Scripts en GNU/Linux

En los sistemas Linux, una *shell* es un entorno de programación completo, que podemos utilizar interactivamente (como hemos hecho hasta ahora), o bien a través de pequeños programas, llamados ***scripts***.

La *shell* original de Unix es **sh**, la *shell* de GNU/Linux, compatible con la anterior, se llama **bash**. Hay otras shells que pueden implementarse, pero bash es la opción por defecto, además de ser una opción muy potente, y la más difundida.

Hola, mundo

Seremos muy originales: nuestro primer script imprimirá la frase "Hola mundo". Para ello:

- Con un editor, creamos y editamos un archivo de texto: **`nano holamundo.sh`** (el ".sh" es una convención, el archivo puede tener cualquier nombre).
- Una vez en el editor, escribimos el siguiente contenido:

```
#!/bin/bash
echo "Hola, mundo"
```

Guardamos (**`Ctrl-O`**) y salimos (**`Ctrl-X`**). La primera línea le indica al sistema cuál es el intérprete (bash, en este caso) que ejecutará el script. La segunda línea es el comando que imprime "Hola, mundo".

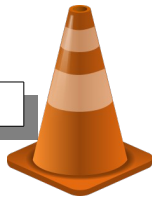
- Le damos permiso de ejecución al archivo: **`chmod u+x holamundo.sh`** (o bien: **`chmod 744 holamundo.sh`**)
- Ejecutamos el archivo, con la ruta completa hacia el mismo. Si está en la carpeta actual, debemos anteponer "./", así: **`./holamundo.sh`**

Variables:

Para asignarle valor a una variable, se utiliza la expresión:

```
nombre_variable="valor"
```

ATENCIÓN: No debe haber espacios en blanco ni antes ni después del signo =



Para **leer** el valor de la variable, se debe anteponer el signo $ a su nombre. Ejemplo:

```
echo $nombre_variable      #Mostrará "valor"
```

(Como habrán adivinado, todo lo que comience con **#**, es un comentario.)

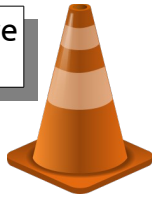
Si queremos que el usuario ingrese por teclado el valor de una variable, podemos utilizar **`read`**, como en el siguiente ejemplo:

```
#!/bin/bash
echo "Ingrese su nombre:" #Esto es un echo común y corriente
read nombre #Guarda en $nombre lo que el usuario ingrese por teclado
echo "Hola, $nombre"
```

O bien:

```
read -p "Ingrese su nombre: " nombre
echo "Hola, $nombre"
```

ATENCIÓN: Cuando se guarda un valor en una variable, se utiliza su nombre solamente. Cuando se "lee" su valor, se antepone el signo $.



Ejecución condicional

Como en cualquier lenguaje de programación, podemos crear nuestro script de manera tal que una o más acciones se ejecuten solamente en caso de que se cumpla una condición. Para ello, vamos a abordar primero cómo se plantean esas condiciones

El comando test o [ ]

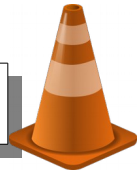
El siguiente ejemplo es autoexplicativo:

```
#!/bin/bash
palabra="HOLA"
if [ $palabra = "HOLA" ]          #Equivale a:  if test $palabra = "HOLA"
then                    #Comienzo de las acciones a ejecutar si la condición es verdadera
    echo "La palabra es HOLA"
else                    #Comienzo de las acciones a ejecutar si la condición es falsa
    echo "La palabra no es HOLA"
fi                      #final del if
```

Como vemos, después de la palabra **`if`** se plantea una condición, que puede ser evaluada como verdadera o falsa.

Dicha condición se escribe [entre corchetes], o bien precedida por la palabra **`test`** (ambas sintaxis son equivalentes).

ATENCIÓN: En caso de que se utilicen los corchetes, debemos dejar un espacio en blanco antes y después de cada corchete.



Luego, se escribe el comando **`then`**, y a continuación todas las acciones que deban ejecutarse si la condición se cumple.

Opcionalmente, podemos agregar un **`else`**, para luego especificar las acciones a seguir si la condición no se cumple.

