TEMA 13: PROGRAMACIÓN DEL INTÉRPRETE DE COMANDOS.

TEMA 13: PROGRAMACIÓN DEL INTÉRPRETE DE COMANDOS.

13.1 INTRODUCCIÓN

Como hemos visto hasta ahora, el shell es un intérprete de órdenes, pero el shell no es solamente eso; los intérpretes de órdenes de Linux son auténticos lenguajes de programación. Como tales incorporan sentencias de control de flujo, sentencias de asignación, funciones, etc. Los programas de shell son interpretados línea a línea por el propio shell. A estos programas se les conoce con el nombre de shell scripts, guiónes o simplemente scripts.

13.2 ¿CÓMO CREAR UN SCRIPT?

La forma de escribir un programa de Shell es muy sencilla, consiste en crear un archivo de texto con un editor (por ejemplo vim, gedit, etc.). Este archivo contendrá las órdenes que el shell va a ir interpretando y ejecutando.

13.3 ¿CÓMO EJECUTAR UN SCRIPT?

Una vez tenemos el archivo de texto, será necesario darle al archivo permiso de ejecución, para ello emplearemos la orden chmod, ya estudiada en temas anteriores.

Una vez cambiados los derechos o permisos, ya podremos ejecutar nuestro programa simplemente invocándolo por su nombre. En caso del que el directorio en el que se halle nuestro script no se encuentre en el PATH, será necesario invocar al script anteponiendo la cadena ./ y a continuación (sin espacio en blanco) el nombre del script.

Nota aclaratoria: Linux tiene una serie de rutas donde buscar los ficheros que son ejecutables. Estas rutas aparecen descritas en la variable del shell PATH. Normalmente el scritpt se crea en el directorio de trabajo, o en subdirectorios creados dentro del directorio de trabajo, que no forman parte del PATH del sistema, y por tanto hay que decirle a linux donde está el programa o script que queremos ejecutar, para ello se antepone al nombre del guión su ubicación ./nombre_script.

13.4 ¿CÓMO INTRODUCIR COMENTARIOS EN NUESTROS SCRIPTS?

Es posible, e incluso recomendable, introducir comentarios en nuestros programas de shell, los comentarios no son más que textos aclaratorios de lo que hace una parte concreta de un guión, son ignorados por el intérprete de órdenes al leer el script, pero son muy útiles para que una persona, cuando edite el guión, pueda entender cómo funciona.

Como en el resto de los lenguajes de programación, hay un carácter o unos caracteres especiales que denotan comentario. En el caso de Linux este carácter

especial es la almohadilla `#', y denota que desde ahí hasta el final de la línea, el resto de los caracteres son un comentario del programador.

13.5 PRIMER PROGRAMA DE SHELL

Vamos a crear a continuación un sencillo Shell script para mostrar cuál va ser la técnica general para crear este tipo de programas.

En **primer lugar** lo que tenemos que hacer es decidir dónde vamos a crear nuestro programa, por ser algo ordenados creémonos un subdirectorio dentro de nuestro directorio de trabajo de nombre ejercicios, en mi caso:

```
$ mkdir /home/noelia/ejercicios
```

En **segundo lugar** decidimos el nombre que deseamos dar a nuestro programa e invocamos a un editor de texto. Vamos a probar con el editor vim y vamos a llamar a nuestro script, **script_1**

```
$ vim ejercicio_1
```

En **tercer lugar**, una vez que tenemos el editor abierto tecleemos el contenido de nuestro programa, que va a ser el siguiente

```
######################
# Shell script de prueba #
######################
who
date
```

En **cuarto lugar**, y una vez guardado el script y cerrado el editor, debemos cambiar sus atributos para que tenga derecho de ejecución:

```
$ chmod +x ejercicio_1
```

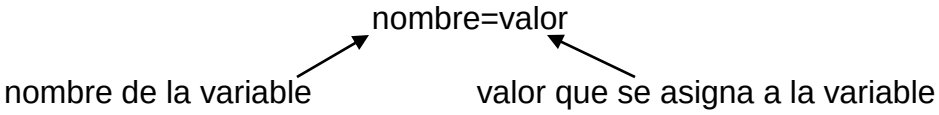
En **quinto y último lugar**, ya podemos ejecutar el script, invocándolo, y recordando que si el directorio en el que lo hemos creado no forma parte del PATH, la forma de invocarlos será:

```
$ ./ejercicio_1
```

13.6 LAS VARIABLES

Como en cualquier lenguaje de programación, en la programación en shell también existen las variables, que son nombres simbólicos para zonas de memoria que almacenan datos que nos interesan. Pero al contrario que en los lenguajes de alto nivel, las variables de los guiones *no tienen tipo*, o quizás sería más apropiado decir que tienen un tipo único y permanente: a todos los efectos se tratan como ristras de caracteres.

Las variables de los guiones no se declaran, nosotros crearemos nuestras variables asignándoles valores. Esto se hará de la siguiente manera:



Hay que tener cuidado al asignar valores a las variables, ya que no se debe dejar ningún espacio entre el signo de asignación (=) y la variable o el valor asignado.

Si deseamos consultar el valor que se ha asignado a una variable debemos utilizar el símbolo $, que deberá anteponerse al nombre de la variable. Entenderemos esto mejor con un ejemplo, creemos el **script_2**, con el siguiente contenido:

```
########################################
# APRENDIENDO A TRABAJAR CON VARIABLES #
########################################
x=25
y=hola
# veamos cual es la salida proporcionada
# por nuestro script si no utilizamos el
# símbolo $
echo x
echo y
# veamos que obtenemos si empleamos el símbolo $
echo $x
echo $y
```

Si ejecutáramos este script la salida proporcionada seria

```
x
y
25
hola
```

Nota recordatoria: Como ya sabemos Linux es case sensitive (distingue entre mayúsculas y minúsculas), por tanto recuerda que no es la misma variable x que X.

