

Linux para Ingeniería:

Shell Scripting

Luis Garreta

luis.garreta@javerianacali.edu.co

Ingeniería de Sistemas y Computación
Pontificia Universidad Javeriana – Cali

23 de febrero de 2018

¿Que es el Shell?

- ▶ Es la interfaz entre el usuario final y el Sistema Operativo.
- ▶ No es el S.O.
- ▶ Existen múltiples versiones. Para averiguar cual tenemos instalada :

```
$ bash --version
```

Las Shells

- ▶ En UNIX existen múltiples Shells:
 - ▶ Bourne shell (sh),
 - ▶ C shell (csh),
 - ▶ Korn shell (ksh),
 - ▶ TC shell (tcsh),
 - ▶ Bourne Again shell (bash).
- ▶ La más popular es la "bash" shell.
 - ▶ Incorpora las prestaciones más útiles de la Korn shell (ksh) y la C shell (csh).
 - ▶ Ofrece mejoras funcionales sobre otras shells desde el punto de vista de programación y de su uso interactivo

Programación o scripting?

- ▶ bash no es únicamente una excelente shell por línea de comandos
- ▶ También es un **lenguaje de scripting** en sí mismo
 - ▶ El shell scripting permite utilizar las capacidades de la shell para automatizar multitud de tareas que, de otra forma, requerirían múltiples comandos introducidos de forma manual

Lenguaje de programación vs. scripting:

- ▶ Los lenguajes de programación son, en general, más potentes y mucho más rápidos que los lenguajes de scripting
- ▶ Los lenguajes de programación comienzan desde el código fuente, que se compilan para crear los ejecutables (lo que permite que los programas sean fácilmente portables entre diferentes SO)

Scripting

- ▶ Un lenguaje de scripting también comienza por el código fuente, pero no se compila en un ejecutable
- ▶ En su lugar, un intérprete lee las instrucciones del fichero fuente y las ejecuta secuencialmente
- ▶ Los programas interpretados son, en general, más lentos que los compilados
- ▶ La principal ventaja reside en que el fichero de código fuente es fácilmente portable a cualquier sistema operativo
- ▶ Además, los lenguajes de script (bash, python, etc.) son generalmente más fáciles de usar:
 - ▶ No declara variables
 - ▶ No tiene que compilar
 - ▶ Puede hacer pruebas en el shell del lenguaje

El primer programa bash

- ▶ Editamos un nuevo archivo con el editor de textos: **vi**

```
$ vi hola.sh
```

- ▶ Escribimos el código

```
#!/bin/bash echo  
"Hola Mundo"
```

- ▶ Cambiamos los permisos

```
$chmod +x hola.sh
```

- ▶ Vemos los permisos:

```
$ ls -l  
-rwx----- hola.sh
```

- ▶ Ejecutamos

```
$ ./hola.sh
```

Modificación de la variable PATH

- ▶ Para evitar el `./` al inicio de la ejecución de los script, modificamos la variable de ambiente **PATH** para que tenga acceso a los ejecutables que están en nuestro directorio actual, así:
 - ▶ Editar el archivo `.profile` que está en la raíz de su *home*
 - ▶ Modificar la variable PATH agregando una nueva instrucción al final

```
# Archivo .profile
...
...
PATH=$PATH:.bin
```

Segundo script en bash

- ▶ A veces es peligroso borrar directorios con el comando **rm** ya que no se pueden recuperar fácilmente si se comete un error.
- ▶ Entonces vamos a realizar nuestro propio comando para borrar moviendo el archivo a una carpeta llamada **TRASH** :
 - ▶ Crear el directorio *TRASH* en su home
 - ▶ Crear el siguiente script del.sh:

```
#!/bin/bash
nombreArchivo=$1

mv $nombreArchivo ~/TRASH
```

- ▶ Mover el archivo a su directorio de binarios: ~/BIN

Ejercicio

Crear el comando *undel.sh*

Variables

- ▶ Como en cualquier lenguaje de programación, en shell scripting se pueden utilizar variables
- ▶ Todos los valores son almacenados como cadenas de texto
- ▶ También hay operadores matemáticos que convierten las variables en números para el cálculo
- ▶ No es necesario declarar una variable, simplemente asignándole un valor a su referencia será suficiente para crearla
- ▶ Ejemplo:

```
#!/bin/bash
STR="Hola Mundo!" # sin espacios en blanco
echo $STR
```