Por último, y en todos los casos "cerramos" el **`if`** con un **`fi`**

Opciones del comando test

Archivos		
test	**[ ]**	**Es verdadero si...**
test -d elemento	[-d elemento]	*elemento* existe y es un directorio
test -e elemento	[-e elemento]	*elemento* existe
test -f elemento	[-f elemento]	*elemento* existe y es un archivo normal
test -L elemento	[-L elemento]	*elemento* existe y es un enlace simbólico
test -r elemento	[-r elemento]	*elemento* existe y es legible (permiso r)
test -w elemento	[-w elemento]	*elemento* existe y es modificable (permiso w)
test -x elemento	[-x elemento]	*elemento* existe y es ejecutable (permiso x)
test -s elemento	[-s elemento]	*elemento* existe y su tamaño es mayor que cero
test elemento1 -ot elemento2	[elemento1 -ot elemento2]	*elemento1* es más antiguo que *elemento2*

Cadenas		
test	**[ ]**	**Es verdadero si...**
test -n cadena	[-n cadena]	la longitud de *cadena* es distinta de cero
test -z cadena	[-z cadena]	la longitud de *cadena* es cero
test cadena1 = cadena2	[cadena1 = cadena2]	*cadena1* y *cadena2* son iguales
test cadena1 != cadena2	[cadena1 != cadena2]	*cadena1* y *cadena2* son distintas
Números		
test	**[ ]**	**Es verdadero si...**
test n1 -eq n2	[n1 -eq n2]	*n1* y *n2* son iguales
test n1 -ne n2	[n1 -ne n2]	*n1* y *n2* son distintos
test n1 -lt n2	[n1 -lt n2]	*n1* es menor que *n2*
test n1 -gt n2	[n1 -gt n2]	*n1* es mayor que *n2*
test n1 -le n2	[n1 -le n2]	*n1* es menor o igual que *n2*
test n1 -ge n2	[n1 -ge n2]	*n1* es mayor o igual que *n2*
Operadores lógicos		
test	**[ ]**	**Es verdadero si...**
test ! expresión	[! expresión]	*expresión* es falsa (NEGACIÓN NOT)
test expresion1 -a expresion2	[expresion1 -a expresion2]	*expresion1* y *expresion2* son ambas verdaderas (AND)
test expresion1 -o expresion2	[expresion1 -o expresion2]	*expresion1* o *expresion2* (o ambas) son verdaderas (OR)

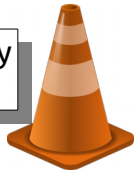
ATENCIÓN: En `[ cadena1 = cadena2 ]` el signo = opera como comparación, no como asignación. En este caso, debemos dejar un espacio en blanco antes y después del signo =

if...elif...else

Los **`if`** se pueden anidar unos dentro de otros, sin límite. Pero frecuentemente resulta más conveniente usar **`elif`**. Por ejemplo, si se desea saber si un número es positivo, negativo, o cero:

```
#!/bin/bash
x=0
if [ $x -gt 0 ]                    #Si $x es mayor que 0... (*)
then
        echo "$x es positivo"
elif [ $x -lt 0 ]                  #Si (*) dio falso, pregunto si $x es menor que 0 (**)
then
        echo "$x es negativo"
else                               #Si tanto (*) como (**) dieron falso
        echo "Es cero"
fi
```

ATENCIÓN: La indentación (dejar sangrías) no es obligatoria, pero es muy recomendable para mejorar la legibilidad del código.



Parámetros posicionales

Podemos enviarle al script parámetros posicionales en el momento en que lo ejecutamos.

Por poner un ejemplo trivial, vamos a hacer un script que recibirá un nombre y un apellido, y mostrará un saludo. Si el script se llama "saludar.sh", lo invocaremos del siguiente modo:

```
./saludar.sh Fulano "de Tal"
```

Entonces, el script recibirá como primer parámetro la cadena "Fulano" y como segundo parámetro la cadena "de Tal". (Si no hubiésemos puesto comillas, el script interpretaría que el segundo parámetro es "de" y el tercero es "Tal").

