Shell Scripts

Se denomina *shell script* a un fichero que contiene órdenes para ser ejecutadas por el *shell*.

Mediante el lenguaje de programación que proporciona el *shell*, podemos escribir *scripts* que realicen tareas más complejas.

El nombre del fichero puede utilizarse posteriormente para ejecutar el conjunto de órdenes almacenado, como si fuera una única orden del *shell*.

La ejecución del *script* finaliza cuando termina la ejecución de todas las órdenes del guión o se produce un error sintáctico en el lenguaje de programación.

```
#!/bin/bash
#won
#mi primer script
echo -n "Fecha y hora: "
date
echo -n "Numero de usuarios del sistema:
"
who | wc -l
echo -e "Directorio actual:\c "
pwd
exit 0
```

Ejecución de un *script*

Podemos ejecutar un *script* de varias formas:

- Creando un *shell* no interactivo y redirigiendo su entrada al *script*

```
$ bash < miscript
```

- Creando un *shell* no interactivo y pasándole el *script* como argumento:

```
$ bash miscript
```

- Dando permiso de ejecución al *script* y ejecutándolo como una orden:

```
$ chmod +x miscript

$ miscript
```

el *shell* detecta que se trata de un *shell script* y crea otro *shell* para que lo ejecute.

- Con la orden "."

```
$. miscript
```

Ejecuta el *script* sin crear un *shell* hijo.

Comentarios

Las líneas que comienzan por el carácter # son comentarios. El *shell* no las interpreta.

Debemos utilizar los comentarios para documentar nuestros *scripts*.

Un caso especial es el uso de "#!" en la primera línea para indicar el intérprete con que se ejecutará el *script*.

Si no se coloca esta línea, el *script* se interpretará con el *shell* activo.

La orden *echo*

Muestra la cadena por la salida estándar

```
$ echo -opciones cadena
```

Opciones:

- n → no realiza salto de línea
- e → habilita la interpretación de caracteres de escape. La cadena debe ir entrecomillada.
  - \n → retorno de carro y alimentación de línea
  - \r → retorno de carro
  - \t → tabulador
  - \c → inhibe el retorno de carro

Variables

Todas las variables son de tipo alfanumérico. Por tanto no se distinguen tipos de datos.

Empiezan por letra o _ (guión bajo).

La variable queda definida al darle valor.

```
$ variable=valor
```

- Crea la *variable* asignándole la cadena de caracteres *valor*
- Si *variable* ya *existe* modifica su valor

```
$ MIDIR=/home/dfsi
$ JO=Pepet
```

Referencias a variables:

- Se referencian utilizando el operador $

```
$ echo $MIDIR
$ echo $MIDIR $YO
$ DIR=$MIDIR/$YO
$ echo $DIR
```

$ `set`

- Muestra todas las variables y su valor, definidas en el *shell* actual.

$ `unset` *`variable`*

- Elimina la variable

➢ Ámbito de las variables

Las variables del *shell* sólo son accesibles en el *shell* en que han sido creadas.

Si el *script* se ejecuta desde un *shell* hijo:

- Las variables creadas en él permanecerán en memoria hasta que finalice.
- Tampoco serán accesibles para el *script* las variables del *shell* padre.

$ `export variable`

- La variable se hace visible para los *shell* hijo, y por tanto para los *scripts.*

➢ Las comillas

Según el tipo de comillas, el *shell* realiza una interpretación distinta del contenido de las mismas:

- Las comillas simples:
  - El contenido se interpreta de forma literal.

```
$ saludo='Hola $YO, que tal'
$ echo $saludo
$ echo '$saludo'
```

- Las comillas dobles:
  - Interpreta las referencias a variable

```
$ echo "Mi directorio es $MIDIR"
$ SALUDO="Hola $YO, que tal"
```

- Las comillas inversas:
  - Permiten asignar la salida de una orden a una variable

```
$ HOY=`date`
$ DIR=`pwd`
```

➢ Variables posicionales

Podemos acceder a los argumentos de un *script* mediante las variables posicionales $1, $2, …, $9. Cada $i hace referencia al argumento i-ésimo.

Además, tenemos otras variables:

- $0 → nombre del *script*
- $* → todos los argumentos (no contiene el nombre del *script*)
- $# → número de argumentos.
- $? → estado de terminación de la última orden
- $$ → pid del proceso en ejecución

Mediante el comando `set` también se puede asignar valores a las variables posicionales:

```
$ set uno dos tres
```

- → $1=uno, $2=dos, $3=tres

```
$ set `date`
```

- $1=sáb, $2=oct, $3=16,...