- ▶ El valor de la variable *STR* se recupera anteponiendo \$, así *\$STR*

Caracteres especiales del Shell

- ▶ Existen algunos caracteres especiales para el shell como:
 - ▶ `?, *, ~, $`
 - ▶ Si quiere usar estos caracteres como símbolos, debe usar el caracter de escape : `\`
 - ▶ Ejemplo:

```
$ ls *           # Muestra todos los archivos
$ ls \*          # Busca archivos que se llamen *
```

Comillas Simples y Dobles

- Las comillas simples mostrarán una cadena de caracteres de forma literal sin resolución de variables:

```
$ var='cadena de prueba'
$ nuevavar='Valor de var es $var'
$ echo $nuevavar
Valor de var es $var

$ otravar='Valor de var es $var'
$ echo $nuevavar
Valor de var es cadena de prueba
```

Comando export

- ▶ El comando export pone una variable en el entorno de forma que sea accesible por los procesos hijos.
- ▶ Ejemplo:

```
$ x=hola
$ bash                # Ejecuta una shell en proceso hijo
$ echo $x             # Nada en x
$ exit                # Volvemos al proceso padre

$ export x
$ bash
$ echo $x
hola                  # Ahí está.
```

Comando export

- ▶ Si el proceso hijo modifica el valor de la variable x, no modificará el valor original del proceso padre.
- ▶ Ejemplo: Verificarlo cambiando el valor de x de la siguiente manera:

```
$ x=hola
$ bash                # Ejecuta una shell en proceso hijo
$ echo $x             # Nada en x
$ x=adios
$ exit
$ echo $x
hola                  # Ahí está.
```

Variables de Entorno

- Hay dos tipos de variables de entorno:
 - **Variables locales:** las que usted crea durante su sesión
 - **Variables del entorno:** Las Variables del entorno se establecen por el sistema y se pueden encontrar utilizando el comando **env**.

```
$ env
CPATH=/home/lg/opt/apps/sys/include
DEFAULTS_PATH=/usr/share/gconf/icewm-session.default.path
GW_APP=/home/lg/cloud/gebixw/Dropbox/dev
JAVA_PATH=/opt/apps/java/jdk
XDG_CONFIG_DIRS=/etc/xdg/xdg-icewm-session:/etc/xdg
DESKTOP_SESSION=icewm-session
PATH=/home/lg/bin:/opt/node_modules/bin::/home/lg/.bin:/opt/bin:/
sbin:/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/
home/lg/cloud/gebixw/opt/apps/bio/uclust:/home/lg/cloud/gebixw/
Dropbox/dev:/DATA/cloud/pathways/eval/tools/programs/
mmtsbt_toolset:/opt/apps/edirect
QT_QPA_PLATFORMTHEME=appmenu-qt5
```

Inicialización Variables de Entorno

- ▶ Las variables del entorno se definen en `/etc/profile`, `/etc/profile.d/` y `~/.bash_profile`.
- ▶ Estos ficheros son de inicialización y son leídos cuando se invoca la bash shell.
- ▶ Cuando la login shell sale, la bash lee `~/.bash_logout`

Variables de Entorno Principales

- ▶ **HOME**: argumento por defecto (directorio home) del comando `cd`.
- ▶ **PATH**: el path de búsqueda de comandos. Es una lista de directorios separados por ':' en los que se busca cuando se teclea cualquier comando
 - ▶ Normalmente, introducimos los comandos de la siguiente manera:

```
$ ./trash.sh
```

- ▶ Estableciendo:

```
PATH=$PATH:.
```

- ▶ Nuestro directorio de trabajo se incluye en path de búsqueda de comando y simplemente podremos introducir:

```
$ trash.sh
```


Otras Variables de Entorno

- ▶ LOGNAME: contiene el nombre de usuario
- ▶ HOSTNAME: contiene el nombre de la máquina
- ▶ MACHTYPE: sistema hardware
- ▶ PS1: secuencia de caracteres mostrados antes del prompt
 - ▶ \t hora
 - ▶ \d fecha
 - ▶ \w directorio actual
 - ▶ \W última parte del directorio actual
 - ▶ \u nombre de usuario
- ▶ \\$ caracter del prompt
- ▶ UID: contiene el id del usuario que no puede ser modificado
- ▶ SHLVL: contiene el nivel de anidamiento de la shell