Dentro de nuestro script, el primer parámetro estará disponible en la variable **`$1`**, y el segundo parámetro en la variable **`$2`**. Obviamente, si tuviéramos más parámetros, estarían disponibles en **`$3`**; **`$4`** …

El script sería el siguiente

```
#!/bin/bash
echo "Hola, $1 $2"     #En nuestro ejemplo, mostrará: Hola, Fulano de tal
```

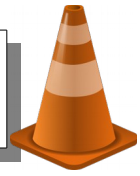
Algunas observaciones:

Supongamos, a modo de ejemplo, que ejecutamos un script con 10 parámetros:

```
./numeros.sh uno dos tres cuatro cinco seis siete ocho nueve diez
```

- La variable **`$#`** contiene la cantidad de parámetros con los que fue ejecutado el script (en el último ejemplo **`$#`** guarda el número 10). Suele ser muy útil para validar la cantidad de parámetros recibidos.
- La variable **`"$@"`** (conviene siempre utilizarla entre comillas[1]), contiene todos los parámetros concatenados (en el último ejemplo, **`$@`** vale **`"uno dos tres cuatro cinco seis siete ocho nueve diez"`**).
- La variable **`$0`** contiene el nombre con que fue ejecutado el script (en el último ejemplo, **`$0`** vale **`"./numeros.sh"`**)
- Si se necesitan más de 9 parámetros, todos los que tengan dos cifras o más, deben ponerse entre llaves. Así, si queremos mostrar el último de los parámetros, deberíamos poner **`echo ${10}`**
  (Si ponemos **`echo $10`** [**¡error!**], nos mostraría **`uno0`**)

ATENCIÓN: 1- Conviene usar la variable **"$@"** siempre entre comillas.
2- Para los parámetros del 10° en adelante, debemos poner el número entre llaves, así: **${14}**



Estados de salida

El estado de salida de un comando o script es un número[2] entre 0 y 255, que brinda información acerca de cómo finalizó el script. El estado 0 (cero) es el estado de salida exitoso.

Por ejemplo, podemos usar **`&&`** (AND) y **`||`** (OR) para ejecutar una acción en caso de que un comando falle:

```
cat hola.txt || echo "Error: archivo inexistente o sin permisos"
```

Si el comando **`cat hola.txt`** tiene un estado de salida **`0`** (exitoso), **no** ejecutará el segundo comando (**`echo "Error..."`**). Pero si el primer comando tiene un estado distinto de **`0`**, ejecutará el segundo comando. Otro ejemplo:

```
mkdir carpeta && cp hola.txt carpeta
```

Si el comando **`mkdir carpeta`** tiene un estado de salida **`0`** (exitoso), ejecutará el segundo comando (**`cp`**). Pero si el primer comando tiene un estado de salida distinto de **`0`**, **no** ejecutará el segundo comando.

1 Para que liste todos los parámetros entre comillas, evitando que un parámetro que contiene espacios en blanco sea interpretado como dos parámetros distintos.

2 Los números mayores a 125 están reservados por el sistema. El estado 1 significa "error no especificado". El estado 2 también suele estar reservado. Cuando programamos un script, se recomienda usar el estado 0 para el caso de éxito, y los estados del 3 al 125 para errores.

El comando exit

Es un comando que causa la interrupción del script. Si va seguido de un número, ese número será el estado de salida del script. Si queremos indicar que un script debe finalizar con éxito, simplemente ponemos el comando **exit 0**
Si el comando **exit** no va seguido de ningún parámetro, el estado de salida del script será el estado de salida de la última instrucción ejecutada dentro del script.
Por ejemplo, hagamos un script que reciba como parámetro el nombre de un archivo, y que muestre su contenido:

```
#!/bin/bash
if [ -f "$1" -a -r "$1" ]     #Si $1 es un archivo y podemos leerlo...
then
    cat "$1"
    exit 0                    #Salimos con estado 0 (éxito)
else
    echo "Error: archivo inexistente o sin permisos de lectura"
    exit 5                    #Salimos con estado 5 (fracaso, porque es distinto de 0)
fi
```

La variable $?

