TEMA 5: Programación Shell

ADMINISTRACIÓN DE SISTEMAS OPERATIVOS

2020/2021

Cristian Martín Fernández

Contenido

- Shell scripts
- Ejecución de scripts
- Parámetros y variables
- Sentencias condicionales
- Bucles
- Funciones

¿Qué es un script?

- El Shell es un interprete de comandos que permite ejecutar una serie de ordenes y la interacción de los usuarios con el sistema.

- Pero también es mucho más que eso: es un **lenguaje de programación** en el que cada instrucción se ejecuta de forma secuencial como un comando.

- Un **shell script** es un fichero de texto que contiene un conjunto de comandos y órdenes interpretables por el Shell, no ejecutables.

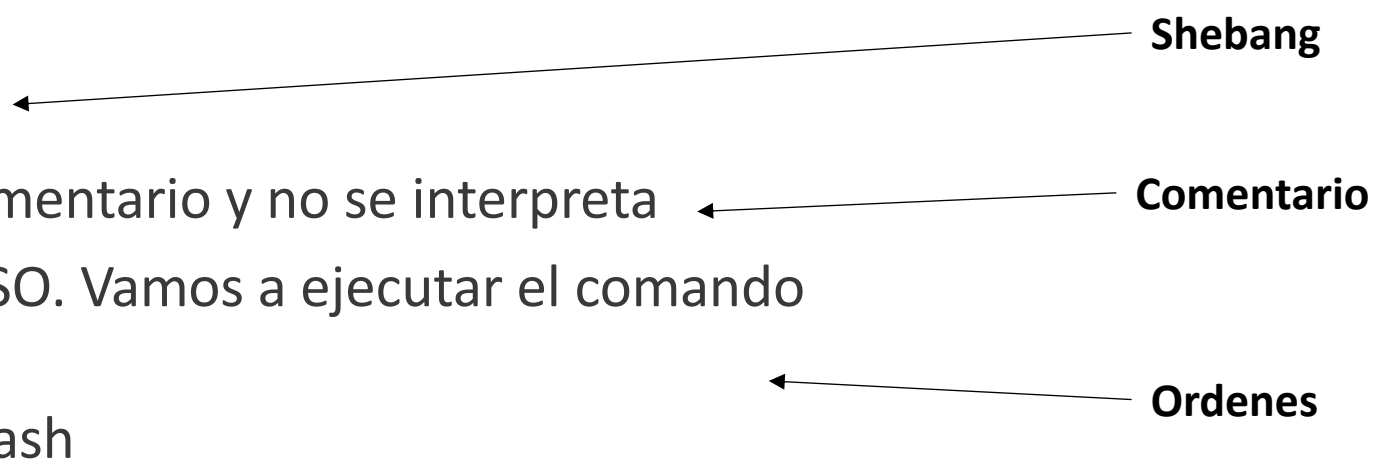
- Es habitual que los shell scripts tengan la extensión ".sh" para ayudar a la identificación del contenido a partir del nombre del archivo.

Bourne Again Shell (Bash)

- Shell más moderno y más utilizado en sistemas GNU/Linux.
- Superconjunto de Bourne Shell (sh). Todos los scripts sh se pueden ejecutar en bash, pero no al contrario!
- Además posee una serie de ventajas como:
  - definición de alias
  - ejecución síncrona y paralela
  - incorporación de operadores de lenguaje de alto nivel
  - control de trabajos, etc.

Sintaxis de un script shell

primer_script.sh



Shebang

- El shebang permite **especificar el intérprete de comandos** con el que deseamos que sea interpretado el resto del script cuando se usa invocación implícita.

- Es imprescindible que el shebang se encuentre en la **primera línea** del script, ya que, en caso contrario, sería interpretado como un comentario (comienza con el carácter #).

- El shebang no es obligatorio, pero es recomendable. Si no se indica en invocación implícita se intentará usar el mismo tipo de shell desde el que se ha invocado el script.