Argumentos de Línea de Comandos en Scripts

- ▶ \$1...\$9: Parámetros posicionales
- ▶ \$#: Número de argumentos

```
#!/bin/bash  
echo Argumento 1 es: $1  
echo Existen $# argumentos
```

- ▶ Otras variables intrínsecas:
 - ▶ \$#: número argumentos
 - ▶ \$*: todos los argumentos de la shell
 - ▶ \$@: semejante a la variable anterior
 - ▶ \$-: opciones suministradas a la shell
 - ▶ \$?: devolver valor de la última orden ejecutada
 - ▶ \$!: identificación del proceso de la última orden que comenzó con &

Comando *exit*

- ▶ El comando `exit` se puede utilizar para finalizar la ejecución de un script o para devolver un valor
- ▶ El valor devuelto estará disponible al proceso padre del script.
- ▶ Cuando un script termina con `exit` sin parámetros, el estado de salida será el del último comando ejecutado en el script

```
#!/bin/bash
COMANDO_1
. . .
# sale con el estado de la ejecución
#del último comando.
ULTIMO_COMANDO
exit
```

- ▶ Cuando quiere retornar un valor diferente de Cero:

```
#!/bin/bash
COMANDO_1
. . .
ULTIMO_COMANDO
exit 999
```

Comando *read*

- ▶ El comando `read` nos permite solicitar un valor de entrada para almacenarlo en una variable:
- ▶ Ejemplo:

```
#!/bin/bash
echo -n "Introduzca nombre de fichero a borrar: "
read fichero
rm -i $fichero
echo "Fichero $fichero borrado!"
```

Opciones de Comando *read*

► Opciones

- `read -s` (no hace echo de la entrada)
- `read -nN` (acepta sólo N caracteres de entrada)
- `read -p "mensaje"` (muestra un mensaje)
- `read -tT` (acepta una entrada por un tiempo máximo de T segundos)

► Ejemplo:

```
$ read -s -n1 -p "si (S) o no (N)?" respuesta
si (S) o no (N) ? S
$ echo $respuesta
S
```

Sustitución de comandos

- ▶ El símbolo `""` tiene un uso diferente de `'``'`. Se utiliza para sustitución de instrucciones.
- ▶ Es decir si dentro de un script aparece el texto `" 'comando'"` entonces se ejecutará lo orden que está entre las `""`
- ▶ Ejemplo:

```
$ LISTA='ls '  
$ echo $LISTA # Lista los archivos  
hola.sh leer.sh
```

- ▶ Otra forma de realizar la sustitución de comandos: **`$(comando)`**

```
$ LISTA=$(ls)  
$ echo $LISTA  
hola.sh leer.sh  
  
$ls $(pwd)
```

Operadores aritméticos

► Operadores:

- + suma
- - resta
- * multiplicación
- / división
- ** exponenciación
- % módulo

► Ejemplo:

```
$ a=(5+2)*3  
$ echo $a  
$ b=2**3  
$ echo $a+$b
```

Evaluación aritmética

- La instrucción **let** se puede utilizar para realizar funciones matemáticas:

```
$ let X=10+2*7
$ echo $X
24
$ let Y=X+2*4      # No es necesario usar $X
$ echo $Y
32
```

- Una expresión aritmética se puede evaluar con `$(expression)` o ``${expression}``

```
$ echo $((123+20))
143
$ VALOR=$((123+20))
$ echo $[123*$VALOR]
1430
$ echo ${2**3}
8
$ echo ${8%3}
```


Ejemplo Script Evaluación Aritmética

```
#!/bin/bash
echo -n "Introduzca un primer número: "; read x
echo -n "Introduzca un segundo número : "; read y
suma=$(( $x + $y ))
resta=$(( $x - $y ))
mul=$(( $x * $y ))
div=$(( $x / $y ))
mod=$(( $x % $y ))
# imprimimos las respuestas:
echo "Suma: $suma"
echo "Resta: $resta"
echo "Multiplicación: $mul"
echo "División: $div"
echo "Módulo: $mod"
```

