



# CFGs IMPLANTACIÓN DE SISTEMAS OPERATIVOS ADMINISTRACIÓN DE SISTEMAS INFORMÁTICOS EN RED



## Ud4.-Iniciación a los scripts

### Índice

- 1.- Introducción.
- 2.- Creación, ejecución y depuración de un script.
- 3.- Salida estándar.
- 4.- Entrada estándar.
- 5.- Estructuras condicionales.
- 6.- Estructuras repetitivas.
- 7.- Operaciones algebraicas.
- 8.- Paso por parámetros.
- 9.- Funciones.
- 10.- Retornar valores entre scripts.

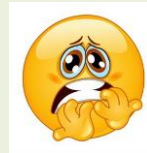


```
#!/bin/bash
```



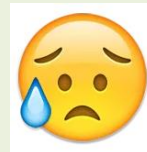
Nota:

1.- No te asustes.



2.- Estos contenidos se irán introduciendo poco a poco.

3.- Los ponemos aquí para tenerlo como referencia para el resto de unidades.





## 1.- Introducción.



Tira de Linux Hlspano

by danigm

¿Qué es un script?

Es un archivo de texto que incluye una serie de comandos y estructuras organizadas secuencialmente.

¿Qué utilidad tiene un script para un administrador?

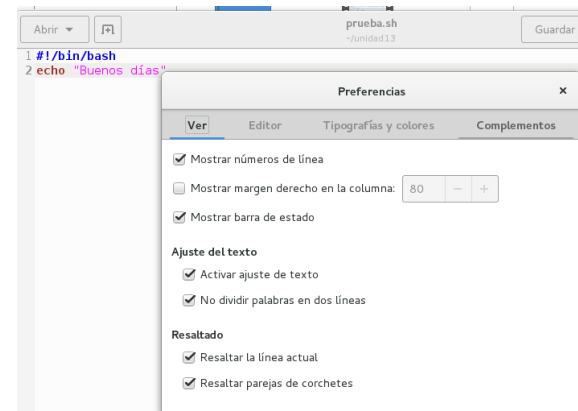
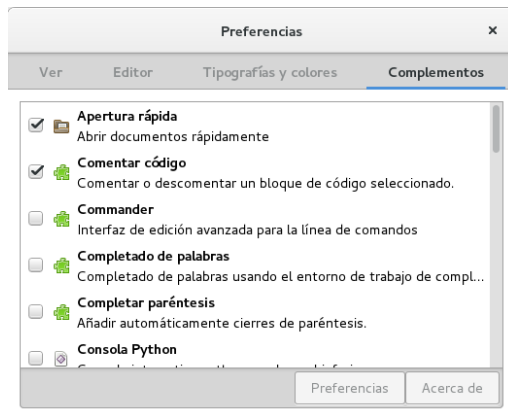
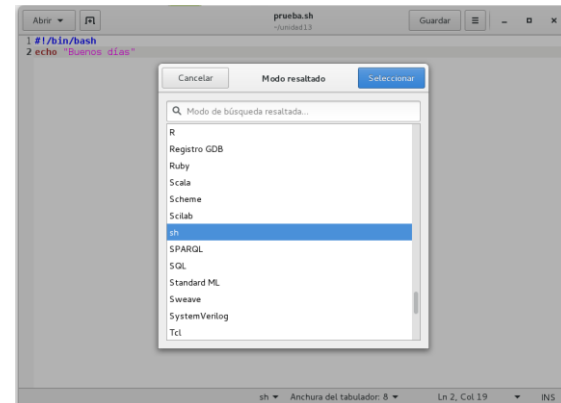
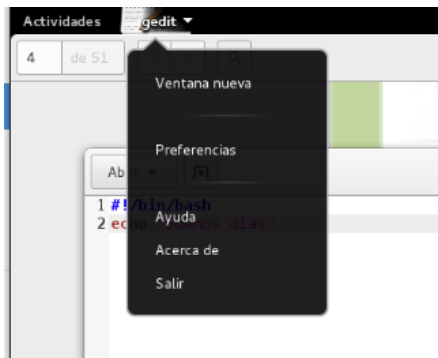
Le permitirá automatizar determinadas tareas.