13.7 PASO DE PARÁMETROS A UN PROGRAMA DE SHELL

A menudo queremos que nuestros programas de shell reciban parámetros desde la línea de comandos para hacerlos más versátiles. Estos parámetros se pueden utilizar dentro del script como cualquier otra variable; es decir, para saber su valor utilizaremos el símbolo $.

Cuando se ejecuta nuestro programa en shell hay una serie de variables que siempre estarán disponibles, entre ellas las que nos permiten acceder a los distintos parámetros que pasamos a nuestro script en el momento de ser invocado.

$0 devuelve o contiene el nombre de nuestro script.

$n devuelve el parámetro pasado en n-ésimo lugar, con n de 1 a 9, ambos inclusive. Más adelante veremos la forma de acceder a más de 9 argumentos (comando shift).

Vamos a poner un ejemplo del shell script que visualiza los cuatro primeros parámetros que le pasemos. Al programa lo denominaremos **script_3**, y su contenido será el siguiente:

```
##########################################
# ESTE SHELL SCRIPT VISUALIZA LOS 4 PRIMEROS   #
# PARÁMETROS QUE LE PASAMOS DESDE LA LÍNEA #
# DE COMANDOS                                  #
##########################################
echo Parámetro 0 = $0
echo Parámetro 1 = $1
echo Parámetro 2 = $2
echo Parámetro 3 = $3
```

Si invocamos el script_3 de la siguiente manera **script_3 uno dos tres**, la salida proporcionada sería:

```
Parámetro 0 = ./script_3

Parámetro 1 = uno

Parámetro 2 = dos

Parámetro 3 = tres
```

Si invocamos el script_3 de la siguiente manera **script_3 uno dos**, la salida proporcionada sería:

```
Parámetro 0 = ./script_3
Parámetro 1 = uno
Parámetro 2 = dos
Parámetro 3 =
```

Si invocamos el script_3 de la siguiente manera **script_3 uno dos tres cuatro**, la salida proporcionada sería:

```
Parámetro 0 = ./script_3
Parámetro 1 = uno
Parámetro 2 = dos
Parámetro 3 = tres
```

13.8 ALGUNAS VARIABLES ESPECIALES DEL SHELL

Dentro de un programa de Shell existen variables con significados especiales, algunas de las cuales se citan a continuación:

#	esta variable guarda el nº de parámetros o argumentos con los que se ha invocado al script, excluyendo el nombre del programa.
*	guarda todos los parámetros separados por un espacio menos $0.
?	guarda el código de retorno de la última orden ejecutada (0 si no hay error y distinto de 0 si hay error).

Vamos a mostrar con un sencillo ejemplo el uso de estas variables. Al programa lo denominaremos **script_4**, y su contenido será el siguiente:

```
################################################
# ESTE SHELL SCRIPT VISUALIZA LAS VARIABLES #, * Y ?  #
################################################
# los caracteres #, *, ?, al ser susceptibles de ser interpretados
# por el shell de otra manera, se preceden del carácter backslash (\),
# de este modo pierden su significado especial
echo La variable \# vale: $#
echo La variable \* vale: $*
cp
echo La variable \? vale: $?
```

Si invocamos el script_4 de la siguiente manera **script_4 uno dos tres cuatro**, la salida proporcionada sería:

```
La variable # vale: 4
La variable * vale: uno dos tres cuatro
cp: faltan argumentos (ficheros)
Pruebe ´cp –help´ para más información
La variable ? vale: 1
```

Como podemos observar, al variable ? toma un valor distinto de cero, puesto que la orden cp se ha ejecutado con errores. Es importante que si dentro de un programa de shell se produce algún error tomemos decisiones al respecto. Como veremos más adelante, podremos evitar que el fallo de un comando aparezca por pantalla.

13.9 ENTRECOMILLADO Y CARACTERES ESPECIALES

(‘) Entrecomillado con comillas simples (es la comilla que comparte la tecla con el símbolo ?): Indica que lo que se pone entre comillas simples se trate de forma literal, es decir, no sea interpretado en una orden.

(“) Entrecomillado con comillas dobles (es la comilla que comparte la tecla con el número 2 y la @): Reconoce las variables y caracteres especiales que se encuentren en la cadena de caracteres encerrada entre comillas dobles y hace las sustituciones pertinentes por los valores correspondientes.

(`) Entrecomillado con comillas invertidas (es la comilla que comparte tecla con el corchete de apertura): Permite la ejecución de una orden para que la salida pueda ser asignada a una variable para ser tratada posteriormente.

(\) Backslash. Cambia el carácter especial de cualquier carácter que le siga, es decir, interpreta el carácter que hay a continuación como tal y no como parámetro.

Vamos a mostrar un ejemplo el uso de los distintos tipos de comillas.

