terminación sin error."
      exit 0;;
   error1)
      echo " -> Terminación con error 1."
      exit 1;;
   error2)
      echo " -> Terminación con error 2."
      exit 2;;
   *)
      echo " -> Terminación con error 3."
      echo " (invocado con un parámetro no previsto o sin parámetro."
      exit 3;;
esac
```

El comando EXPR

Este comando recibe **números y operadores aritméticos como argumentos, efectúa los cálculos indicados y devuelve el resultado**. Cada **argumento debe estar separado por espacio en blanco**. Opera sólo con **números enteros** y realiza las operaciones suma (+), resta (-), multiplicación (*), división entera (/), resto de división entera (%).

Los símbolos * y / deben ser escapados escribiendo * y \/, al igual que los paréntesis, que deben escribirse \(y \). **El comando expr usa la convención de C** para cierto y falso: 0 es falso, y distinto de 0 es cierto. **No confundir con la convención que toma el shell en sus valores true y false, que es la contraria.**

```
expr 4 + 5                      # devuelve 9 ( 4 + 5 = 9 )
expr 3 \* 4 + 6 \/2             # devuelve 15 ( 3 * 4 + 6 /2 = 15 )
expr 3 \* \( 4 + 3 \) \/2       # devuelve 10 ( 3 * (4 + 3) / 2 = 10 )
```

El comando expr **también realiza operaciones lógicas** de comparación, aceptando los operadores =, !=, >, <, >= y <=. El operador != es "no igual"; el ! se usa para negar. Estos caracteres también requieren ser escapados.

```
echo `expr 6 \< 10`
echo `expr 6 \> 10`
echo `expr abc \< abd`
```

6. Estructuras iterativas o bucles

Las principales **estructuras iterativas** que podemos usar en shell scripts son **for**, **while**, **until**, y **select**.

La estructura FOR

La estructura **for** es la siguiente, donde **variable** va tomando los **valores del conjunto**:

```
for variable in conjunto; do
   estas líneas se repiten una vez por cada elemento del conjunto
done
```

Ese conjunto que aparece en la estructura del for, es normalmente un **conjunto de valores cualesquiera**, separados por espacios en blanco o retornos de línea. Así, si queremos mostrar los días de la semana por pantalla este script lo haría:

```
for dia in lunes martes miércoles jueves viernes sábado domingo; do
   echo "El día de la semana procesado es $dia"
done
for NUMERO in 1 2 3 4 ; do echo $NUMERO ; done

for NOMBRE in alfa beta gamma ; do echo $NOMBRE ; done

for ARCH in * ; do echo "Nombre de archivo: $ARCH" ; done
```

El **carácter *** es **expandido por el shell** colocando en su lugar **todos los nombres de archivo del directorio** actual. Pero lo interesante de ese **conjunto de valores** es que también **podemos obtenerlo mediante la ejecución de órdenes** que nos devuelvan una salida de ese tipo. Por ejemplo, si ejecutamos la orden:

```
find ~ -iname "*sh" 2> /dev/null
```

Obtendremos una **lista de todos los ficheros que terminen en sh** (normalmente es como acaban los scripts, si es que decidimos usar esa extensión) que están dentro de nuestro directorio de usuario (~) y enviando los posibles mensajes de error de la orden a /dev/null (para no verlos por pantalla). Por lo que la **salida de esta orden** es un **conjunto que podemos procesar con un for**:

```
#!/bin/bash
for programa in $( find ~ -iname "*sh" 2> /dev/null ); do
   echo "Uno de mis scripts: " $programa
done
```

Imaginemos que queremos copiar a un USB (montado en **/media/usbdisk** por ejemplo) todos los scripts que tengamos en nuestro directorio home, sin importar en que directorio estén, podríamos hacerlo con este script:

```
for programa in $( find ~ -iname "*sh" 2> /dev/null ); do
   echo "Copiando el script :" $programa
   cp $programa /media/usbdisk
done
```

Es posible mejorar el script anterior para que cree un directorio scripts en nuestro USB si no existe.