## 2.- Creación, ejecución y depuración de un script.

### 2.1.- Creación

1.- Abrimos el editor de texto: gedit y lo configuramos en preferencias:

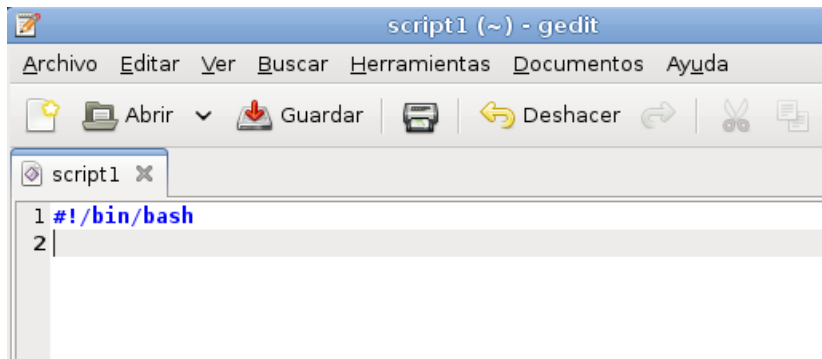




## 2.- Creación, ejecución y depuración de un script.

### 2.1.- Creación

2.- La primera línea de nuestro script es:



\*Shell → Es el programa que interpreta las órdenes introducidas por el usuario.

La más extendida es **/bin/bash** (Bash (Bourne-Again SHell)).

**#!** → Se conoce con el nombre de *Sha Bang*.

**#!/bin/bash**

Esto no es obligatorio, pero es conveniente porque:

.- Cuando ejecutemos el script, la shell detecta el carácter **#!** y lanza **/bin/bash** para que procese el script.

.- Nos asegura que siempre será ejecutado por bash, aunque nuestra shell actual sea otra (csh, sh...)



## 2.- Creación, ejecución y depuración de un script.

### 2.2.- Ejecución



1ª forma:

```
$ sh nombre_script.sh
```

2ª forma:

```
$ chmod a+x nombre_script.sh
```

```
$ ./nombre_script.sh
```



## 2.- Creación, ejecución y depuración de un script.

### 2.3.- Forma de trabajar.

The screenshot shows a Linux desktop with a blue background. On the left, a text editor window titled 'script0 (~) - gedit' is open. It contains two lines of code: `1 #!/bin/bash` and `2 echo "Bienvenidos futuros Administradores de Sistemas"`. The status bar at the bottom of the editor shows 'sh', 'Ancho de la tabulación: 8', 'Ln 2, Col 55', and 'INS'. On the right, a terminal window titled 'usuario@debian-pc200: ~' is open. It shows the following commands and output: `usuario@debian-pc200:~$ ls -l` resulting in a directory listing for 'Desktop' and 'script0'; `usuario@debian-pc200:~$ sh script0` resulting in the output 'Bienvenidos futuros Administradores de Sistemas'; and `usuario@debian-pc200:~$` with a cursor.

\* Cuando el script esté terminado, le daremos permisos de ejecución y ya lo lanzaremos como `./script`





## 2.- Creación, ejecución y depuración de un script.

### 2.3.- Forma de trabajar.

\* Buenas prácticas: Encabezado del script

```
1 #!/bin/bash
2 # Author: Manuel Domínguez
3 # Versión: 1.0
4 # Descripción: Programa bienvenida
5 Clear
6 echo "Bienvenidos, futuros administradores"
7
```



## 2.- Creación, ejecución y depuración de un script.

### 2.4 Depurar el script

`$sh -x nombre_script`

Depurar el código sirve para ver como se ejecuta paso por paso el script, que valores toman sus variables, si has cometido un fallo saber en que parte del código ha sido, etc.

```
script0 (~) - gedit
Archivo Editar Ver Buscar Herramientas Documentos Ayuda
Abrir Guardar Deshacer

script0 x
1 #!/bin/bash
2 echo "Bienvenidos futuros Administradores de Sistemas"

usuario@debian-pc200: ~
Archivo Editar Ver Terminal Ayuda
usuario@debian-pc200:~$ sh -x script0
+ echo Bienvenidos futuros Administradores de Sistemas
Bienvenidos futuros Administradores de Sistemas
usuario@debian-pc200:~$
```

+ → muestra la línea del script. La siguiente línea nos muestra el resultado.



## 3.- Salida estándar

### 3.1.- Sintaxis:

`echo [opciones] [cadena]`

Opciones:

“\n” → Si escribimos \n dentro de la cadena, se produce un salto de línea.

Ejemplo:

`echo “Bienvenidos: \n futuros administradores”`



### 3.- Salida estándar

#### 3.2.- Entrecorillados

' ' → Comillas simples.	“ ” → Comillas dobles.	`comando` → tildes invertidas.
Todo lo que vaya entre estas comillas será interpretado literalmente.	Lo mismo que el anterior, pero interpretando caracteres especiales como \$ y `.	Todo lo que vaya entre estas comillas será ejecutado.



