8

OBJETIVOS

I. Conocer las variables locales y de entorno, su utilidad y saberlas utilizar y modificar.

II. Enumerar las estructuras de control y saber emplearlas correctamente.

III. Saber qué es un script e identificar sus aplicaciones.

IV. Construir scripts a partir de especificaciones.

LINUX. Programación del shell

Índice

Notas del autor

El propósito de esta documentación es proporcionar al alumno una guía que incluya los contenidos que considero necesarios para el desarrollo de esta unidad de trabajo.

Estos contenidos han sido extraídos de diferentes libros, webs y revistas citados todos en el apartado bibliográfico, así como de experiencias personales.

Si llega a tu mano esta documentación, no dudes en usarla, ya que ese es el objetivo que persigo al realizarla, sólo debes tener en cuenta una cosa, **aglutinar toda esta información lleva mucho tiempo por lo que si vas a usar este documento ten el detalle de citar al autor del mismo**.

Víctor Manuel Garrido Cases

Profesor Técnico de Formación Profesional
Sistemas y Aplicaciones Informáticas

Scripts en Linux

Un script es un **archivo de texto plano que contiene órdenes** o comandos para realizar una o varias tareas que haya que ejecutar repetida o frecuentemente.

En Linux, los shell scripts son ficheros de texto que contienen comandos que se irán interpretando por la shell

Para que un **shell script puede ejecutarse** hay que **añadirle al fichero de texto permisos de ejecución** y habrá que **tenerlo en un directorio que esté incluido en la variable PATH** como directorio donde el sistema va a ir buscando los comandos a ejecutar.

En Linux, cuando hagamos shell scripts hay que avisar al sistema para que sepa con qué shell debemos ejecutar el script. Para hacer esto, **como primera línea del fichero incluiremos el texto `#!/bin/bash`** si queremos que sea la shell basica la que ejecute nuestro fichero. También es frecuente añadirle la extensión `.sh` al fichero aunque no es obligatorio.

Variables

Una variable se puede definir como un lugar o **espacio de memoria** del sistema **donde** podemos **almacenar una información** de un tipo determinado. Las variables pueden ser de varios tipos dependiendo del área de memoria donde estén almacenadas

Variables locales

Las **variables locales solo serán accesibles por el programa** (shell script) que se está ejecutando en ese momento. Se suelen escribir en minúscula

Variables de entorno

Las **variables de entorno almacenan valores que pueden ser accedidos por todos los programas**. Se suelen escribir en mayúscula para diferenciarlas de las variables locales. Estas variables pueden ser usadas por cualquier usuario y desde cualquier terminal.

Para definir una variable basta con escribir desde la línea de comandos

variable=”valor”

Muy importante **no poner espacios** a ambos lados del símbolo =

```
a=1; b=2; ab=3; c=”1   3”

echo $ab #Salida: 3

echo ${a}b #Salida: 1b

echo “$c” #Salida: 1    3

echo $c #Salida: 1 3
```

Por defecto, cuando creamos una variable es local a menos que se indique que queremos que sea global en cuyo caso habrá que escribir lo siguiente

VARIABLE=”valor”

export VARIABLE

o hacerlo directamente **export VARIABLE =”valor”**

Algunas de las variables de entorno más usadas en Linux y que ya tenemos predefinidas son:

HOME. Ruta de nuestro directorio personal.

USER. Nombre de usuario.

SHELL. Ruta del intérprete de comandos (shell) que se está ejecutando.

PWD. Directorio de trabajo actual.

PATH. Rutas en las que el intérprete de comandos busca las órdenes a ejecutar cuando no especificamos dónde están.

Para conocer la **lista de variables de entorno del usuario actual** usaremos el comando `env`.

Para **acceder al valor que almacena una variable** lo tendremos que hacer **anteponiendo el carácter $ al nombre de la variable**. Por ejemplo, si

ejecutamos en el terminal `echo $HOME` nos mostrará la ruta de nuestro directorio personal.