➢ **Ejemplo**

```
#!/bin/sh
#Per veure els arguments
echo Arguments de l\'ordre $*
echo nombre d\'arguments $#
echo comando $0
echo argument 1 $1
echo argument 2 $2
echo argument 3 $3
echo agrument 4 $4
echo estat $?
echo pid $$
echo Tota l\'ordre: $0 $*
exit 0
```

➢ **Salida por pantalla**

```
$ arguments uno dos tres quatre cinc sis
Arguments de l'ordre uno dos tres quatre cinc sis
nombre d'arguments 6
comando ./arguments
argument 1 uno
argument 2 dos
argument 3 tres
argument 4 quatre
estat 0
pid 11537
Tota l'ordre: ./arguments uno dos tres quatre cinc sis
```

Estado de terminación de una orden

Cuando una orden finaliza devuelve al *shell* un valor que indica el estado de terminación del proceso.

- 0 → Finalización correcta (éxito)
- ≠0 → Finalización incorrecta (error)

El *shell* interpreta el estado de terminación de la orden como una condición que puede ser

- Verdadera → 0
- Falsa → ≠0

Un *script* puede devolver un estado de terminación mediante el comando `exit`

```
exit valor
```

- Fuerza la terminación del *script* y devuelve *valor*

Operadores y expresiones lógicas

➢ **Expresión lógica**

Se construye del siguiente modo:

```
expresion <op> expresion <op> expresion ...
```

Donde <op> indica un operador lógico y las expresiones pueden ser numéricas, alfanuméricas o de ficheros.

➢ **Operadores lógicos**

- and → -a
- or → -o
- not → !

➢ **Expresiones relacionales numéricas**

Se construye mediante la asociación de valores numéricos (M, N) con operadores relacionales:

```
M <operador relacional> N
```

La codificación de los operadores relacionales es la siguiente:

- = → -eq
- ≠ → -ne
- > → -gt
- ≥ → -ge
- < → -lt
- ≤ → -le

➢ **Expresiones relacionales alfanuméricas**

Se construye mediante la asociación de cadenas alfanuméricas (S1, S2) con operadores relacionales:

```
S1 <operador relacional> S2
```

La codificación de los operadores relacionales es la siguiente:

- = → =
- ≠ → !=
- Cadena de longitud no nula → -n S1
- Cadena vacía → -z S1

➢ **Expresiones con ficheros**

Se construye mediante la aplicación de ciertos operadores sobre un fichero:

```
<operador> fichero
```

La codificación de los operadores que actúan sobre los ficheros es la siguiente:

- El fichero existe y no está vacío → -s fichero
- El fichero existe y es regular → -f fichero
- El fichero existe y es un directorio → -d fichero
- El fichero existe y tiene permiso de lectura → -r fichero
- El fichero existe y tiene permiso de escritura→ -w fichero
- El fichero existe y tiene permiso de ejecución → -x fichero

➢ **Evaluación de expresiones lógicas. La orden *test***

Para evaluar una expresión lógica debemos utilizar el comando `test`. Esta orden evalúa la expresión y según su valor de verdad devuelve un estado de terminación Verdadero o Falso. Una expresión lógica sin evaluar con este operador, no es interpretable por el *shell*.

```
$ test expresión
```

- Evalúa la expresión que tiene como argumento.
- Devuelve
  - 0 → expresión verdadera
  - 1 → expresión falsa

Sirve para evaluar expresiones lógicas que forman parte de una estructura condicional.

```
if test expresión
```

La orden *test* posee una forma abreviada totalmente equivalente []

```
$ test expresión ≡ $ [ expresión ]
```

- La separación de la menos un blanco entre la expresión y los corchetes es necesaria.

Evaluación de expresiones aritméticas

Existen varias formas de evaluar expresiones aritméticas. Según el comando utilizado, cambia la sintaxis, valores y los operadores soportados.

➢ El comando *expr*

Evalúa una expresión aritmética o relacional devolviendo el resultado por la salida estándar (*stdout*).

```
expr arg1 <op> arg2
```

- Sólo soporta argumentos enteros
- Los argumentos y el operador se separan por espacio.
- Algunos operadores deben escribirse con secuencias de escape: * \< \>. También los paréntesis \(… \)

Soporta los siguientes operadores:

- Aritméticos: +, -, *, /, %
- Relacionales: <, <=, >, >=, =, !=

➢ Evaluación de expresiones con órdenes internas

El *Shell* dispone de órdenes internas para la evaluación de expresiones. El resultado no se devuelve por la salida estándar, sino que lo recibe el propio *shell*. Por lo tanto lo debemos capturar mediante una variable o utilizar la orden `echo` si queremos mostrarlo por la salida estándar.