```
if ! [ -d /media/usbdisk/scripts ]; then
   mkdir /media/usbdisk/scripts
fi

for programa in $( find ~ -iname "*sh" 2> /dev/null ); do
   echo "Copiando el script: " $programa
   cp $programa /media/usbdisk/scripts
done
```

Las estructuras WHILE y UNTIL

Cuando no queremos recorrer un conjunto de valores, sino **repetir algo mientras se cumpla una condición**, **o hasta que se cumpla una condición**, podemos usar las estructuras **while** y **until**. La **estructura while** es la siguiente:

```
while [ expresión ]; do
   estas líneas se repiten MIENTRAS la expresión sea verdadera
done
```

Y la **estructura until** es la siguiente:

```
until [ expresión ]; do
   estas líneas se repiten HASTA que la expresión sea verdadera
done
```

Ambas **estructuras**, tanto while como until, **realizan exactamente lo mismo**, al efectuar la comprobación de la expresión en la primera línea, no como en otros lenguajes. Veamos un ejemplo de un script usando la **estructura while** (mientras).

```
#!/bin/bash
# doble.sh
# solicita números y muestra el doble de dichos números
#
echo "Dime un número (0 para salir): "
read NUMERO
while [ $NUMERO -ne 0 ]; do
   echo "El doble de $NUMERO es: " $(($NUMERO*2))
   echo "Dime un número (0 para salir): "
   read NUMERO
done
# fin doble.sh
```

Ahora veamos como queda el mismo script, usando la **estructura until** (hasta).

```
#!/bin/bash
# doble.sh
# solicita números y muestra el doble de dichos números
#
echo "Dime un número (0 para salir): "
read NUMERO

until [ $NUMERO -eq 0 ]; do
   echo "El doble de $NUMERO es :" $(($NUMERO*2))
   echo "Dime un número (0 para salir): "
   read NUMERO
done
```

Otro ejemplo, vamos a mostrar por pantalla los número del 1 al 20:

```
#!/bin/bash
#
NUMERO=1
until [ $NUMERO -gt 20 ]; do
   echo "Número vale :" $NUMERO
   let NUMERO=NUMERO+1
done
```

Existe una orden en Linux que es **seq**, que **nos permite mostrar una secuencia de números**, cuyo formato es **seq primero incremento último**. Esto **nos permite realizar este tipo de estructuras con for**.

```
#!/bin/bash
#
for i in $( seq 1 1 20 ); do
   echo "Número vale :" $i
done
```

7. Funciones

Usar **funciones en los scripts** es muy simple. Basta con usar la siguiente estructura al principio del script:

```
function nombre_función {
   líneas de la función
}
```

Estas líneas de la función no se ejecutarán al procesar el script, sino que **solo se ejecutarán cuando en el cuerpo del script usemos el nombre_funcion**. Ejemplo:

```
#!/bin/bash
function doble {
   echo "Voy a calcular el doble del valor del número."
   let NUMERO=NUMERO*2
}

$NUMERO=3
echo ' $NUMERO tiene un valor de: ' $NUMERO      # comillas simples
doble                                            # llamamos a la función
echo ' $NUMERO tiene un valor de: ' $NUMERO
```

Podría parecer que estamos pasando por let NUMERO=NUMERO*2 antes de asignarle el valor 3. No es así, ya que aunque veamos esas líneas físicamente anteriores a la asignación, **solo serán procesadas cuando en el script escribimos doble**.

Por defecto, todas las variables que usemos son **globales**, es decir, que las funciones y el script las comparten, pueden modificar sus valores. Sin embargo, en determinadas ocasiones nos puede interesar que **las variables sean locales a la función**, y si la función modifica su valor, este valor no afecte al script.

```
#!/bin/bash
function saludo {
   NOMBRE="Jose Antonio"
   echo "Hola señor $NOMBRE encantado de conocerle"
}
NOMBRE="Juana"
saludo
echo "En el script principal, mi nombre es $NOMBRE"
```

En este ejemplo, vemos cómo aparece en el script principal, mi nombre es Jose Antonio, ya que cuando en la función se modifica NOMBRE, se modifica en todo el ámbito del programa.