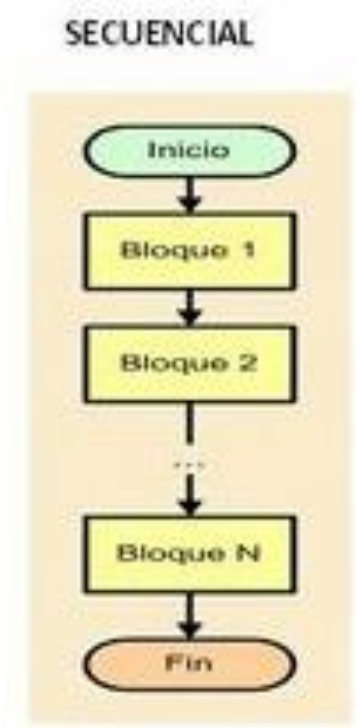
Para **borrar una variable** escribiremos `unset <variable>`

Estructuras de control

El teorema de Bohm y Jacopini afirma que **todo algoritmo**, **por muy complejo que sea**, **puede resolver utilizando única y exclusivamente 3 tipos de estructuras de control**: secuencial, alternativa y repetitiva.

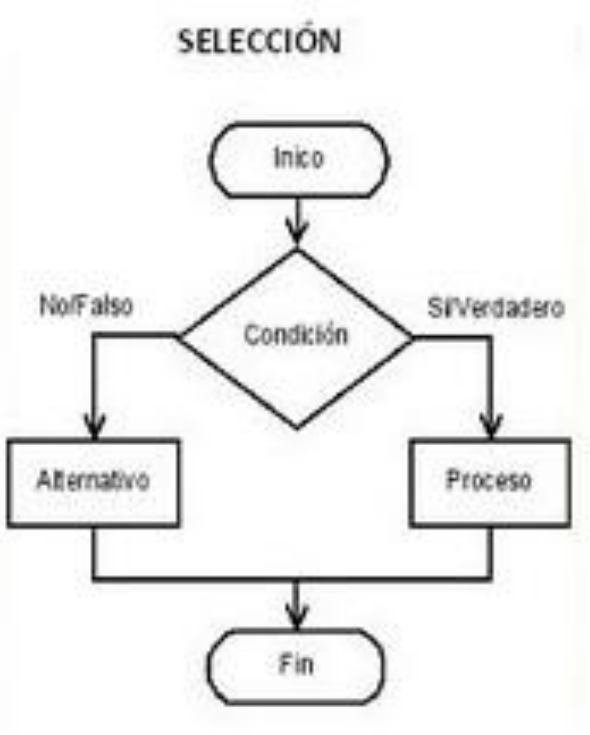
Estructura secuencial

Una estructura secuencial es aquella que ejecuta las acciones sucesivamente una a continuación de la otra sin posibilidad de omitir ninguna de ellas.



Estructura alternativa

Una estructura alternativa es aquella que dependiendo del valor de una determinada condición ejecuta una sentencia u otra.



La sintaxis con la que se expresa esta estructura es la siguiente:

```
if [ condición ]; then
     comando
else
     comando
fi
```

O si queremos que no se ejecute nada si no se cumple la condición bastaría con escribir

```
if [ condición ]; then
     comando
fi
```

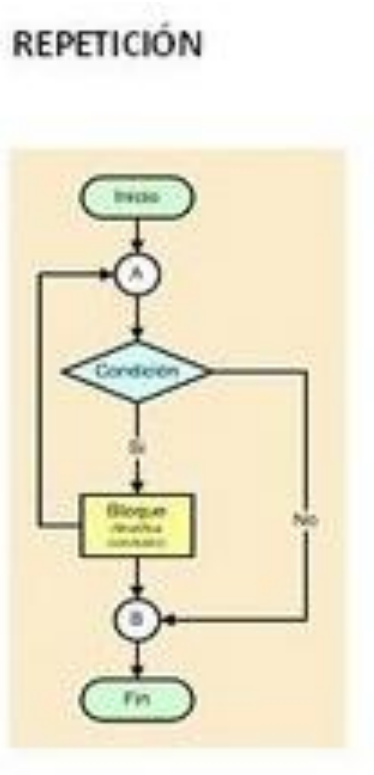
Si tenemos muchas condiciones que analizar podemos hacer uso de la siguiente sintaxis.

```
case valor in
patrón1)
 c1
 c2
 ..
 ;;
patrón2)
 ca
 cb
 ..
 ;;
*)
 cx
 cy
 ..
esac
```

```
case $v in
A|a) echo "Ha introducido A" ;;
B|b) echo "Ha introducido B" ;;
C|c) echo "Ha introducido C" ;;
*) echo "Opción incorrecta" ;;
esac
```

Estructura repetitiva

Las estructuras repetitivas o iterativas son aquellas en las que las acciones se ejecutan un número determinado de veces, siendo el número de repeticiones un valor predefinido o una determinada condición.