Crea un programa al que denominaremos **script_5**, cuyo contenido será el siguiente:

```
#################################################################
# SHELL SCRIPT MUESTRA LAS DIFERENCIAS EN EL USO DE LAS DIFERENTES COMILLAS #
#################################################################
nombre=pedrito
saludo1=‘hola $nombre’          # utilizo comillas simples
saludo2=“hola $nombre”          # utilizo comillas dobles
saludo3=`hola $nombre`          # utilizo comillas invertidas
echo $saludo1
echo $saludo2
echo $saludo3
echo `whoami`
```

Pasamos a ejecutar el script_5 para observar su salida:

script_5, la salida proporcionada sería:

```
hola $nombre

hola pedrito

error porque se esperaba que lo de dentro fuera un comando y no lo es

noelia
```

Otros caracteres especiales con los que ya hemos trabajado pero que no está de más recordar son:

$	Utilizado para acceder al valor asignado a las variables.
?, *, [ ]	Los metacaracteres estudiados en el tema 7 empleados para localizar a archivos que se ajusten a determinados patrones.
espacio en blanco	Utilizado como delimitador de argumentos.
< , >, >>, 2>, 2>>	Empleados para la redirección de la E/S y de los errores (la redirección de la entrada, la salida y los errores, se explicaron en el tema 7).
|	Tubería o pipeline (símbolo que comparte tecla con el número 2 y el símbolo de exclamación), ya estudiado en el tema 7, para la interconexión de procesos.

13.10 CONSTRUCCIONES DEL LENGUAJE

Vamos a ver a continuación las construcciones del lenguaje típicas empleadas en los programas de Shell. No realizaré una descripción exhaustiva de todas y cada una de ellas, sino que me centraré en lo empleado más comúnmente.

shift

Sintaxis: shift n

Esta orden se utiliza para desplazar los argumentos. Este desplazamiento afecta también a las variables # y *, pero no al parámetro de posición 0 o nombre del programa.

Aunque se pueden dar más de nueve parámetros a un guión para el intérprete de órdenes, ya vimos que sólo se puede acceder de forma directa a los nueve primeros. La orden shift permite desplazar los parámetros de sitio, de tal forma que sean accesibles los que estén más allá del noveno, con el inconveniente de no poder acceder a los primeros.

La manera más simple de entender el funcionamiento de la misma es con algunos ejemplos:

Vamos a crear un script al que denominaremos **script_6**, y su contenido será el siguiente:

```
################################################
# PROGRAMA DE SHELL QUE MUESTRA EL USO DE SHIFT #
################################################
echo \$1 vale: $1
echo \$2 vale: $2
echo \$3 vale: $3
shift 2
echo Ahora \$1 vale: $1
echo Ahora \$2 vale: $2
echo Ahora \$3 vale: $3
```

En el ejemplo, al desplazar dos lugares tendremos que $5 pasa a ser $3, $4 pasa a ser $2 y $3 pasa a ser $1. Los argumentos $1 y $2, se pierden después del desplazamiento. Vamos e ejecutar el programa anterior:

$ ./script_6 uno dos tres cuatro cinco

```
$1 vale: uno
$2 vale: dos
$3 vale: tres
Ahora $1 vale: tres
Ahora $2 vale: cuatro
Ahora $3 vale: cinco
```

Vamos a crear un script al que denominaremos **script_7**, y su contenido será el siguiente:

```
#########################################################
# OTRO PROGRAMA DE SHELL QUE MUESTRA EL USO DE SHIFT #
#########################################################
# Otro ejemplo con shift
echo \$# vale: $#
echo \$* vale: $*
shift 2
echo Ahora \$# vale: $#
echo Ahora \$* vale: $*
```

Al ejecutar el anterior programa, se produce el siguiente resultado:

$ ./script_7 uno dos tres cuatro cinco

```
$# vale: 5
$* vale: uno dos tres cuatro cinco
Ahora $# vale: 3
Ahora $* vale: tres cuatro cinco
```

echo

Sintaxis: read [–n] [cadena de caracteres]

La orden echo, muestra por pantalla todo lo que pasemos como parámetro. La opción –n se empleaba para evitar el retorno de carro. Esta orden, ya estudiada en el tema 7, la volvemos a recordar aquí porque como ya comentamos será un comando muy utilizado en la construcción de scripts.

No se incluirá ningún ejemplo ya que en todos los ejemplos hechos hasta ahora hemos venido utilizando este comando y conocemos perfectamente a estas alturas su funcionamiento.

read

Sintaxis: read variable (s)

La orden read se usa para leer información escrita en el terminal de forma interactiva. Si hay más variables en la orden read que palabras escritas, las variables que sobran por la derecha se asignarán a NULL. Si se introducen más palabras que variables haya, todos los datos que sobran por la derecha se asignarán a la última variable de la lista. Pero para entender mejor el funcionamiento de esta orden los veremos con varios ejemplos:

Vamos a crear un script al que denominaremos **script_8**, cuyo contenido será el siguiente:

```
################################################
# PROGRAMA QUE ILUSTRA EL USO DE LA ORDEN READ #
################################################
echo –n Introduce una variable:
read var
echo La variable introducida es: $var
```

Al ejecutar el anterior programa, se produce el siguiente resultado:

$ ./script_8 hola

```
Introduce una variable: hola
La variable introducida es: hola
```

Vamos a crear un nuevo script al que denominaremos **script_9**, cuyo contenido será el siguiente:

```
#############3############################################
# OTRO PROGRAMA QUE ILUSTRA EL USO DE LA ORDEN READ #
#####################################################
echo –n Introduce las variables:
read var1 var2 var3
echo Las variables introducidas son:
echo var1 =  $var1
echo var2 =  $var2
echo var3 =  $var3
```

Vamos a ejecutar este script de tres formas distintas:

- Veamos una ejecución normal en la que leemos tres variables:

  $ ./script_9 34 hola 938

  Introduce las variables: 34 hola 938

  Las variables introducidas son:

  var1 = 34

  var2 = hola

  var3 = 938

- Veamos una ejecución en la que introducimos sólo dos parámetros:

  $ ./script_9 34 hola 938

  Introduce las variables: 34 hola

  Las variables introducidas son:

  var1 = 34

  var2 = hola

  var3 =

Como podemos observar, la variable var3 queda sin asignar, puesto que solo hemos introducido dos valores.