Estructuras de control: Condicional

- La forma más básica es:

```
if [ expresión ]; then
    instrucciones
elif [ expresión ]; then
    instrucciones
else
    instrucciones
fi
```

- Las secciones elif(`else if`) y `else` son opcionales

Expresiones

- ▶ Una expresión puede ser:
 - ▶ comparación de cadenas,
 - ▶ comparación numérica,
 - ▶ operadores de fichero y
 - ▶ operadores lógicos y
- ▶ Se representa mediante [expresión]:

Expresionea de Comparación de Cadenas

► Operadores:

- =
- !=
- -n evalúa si la longitud de la cadena es superior a 0
- -z evalúa si la longitud de la cadena es igual a 0

► Ejemplos:

```
[ s1 = s2 ]      #(true si s1 es igual a s2, sino false)
[ s1 != s2 ]     #(true si s1 no es igual a s2, sino false
)
[ s1 ]           #(true si s1 no está vacía, sino false)
[ -n s1 ]        #(true si s1 tiene longitud mayor que 0,
sino false)
[ -z s2 ]        #(true si s2 tiene longitud 0, sino false
)
```

Ejemplo Comparación Cadenas

```
#!/bin/bash

echo -n "Introduzca su nombre de usuario: "
read login

if [ "$login" = "$USER" ]; then
    echo "Hola, $login. Cómo está hoy?"
else
    echo "Tú no eres $login!!!"
fi
```

Expresiones de Comparación numérica

► Operadores:

- -eq
- -ge
- -le
- -ne
- -gt
- -lt

► Ejemplos:

```
[ n1 -eq n2 ]  
[ n1 -ge n2 ]  
[ n1 -le n2 ]  
[ n1 -ne n2 ]  
[ n1 -gt n2 ]  
[ n1 -lt n2 ]
```

Ejemplo Comparación numérica

```
#!/bin/bash

echo -n "Introduzca un número 1 < x < 10: "
read num

if [ "$num" -lt 10 ]; then
    if [ "$num" -gt 1 ]; then
        echo "$num*$num=$(($num*$num))"
    else
        echo "Número fuera de rango !"
    fi
else
    echo "Número fuera de rango !"
fi
```

Expresiones para Comparación en Archivos

► Operadores de archivos:

- -d verifica si el path dado es un directorio
- -f verifica si el path dado es un archivo
- -s verifica si el path dado es un link simbólico
- -e verifica si el fichero existe
- -s verifica si el fichero tiene un tamaño mayor a 0
- -r verifica si el fichero tiene permiso de lectura
- -w verifica si el fichero tiene permiso de escritura
- -x verifica si el fichero tiene permiso de ejecución

► Ejemplos:

```
[ -d nombre_fichero ]  
[ -f nombre_fichero ]  
[ -e nombre_fichero ]  
[ -s nombre_fichero ]  
[ -r nombre_fichero ]  
[ -w nombre_fichero ]  
[ -x nombre_fichero ]
```


Ejemplo Comparaciones con Archivos

```
#!/bin/bash

if [ -f /etc/fstab ]; then
    cp /etc/fstab .
    echo "Hecho."
else
    echo "Archivo /etc/fstab no existe."
    exit 1
fi
```

Otros Operadores Lógicos

- ▶ Operadores lógicos:

- ▶ NOT: !

- ▶ AND:

- ▶ -a

- ▶ &&

- ▶ OR:

- ▶ -o

- ▶ ||

- ▶ Ejemplo:

```
#!/bin/bash
echo -n "Introduzca un número entre 1 < x < 10:"
read num

if [ "$num" -gt 1 -a "$num" -lt 10 ]; then
    echo "$num*$num=$(( $num*$num ))"
else
    echo "Número introducido incorrecto !"
fi
```

Parámetros de la shell

- ▶ Parámetros Posicionales:
 - ▶ Se asignan desde la shell cuando se invoca.
 - ▶ Parámetro posicional "N" se referencia como "\${N}", o "\$N" cuando "N" lo forma un sólo dígito
- ▶ Parámetros especiales:
 - ▶ \$# número de parámetros pasados
 - ▶ \$0 devuelve el nombre del shell script que se está ejecutando y su ubicación en el sistema de archivos
 - ▶ \$* devuelve en una cadena de caracteres todos los parámetros pasados al script
 - ▶ @\$ devuelve un array con los parámetros pasados al script
- ▶ Ejemplo (parametros.sh):

```
#!/bin/bash
echo "$#; $0; $1; $2; $*; @$"
```

```
$ parametros.sh estudiante1 estudiante2
```

```
2; ./parametros.sh; estudiante1; estudiante2;
    estudiante1 estudiante2; estudiante1 estudiante2
```