### 3.- Salida estándar

#### 3.2.- Entrecorridos

#### Ejercicio:

Indica la salida esperada y a continuación compruébalo con el ordenador.

Comando	Salida en pantalla
\$x=10	
\$echo ' El valor de x=\$x'	
\$echo " El valor de x=\$x"	
\$echo Hoy es `date`	



## 4.- Entrada estándar.

### 4.1.- Leer variables

.- Dar valor a una variable: `nombre_variable=valor`

Ejemplos:

```
var1="Hola"
```

```
var2='Adios'
```

```
var3=523
```

```
var4=$var1
```



## 4.- Entrada estándar.

### 4.1.- Leer variables

.- Leer por teclado : `read [opciones] nombre_variable1 nombre_variable2`

Opciones:

`read -p "texto" variable` → Muestra un texto.

Ejemplo:

`read -p "Introduce tu nombre:" nombre`



## 4.- Entrada estándar.

### 4.2.- Utilizar y mostrar variables

Cuando queremos **utilizar el valor de una variable** en el código, nos referiremos a éste como:

```
$nombre_variable
```

Cuando queremos **mostrar el valor de una variable**, utilizaremos:

```
echo $nombre_variable
```





## 4.- Entrada estándar.

### 4.3.- Asignando resultados de comandos a variables

Variable=`comando`

Variable=\$(comando)

#### Ejemplos:

fecha1=`date`

fecha2=\$(date)

echo "\$fecha1 \$fecha2"



## 4.- Entrada estándar.

### 4.4.- Variables del entorno

**Comando export:** cuando le pasamos el nombre de una variable, ésta es trasladada desde el área local de datos al entorno.

Ej: export TERM (A partir de este momento la variable TERM es conocida por cualquier proceso iniciado desde el intérprete de comandos.

Ej: export x=8

**Comando set:** Informa de los nombres y valores de todas las variables del Shell en el área local de datos y en el entorno.

**Comando env:** Informa de los valores y nombres de todas las variables de entorno.

**Comando unset:** borra el valor de las variables del área local de datos (sin argumentos).

Ej: unset pepe



## 4.- Entrada estándar.

### 4.4.- Variables del entorno

#### uno.sh

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

¿Sería conocida por el script dos.sh?

```
x=10
```

export x → A partir de estos momentos cualquier subproceso (subscript)

Sh dos.sh conoce la variable.

#### dos.sh

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

```
echo "El valor de x es $x"
```



## 4.- Entrada estándar.

### 4.5.- Ejercicios

#### Bienvenida.sh

Realizar un script que pregunte tu nombre y muestre en pantalla un texto dándote la bienvenida:

#### Contraseña.sh

Realizar un script que nos pregunta una contraseña y a continuación se la mostramos en pantalla. La contraseña introducida, no debe aparecer en pantalla.

Investiga como anular el eco con el comando **stty**



## 4.- Entrada estándar.

### 4.5.- Ejercicios

#### Contraseña.sh

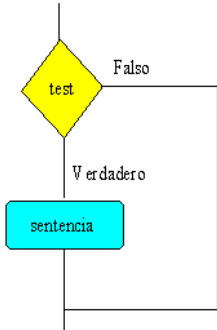
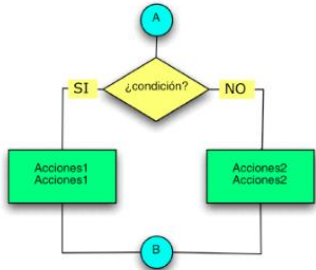
```
1 #!/bin/bash
2 # Author: Manuel Domínguez
3 # Versión: 1.0
4 # Descripción: Programa bienvenida
5 clear
6 stty -echo #Anula el eco del teclado
7 read -p "Introduce la contraseña:" contra
8 echo " \n La contraseña que has introducido es: $contra"
9 stty echo
```



## 5.- Estructuras condicionales.

### 5.1.- If

La finalidad de las expresiones condicionales es la de ramificar la ejecución de sentencias del script en función a una variable u otro elemento.