- Veamos una ejecución en la que introducimos cuatro variables:

  $ ./script_9 34 hola 938 saludos

  Introduce las variables: 34 hola 938 saludos

  Las variables introducidas son:

  var1 = 34

  var2 = hola

  var3 =938 saludos

En este caso a var3 se le asignan toda la ristra de variables a partir de la dos

expr

Sintaxis: expr arg1 op arg2 [op arg3 …..]

La orden expr evalúa sus argumentos y escribe el resultado en la salida estándar.

El uso más habitual de la orden expr es para efectuar operaciones aritméticas simples y, en menor medida, para manipular cadenas.

Antes de realizar ningún ejemplo de script dónde empleemos esta orden estudiaremos los tipos de operadores existentes, ya que el comando expr realiza operaciones.

13.10.1 OPERADORES

a) OPERADORES ARITMÉTICOS

Se utilizan para evaluar operaciones matemáticas. Las operaciones que podemos realizar son las siguientes: suma, resta, multiplicación, división entera y cálculo del resto de la división entera.

+	Suma **arg2** a **arg1**
-	Resta **arg2** a **arg1**
*****	Multiplica los argumentos
/	Divide **arg1** a **arg2** (división entera)
%	Resto de la división entera entre **arg1** y **arg2**

En el caso de utilizar varios operadores, las operaciones de suma y resta se evalúan en último lugar, a no ser que vayan entre paréntesis.

No hay que olvidar que los símbolos *****, **(** y **)** tienen un significado especial para el shell, por lo que deben ser precedidos por el símbolo *backslash*, ****, o bien encerrados **entre comillas simples**.

Vamos a crear un nuevo script al que denominaremos **script_10**, cuyo contenido será el siguiente:

```
################################################
# PROGRAMA DE SHELL QUE MULTIPLICA DOS VARIABLES #
# LEÍDAS DESDE EL TECLADO                        #
################################################
echo
echo Multiplicación de dos variables
echo ----------------------------------------
echo
echo –n Introduce la primera variable:
read arg1
echo –n Introduce la segunda variable:
read arg2
resultado=`expr $arg1 \* $arg2`
echo Resultado=$resultado
```

Vamos a ejecutar el script creado:

$ ./script_10

```
Multiplicación de dos variables

----------------------------------------

Introduce la primera variable:5

Introduce la segunda variable:3

Resultado=15
```

let

Aprovecho para explicar el comando let, que nos permite realizar operaciones aritméticas.

Sintaxis: let variable=arg1operadorarg2

Sin espacios entre argumentos y operador

El comando let puede ser sustituido por corchetes. El intérprete tratará como una expresión aritmética lo que esté incluido entre corchetes, la evaluar y devolverá su valor.

Vamos a crear un nuevo script al que denominaremos **script_11**, que realice lo mismo que el anterior pero utilizando ahora el comando let:

```
################################################
# PROGRAMA DE SHELL QUE MULTIPLICA DOS VARIABLES #
# LEÍDAS DESDE EL TECLADO                        #
################################################
echo
echo Multiplicación de dos variables
echo ----------------------------------------
echo
echo -n Introduce la primera variable:
read arg1
echo -n Introduce la segunda variable:
read arg2
let resultado=$arg1\*$arg2
echo Resultado=$resultado
```

b) OPERADORES RELACIONALES

Estos operadores se utilizan para comparar dos argumentos. Los argumentos pueden ser también palabras. Si el resultado de la comparación es cierto, el resultado es 1; si el resultado de la comparación es falso, el resultado es 0. Estos operadores se utilizan mucho para comparar operandos y tomar decisiones en función de los resultados de la comparación. Veamos los distintos tipos de operadores relacionales que existen:

=	¿Son los argumentos iguales?
!=	¿Son los argumentos distintos?
>	¿Es **arg1** mayor que **arg2**?
>=	¿Es **arg1** mayor o igual que **arg2**?
<	¿Es **arg1** menor que **arg2**?
<=	¿Es **arg1** menor o igual que **arg2**?

No olvides que los símbolos **>** y **<** tienen significado especial para el shell, por lo que deben ser **entrecomillados** o precedidos de ****

Vamos a crear un nuevo script al que denominaremos **script_12**, cuyo contenido será el siguiente:

```
#####################################################
# PROGRAMA DE SHELL QUE DETERMINA SI DOS VARIABLES #
# LEÍDAS DESDE EL TECLADO SON IGUALES O NO          #
#####################################################
echo
echo Comparamos dos variables para ver si son iguales
echo ------------------------------------------------------------------
echo
echo –n Introduce la primera variable:
read arg1
echo –n Introduce la segunda variable:
read arg2
resultado=`expr $arg1 = $arg2`
echo Resultado=$resultado
```

Vamos a ejecutar el script creado, el programa devolverá 1 si las dos variables introducidas son iguales y 0 si son distintas (lo ideal sería que nos mostrara un mensaje que nos dijera literalmente si las variables son iguales o no, no pasa nada, llegaremos a eso, pero antes debemos conocer las estructuras de programación a las que aún no hemos llegado)

$ ./script_12

```
Comparamos dos variables para ver si son iguales

------------------------------------------------------------------

Introduce la primera variable:12

Introduce la segunda variable:12

Resultado=1
```