Vamos a ver dos estructuras repetitivas:

MIENTRAS

Mientras se cumpla una determinada condición se ejecutan las acciones. Las sintaxis con la que se expresa esta estructura es la siguiente:

```
while [ condición ]
do
     comando
done
```

DESDE HASTA

Se ejecutan las acciones un número determinado de veces. Las sintaxis con la que se expresa esta estructura es la siguiente:

```
for $variable in valores

do

     comando

done
```

REPETIR HASTA

Se repite una acción hasta que se cumpla una condición

```
until condición

do

 comando1

 comando2

 ...

done
```

Expresiones condicionales

Para evaluar el valor devuelto por la condición usaremos las siguientes expresiones dentro de los corchetes, es decir `[ condición ]` o con la orden `test` que veremos a continuación

Condición	Devuelve verdadero….
`-f fichero`	El fichero existe y es un fichero regular
`-d fichero`	El fichero existe y es un directorio
`-a fichero`	El directorio o fichero existe
`-r fichero`	El usuario tiene permiso de lectura sobre el fichero
`-w fichero`	El usuario tiene permiso de escritura sobre el fichero
`-x fichero`	El usuario tiene permiso de ejecución sobre el fichero
`-L fichero`	El fichero es un enlace simbólico
`-O fichero`	El usuario es el propietario del fichero
`-G fichero`	El usuario pertenece al grupo del fichero
`-s fichero`	El fichero existe y tiene tamaño > 0
`f1 -nt f2`	El fichero f1 es más nuevo que el fichero f2
`f1 -ot f2`	El fichero f1 es más viejo que el fichero f2
`f1 -ef f2`	El f1 y f2 son enlaces duros del mismo archivo

Si queremos **comparar valores numéricos** tendremos que usar los siguientes operadores entre una variable o un valor numérico, o entre dos variables que representen un valor numérico:

Comparador	Significa
`-eq`	Igual
`-ne`	Distinto
`-lt`	Menor que
`-gt`	Mayor que
`-le`	Menor o igual
`-ge`	Mayor o igual

Si queremos **comparar cadenas de caracteres** lo haremos de la siguiente forma

Comparador	Significa
`c1 = c2`	c1 es igual a c2 carácter a carácter. Es obligatorio los espacios entre =
`c1 != c2`	c1 no es igual a c2
`-n cadena`	La longitud de la cadena no es 0
`-z cadena`	La longitud de la cadena es 0
`c1 > c2`	c1 precede alfabéticamente a c2
`c2 > c1`	c1 antecede alfabéticamente a c2

Para las operaciones lógicas usaremos

Comparador	Significa
`&&`	Y lógico
`||`	O lógico
`!`	Negación

Orden test

Para evaluar una expresión, además de usar los corchetes, también se puede utilizar la orden test. Esta orden evalúa una expresión, si la expresión es verdad, devuelve un estado de salida 0, si la expresión no es verdad, devuelve un estado de salida no cero.

Esta orden permite evaluar cadenas, enteros y el estado de archivos del sistema operativo. Por ejemplo, el **número de argumentos** datos a un programa de Shell se mantiene en la **variable** de cadena #. Si un usuario

intenta ejecutar un programa sin argumentos, test puede ser utilizado para proporcionar un mensaje de diagnóstico.

```
if test $# -eq 0 then
     echo "Error. No hay parámetros"
fi
```

Argumentos en los programas shell

Cuando se ejecuta un programa Shell, las variables Shell se fijan automáticamente para que coincidan con los argumentos de la línea de órdenes dados al programa. Estas variables se conocen como parámetros posicionales y permiten al programa acceder y manipular información de la línea de órdenes. El parámetro **$#** es el número de argumentos que se pasan al guion. Los parámetros **$1, $2, $3** hacen referencia al primero, segundo, tercero y así sucesivamente argumentos en la línea de orden, **${10}** hace referencia al décimo parámetro. El parámetro **$0** es el nombre del programa de Shell. El parámetro **$*** hace referencia a todos los argumentos de la línea de órdenes.