<p><b>Alternativa simple</b></p> <pre> if [ condición ] then     bloque fi </pre> 	<p><b>Alternativa doble</b></p> <pre> if [ condición ] then     bloque 1 else     bloque 2 fi </pre> <p>Si la condición se cumple entraría por el <i>then</i>, en caso de que no, por el <i>else</i>.</p> 
<p><b>Alternativa múltiple</b></p> <pre> if [ condición 1 ] then     bloque1 elif [condición 2 ] then     bloque2 else     bloque fi </pre>	<p>Ahora lo que hace es evaluar la condición, si es verdadera entra por el <i>then</i>, pero si no y se da el caso de otra condición entraría por el <i>elif</i>, los <i>elif</i> no se cierran, solamente el <i>fi</i> final corresponde a la apertura del <i>if</i>.</p>



## 5.- Estructuras condicionales.

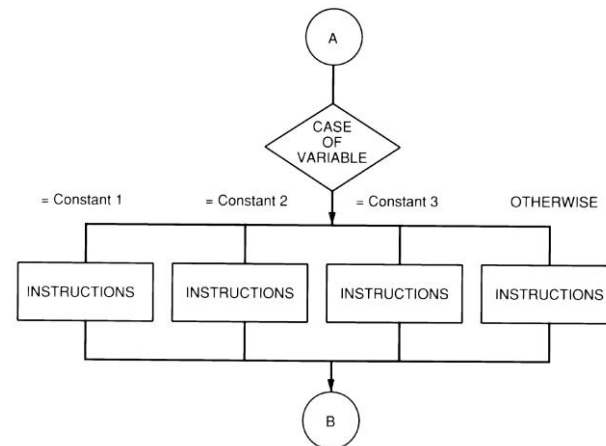
### 5.2.- case

Según el valor de la expresión se hará un caso u otro. Sería equivalente a if anidados, pero la estructura es más ordenada.

```

case $variable in
valor1)
    bloque1
    ;;
Valor2)
    bloque2
    ;;
*)
    bloque por defecto
    ;;
esac
  
```

Nota:  
Los paréntesis son obligatorios.





## 5.- Estructuras condicionales.

### 5.3.- Condiciones

[ Condición ]

NOTAS:

1.- Hay que dejar un espacio después y antes de los corchetes.

2.- Hay que dejar espacios entre el operando y operadores.

“A” = “B”

3.- Para anular una expresión: [ ! **Expresión** ] → Espacio después del !.

4.- Para anular la expresión global: ! [ **Expresión** ] → Espacios después del !





## 5.- Estructuras condicionales

### 5.4.- Operadores

<i>Operadores de comparación de cadenas alfanuméricas</i>	
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)



## 5.- Estructuras condicionales

### 5.4.- Operadores

<i>Operadores de comparación de valores numéricos.</i>	
Numero1 <b>-eq</b> Numero2	Verdadero si Numero1 o Variable1 es IGUAL a Numero2 o Variable2
Numero1 <b>-ne</b> Numero2	Verdadero si Numero1 o Variable1 NO es IGUAL a Numero2 o Variable2
Numero1 <b>-lt</b> Numero2	Verdadero si Numero1 o Variable1 es MENOR a Numero2 o Variable2
Numero1 <b>-gt</b> Numero2	Verdadero si Numero1 o Variable1 es MAYOR a Numero2 o Variable2
Numero1 <b>-le</b> Numero2	Ver. si Numero1 o Variable1 es MENOR O IGUAL a Numero2 o Variable2
Numero1 <b>-ge</b> Numero2	Ver. si Numero1 o Variable1 es MAYOR O IGUAL a Numero2 o Variable2



## 5.- Estructuras condicionales.

### 5.4.- Operadores

#### Ejercicio:

Evalúa las siguientes expresiones, e indica si hay errores de sintaxis.

1.- [4 -eq 4]

2.- [ "José" -eq "José" ]

3.- [ 3 = 4 ]

4.- [ "José"="José" ]



## 5.- Estructuras condicionales.

### 5.4- Operadores

#### Condiciones sobre ficheros

<b>-a fichero</b>	Verdadero si fichero existe
<b>-d fichero</b>	Verdadero si fichero existe, y es un fichero de tipo directorio
<b>-f fichero</b>	Verdadero si fichero existe, y es un fichero regular.
<b>-r fichero</b>	Verdadero si fichero existe y se puede leer
<b>-w fichero</b>	Verdadero si fichero existe y se puede escribir
<b>-x fichero</b>	Verdadero si fichero existe y se puede ejecutar
<b>fichero1 -nt fichero2</b>	Verdadero si fichero1 es más nuevo que fichero2
<b>fichero1 -ot fichero2</b>	Verdadero si fichero1 es más viejo que fichero2



## 5.- Estructuras condicionales.

### 5.4.- Operadores

#### Concatenar expresiones

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
    if [ $numero -lt 100 ] || [ $numero -gt 999 ]
```

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

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



## 5.- Estructuras condicionales.

### Ejercicios

#### Calendario.sh

Realizar un script que muestre el siguiente menú.

Menú

- 
- 1.- Fecha actual.
  - 2.- Calendario.
  - 3.- Salir.