$ $ [...]

```
$ echo $[expresion] || $ r=$[expresion]
```

- Sólo admite argumentos enteros
- No es necesario separar por espacios
- No necesitamos utilizar secuencias de escape

$ $ ((...))

```
$ echo $((expresión)) || $ r=$((expresión))
```

- Totalmente equivalente a la anterior

Soportan los siguientes operadores:

- Aritméticos: +, -, *, /, %
- Relacionales: <, <=, >, >=, ==, !=
- Lógicos a nivel de bit: &, | , !, ^ (or exclusivo)

➢ Evaluación de expresiones con números reales

Mediante el comando `bc`, podemos evaluar expresiones con valores no enteros y además con la posibilidad de utilizar la división real e incluso la potencia. Realmente `bc` soporta un lenguaje de programación de sintaxis similar a C, pudiendo realizar sentencias de control de flujo entre otras cosas.

Por defecto, activa la versión interactiva.

```
$echo expresion | bc
```

- Evalúa una expresión numérica introducida por la entrada estándar.
- Admite números reales
- Soporta la división real y la potencia: / ^
- No es necesario separar por espacios.
- Admite paréntesis con secuencia de escape o encerrar la expresión con comillas.

Asignación de valores

Para asignar un valor a una variable, se utiliza el operador de asignación =. Ahora bien, si queremos asignar a una variable el resultado de una expresión, el operador de asignación sin más, ya no es válido.

Existen distintas maneras de realizar la asignación del resultado de una expresión a una variable.

➢ El comando *let*

Asigna a una variable el resultado de la expresión correspondiente

```
$ let x=expresion
```

- Admite las operaciones aritméticas (+ - * / %)
- Sólo soporta números enteros

➢ **Los evaluadores de expresiones**

Utilizando los evaluadores de expresiones y redireccionando, en su caso, la salida a la variable, podemos realizar la asignación a una variable del resultado de una expresión.

```
$ x=$[expresion]
$ x=$((expresion))
$ x=`expr expresion`
$ x=`echo expresion | bc `
```

➢ **La orden *read***

Mediante la orden `read` podemos leer valores de la entrada estándar para guardarlos en la variable o variables que se indiquen. Los datos introducidos por la entrada estándar deben distinguirse por separadores (espacio o tabulador).

```
$ read var1 var2 … varN
```

▪ Ejemplo

```
#!/bin/bash
#Saludo
echo -e "Tu nombre es: \c"
read nombre
echo -e "Tus apellidos son: \c"
read apellidos
echo Hola $nombre $apellidos
exit 0
```

Control de flujo

El *shell* posee sentencias para el control del flujo similares a las existentes en otros lenguajes de programación

➢ **if…then**

```
if condición
then
  orden1
  orden 2
  ...
  orden n
fi
```

▪ Ejemplo

```
#!/bin/bash
#micp
#cp con mi propio mensaje de error

error=0
if !cp $1 $2 2>/dev/null
    then
  error=1
    echo No se ha copiado $1 en $2
fi
exit $error
```

➢ **if…then…else**

```
if condición
  then
  orden1
       ...
  orden n
else
  orden1
  ...
  orden n
fi
```

▪ Ejemplo

```
#!/bin/bash
#micp
#cp con mi propio mensaje de error

if cp $1 $2 2>/dev/null
    then
    error=0
    echo Copia de $1 en $2 correcta
else
    error=1
    echo No se ha copiado $1 en $2
fi
exit $error
```

- **if…then…elif…else**

```
if condición
   then
   orden1
   ...
elif condición
   then
   orden1
   ...
elif condición
   ...
...
else
   ...
fi
```

- Ejemplo

```
#!/bin/bash
#max
#El maximo de 3 numeros
#
echo Introduce 3 numeros
read n1 n2 n3
echo n1=$n1  n2=$n2  n3=$n3
if test $n1 -gt $n2 -a $n1 -gt $n3
    then
    echo El mayor es: $n1
elif [ $n2 -gt $n1 -a $n2 -gt $n3 ]
    then
    echo El mayor es: $n2
else
    echo El mayor es: $n3
fi
exit 0
```

- **Bifurcación múltiple *case***

```
case var in
  caso1)
       orden1
       ...
       ordenN
  ...
  casoN)
       ordenes
  *)
       ordenes
esac
```