- Para consultar la Shell por defecto se puede utilizar el siguiente comando:
  - `$ echo $SHELL`

Ejecución de scripts

- Explícita: escribiendo explícitamente qué shell se desea invocar. Esto creará un nuevo proceso en memoria para dicha Shell. En este caso se ignora el shebang. Ejemplos:
  - `/bin/sh primer_script.sh`
  - `/bin/dash primer_script.sh`
  - `/bin/bash primer_script.sh`

- Implícita: invocando al script como si fuera un ejecutable, lo que requiere asignar permisos de ejecución al script. Se lee el shebang para determinar qué shell deberá usarse para leer el script, cargándose en memoria un proceso hijo (subshell) para dicha Shell.
  - `./primer_script.sh`

- Implícita con "." o con source: el script será interpretado por el *mismo proceso* del shell responsable de la línea de comandos desde la que se está invocando el script (luego aquí no se abre ningún subshell). Script interactivo. En este caso también se ignora el shebang.
  - `. primer_script.sh`
  - `source primer_script.sh`

Parámetros y variables en Shell

- Hay tres tipos de parámetros en Shell:

1. Si el nombre es un número se denominan parámetros posicionales.
2. Si el nombre es un carácter especial se denominan parámetros especiales.
3. El resto son variables.

Parámetros posicionales

- Los parámetros posicionales son los encargados de recibir los argumentos de un script y los parámetros de una función.

- Para acceder a ellos utilizamos el símbolo $ seguido de la posición del parámetro. Ejemplo: $1, $2, $3, etc.

- El parámetro posicional 0 almacena el nombre del script donde se ejecuta, o la Shell si se ejecuta directamente.
- posicionales.sh

```
#!/bin/bash

# Ejemplo de script que recibe parametros y los imprime
echo "El script $0"
echo "Recibe los argumentos $1 $2 $3 $4"
```

Parámetros especiales

$*	Se expande a todos los parámetros posicionales desde el 1. Si se usa dentro de comillas dobles, se expande como una única palabra formada por los parámetros posicionales separados por el primer carácter de la variable IFS.
$@	Se expande a todos los parámetros posicionales desde el 1, como campos separados, incluso aunque se use dentro de comillas dobles.
$-	Opciones actuales del shell (modificables con el comando set). Consulte las opciones disponibles con el comando man dash .
$#	Nº de argumentos pasados al script (no incluye el nombre del script).
$?	Valor devuelto por el último comando, script, función o sentencia de control invocado.
$$	PID del proceso shell que está interpretando el script.
$!	PID del último proceso puesto en segundo plano.

Parámetros especiales

- **especiales.sh**

```
#!/bin/bash

IFS=','
echo "El script $0 con PID $$ recibe $# argumentos: $*"
echo "El script $0 con PID $$ recibe $# argumentos: $@"
echo "El ultimo proceso puesto en segundo plano es $!"
echo "El ultimo valor devuelto es $?"
```

Variables

- No es necesario definir las variables previamente a su uso, ya que se crean al asignarles un valor la primera vez.

- Al definir una variable sin inicialización, su valor por omisión es la cadena nula. Las siguientes entradas son equivalentes:
  - `VAR=`
  - `VAR=""`

- Por defecto, todas las variables son tratadas por defecto como cadenas de caracteres.

Variables

- Es importante no incluir ningún espacio ni antes ni después del signo =. Si se hace, el shell intentará interpretar el nombre de la variable como un comando:
  - `VAR=5 # Asignación correcta`
  - `VAR = 5 # Comando VAR`

- Para el acceso al valor de una variable utilizamos $:
  - `$VAR`
  - `${VAR}`

- Para declarar variables de tipo entero usamos la opción declare:
  - `declare -i VAR`
  - `VAR=1`
  - `VAR=VAR+1`
  - `echo $VAR`

Variables

variables.sh

```
#!/bin/bash

VAR = 1
VAR=1
declare -i var=3

echo "Variables: $VAR $var"
VAR=$VAR+1
var=var+1
echo "Variables: $VAR $var"
echo "Variable VAR seguida de 1 (requiere llaves): ${VAR}1"
echo 'Comillas simples: $VAR'
echo "Valor: $VAR-1"
```

Operadores de cadena

- Los operadores de cadena nos permiten manipular cadenas sin necesidad de escribir rutinas de procesamiento de texto.

- En particular los operadores de cadena nos permiten realizar las siguientes operaciones:
  - Asegurarnos de que una variable existe (que está definida y que no es nula).
  - Asignar valores por defecto a las variables.
  - Tratar errores debidos a que una variable no tiene un valor asignado.
  - Coger o eliminar partes de la cadena que cumplen un patrón.

Operadores de cadena. Asignación

- **$ {*var*:-*dato*}** Si *var* existe y no es nula retorna su valor, sino retorna dato.

- **${*var*:+*dato*}** Si *var* existe y no es nula retorna dato, sino retorna una cadena nula.

- **${*var*:=*dato*}** Si *var* existe y no es nula retorna su valor, sino asigna *dato* a *var* y retorna su valor.