```

1 #!/bin/bash
2 # Calendario
3 clear
4 echo "
5 echo "Menú"
6 echo "
7 echo "1.- Fecha actual"
8 echo "2.- Calendario"
9 echo "3.- Salir"
10 echo
11 read -p "Introduce una opción:" opcion
12 case $opcion in
13 1)
14     echo "La fecha actual es `date`";;
15 2)
16     echo "El calendario es :\n `cal`";;
17 3)
18     echo "Hasta luego"
19     exit 0;;
20 *)
21     echo "No has pulsado ninguna opción correcta";;
22 esac
23 |
  
```

#### Nota:

exit → Lo utilizaremos para salir del script.

exit n → sale del script con valor n



## 6.- Estructuras repetitivas

### 6.1.- Bucles for

<b>for variable in conjunto</b> do bloque [break   continue] done	La variable irá tomando cada uno de los valores del conjunto. Para cada valor del conjunto se ejecutará el bloque.  Variable sin \$
---	--

break → Rompe el bucle.

continue → Salta al siguiente valor de la lista.

**Nota:** Los valores del conjunto se pueden dar de diversas formas:

.- Separados por espacios.

.- Líneas de un fichero.



## 6.- Estructuras repetitivas

### 6.1.- Bucles for

#### Amigos1.sh

```
#!/bin/bash
clear
for i in "Bob Esponja" Patricio Calamardo
do
    echo "Mi mejor amigo es $i"
done
```

Nota: El espacio o el salto de línea lo interpreta como un nuevo valor de la variable.

#### Amigos2.sh

```
#!/bin/bash
clear
for i in `cat amigos.txt`
do
    echo "Mi mejor amigo es $i"
done
```

```
usuario@servidor2xx:~$ cat amigos.txt
Bob_Esponja
Patricio
Calamardo
```

→ For i in \$(cat amigos.txt)





## 6.- Estructuras repetitivas

### 6.1.- Bucles for → Leer líneas de un fichero (Mejor While read)

La **variable IFS** (Internal Field Separator) es una variable interna de bash que especificará el carácter separador.

Si ponemos IFS=""

Podemos procesar cada línea de un fichero sin mayor problema utilizando el for.

#### Amigos3.sh

```
#!/bin/bash
#Utilizamos la variable IFS para indicar el separador.
OLDIFS=$IFS
#Definimos un nuevo IFS
IFS=""
for i in $(cat amigos.txt)
do
    echo $i
done
IFS=$OLDIFS # Lo dejamos como estaba
```

```
usuario@debian-pc100:~/Documentos/scripts$ cat amigos.txt
```

```
Bob esponja
Patricio
Calamardo
```



## 6.- Estructuras repetitivas

### 6.1.- Bucles for: variedad

```
#!/bin/bash
#VARIEDADES DEL FOR
echo "Variedad del for i in 1 2 3 4 5"
for i in 1 2 3 4 5
do
    echo "i=$i"
done

echo "Variedad del for con seq"
for i in $( seq 1 2 10 ) # Primer valor, incremento y último valor. Cuidado con los espacios
do
    echo "i=$i"
done
```

Variedad del for i in 1 2 3 4 5  
i=1  
i=2  
i=3  
i=4  
i=5  
Variedad del for con seq  
i=1  
i=3  
i=5  
i=7  
i=9  
usuario@debian-pc200:~\$

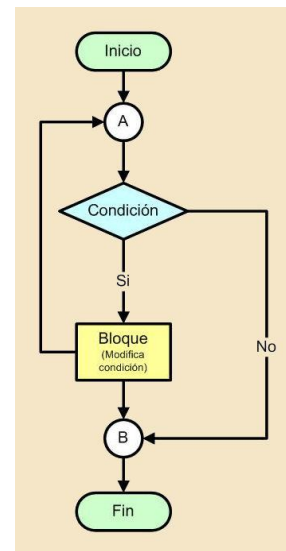


## 6.- Estructuras repetitivas

### 6.2.- Bucles while

```
while [ condición ]  
do  
    bloque  
done
```

Mientras que la condición sea verdadera, se repetirá el bloque.





## 6.- Estructuras repetitivas

### 6.2.- Bucles while

#### Mayor5.sh

```
#!/bin/bash
clear
read -p "Introduce un numero >,5:" num
while [ $num -lt 5 ]
do
    read -p "Introduce un numero >,5:" num
done
```

#### BucleInfinito.sh