Volvemos a ejecutar el script pero ahora voy a introducir dos variables que son argumentos distintos:

$ ./script_12

```
Comparamos dos variables para ver si son iguales

------------------------------------------------------------------

Introduce la primera variable:143

Introduce la segunda variable:85

Resultado=0
```

c) OPERADORES LÓGICOS

Los operadores lógicos son los siguientes:

&&	equivale a and
||	equivale a or
!	equivale a not

&&

Al igual que en otros lenguajes el segundo operando sólo se evalúa si el primero no determina el resultado de la condición. Según esto la operación:

```
if cd /tmp && cp 001.tmp $HOME ; then
....
fi
```

Ejecuta el comando cd /tmp, y si éste tiene éxito (el código de terminación es 0), ejecuta el segundo comando cp 001.tmp $HOME, pero si el primer comando falla ya no se ejecuta el segundo porque para que se cumpla la condición ambos comandos deberían de tener éxito (si un operando falla ya no tiene sentido evaluar el otro).

Muchas veces se utiliza el operador && para implementar un if. Para ello se aprovecha el hecho de que si el primer operando no se cumple, no se evalúa el segundo. Por ejemplo si queremos imprimir un mensaje cuando elementos sea 0 podemos hacer:

```
[ elementos = 0 ] && echo "No quedan elementos"
```

||

El operador || por contra ejecuta el primer comando, y sólo si éste falla se ejecuta el segundo. Por ejemplo:

```
if cp /tmp/001.tmp ~/d.tmp || cp /tmp/002.tmp ~/d.tmp then
....
Fi
```

Aquí si el primer comando tiene éxito ya no se ejecuta el segundo comando porque para que se cumpla la condición basta con que uno de los dos tenga éxito.

!

Por último el operador ! niega un código de terminación. Por ejemplo:

```
if ! cp /tmp/001.tmp ~/d.tmp; then
        Procesa el error
fi
```

13.10.2 EVALUACIONES. LA ORDEN test

La orden test se usa para evaluar expresiones y generar un valor de retorno; este valor no se escribe en la salida estándar, **el valor de retorno será 0 si la expresión es verdadera y será 1 si la expresión evaluada es falsa**.

La orden test puede evaluar tres tipos de elementos: archivos, cadenas y números

test (para evaluar archivos)

Sintaxis: test –operador fichero

El comando tiene una sintaxis alternativa, que es con mucho la más empleada, y que es la siguiente:

Sintaxis: [–operador fichero]

Operadores para el manejo de ficheros

Los operadores de ficheros que acepta la orden test son los siguientes:

Operadores del comando test para el manejo de ficheros	
-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
-s fichero	Verdadero si fichero existe y tienen un tamaño mayor que cero
fichero1 -nt fichero2	Verdadero si fichero1 es más nuevo que fichero2
fichero1 -ot fichero2	Verdadero si fichero1 es más viejo que fichero2

La sintaxis para los operadores binario es la que aparece en la tabla (dos últimos operadores de la tabla)

script_13

```
##############################
# COMANDO TEST CON ARCHIVOS #
##############################
test –f $1
echo $?
```

Ejecutamos el script_13, pasándole como argumento un archivo que no existe:

$ script_13 /home/noelia/ficherito

1 ⟵ El archivo no existe

Ejecutamos el script_13, pasándole como argumento un archivo que si existe:

$ script_13 /etc/passwd

0 ⟵ El archivo sí existe

test (para evaluación de cadenas)

Sintaxis: test cadena1 operador cadena2

Sintaxis alternativa:

Sintaxis: [cadena1 operador cadena2]

Operadores para el manejo de cadenas

Los operadores de ficheros que acepta la orden test son los siguientes:

Operadores del comando test para el manejo de cadenas	
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 (está vacía o no definida)

La sintaxis para los operadores unarios es la misma que para el manejo de ficheros (dos últimos operadores de la tabla)

script_14

```
##############################
# COMANDO TEST CON CADENAS #
##############################
a=palabra1
[ $a = palabra2 ]
echo $?
[ $a = palabra1 ]
echo $?
```

Ejecutamos el script_14 y observamos la salida proporcionada:

$ script_14

```
1
0
```

test (para evaluación numérica)

Sintaxis: test numero1 operador numero2

Sintaxis alternativa:

Sintaxis: [numero1 operador numero2]

En evaluaciones numéricas esta orden es sólo válida con números enteros.

Operadores para el manejo de números

Hemos explicado que no existen los tipos en las variables del intérprete, pero aún así es posible que las ristras “1” o “20” sean tratadas como números, así que son necesarios los operadores para el manejo de números. Los operadores de manejo de números son siempre binarios, y son los siguientes:

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 Var2
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	Ver. si Numero1 o Variable1 es MENOR O IGUAL a Numero2 o Var2
Numero1 -ge Numero2	Ver. si Numero1 o Variable1 es MAYOR O IGUAL a Numero2 o Var2

script_15

```
################################
# COMANDO TEST CON NÚMEROS #
################################
a=23
[ $a –lt 55 ]
echo $?
test $a –ne 23
echo $?
```

Ejecutamos el script_15 y observamos la salida proporcionada:

```
$ script_15
0
1
```

Puedes encontrar otro ejemplo del comando test para evaluar ficheros en el script_16

Puedes encontrar otro ejemplo del comando test para evaluar cadenas en el script_17

Puedes encontrar otro ejemplo del comando test para evaluar números en el script_18

Es un comando muy utilizado y además de en los ejemplo citados lo encontrarás en otros scripts de los desarrollados en el tema a partir de ahora.