- **${var:?mensaje}** Si var existe y no es nula retorna su valor,
  sino imprime var:mensaje y aborta el script que se esté ejecutando (sólo en shells no interactivos).

Operadores de cadena. Asignación

asignacion.sh

```
#!/bin/bash

unset VAR
echo 1: ${VAR:-1}
VAR=2
echo 2: ${VAR:+3}
unset VAR # eliminamos la variable VAR
echo 3: ${VAR:=4}
echo 4: $VAR
unset VAR
${VAR:?variable no declarada}
echo Adios
```

Operadores de cadena. Búsqueda de patrones

- **${var%sufijo}** Elimina el sufijo más pequeño del valor de la variable. Si en vez de % se pone %% se elimina el sufijo más grande.

- **${var#prefijo}** Elimina el prefijo más pequeño del valor de la variable. Si en vez de # se pone ## se elimina el prefijo más grande.

- **${var:offset:longitud}** Retorna una subcadena de var que empieza en offset y tiene longitud caracteres. Si se omite longitud la subcadena empieza en offset y continua hasta el final de var.

Operadores de cadena. Búsqueda de patrones

patrones.sh

```
#!/bin/bash

TEXT="ASO 1 ASO2021 ASODECEMBER"
echo ${TEXT%A*}
echo ${TEXT%%A*}

FICH=/usr/bin/aso
echo ${FICH#*/}
echo ${FICH##*/}

echo ${TEXT:6:8}
```

Sustitución de comando

- Permite que la salida estándar de un programa se utilice como parte de una variable u otro comando.

- Existen dos opciones, con el mismo funcionamiento:
  - `$(comando)`
  - `` `comando` ``

- Por ejemplo, para almacenar en una variable el nombre de todos los ficheros con extensión .sh del directorio actual, podría escribir:
  - `` VAR=`ls *.sh` ``

- O, por ejemplo, para matar el proceso con nombre “python”, podría usar:
  - `kill -9 $(pidof python)`

Sustitución de comando

sus_comando.sh

```
#!/bin/bash


echo El directorio $1 tiene $(find $1 -type d | wc -l) directorios
echo Y $(find $1 -type f | wc -l) ficheros
file $(find $1 -name "*.sh" -type f)
```

Expansión aritmética

- Permite evaluar las cadenas indicadas en la expresión como enteros, admitiendo gran parte de los operadores usados en el lenguaje C.

- Tras la evaluación aritmética, el resultado vuelve a ser convertido a una cadena.

- El formato para realizar una expansión aritmética es el siguiente:
  - `$((expresion))`

- Ejemplo:
  - `$(($VAR+1))`

- Si se usan variables en la expresión, no es necesario que vayan precedidas por el carácter $ si ya contienen un valor entero válido.

- Otra forma de expansión aritmética es usar el operador let
  - `let VAR=2+5`

Expansión aritmética

aritmetica.sh

```
#!/bin/bash

VAR=1
VAR=$VAR+1
echo $VAR
RES=$(($VAR))+1
echo $RES
let RES=$RES+1
echo $RES
VAR1=b
echo $(($VAR1+1)) # VAR1 necesita $
```

Sentencias condicionales

- La sentencia condicional (verdadera con 0) tiene el siguiente formato:

```
if condicion
then
    sentencias
elif condicion
then
    sentencias
else
    sentencias
fi
```

- Es equivalente a:

```
if condicion ; then
    sentencias
elif condicion ; then
    sentencias
else
    sentencias
fi
```

- También se puede poner todo en una línea:
  - `if condicion; then sentencias; elif condicion; then sentencias; else sentencias;`

Sentencias condicionales

if.sh

```
#!/bin/bash

if [ -f $1 ]; then
    echo el fichero $1 existe
else
    echo fichero no encontrado
fi
```

Sentencias condicionales. Case

- La estructura case se puede utilizar directamente con cadenas de caracteres, al contrario que otros lenguajes de programación.
- El doble punto y coma ;; permite determinar el final de los elementos a interpretar cuando se cumpla su patrón asociado. Por ese motivo, el ;; del último patrón puede omitirse.

```
case cadena_texto in
    patron1) lista-compuesta1;;
    patron2) lista-compuesta2;;
    ...
    * ) lista-defecto [;;] #coincide con todo
esac
```

Sentencias condicionales. Case

case.sh

```
#!/bin/bash

case $1 in
   texto.txt)
      echo Prueba case correcta! ;;
   *.c)
      echo Fichero C ;;
   *)
      echo Default. Pruebe otro
esac
```

Operadores lógicos

- Podemos combinar varios códigos de terminación de comandos mediante los **operadores lógicos** and (representada con &&) or (representada con ||) y not (representada con !).