```
#!/bin/bash
clear
while true
do
    read -p "Introduce la palabra fin:" fin
    if [ "$fin" = "fin" ]
    then
        exit 2
    fi
done
```



## 6.- Estructuras repetitivas

### 6.2.- Bucles while → Leer líneas de un fichero

Con el *while* se puede leer línea por línea un fichero, para ello lo único que hay que hacer es poner un redireccionamiento de entrada en el *done* con la ruta del fichero que queremos leer ( esto es muy útil, ya que puedes crear usuarios a partir de un fichero, etc. Tan solo tenéis que poner `done < /ruta_fichero` ).

#### LeerFichero.sh

```
#!/bin/bash
while read linea
do
    echo "Hola $linea"
done <amigos.txt
```

```
usuario@debian-pc200:~$ cat amigos.txt
Dora
Botas
El mapa
usuario@debian-pc200:~$ █
```

Nota: Dentro del do-done si ponemos un read, lo lee del fichero.



## 6.- Estructuras repetitivas

### 6.3.- Bucles until

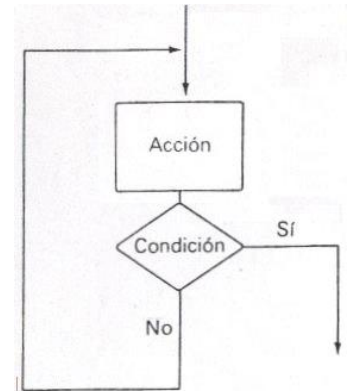
La estructura repetitiva *until* es de la siguiente forma:

```
until condicion
do
break
done
```

La estructura *until* se usa para repetir un conjunto de comando hasta que se cumpla la condición, cuando se cumple el script sale del *until*. Las condiciones y el *break* es lo mismo que en el *while*, si se usa una variable en el *until* se debe declarar antes.

Ejemplo:

```
BorrarFicheros.sh
#!/bin/bash
clear
directorio=malo
until `cd $directorio 2> /dev/null`
do
clear
read -p "Introduce un directorio " directorio
done
echo "borrado fichero"
rm -i $directorio/*
```





## 6.- Estructuras repetitivas

### 6.4.- Salir y saltar dentro de un bucle

Break → Dentro de un bucle se interrumpe y continua la ejecución después del bucle.

```
#!/bin/bash
for i in 1 2 3 4 5 6
do
if [ $i -eq 3 ]
then
    break
fi
echo "El valor de bucle es $i"
done
echo " Hemos terminado"
```

```
usuario@debian-pc200:~$ sh bucle.sh
El valor de bucle es 1
El valor de bucle es 2
Hemos terminado
usuario@debian-pc200:~$
```

Continue → Dentro de un bucle fuerza la siguiente iteración del bucle.

```
#!/bin/bash
for i in 1 2 3 4 5 6
do
if [ $i -eq 3 ]
then
    continue
fi
echo "El valor de bucle es $i"
done
echo " Hemos terminado"
```

```
usuario@debian-pc200:~$ sh bucle.sh
El valor de bucle es 1
El valor de bucle es 2
El valor de bucle es 4
El valor de bucle es 5
El valor de bucle es 6
Hemos terminado
```



## 6.- Estructuras repetitivas

### 6.5.- Diseño de bucle.

Bucle controlado por centinela.

LEER (variable)

MIENTRAS (condición)

BLOQUE

LEER (variable)

FIN-MIENTRAS





## 7.- Operaciones algebraicas

### 7.1.- Operaciones numéricas con valores enteros

`expr arg1 operación arg2`

El resultado del comando es enviado a la salida estándar.

Entre cada argumento y la operación debe haber un espacio, para que expr pueda distinguir los argumentos y su operando.

Si queremos recoger el resultado en una variable:

`Z=`expr arg1 operación arg2``



## 7.- Operaciones algebraicas

### 7.1.- Operaciones numéricas con valores enteros

**expr num1 + num2** → Devuelve la suma de num1 + num2

**expr num1 - num2** → Devuelve la resta de num1 - num2

**expr num1 \\* num2** → Devuelve el producto de num1 \* num2

**expr num1 / num2** → Devuelve la división de num1 / num2

**expr num1 >= num2** → Devuelve 0(true) si num1 >= num2

**expr num1 > num2** → Devuelve 0(true) si num1 > num2

**expr num1 <= num2** → Devuelve 0(true) si num1 <= num2

**expr num1 < num2** → Devuelve 0(true) si num1 < num2

**expr num1 != num2** → Devuelve 0(true) si num1 es distinto de num2

**%** → Módulo (Resto de una división)



## 7.- Operaciones algebraicas

### 7.1.- Operaciones numéricas con valores enteros

Ejemplos:

$4+5$   $\rightarrow$  expr 4 + 5

$3.4 + 6/2$   $\rightarrow$  expr 3 \\* 4 + 6 / 2

$4. (4+3)/2$   $\rightarrow$  expr 4 \\* \(( 4 + 3 \) / 2



## 7.- Operaciones algebraicas

### 7.1.- Operaciones numéricas con valores enteros

#### Cuentas.sh

Realizar un script que pregunte dos variables matemáticas, x e y.

A continuación debe mostrar en pantalla las operaciones:

+, -, \*, :, %