Es una variable que guarda el último estado de salida.
Podríamos reescribir el último script del siguiente modo:

```
#!/bin/bash
cat $1 2> /dev/null     #Ejecutamos el cat sin validar, descartando mensajes de error.
if [ $? -eq 0 ]         #Si el cat finalizó exitosamente...
then
   exit 0               #Salimos con estado 0 (éxito)
else
    echo "Error: archivo inexistente o sin permisos de lectura"
    exit 5              #Salimos con estado 5 (fracaso, porque $? es distinto de 0)
fi
```

La estructura case

Es una alternativa más conveniente a anidar muchos **if**, cuando las acciones a ejecutar dependen del valor de una variable.
Lo explicaremos con un ejemplo trivial:

```
#!/bin/bash
read -p "Ingrese una palabra" palabra
case $palabra in
    "hola")                                 #Si $palabra vale "hola"...
        echo "Ud. ingresó la palabra hola"
        ;;                                  #Fin de los comandos a ejecutar
    "chau")
        echo "Ud. ingresó la palabra chau"  #Si $palabra vale "chau"...
        ;;
    *)
        echo "Ud. ingresó otra palabra"     #Si $palabra vale cualquier otra cosa...
        ;;
esac                                        #Fin del case
```

Observaciones:

- La estructura comienza con: **`case $variable in`**
- Cada uno de los posibles valores de **`$variable`** va seguido del paréntesis que cierra: **`)`**
- Luego va la lista de comandos a ejecutar si **`$variable`** coincide con la opción.
- Para indicar que ha terminado la lista de comandos a ejecutar, debemos escribir el carácter ";" dos veces: `;;`
- Para indicar una serie de comados a ejecutar en caso de que **`$variable`** no coincida con ninguna de las opciones, escribimos **`*)`**
- La estructura termina con **`esac`** ("case" escrito al revés).
- Puede utilizarse el símbolo **`|`** para representar opciones alternativas. Por ejemplo, si queremos que el script anterior considere mayúsculas, la línea que dice: **`"hola")`** debería decir: **`"hola"|"Hola"|"HOLA")`**

Bucles while y until

Bucles while

Como cualquier lenguaje de programación, bash permite utilizar bucles. Una de las estructuras que cumplen con este objetivo es **while**. Su sintaxis es relativamente sencilla:

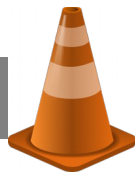
```
while condicion
do
    un comando
    otro comando
    …
    ultimo comando
done
```

Mientras la *condición*[3] se evalúe como verdadera, se ejecutarán los comandos que se encuentren entre **`do`** y **`done`**.

Un ejemplo trivial, que imprimirá 1; 2; 3 y 4, podría ser el siguiente:

```
#!/bin/bash
contador=0
while [ $contador -lt 4 ]
do
       contador=$(($contador + 1))      #O bien: ((contador++))
       echo $contador
done
```

Atención: Si al programar un script cometemos el error de dejar un bucle infinito, podemos suspender su ejecución con la combinación de teclas Ctrl-C



break y continue

El comando **`break`** sirve para salir inmediatamente del bucle.

El comando **`continue`** sirve para salir inmediatamente de la iteración actual, volviéndose a evaluar la condición, e iterando nuevamente si ésta es verdadera.

Bucles until

La estructura **until** itera mientras la condición sea falsa, y sale del bucle cuando esta sea verdadera. El siguiente ejemplo imprime 1; 2; 3 y 4:

```
#!/bin/bash
contador=0
until [ $contador -eq 4 ]
do
       contador=$(($contador + 1))      #O bien: ((contador++))
       echo $contador
done
```

3 La "condición", puede ser cualquier comando, en ese caso el bucle terminará cuando el comando falle (cuando tenga un estado de salida distinto de cero).