Variables especiales para programas Shell.

El Shell dispone de varias variables predefinidas que son útiles en los guiones. Éstas proporcionan información sobre aspectos de su entorno que pueden ser importantes a los programas de Shell, tales como parámetros posicionales y procesos.

1. * Contiene los valores del conjunto actual de parámetros posicionales.

2. $ El ID del proceso del Shell actual.

3. # Contiene el número de parámetros posicionales. Esta variable se utiliza dentro de los programas para comprobar si existen argumentos de líneas de órdenes, y si es así, cuántos.

4. ? Es el calor devuelto por la última orden ejecutada. Cuándo se ejecuta una orden, devuelve un número al Shell. Este número indica si tuvo éxito (se ejecutó por completo) o falló (encontró un error). La convención es que 0 se devuelve en caso de una orden con éxito y un valor no cero cuando la orden falla.

5. ! Contiene el ID del proceso del último proceso subordinado. Resulta útil cuando un guion necesita eliminar un proceso subordinado que ha iniciado previamente.

Entrada y salida

El Shell proporciona dos órdenes internas para la escritura de la **salida** (`echo`) y para la lectura de la **entrada** (`read`).

La orden `echo` permite escribir la salida desde un programa de Shell. Esta orden escribe sus argumentos sobre la salida estándar. Puede utilizarse directamente cómo una orden regular o como un componente de un guion Shell. Esta orden se suele utilizar para examinar el valor de los parámetros del Shell y las variables, por ejemplo `echo $PATH`

La orden `read` permite insertar la entrada del usuario en su guion. Lee sólo una línea de la entrada del usuario y la asigna a una o más variables del Shell.

```
read v1  #Lee la variable v1

read -p msg v1  #Escribe un mensaje msg antes de solicitar la entrada

read -n 1 v1  #Acepta un máximo de 1 carácter
```

Funciones

Al igual que en otros muchos lenguajes de programación, shellscript nos permite crear funciones, vamos a ver cómo hacerlo.

```
function nomombreFuncion()

{
```

```
 comando1
 comando2
 ...
 return valor
}
---------------------------------
nom_fun a1 a2 a3
```

Los parámetros que reciba la función se tratan igual que en un script (`$1`, `$2`,..), con una excepción, `$0` que no cambia de valor, en su lugar tenemos `$FUNCNAME` que tiene el nombre de la función. El comando `return` termina la función y devuelve el valor calculado.

Si se quiere declarar una variable como local dentro de la función debe declararse como `local variable=valor`

Un ejemplo sencillo, una función para sumar dos números

```
a=4
b=4
function suma()
{
return $[$1+$2]
}
suma $a $b
echo "La suma es $?"
```

Usamos `$?` para **recuperar el valor de la última orden ejecutada**, que **en nuestro caso concreto es la función suma**, y más concretamente **el valor devuelto por** `return` que es **el valor de la suma.**

Expresiones aritméticas

Las expresiones aritméticas las podemos expresar de dos formas

`$[exp]`

`$((exp))`

Es decir `echo $[(5*4)+2]` es lo mismo que `echo $(((5*4)+2))`

Ejecución de scripts

Por convenio, los ficheros que contengan comandos de Shell deben acabar con la extensión .sh. Para ejecutar un Shellscript, lo primero que tenemos que hacer es darle permisos de ejecución para, al menos, el propietario

`chmod u+x`

A continuación, y si la ruta en la que nos encontramos no se encuentra en la variable PATH, tendremos que teclear para ejecutar el script la siguiente orden

`./<nombreScript.sh`

Si la ruta en la que nos encontramos se encuentra en la variable PATH, bastará con escribir el nombre del fichero directamente.

Algunos ejemplos de scripts

Ejemplo 1.

Ejemplo sencillo para ver diferentes formas de expresar un if

```
a=5
b=4
if [ $a -lt $b ]; then
      echo "a es menor que b"
elif [ $a -eq $b ]
then
      echo "a es igual a  b"
elif test $a -gt $b
then
      echo "a es mayor que b"
fi
```

Ejemplo 2.