```
#!/bin/bash
function saludo {
   local NOMBRE="Jose Antonio"
   echo "Hola señor $NOMBRE encantado de conocerle"
}
```

```
NOMBRE="Juana"
saludo
echo "En el script principal, mi nombre es $NOMBRE"
```

Vemos como ahora, **al anteponer local** a la variable NOMBRE en la función, las modificaciones que se realicen **sólo afectan a la propia función**, por lo que en pantalla vemos como aparece en el script principal, mi nombre es Juana.

8. Paso de parámetros

Podemos **pasar parámetros tanto a scripts como a funciones**. Los parámetros en **bash** se indican como un **símbolo dólar** (**$**) seguido de un **número** o **carácter**. Los principales parámetros que se pueden usar son:

Parámetros	
$1	Devuelve el 1º parámetro pasado al script o función al ser llamado.
$2	Devuelve el 2º parámetro.
$3	Devuelve el 3º parámetro. Podemos usar hasta $9.
$*	Devuelve todos los parámetros separados por espacio.
$#	Devuelve el número de parámetros que se han pasado.
$0	Devuelve el parámetro 0, es decir, el nombre del script o de la función.
$?	Devuelve si el último comando se ejecutó o no correctamente.
$$	Devuelve el PID del shell script.

Podemos entenderlo mucho mejor con un script como el siguiente:

```
#!/bin/bash
# parametros.sh
# script para demostrar el funcionamiento de los parámetros.
#
echo "El primer parámetro que se ha pasado es " $1
echo "El tercer parámetro que se ha pasado es " $3
echo "El conjunto de todos los parámetros: " $*
echo "Me has pasado un total de " $# " parámetros"
echo "El parámetro 0 es: " $0
```

También podemos pasarle parámetros a las funciones, usando el mismo método y las mismas posibilidades que para los scripts completos. Así por ejemplo:

```
#!/bin/bash
#
function mayor_edad {
   if [ $1 -ge 18 ]; then
      echo Sí, es mayor de edad.
   else
      echo No, es menor de edad.
   fi
}
read -p "Dime la edad del que quiere entrar: " EDAD
echo "Voy a comprobar si puede entrar o no."
mayor_edad $EDAD
```

Existe un parámetro especial, el **$? que nos devuelve el valor del resultado de la ultima orden**. Es decir, después de ejecutar cualquier orden del sistema podemos comprobar el valor de $? que tendrá un 0 si todo ha ido bien, y otro valor cualquiera en caso de que haya fallado. Este parámetro puede sernos muy útil realizando scripts, ya que nos permite una forma rápida y cómoda de ver si todo está funcionando de forma correcta.

El siguiente programa muestra los parámetros que recibe al ser invocado:

```
# ecopars.sh
# muestra los parámetros recibidos
#
echo "Cantidad de parámetros: " $#
for VAR in $*
do
   echo $VAR
done
# fin ecopars.sh
```

La **variable $* contiene la lista de parámetros**, y **$# la cantidad**. Si no sabemos con antelación cuántos parámetros va a recibir un shell script, utilizamos el comando shift, que desplaza los parámetros que hemos recibido desde la línea de comandos hacia la izquierda, de manera que el primero se pierde.

Dentro del script, los parámetros recibidos pueden referenciarse con **$1, $2, $3,..., $9, siendo $0 el nombre del propio programa**. Debido a que se los reconoce por su ubicación, se llaman **parámetros posicionales**. El siguiente programa se invoca con tres parámetros y muestra sus valores:

```
# mostrar3.sh
# se invoca con 3 parámetros y los muestra
#
echo "Nombre del programa: " $0
echo "Parámetros recibidos:"
echo $1; echo $2; echo $3
echo
# fin mostrar3.sh
```

9. Interfaces de usuario

El comando READ

Podemos **pedir datos al usuario para que los script que se programen tengan mucho más sentido**. Se hace con la orden **read**:

```
read -p "texto de la pregunta" variable
```

La ejecución del script se parará, **mostrará por pantalla el texto de la pregunta, y dejará que el usuario escriba la respuesta**. Cuando se pulse INTRO, la respuesta dada se introducirá como valor de la variable.

También puede ser usado **sin el parámetro -p**, de la forma **read variable**. Podemos hacer que lea un determinado número de caracteres, sin obligar a que el usuario pulse intro, con el **parámetro -n número_de_caracteres**. El **parámetro -s silencia el eco** (no se ve por pantalla lo que el usuario escribe).

Un ejemplo un programa que nos permita indicar si un número introducido es par o impar.