Uso de while para leer un archivo de texto línea por línea

Supongamos que tenemos un archivo de texto llamado semana.txt con tres líneas:

```
Lunes
Martes
Miércoles
```

Podemos leer ese archivo línea por línea, así:

```
#!/bin/bash
i=0
while read linea
do
    i=$(($i+1))          #O bien: ((i++))
    echo "La línea $i dice: $linea"
done < semana.txt
```

En el último renglón del script, se redirige la entrada, con `<` . Por lo tanto, el comando **`read`** **no** esperará que el usuario ingrese el valor de **`$linea`** por teclado, sino que lo tomará de **`semana.txt`**. Como el archivo tiene tres líneas, al intentar ejecutar el bucle por cuarta vez, el comando read fallará y saldrá del while. La salida del script será:

```
La línea 1 dice: Lunes
La línea 2 dice: Martes
La línea 3 dice: Miércoles
```

Bucles for

La sintaxis de esta estructura es la siguiente:

```
for nombre_variable in lista
do
      comando 1
      comando 2
      comando 3
done
```

En la primera iteración, el primer elemento de la lista se guarda en **`$nombre_variable`**. En la segunda iteración, **`$nombre_variable`** contiene el segundo elemento de la lista, etc. La estructura itera tantas veces como elementos tenga la lista.

Ejemplo:

```
#!/bin/bash
for dia in lunes martes miercoles jueves viernes
do
      echo "Hoy es $dia"
done
```

Mostrará:

```
Hoy es lunes
Hoy es martes
Hoy es miercoles
Hoy es jueves
Hoy es viernes
```

Atención: El bucle for en bash tiene un funcionamiento distinto al que tiene en otros lenguajes abordados en la carrera.

Un ejemplo más útil podría ser: mostrar el contenido de todos los archivos con sufijo **`.txt`** de la carpeta actual:

```
#!/bin/bash
#La expresión "*.txt" se reemplaza por una lista de los archivos que cumplan este patrón:
for archivo in *.txt
do
      echo "Contenido del archivo $archivo: "
      cat "$archivo"
done
```

Si se omite la expresión **`in lista`**, se toma por defecto la expresión **`in $@`** Dicho de otro modo, la ***`lista`*** está conformada por todos los parámetros posicionales ingresados al ejecutar el script.
Ejemplo: se ejecuta el script llamado "programando.sh", cuyo código es:

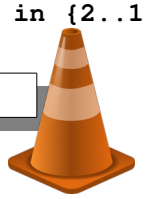
```
#!/bin/bash
for dia           #Se omite la palabra in y la lista
do
    echo "Programo en bash los $dia"
done
```

Si ejecuto el script así: **`./programando.sh lunes`** mostrará: **`Programo en bash los lunes`**
Si ejecuto el script así: **`./programando.sh martes viernes`** mostrará: **`Programo en bash los martes`**
(Si ejecutamos el script sin parámetros, no mostrará nada.) **`Programo en bash los viernes`**

Listas con números consecutivos

Si queremos ejecutar una acción un determinado número de veces (es decir, queremos que la lista que sigue a la palabra **`in`** esté compuesta por números consecutivos), podemos utilizar la siguiente sintaxis.
Supongamos que queremos que la lista esté compuesta, por ejemplo, por los primeros 10 números naturales, podemos usar la expresión **`for variable in {1..10}`** (*llave-valor inicial-punto-punto-valor final-llave*). Si queremos que esté formada, por ejemplo, por los 5 primeros números pares: **`for variable in {2..10..2}`** (*llave-valor inicial-punto-punto-valor final-punto-punto-incremento-llave*).

Atención: Esta sintaxis no funciona en versiones antiguas de bash.



Por ejemplo, si queremos mostrar la suma de los múltiplos de 3 menores que 100:

```
#!/bin/bash
acumulador=0
for num in {3..100..3}  #Equivale a: for num in 3 6 9 12 15 (etc,etc) 93 96 99
do
      acumulador=$(($acumulador+$num))
done
echo "La suma es $acumulador"
```

Sintaxis "de tres expresiones"

"A pedido" de los programadores habituados al lenguaje C y todos los que heredaron su sintaxis (como javascript y PHP), bash acepta la sintaxis "de tres expresiones". El ejemplo anterior con esta sintaxis es:

```
#!/bin/bash
acumulador=0
for ((num=3;num<=100;num+=3))
do
      acumulador=$(($acumulador+$num))
done
echo "La suma es $acumulador"
```

- Las tres expresiones van encerradas entre ((doble paréntesis)) y separadas por punto y coma.
- No se utiliza el signo **`$`** antes del nombre de las variables.
- Pueden utilizarse los signos **`<`** (menor) y **`>`** (mayor).