```
1 #!/bin/bash
2 clear
3 echo " Calculadora"
4 echo " _____ \n"
5 read -p "Introduce dos números separados por un espacio:"
   num1 num2
6 suma=`expr $num1 + $num2`
7 resta=`expr $num1 - $num2`
8 multiplicacion=`expr $num1 \* $num2`
9 division=`expr $num1 / $num2`
10 resto=`expr $num1 % $num2`
11 echo "\n La suma es $suma"
12 echo "\n La resta es $resta"
13 echo "\n La multiplicación es $multiplicacion"
14 echo "\n La división es $division"
15 echo "\n El resto es $resto"
16
17
```



## 7.- Operaciones algebraicas

### 7.2.- Operaciones numéricas con decimales

#### Regla mnemotécnica

bc: **B**asic **C**alculator

El comando bc es una calculadora que se puede usar desde la línea de comandos. Esta herramienta ofrece muchas posibilidades. En nuestro caso la presentaremos para realizar operaciones muy básicas, y más adelante veremos su utilidad en los scripts.

Si ejecutamos bc y entramos en su consola y podemos realizar operaciones y nos irá imprimiendo el resultado. Veamos un ejemplo:

```
$ bc
7+5
12
((7+5)/2)^2
36
1.1*3
3.3
quit
```

Su utilidad dentro de los scripts reside en la posibilidad de realizar operaciones utilizando las tuberías para pasarle operaciones matemáticas y que nos devuelva el resultado. veamos unos ejemplos:

```
$ echo 7+5*2 | bc
17
$ echo (3^5-sqrt(16))/2 | bc
119
```

Podemos actualizar el valor de variables. Guardamos en actualizamos el valor de una variable con una operación matemática  $y=y+1$  en la variable y:

```
$ y=1
$ y=$(echo $y+1 | bc)
$ echo $y
2
```

```
#!/bin/bash
clear
read -p " Introduce el numero A:" A
read -p " Introduce el numero B:" B
echo "Operaciones elementales"
echo "A=$A y B=$B"
C=$(( $A + $B ))
echo "A+B= $C"
C=$(( $A - $B ))
echo "A-B=$C"
C=$(( $A * $B ))
echo "A*B: $C"
C=$(( $A / $B ))
echo "A/B: $C"
C=$(( ($A + $B) * 2 ))
echo "(A+B)*2= $C"
```



## 7.- Operaciones algebraicas

### 7.2.- Operaciones numéricas con decimales

```
#!/bin/bash
clear
read -p " Introduce el numero A:" A
read -p " Introduce el numero B:" B
resultado=$(echo "scale = 2;  $A/$B " | bc)
echo "$resultado"
```

```
#!/bin/bash
clear
read -p " Introduce el numero A:" A
read -p " Introduce el numero B:" B
echo "scale = 2;  $A/$B " | bc
```



## 8.- Paso de parámetros

Podemos pasar parámetros tanto a los scripts como a las 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.



## 8.- Paso de parámetros

Ejercicio: Dado el siguiente script, indica lo que se mostraría en pantalla.

Podemos entenderlo mucho mejor con un script como el siguiente:

```
#!/bin/bash
# parámetros.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
```

Si hacemos este script ejecutable, y lo llamamos como:

```
./parámetros.sh Caballo Perro 675 Nueva Cork
```

### Parámetros.sh

Realiza un script que reciba como parámetros dos nombres separados por espacio. Debe saludar individualmente a cada uno. Comprobar que se le han pasado dos parámetros.

Modificarlo para que salude a todos los que se les pase por parámetro.





## 9.- Funciones

Una función es un conjunto de instrucciones que se agrupan para realizar una tarea concreta y que se pueden reutilizar fácilmente.

### Definición de la función.

Nombre\_funcion ()

{

Bloque

}

### Llamada a la función.

Nombre\_función parámetros

Ejemplo:

Nombre\_función hola adios

\$1 \$2

Le hemos pasado a la función: hola y adios.

Esos dos parámetros son recogidos dentro de la función como \$1 y \$2.

No debemos confundirlos con el \$1 y \$2 que se le pasa al script en la línea de comandos.



## 9.- Funciones

### Ejemplo: Mayor18.sh