```
# parimpar.sh
# script que nos pide un número e indica si es par o impar.
```

```
#
clear
read -p "Introduzca un número: " NUMERO
let RESTO=NUMERO%2
if [ $RESTO -eq 0 ]; then
   echo "El número $NUMERO es par"
else
   echo "El número $NUMERO es impar"
fi
```

El ejemplo siguiente obtiene datos del usuario, los repite en pantalla, solicita confirmación y emite un mensaje.

```
# yo.sh
# captura datos del usuario
#
clear
echo "Datos del usuario."
echo -n "Nombre y apellido: "; read NOMBRE
echo -n "Documento de identidad: "; read DNI
echo
echo "Ha ingresado los siguientes datos:"
echo "Nombre y apellido: " $NOMBRE
echo "Documento de identidad: " $DNI
echo -n "¿Es correcto?(S/N): "; read RESP
if [ "$RESP" = "s" -o $RESP = "S" ]
then
   echo "Fin de ingreso."
else
   echo "Debe ingresar sus datos nuevamente."
fi
```

La estructura SELECT

Otra de las **funciones que nos permiten interactuar con el usuario** es **select**. Ésta nos permite realizar una **iteración o bucle, pero presentando un menú por pantalla para que el usuario escoja una opción**. Su estructura general es la siguiente:

```
select VARIABLE in conjunto_opciones; do
   Aquí variable toma el valor de una de las opciones del conjunto
done
```

La **estructura** como vemos **es muy parecida a la del for**, pero presenta la principal diferencia en que por definición **se crea un bucle sin final**, no hay un valor inicial y un valor límite, el bucle se repetirá eternamente, lo que nos obliga a salirnos del mismo bien con **break** que nos permite **salirnos del bucle** o con **exit** que nos permite **salirnos del script entero**. Veamos un ejemplo:

```
#!/bin/bash
#
select OPCION in Chiste Refrán Proverbio Salir; do
   if [ $OPCION = "Chiste" ]; then
      echo "Van dos por la calle y se cae el de en medio."
   elif [ $OPCION = "Refrán" ]; then
      echo "Quien cría cuervos y tendrá muchos."
   elif [ $OPCION = "Proverbio" ]; then
```

```
      echo "Ten cerca a tus amigos y mucho mas cerca a tus enemigos."
   else
      # Ha escogido Salir
      break
   fi
done
echo "Hemos acabado el menú con select."
```

Veremos por pantalla que se nos muestra un menú parecido al siguiente:

```
1)Chiste
2)Refrán
3)Proverbio
4)Salir
#?
```

Automáticamente realiza un read (el #?) para pedir al usuario que escoja una opción mediante los números asignados a cada una. Al escoger la opción 4 en nuestro script realizamos un **break**, cuya misión es salir del bucle, con lo que por pantalla veremos el mensaje "Hemos acabado el menú con select" ya que break continua la ejecución del script justo debajo de done.

Si en lugar de break hubiéramos utilizado **exit**, no veríamos este mensaje en pantalla, ya que se **termina el script completo**, y en muchas ocasiones incluso cierra el terminal. Al igual que sucedía con el for, es perfectamente posible crear el conjunto mediante una instrucción. Así por ejemplo, la **instrucción ls** nos devuelve un conjunto formado por todos los ficheros del directorio actual:

```
select FICHERO in $( ls ); do
   echo Has seleccionado el fichero $FICHERO
   echo Ahora preguntamos si se desean borrar, copiar, visualizar, etc.
done
```

Si en el conjunto ponemos directamente el símbolo asterisco (*) veremos que tiene la misma función que un ls, devuelve el listado de ficheros del directorio actual.

Vamos a mostrar por pantalla **un menú con todos los mp3 que existan en el directorio home del usuario** actual y que el usuario escoja uno de ellos para reproducirlo. Para ello, usamos un reproductor de mp3 desde línea de comandos que se instala con **apt-get install mpg321** (también puede instalarse cualquier otro).

```
select MP3 in $( find . -iname "*mp3" ); do
   echo "Voy a reproducir el mp3: " $MP3
   mpg321 $MP3 &> /dev/null
done
```