Funciones en bash

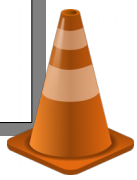
Como en casi todos los lenguajes de programación, en nuestros scripts podemos **definir** y luego **invocar** funciones. La sintaxis es:

```
#Definición:
nombre_funcion() {
comando1
comando2
}
#Invocación:
nombre_funcion
```

```
#EJEMPLO
saludar() {
echo "Buenas noches"
}
#Invocación:
saludar     # Mostrará "Buenas noches"
```

Las funciones **deben estar definidas en el código antes de ser invocadas por primera vez.** Una vez definidas, funcionan como un comando. Es decir: la invocación **no** lleva paréntesis.
Para pasar datos a una función, se utilizan los parámetros posicionales ($1, $2, etc), como en cualquier otro comando.

> 1- Hay que definir la función **ANTES** de invocarla.
>
> 2- La definición lleva los paréntesis siempre vacíos.
>
> 3- La invocación **no** lleva paréntesis



Definición:

```
#!/bin/bash
nombre_funcion() {
  #El 1er parametro recibido es $1, etc
  comandoX $1
  comandoY $2
}
#Invocación:
nombre_funcion param1 param2
```

Ejemplo

```
#!/bin/bash
saludar() {
  if [ $# = 0 ] #Si no hay parámetos...
      then
      echo "Hola, che"
  else
      echo "Hola, $1"
  fi
}
nombre="Laura"
saludar "Juan"  #Mostrará "Hola, Juan"
saludar $nombre  #Mostrará "Hola, Laura"
saludar  #Mostrará "Hola, che"
```

Funciones que retornan valores

Para retornar inmediatamente al programa principal se usa **`return`**. Return puede ir acompañado de un valor, pero solo puede ser numérico y menor a 256 (como los exit status). Ejemplo:

```
#!/bin/bash
saludar() {
  #Si $1 no existe o está vacío:
  if test -z $1
  then
      return 10
  else
      echo "Hola, $1"
      return 0
  fi
}
nombre=$1 #Asigna en la variable nombre el 1er parámetro pasado al script
saludar $nombre
resultado=$? #Guardo en la variable $resultado el estado de salida retornado por "saludo"
echo $resultado #Muestra el estado de salida
```

> ATENCIÓN: No se puede retornar valores no numéricos con **`return`**. Solamente enteros entre 0 y 255.

Sin embargo, a veces podemos necesitar que nuestra función retorne valores no numéricos (o números fuera del rango 0..255). En esos casos, se puede recurrir a una variable global o a la sustitución de comandos, como muestran estos ejemplos:

Ejemplo de "retorno" con variable global

```
#!/bin/bash
saludo=      #Defino la variable, sin asignar valor
retorna_saludo() {
  #Las variables definidas en el programa principal ANTES de la definición, son globales:
  $saludo="Hola, $1"    #$saludo guarda el string
}
retorna_saludo "Juan"
echo $saludo #Muestra "Hola, Juan"
```

Ejemplo de "retorno" con sustitución

```
#!/bin/bash
#Ejemplo de retorno con sustitución
retorna_saludo() {
  echo "Hola, $1";
}
#Invocación con sustitución : $saludo guarda lo impreso por la función. No muestra nada.
saludo=$(retorna_saludo "Juan")
echo $saludo #Muestra "Hola, Juan"
```

Software, licencia e impresión

- Este documento fue creado íntegramente con Software Libre.
- Si es necesario imprimir este documento, considere hacerlo en doble faz, para que menos árboles sean talados, y reducir la contaminación producida por la industria papelera.
- El conocimiento es libre: este documento está bajo una Licencia Creative Commons Atribución-NoComercial-CompartirIgual 4.0 Internacional. Ud. es libre de copiar, distribuir, exhibir y ejecutar la obra; así como de hacer obras derivadas de la misma, siempre que atribuya correctamente la autoría, redistribuya las obras derivadas bajo esta misma licencia, y no la utilice con fines de lucro.