Con esta comando (dentro de los []) también se pueden usar **operadores lógicos**, pero en este caso debemos de usar los operadores - a (para and) y -o (para or).

Son válidos en una expresión a la hora de evaluar tanto archivos como cadenas o números, son los siguientes:

Operadores lógicos para test	
-o	or
-a	and
!	not

Por ejemplo, la siguiente operación comprueba que $reintegro sea menor o igual a $saldo y que $reintegro sea menor o igual a $max_cajero:

```
if [ \( $reintegro -le $saldo \) -a \( $reintegro -le $max_cajero \) ]
then
....
fi
```

13.11 CONTROL DEL FLUJO DEL PROGRAMA

Ahora es cuando vamos a empezar a ser capaces de programar algo en shell. Las estructuras de control de flujo del programa que tenemos disponibles cuando programamos en shell son el if, case, while, for y until. Además, veremos algunas órdenes especiales y algunas construcciones un poco más raras, que sirven también para controlar el curso de los acontecimientos.

13.11.1 Estructuras condicionales

if

Sintáxis:

```
if condicion1
        then orden1
[elif condicion2
        Then orden2]
…
[else odenn]
fi
```

La estructura condicional if se emplea para tomar decisiones a partir de los códigos de retorno, normalmente devueltos por la orden **test**.

Su funcionamiento es el siguiente:

- se evalúa la condición1.
- si el valor de retorno de condición1 es verdadero (0), se ejecutará la orden1.
- si no es así y se cumple la condición2, se ejecutará la orden2
- ……
- en cualquier otro caso se evaluará la orden especificada en la cláusula del else, ordenn.

Hay que tener mucho cuidado con los espacios en blanco: Los corchetes llevan espacios en blanco tanto a izquierda como derecha y siempre hay que poner espacios en blanco en las expresiones.

Vamos a crear varios ejemplos para ilustrar el funcionamiento de esta estructura de programación:

script_16

```
#####################################
# PROGRAMA PARA PROBAR EL USO DE LA #
# ESTRUCTURA CONDICIONAL IF         #
#####################################
if test –f /home/noelia/prueba
then
        cat /home/noelia/prueba
else
        echo El archivo no existe
fi
```

Recordad que el uso de corchetes es similar al uso de test

script_17

```
#####################################
# PROGRAMA PARA PROBAR EL USO DE LA #
# ESTRUCTURA CONDICIONAL IF         #
#####################################
NOMBRE=”Pablo”
if [ $NOMBRE = “Pablo” ]
then
        echo Bienvenido Pablo
fi
```

script_18

```
#####################################
# PROGRAMA PARA PROBAR EL USO DE LA #
# ESTRUCTURA CONDICIONAL IF         #
#####################################
NUMERO=12
if [ $NUMERO -eq 12 ]
then
        echo Efectivamente, el número es 12
fi
```

script_19

Vamos a hacer un script que admita un argumento. Si el argumento dado coincide con el nombre de un directorio que ya existe el programa mostrará un mensaje diciendo que ya existe un directorio con ese nombre, si no existe creará un directorio con ese nombre. Además modificará los permisos del directorio creado para que solo el propietario tenga acceso al mismo.

```
#########################################################
# PROGRAMA QUE CREA, SI NO EXISTE, EL DIRECTORIO QUE    #
# INDIQUEMOS DESDE TECLADO. AL DIRECTORIO RECIÉN        #
# CREADO SÓLO TENDRÁ ACCESO EL PROPIETARIO DEL MISMO    #
#########################################################
if [ -d $1 ]
then
        echo El directorio ya existe
else
        mkdir $1
        chmod 700 $1
fi
```

script_20

```
#################################################################
# PROGRAMA SHELL QUE COMPRUEBA SI EXISTE UN ARCHIVO PASADO #
# COMO ARGUMENTO Y SI EXISTE MUESTRA DE QUE TIPO ES             #
#################################################################
if [ $# -eq 0 ]
then
        echo Debes introducir al menos un argumento
        exit 1
fi
if [ -f  $1 ]
then
        echo –n $1 es un archivo regular” “
        if [ -x $1 ]
        then
                echo ejecutable
        else
                echo no ejecutable
        fi
elif [ -d $1 ]
then
        echo $1 es un directorio
else
        echo $1 es una cosa rara o no existe
fi
```

Veamos algunos **errores muy comunes**, para que no los cometáis:

`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
`if [ 3 > 4 ]` `then`	Esto devuelve verdadero. Sirva como prueba para no usar operadores de cadena para comparar números.
`If [ jose=Antonio ]` `then`	No hemos dejado espacios en la condición =. Devuelve verdadero.