```
#!/bin/bash
# Definición de funciones
mayor18 ()
{
  # $1 es el parámetro que se le pasa a la función
  if [ $1 -ge 18 ]
  then
    echo "Eres mayor de edad"
  else
    echo "Eres menor de edad"
  fi
}
# Inicio
clear
read -p "Introduce tu edad:" edad
mayor18 $edad
```

### doble.sh

Diseña una función a la que le pasamos un número y nos muestra el valor doble.



## 9.- Funciones

### Ejemplo: saludo.sh

Por defectos, todas las variables que usamos son **globales**, es decir, que las funciones y el script las comparten, pueden modificar sus valores, leer las modificaciones realizadas, etc. Sin embargo, en determinadas ocasiones nos puede interesar que las variables sean **locales** a la función, es decir, que si la función modifica su valor el script no se entera...

```
#!/bin/bash
#Definición de funciones
saludo ()
{
  Nombre="Bob Esponja"
  echo "Encantado de conocerte, $Nombre"
}
#Inicio
Nombre="Patricio"
saludo
echo "Mi nombre es $Nombre"
```

```
#!/bin/bash
#Definición de funciones
saludo ()
{
  local Nombre="Bob Esponja"
  echo "Encantado de conocerte, $Nombre"
}
#Inicio
Nombre="Patricio"
saludo
echo "Mi nombre es $Nombre"
```



## 9.- Funciones

Ejercicio: [suma.sh](#)

Realizar un script que tenga una función llamada suma, que reciba dos números y nos muestre la suma.



## 9.- Funciones

Ejercicio: [FechaCalendario.sh](#)

Diseñar el siguiente menú.

---

Menú

---

- 1.- Fecha del sistema.
- 2.- Calendario.
- 3.- Salir.

El menú se repetirá mientras que no se pulse salir.  
Crear una función para el menú.



## 10.- Retornar valores entre scripts

### Retornar un valor: return n

Finaliza la ejecución del script retornando un valor numérico que puede ser leído por otro script. Queda almacenado en la variable \$?.

entrada.sh

```
#!/bin/bash
#Script que llama al script descuento.sh
#para que calcule el valor de la entrada
#Definición de contantes:
ENTRADA=10
clear
echo "_____"
echo "PRECIO DE LA ENTRADA"
echo "\t $ENTRADA euros"
echo "_____"
echo
read -p "Cual es tu edad:" edad
#LLamo al segundo script
sh descuento.sh $edad
reduccion=$?
echo "\nEl descuento es $reduccion"
echo "\nLa entrada es `expr $ENTRADA - $reduccion` euros\n"
```

descuento.sh

```
#!/bin/bash
#Script que recibe una edad y devuelve un descuento
echo
echo "_____"
echo "CALCULO DEL DESCUENTO"
echo "_____"
if [ $1 -lt 18 ]
then
    echo "\nEres menor de edad"
    return 3 #Finaliza el programa y devuelve el valor al script origen
else
    echo "\nEres mayor de edad"
    return 0 #No hay descuento
fi
echo "FIN DEL SCRIPT" # Esto nunca lo escribiría.
```



## 10.- Retornar valores entre scripts

\$?

Existe un parámetro especial, el \$? que nos devuelve el valor del resultado de la ultima orden. Es decir, despues de ejecutar cualquier orden del sistema (o casi cualquier orden mejor dicho) podemos comprobar el valor de \$? que tendrá un 0 si todo ha ido bien, y otro valor cualquiera en caso de que haya fallado. Comprobarlo es muy simple:

### Ejemplo:

```
usuario@debian-pc200:~$ cd /tmp
usuario@debian-pc200:/tmp$ echo $?
0
usuario@debian-pc200:/tmp$ cd /lunares
bash: cd: /lunares: No existe el fichero o el directorio
usuario@debian-pc200:/tmp$ echo $?
1
usuario@debian-pc200:/tmp$ █
```



## Sugerencias/mejoras del tema



### Sugerencias /mejoras del tema





## Referencias

- ❑ Los logotipos del Dpto de informática han sido diseñados por Manuel Guareño.
- ❑ Algunas de las imágenes proceden de Internet y pueden tener copyright.
- ❑ Guía de programación de scripts Bash. Fernando Sosa Gil
- ❑ Sistemas Informáticos Monousuarios y Multiusuarios. Scripts en Linux. IES Fco Romero Vargas.