- Ejemplo

```
#!/bin/bash
#diasem
#Nombres de los dias de la semana
#Invocar con un nunero del 0 al 6;
#0 es domingo
case $1 in
    0)        echo Domingo;;
    1)        echo Lunes;;
    2)        echo Martes;;
    3)        echo Miercoles;;
    4)        echo Jueves;;
    5)        echo Viernes;;
    6)        echo Sabado;;
    *)        echo Nunero de 0 a 6;;
esac
exit 0
```

➢ **El bucle for**

```
for variable in lista_variables
do
orden1
...
ordenN
done
```

- Ejemplo 1

```
#!/bin/bash
#listfile
for i in a e i o u
  do
  echo Ficheros que empiezan por
/usr/bin/$i
  echo ------------------------------------
-
  ls /usr/bin/$i*
done
exit 0
```

- Ejemplo 2

```
#!/bin/bash
#listar
num=0
for i in  `ls`
do
echo fichero $num: $i
let num=num+1
#num=$[num+1]
#num=$((num+1))
#num=`expr $num + 1`
done
exit 0
```

- **La orden seq**

```
$ seq primer increment ultim
```

  - Muestra los números desde *primer* hasta *ultim* en incrementos de *increment*.
  - El valor predeterminado para primer o *increment* es 1.
  - *primer*, *increment* i *ultim* se interpretan como valores de coma flotante. (decimal después de ,)
  - *increment* debe ser positivo si *primer* es menor que *ultim*, de otra manera, negativo

  - Ejemplo

```
#!/bin/bash
#listnum
for i in `seq 1 20`
do
echo $i
done
exit 0
```

- **El bucle *while***

```
while condición
do
orden1
...
ordenN
done
```

  - Ejemplo

```
#!/bin/bash
CONT=0
while [  $CONT -lt 10 ]
  do
  echo El contador es $CONT
  let CONT=CONT+1
done
exit 0
```

- **El bucle *until***

```
until condicion
do
orden1
...
ordenN
done
```

  - Ejemplo

```
#!/bin/bash
#conectado
#comprueba si un usuario esta conectado

until who | grep "$1" >/dev/null
  do
  sleep 30
done
echo -e "\07$1 esta conectado"
exit 0
```

Las órdenes *true* y *false*

La orden *true* siempre devuelve el valor "verdadero", es decir 0.

La orden *false* siempre devuelve el valor "falso", es decir distinto de 0.

Pueden combinarse con *while* y *until* para realizar bucles infinitos

- Ejemplo

```
#!/bin/bash
#usuarios
while true
  do
  echo "El numero de usuarios es `who | wc -l`"
  sleep 10
done
```

Funciones

El lenguaje de programación del *shell* permite crear funciones para realizar tareas repetitivas fácilmente. El funcionamiento es parecido al que posee cualquier lenguaje de programación, en el cual se agrupan conjunto de comandos y se los llama por un nombre. El formato de las funciones es el siguiente

```
function nomfunc
{
orden1
…
ordenN
}
```

Las características principales de las funciones son:

- Pueden definirse en cualquier lugar
- Para invocar la función hay que escribir el nombre como si fuera un comando.
- Podemos pasarle parámetros utilizando las variables posicionales.
- Puede devolver un valor con *return* entre 0-255, accesible mediante $?.
- Puede utilizar variables globales o locales con *local.*

- Ejemplo 1

```
#!/bin/bash
#continuar
function cont
{
    until false
      do
      echo "Desea continuar?"
      read resp
      if [ $resp = s ] then
          return 0
      elif [ $resp = n ] then
          return 1
      else
          echo Contesta s o n
      fi
    done
}

while cont
  do
  echo Continuamos...
  sleep 2
done
exit 0
```

- Ejemplo 2

```
#!/bin/bash

function suma
{
    local sum=$[$1+$2]
    return $sum
}

if [ $# -eq 2 ]
    then
    suma $1 $2
    echo $1 + $2 = $?
else
    echo "Introduce dos argumentos"
fi
exit 0
```

- Ejemplo 3

```
#!/bin/bash

function suma
{
    let sum=$1+$2
}

if [ $# -eq 2 ]
    then
    suma $1 $2
    echo $1 + $2 = $sum
else
    echo "Introduce dos argumentos"
fi
exit 0
```

Depuración de scripts

```
$ bash -opcion script
```

- opciones:
  - n → Lee las órdenes pero no las ejecuta. Detecta los errores sintácticos.
  - v → Muestra las líneas de entrada tal como las lee junto con el resultado de la ejecución.
  - x → Muestra las líneas de entrada tal como se ejecutan (con la sustitución de variables) junto con el resultado de la ejecución.
- Lanzando el *script* con la opción adecuada, podemos obtener información para depurarlo.