Otro error muy común es el siguiente:

```
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, me he encontrado un [(corchete abierto) y luego un operador (el =) sin nada en medio, y eso no funciona.

Revisando el programa anterior, vemos como nos hemos equivocado en el nombre de la variable, por lo cual $profsor no tiene ningún valor (es nula) y por lo tanto al no valer nada, el programa lo que ve es lo siguiente: if [= “Juana”].

Hagamos por ejemplo un programa que nos permita indicar si un número introducido es negativo o positivo:

script_21

```
#####################################################
# SCRIPT QUE NOS PIDE UN NÚMERO E INDICA SI ES MAYOR #
# O MENOR QUE CERO                                   #
#####################################################
clear
echo introduzca un numero
read NUMERO
if [ $NUMERO -gt 0 ]
then
        echo “El número $NUMERO es mayor que cero”
elif [ $NUMERO –lt 0 ]
then
        echo “El numero $NUMERO es menor que cero”
else
        echo “El numero es cero”
fi
```

Hagamos un script un poco más complicado… vamos a pedir al usuario un número de 3 cifras y vamos a indicar si es capicúa:

script_22

```
##################
# SCRIPT CAPICÚA  #
##################
clear
echo Dame un número entre 100 y 999
read NUMERO
if [ $NUMERO -lt 100 ]
then
        echo “Lo siento, has introducido un número menor de 100”
elif [ $NUMERO -gt 999 ]
then
        echo “Lo siento, has introducido un número mayor de 999”
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 ni es capicúa ni ná”
        fi
fi
```

case

Hemos visto la principal estructura condicional que es el if, pero tenemos alguna otra a nuestra disposición, como el case.

Sintáxis:

```
case $variable in
valor1)
        acción1;;
valor2)
        acción2;;
…
*)
        instrucción/es que se ejecutan si variable no toma ninguno de los valores anteriores;;
esac
```

Veámoslo mediante otro ejemplo:

script_23

```
###################################
# SCRIPT EJEMPLO ESTRUCTURA CASE #
###################################
# script que nos pide el año de nacimiento y nos dice cual
# es el animal que nos corresponde en el horóscopo chino.
clear
echo introduce el año en que naciste, usa 4 cifras
read anno
let resto=anno%12
case $resto in
        0) animal=”el MONO” ;;
        1) animal=”el GALLO” ;;
        2) animal=”el PERRO” ;;
        3) animal=”el CERDO” ;;
        4) animal=”la RATA” ;;
        5) animal=”el BUEY” ;;
        6) animal=”el TIGRE” ;;
        7) animal=”el CONEJO” ;;
        8) animal=”el DRAGON” ;;
        9) animal=”la SERPIENTE” ;;
        10) animal=”el CABALLO” ;;
        11) animal=”la CABRA” ;;
        *) echo Imposible, el resto de dividir entre 12 es un nº de 0 a 11
esac
echo “Si naciste en $anno según el horóscopo chino te corresponde $animal”
```

13.11.2 Estructuras repetitivas o bucles

for

Sintáxis:

```
for variable in lista
do
        orden (es)
done
```

donde,

variable puede ser cualquier variable de Shell

lista es una lista compuesta de cadenas separadas por espacios en blanco o por tabuladores.

y el funcionamiento de la estructura es el siguiente:

- se asigna a variable la primera cadena de la lista
- se ejecuta orden
- se asigna a la variable la segunda cadena de la lista
- se ejecuta orden
- …, y así se repite el proceso hasta que se hayan usado todas las cadenas de la lista
- después de que haya acabado el bucle, la ejecución continúa en la primera línea que sigue a la palabra clave done

script_24

```
###################################
# SCRIPT EJEMPLO ESTRUCTURA FOR  #
###################################
# script que muestra por pantalla los días de la semana
for dia in lunes martes miércoles jueves viernes sabado domingo
do
        echo el dia de la semana procesado es $dia
done
```

La “gracia” de ese conjunto de valores, es que podemos obtenerlo mediante la ejecución de ordenes 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 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). Bien, pues la salida de esta orden es un conjunto que podemos procesar con un for:

script_25

```
###################################
# SCRIPT EJEMPLO ESTRUCTURA FOR  #
###################################
# script que muestra por pantalla el nombre de cada uno de
# los scripts que tengo creados en mi carpeta de trabajo
for programa in `find ~ -iname “*sh” 2> /dev/null`
do
        echo  $programa
done
```

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

script_26

```
###################################
# SCRIPT EJEMPLO ESTRUCTURA FOR  #
###################################
for programa in  `find ~ -iname “*sh” 2> /dev/null`
do
        cp $programa /media/usbdisk
done
```

Ya que estamos, mejoremos el script anterior para que cree un directorio scripts en nuestro llaverito, pero únicamente si no existe.

script_27

```
#################################
# SCRIPT EJEMPLO ESTRUCTURA FOR  #
#################################
if ! [ -d /media/usbdisk/scripts ]
then
        mkdir /media/usbdisk/scripts
fi
for programa in `find ~ -iname "*sh" 2> /dev/null`
do
        cp $programa /media/usbdisk/scripts
done
```

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**:

while

Sintáxis:

```
while condicion
do
        orden (es)
done
```

El funcionamiento de la estructura es el siguiente:

- se evalúa la condición
- si el código devuelto por la condición es 0 (**verdadero**), se ejecutará la orden u órdenes y se vuelve a iterar
- si el código de retorno de la condición es distinto de 0 (**falso**), se saltará a la primera orden que haya después de la palabra reservada done

Veamos un ejemplo de un script usando la estructura while:

script_28

```
###################################
# SCRIPT EJEMPLO ESTRUCTURA WHILE  #
###################################
#script que pide números y muestra el doble de dichos números.
# el script continua ejecutándose mientras que el nº introducido no sea 0.
echo –n “Introduce un numero: “
read numero
while [ $numero -ne 0 ]
do
        resultado=`expr $numero \*2`
        echo “El doble de $numero es:” $resultado
        echo introduce otro numero o pulsa 0 para salir
        read numero
done
```

until

Sintáxis:

until condicion

do

orden (es)

done

El funcionamiento de la estructura es el siguiente:

- se evalúa la condición
- si el código devuelto por la condición es distinto de 0 (**falso**), se ejecutará la orden u órdenes y se vuelve a iterar
- Si el código de retorno de la condición es 0 (**verdadero**), se saltará a la primera orden que haya después de la palabra reservada done

Ambas estructuras, tanto while como until realmente realizan exactamente lo mismo, al efectuar la comprobación de la expresión en la primera línea, no como ocurre en otros lenguajes de programación.

Veamos cómo queda el script anterior, usando la estructura until:

script_29