- Estas operaciones, al igual que en otros lenguajes como C o Java, funcionan en cortocircuito, es decir el segundo operando solo se evalúa si el primero no determina el resultado de la condición.

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

Test condicionales

- La sentencia if lo único que sabe es evaluar códigos de terminación. Pero esto no significa que sólo podamos comprobar si un comando se ha ejecutado bien.

- Podemos utilizar el operador [] dentro del cual metemos la condición a evaluar.

- Los espacios entre los corchetes y los operandos, o entre los operandos y el operador de comparación = son obligatorios.

- Ejemplo:
  - [cadena1 = cadena2]

Test condicionales. Comparación de cadenas

- Las comparaciones que realizan < y > son lexicográficas, es decir comparaciones de diccionario, donde por ejemplo q es mayor que perro.
- No existen operadores <= y >= ya que se pueden implementar mediante operaciones lógicas.

Operador	**Verdadero si ...**
`str1 = str2`	Las cadenas son iguales
`str1 != str2`	Las cadenas son distintas
`str1 < str2`	`str1` es menor lexicográficamente a `str2`
`str1 > str2`	`str1` es mayor lexicográficamente a `str2`
`-n str1`	`str1` es no nula y tiene longitud mayor a cero
`-z str1`	`str1` es nula (tiene longitud cero)

Test condicionales. Comparación de enteros

- El shell también permite comparar variables que almacenan cadenas interpretando estas cadenas como números.

Operador	Descripción
`-lt`	Less Than
`-le`	Less than or Equal
`-eq`	EQual
`-ge`	Greater than or Equal
`-gt`	Greater Than
`-ne`	Not Equal

Test condicionales. Comparación de ficheros

- El tercer tipo de operadores de comparación nos permiten comparar atributos de ficheros. Existen 20 operadores de este tipo:

Operador	**Verdadero si ...**
`-a` *fichero* *fichero*	existe
`-b` *fichero* *fichero*	existe y es un dispositivo de bloque
`-c` *fichero* *fichero*	existe y es un dispositivo de carácter
`-d` *fichero* *fichero*	existe y es un directorio
`-e` *fichero* *fichero*	existe (equivalente a `-a`)
`-f` *fichero* *fichero*	existe y es un fichero regular
`-g` *fichero* *fichero*	existe y tiene activo el bit de setgid
`-G` *fichero* *fichero*	existe y es poseído por el group ID efectivo
`-h` *fichero* *fichero*	existe y es un enlace simbólico
`-k` *fichero* *fichero*	existe y tiene el stricky bit activado
`-L` *fichero* *fichero*	existe y es un enlace simbólico
`-N` *fichero* *fichero*	existe y fue modificado desde la última lectura
`-O` *fichero* *fichero*	existe y es poseído por el user ID efectivo
`-p` *fichero* *fichero*	existe y es un pipe o named pipe
`-r` *fichero* *fichero*	existe y podemos leerlo
`-s` *fichero* *fichero*	existe y no está vacío
`-S` *fichero* *fichero*	existe y es un socket
`-u` *fichero* *fichero*	existe y tiene activo el bit de setuid
`-w` *fichero* *fichero*	existe y tenemos permiso de escritura
`-x` *fichero* *fichero*	existe y tenemos permiso de ejecución, o de búsqueda si es un directorio

Test condicionales

ficheros.sh

```
#!/bin/bash

if [ $# -eq 0 ]; then
        echo Falta el nombre del fichero
else
        if [ -f $1 ]; then
                if [ -x $1 ]; then
                        echo El fichero $1 es ejecutable
                else
                        echo El fichero $1 NO es ejecutable
                fi
        else
                echo El argumento $1 no es un fichero
        fi
fi
```

Test condicionales

- Se permite crear condiciones múltiples mediante los operadores:

- condicion1 **-a** condicion2
  - AND: Verdadero si ambas condiciones son verdaderas
- condicion1 **-o** condicion2
  - OR: Verdadero si se cumple alguna de las dos condiciones

- Ejemplo, el fichero existe y podemos leerlo pero no se puede modificar:
  - [-r $FILE -a -w $FILE];

- Ejemplo, o el fichero no existe o es un enlace simbólico:
  - [! -e $FILE -o -h $FILE];

Ejercicio de clase

- Ejecuta este script con source o ‘.’