Shell script que lee 2 números por teclado y muestra por pantalla qué número es mayor.

```
#!/bin/bash
echo "Introduce el primer número: "
read numero1
echo "Introduce el segundo número: "
read numero2
if [ $numero1 -lt $numero2 ]; then
      echo "El segundo número es mayor que el primero"
elif [ $numero1 -gt $numero2 ]; then
      echo "El primer número es mayor que el segundo"
else
      echo "Los dos números son iguales"
fi
```

Usando la opción `read -p "mensaje" variable` podemos con una sola sentencia mostrar un texto por pantalla antes de solicitar su valor correspondiente

Ejemplo 3.

Shell script que pide un número por teclado, y nos dice si el número es menor que 5 o es mayor que 5 o es igual a 5.

```
#!/bin/bash
read -p "Introduce el número: " numero
if [ $numero -lt 5 ]; then
      echo "El número es menor que 5"
elif [ $numero -gt 5 ]; then
      echo "El número es mayor que el 5"
elif [ $numero -eq 5 ]; then
      echo "El número es igual a 5"
fi
```

Ejemplo 4.

Shell script que pide el nombre de un fichero, si el fichero existe muestra la información larga del mismo (`ls -l`), en caso de que no exista muestra un error indicándolo.

```
#!/bin/bash
read -p "Introduce el nombre del fichero: " fichero
if [ -f $fichero ]; then
      ls -l $fichero
else
      echo "El fichero con nombre $fichero no existe"
fi
```

Ejemplo 5.

Shell script que recibe el nombre de un archivo como parámetro y nos dice si dicho archivo tiene permisos de lectura, permisos de escritura y permisos de ejecución.

```
#!/bin/bash
if [ -r $1 ]; then
      echo "El fichero $1 tiene permisos de lectura"
else echo "El fichero $1 no tiene permisos de lectura"
fi
if test -w $1 ; then
      echo "El fichero $1 tiene permisos de escritura"
else echo "El fichero $1 no tiene permisos de escritura"
fi
if [ -x $1 ]
then
      echo "El fichero $1 tiene permisos de ejecución"
else echo "El fichero $1 no tiene permisos de ejecución"
fi
```

Ejemplo 6.

Hacer un shell script que recibe varios nombres de archivos como parámetro y por cada uno de ellos comprueba si el archivo existe y si existe muestra su contenido (comando `cat`)

```
#!/bin/bash
for fichero in $*
do
      if [ -f $fichero ]; then
            echo "El fichero $fichero existe y su contenido es"
            cat $fichero
      else echo "El fichero $fichero no existe"
      fi
done
```

Ejemplo 7.

Programar que pide al usuario que introduzca dos números y realiza la suma de ambos.

```
#!/bin/bash
echo "Introduce el primer número:"
read numero1
echo "Introduce el segundo número:"
read numero2
suma=$[$numero1 + $numero2]
echo "La suma de $numero1 y $numero2 es $suma"
```

Ejemplo 8.

Hacer un shell script que reciba un número positivo en decimal como mucho de 4 cifras y lo convierta a binario.

```
#!/bin/bash
echo "Introduce el número:"
```

```
read numero
#Comprobamos que el dato introducido es un número positivo de 4 cifras como máximo
#Podríamos haber usado también read -n 4 numero y ya lo hubiéramos limitado por arriba
if [ $numero –ge 0 ] && [ $numero –le 9999 ]; then
      echo “Has introducido un número válido, vamos a pasarlo a binario”
      dividendo=$numero
      binario=””
      while [ $dividendo –gt 1 ]
      do
            dividendo2=$dividendo
            resto=$[$dividendo % 2]
            dividendo=$[$dividendo2 / 2]
            binario=”$resto””$binario”
      done
      binario=”$dividendo””$binario”
      echo “$numero en binario es $binario”
else
  echo “No has introducido un número válido”
fi
```

Bibliografía

Sistemas operativos monopuesto. Editorial Paraninfo. 1ª edición.

Sistemas operativos monopuesto. Editorial McGraw Hill. 1ª edición.