```
###################################
# SCRIPT EJEMPLO ESTRUCTURA UNTIL  #
###################################
#!/bin/bash
#script que pide números y muestra el doble de dichos números.
# el script continua ejecutándose mientras que el nº introducido no sea 0.
echo –n “Introduce un numero: “
read numero
until [ $numero -eq 0 ]
do
        resultado=`expr $numero \*2`
        echo “El doble de $numero es:” $resultado
        echo introduce otro numero o pulsa 0 para salir
        read numero
done
```

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

script_30

```
###################################
# SCRIPT EJEMPLO ESTRUCTURA UNTIL  #
###################################
#!/bin/bash
NUMERO=1
until [ $NUMERO -gt 20 ]
do
        echo “Número vale :” $NUMERO
        let NUMERO=NUMERO+1
done
```

13.11.3 Instrucciones de interrupción

break

break [n]	Hace que cualquier bucle for, while o until termine y pase el control a la siguiente orden que se encuentre después de la palabra clave done.

continue

continue [n]	Detiene la iteración actual de un bucle for, while o until y empieza la ejecución de la siguiente iteración.

exit

exit [n]	Detiene la ejecución del programa del Shell y asigna n al código de retorno (normalmente o implica éxito, y distinto de 0, error).

Vamos a crear un script que al ejecutarse muestre el siguiente menú:

1.- Borrar la pantalla

2.- Listar el contenido del directorio actual

3.- Identificación del usuario

4.- Salir

A continuación nos pida que introducir una opción y dependiendo de la tecla pulsada:

Si introducimos un 1 se limpia la pantalla y vuelve a aparecer el menú

Si introducimos un 2 lista el contenido del directorio actual y a continuación vuelve a dibujar el menú.

Si tecleamos el 3 muestra un mensaje diciéndonos que usuario somos y a continuación el menú.

Al pulsar el 4 terminamos la ejecución.

Si pulsamos cualquier otra tecla mostrará el mensaje “Opción no válida” y finalizará la ejecución.

script_31

```
##########################################
# SCRIPT CON MENÚ Y FINALIZACIÓN DE BUCLE  #
##########################################
opción = 0
while [ $opcion –ne 4 ]
do
        echo 1.- Borra la pantalla
        echo 2.- Lista el contenido del directorio actual
        echo 3 .- Díme quién soy
        echo 4.- Salir
        echo –n “Introduce una opción: “
        read opcion
        case $opcion in
                1)  clear
                    ;;
                2)  ls –l
                    ;;
                3)  echo tu eres `whoami`
                    ;;
                4)
                    ;;
                *)  echo opción no válida
                    break
                    ;;
        esac
done
```

13.12 USO DE FUNCIONES EN PROGRAMAS DE SHELL

Dentro de los programas de Shell se puede hacer uso de funciones. En una función podemos agrupar un conjunto de órdenes que se ejecuten con cierta frecuencia. Las funciones hay que declararlas antes de utilizarlas.

script_32

```
#########################
# SCRIPT EJEMPLO FUNCION  #
#########################
# el funcionamiento de este script es el siguiente:
# Si no se le pasa ningún parámetro al programa
# se invoca a la función error que mostrará
# un mensaje en pantalla. En caso contrario
# se muestra en pantalla el número de parámetros
# pasados en la llamada al script
error ()
 {
        echo Error de sintaxis
        exit 2
}
if [ $# = 0 ]
then
        error
else
        echo Hay $# argumentos
fi
```

declaración de la función

llamada a la función

Las funciones además pueden colocarse en otro archivo aparte. De esta forma podemos diseñar una biblioteca de funciones y reutilizarlas en nuestros programas.

Como ejemplo vamos a diseñar una función que denominaremos espacio_ocupado, que obtenga la cantidad de memoria ocupada en una partición de disco pasada como argumento. Esta función la vamos a situar en un archivo aparte denominado funciones:

función declarada en el fichero funciones

funciones

```
espacio_ocupado()
{
        espacio=`df –k | grep /dev/$particion | tr –s ‘ ‘ | cut –d ‘ ‘ –f 3`
}
```

Para poder entender el funcionamiento de la función espacio_ocupado, ejecuta el comado df (estudiado en el tema8) y observa la salida que proporciona,

observarás que la 3ª columna indica el espacio libre en las particiones. El modificador -k hace que el resultado de df esté expresado en Kb.

Utilizamos la orden grep para localizar la línea que contiene la información sobre la partición que nos interesa.

La orden tr (estudiada en el tema7) con el modificador –s suprime los espacios en blanco duplicacados para que cut pueda usarlos como delimitadores de campos de forma correcta.

La orden cut nos permite cortar el campo nº 3 que es el que contiene la información que nos interesa.

Para hacer uso de esta función desde otro script es necesario indicar en qué archivo se encuentra. Para ello se coloca al principio de la línea un punto, un espacio y el nombre del archivo que contiene la función con su ruta si fuera necesario. Veámoslo con un ejemplo, construiremos un script que reciba como argumento el nombre de una partición y muestre por pantalla un mensaje informando del espacio ocupada en dicha partición.

script_33

```
. ./funciones
particion=$1
espacio_ocupado
echo La partición $1 tiene ocupados $espacio kb
```