- Compruebe que se le pasa al menos un parámetro, en caso contrario termina (return) el script con código 1.

- Compruebe que se le pasa un directorio como argumento, en caso contrario termina con código 2.

- Si se le pasa un directorio, debe de comprobar que no es el directorio actual y tienes permisos de búsqueda, en caso contrario devuelve el código 3. En caso afirmativo, nos desplazamos hacia él.

Bucles. For

- El bucle for en Bash es un poco distinto a los bucles for tradicionales de otros lenguajes como C o Java, sino que se parece más al bucle for each de otros lenguajes, ya que aquí no se repite un numero fijo de veces, sino que se procesan las palabras de una frase una a una.

- La estructura for define la variable “i” si no ha sido previamente definida.

- lista_valores se corresponde con un conjunto de valores (tomándose cada valor como una cadena de caracteres que puede ser objeto de expansión y como caracteres de separación los caracteres definidos en la variable IFS).

```
for i in lista_valores; do
    sentencias
done
```

Bucles. For

bucle_for.sh

```
#!/bin/bash

for i in 1 2 3; do
        echo "Iteracion: $i"
done

for file in $(ls $1); do
        echo "Fichero: $file"
done

for i in $(seq $2 1 $3); do
        echo "Iteracion: $i"
done
```

Ejercicio de clase

- Desarrolle un script que liste todos los ficheros ejecutables que haya en un directorio pasado como argumento.

Bucles. While y until

- **While** se ejecuta mientras que el código de terminación del comando sea exitoso, es decir 0, mientras que **until** se ejecuta hasta que el código de terminación sea exitoso.
- Until se puede interpretar como ejecutar varias veces un comando hasta que tenga éxito.

```
while condicion; do
    sentencias
done

until condicion; do
    sentencias
done
```

Bucles. While y until

- bucles_while_until.sh

```
#!/bin/bash

CONTADOR=0
while [ $CONTADOR -lt 10 ]; do
    echo El contador while es $CONTADOR
    let CONTADOR=$CONTADOR+1
done

CONTADOR=0
until [ $CONTADOR -ge 10 ]; do
    echo El contador until es $CONTADOR
    let CONTADOR=$CONTADOR+1
done
```

Funciones

- El formato de una función en bash es el siguiente:

```
nombrefuncion () {
    comandos bash ···
}
```

- El paréntesis siempre debe estar vacío (sólo indica que se está definiendo una función). Pueden insertarse espacios antes, entre y después del paréntesis

- Cuando definimos una función se almacena como una variable de entorno (con su valor almacenando la implementación de la función).

- Las funciones reciben parámetros posiciones de la misma forma que un script y sólo pueden devolver enteros.

- Para ver una función definida *declare –f nombrefn.* Para borrar una función podemos usar el comando unset -f *nombrefn*.

Funciones

- funciones.sh

```
#!/bin/bash

suma () {
    RES=$(($1+$2))
    echo "Suma: $RES"
    return $RES
}
suma 2 3
suma $1 $2 #suma los 2 primeros argumentos pasados al script
echo "Valor devuelto de la ultima suma: " $?
```

Ejercicio de clase

- Sabiendo que el operador ${#var} devuelve el tamaño de una cadena, y utilizando el operador para obtener una subcadena ${var:offset:longitud}, cree una función Reverse que devuelva a través de variable global una cadena de caracteres invertida.

Variables globales y locales

- Las variables de un script se liberan después de su ejecución si utilizamos scripts no interactivos.

- Por defecto, todas las variables que definimos en un script o función son globales, es decir, una vez definidas en el script son accesibles (y modificables) desde cualquier función.

- Si queremos que una variable no posicional sea local debemos de utilizar el modificador **local**, el cual solo se puede usar dentro de las funciones (no en los scripts). Ejemplo:
  - `local var=2`

Variables globales y locales

globables.sh

```
#!/bin/bash

CheckVar () {
        var='Dentro de la funcion'
}
var='En el script'
echo $var
CheckVar
echo $var

echo Ejemplo usando variables locales

LocalCheckVar () {
        local var='Dentro de la funcion'
}
var='En el script'
echo $var
LocalCheckVar
echo $var
```

Bibliografía

- Programación Shell-script: http://trajano.us.es/~fjfj/shell/shellscript.htm#_Toc444081200
- Shell Command Language: https://pubs.opengroup.org/onlinepubs/009695399/utilities/xcu_chap02.html
- Shell Script: https://www.shellscript.sh/