Instrucción Case

► Sintaxis

```
case $var in
val1)
    instrucciones;;
val2)
    instrucciones;;
*)
    instrucciones;;
esac
```

► Ejemplo(case.sh)

```
#!/bin/bash
echo -n "Introduzca un número entre 1 < x < 3: "
read x
case $x in
1) echo "Valor de x es 1.";;
2) echo "Valor de x es 2.";;
3) echo "Valor de x es 3.";;
0 | 3) echo "Número incorrecto.";;
*) echo "Valor no reconocido.";;
esac
```

Estructura for

► Sintaxis:

```
for var in lista
do
    statements
done
```

► Ejemplo (for1.sh):

```
#!/bin/bash
let sum=0
for num in 1 2 3 4 5
do
    let "sum = $sum + $num"
done

echo $sum
```

Ejemplos for con cadenas

```
Ejemplo(for2.sh)
#!/bin/bash
for x in papel lapiz boli; do
    echo "El valor de la variable x es: $x"
    sleep 1
done
```

```
#!/bin/bash
for x in "papel A4" "lapiz STADTLER" "boli BIC"; do
    echo "El valor de la variable x es: $x"
    sleep 1
done
```

```
#!/bin/bash
lista="antonio luis maria pepa"
for x in $listado
    echo "El valor de la variable x es: $x"
    sleep 1
done
```

Ejemplos for con manejo de archivos

```
#!/bin/bash
# Lista todos los ficheros del directorio actual
for x in * do
    ls -l "$x"
    sleep 1
done
```

```
#!/bin/bash
# Lista todos los ficheros del directorio /bin
for x in /bin do
    ls -l "$x"
done
```

Ejemplo for usando comandos del sistema

```
#!/bin/bash

read -p "Introduzca el nombre de un directorio: " directorio

echo "enlaces simbólicos en el directorio $directorio "

for fichero in $( find $directorio -type l ) do
    echo "$fichero"
done
```


Arrays con bucles

- Crear un array:

```
mascota[0]=perro  
mascota[1]=gato  
mascota[2]=pez  
pet=( perro gato pez )
```

- Longitud máxima de un array son 1024 elementos.
- Para extraer una entrada del array `${array[i]}`

```
$ echo ${mascota[0]}  
perro  
$ echo ${mascota[2]}  
pez
```

Otras operaciones con Arrays

- ▶ Para extraer todos los elementos se utiliza un asterisco:

```
echo ${array[*]}
```

- ▶ Para saber cuántos elementos hay en el array:

```
echo ${#array[*]}
```

- ▶ Podemos combinar los arrays con bucles utilizando for:

```
for x in ${array[*]} do  
    echo ${array[$x]}  
done
```

Estructura tipo C alternativa para for

► Sintaxis:

```
for (( EXPR1 ; EXPR2 ; EXPR3 )) do
    instrucciones
done
```

► Ejemplo(for8.sh)

```
#!/bin/bash
echo "Introduzca un número: "; read x

let sum=0
for (( i=1 ; $i<$x ; i=$i+1 )) ; do
    let "sum = $sum + $i"
done

echo "La suma de los primeros $x números es: $sum"
```

Estructura while

► Sintaxis:

```
while expresion_evalua_a_true do
    instrucciones
done
```

► Ejemplo(while.sh):

```
#!/bin/bash
echo -n "Introduzca un número: "; read x

let sum=0; let i=1
while [ $i -le $x ]; do
    let "sum = $sum + $i"
    let "i = $i + 1"
done

echo "La suma de los primeros $x números es: $sum"
```

Estructura until

► Sintaxis:

```
until [expression_evalua_a_true] do
    instrucciones
done
```

► Ejemplo(until.sh):

```
#!/bin/bash
echo "Introduzca un número: "; read x
echo ;
until [ "$x" -le 0 ]; do
    echo $x
    x=$(( $x - 1 ))
    sleep 1
done
echo ; echo FIN
```