

Programación con BASH Shell Scripts



Inicialización y terminación

- En función de la configuración del shell se pueden ejecutar al inicio de sesión los siguientes scripts:
 - /etc/profile
 - /etc/bashrc
 - ~/.bash_profile ~ es equivalente a \$HOME
 - ~/.bash_login
 - ~/.profile
- Se pueden ejecutar los siguientes scripts al salir del sistema con exit
 - ~/.bashrc
 - ~/.bash_logout

Ejemplo de ./bash_profile

```
# ~/.bash_profile: executed by bash(1) for login shells.  
# see /usr/share/doc/bash/examples/startup-files for  
examples.  
# the files are located in the bash-doc package.  
umask 022  
# the rest of this file is commented out.  
# include .bashrc if it exists  
if [ -f ~/.bashrc ]; then  
    source ~/.bashrc  
fi  
# set PATH so it includes user's private bin if it exists  
if [ -d ~/bin ] ; then  
    PATH=~/bin:"$PATH"  
fi  
PATH=.:$PATH  
PATH=$PATH:~/scripts  
alias ll='ls -laF -color=auto'  
  
#directorio donde esta instalado pvm  
export PVM_ROOT=/usr/lib/pvm3  
#directorio donde esta el demonio pvmd  
export PVM_DPATH=$PVM_ROOT/lib/pvmd  
#arquitectura del equipo  
export PVM_ARCH=LINUX  
PATH=$PATH:$PVM_ROOT/lib  
PATH=$PATH:$HOME/pvm3/bin/$PVM_ARCH
```

Nuestro primer script

- ¿Qué es un script?

Un shell script no es más que un conjunto de instrucciones y comandos ejecutables por Bash dentro de un fichero de texto al cual se le da permiso de ejecución.

- Utilizamos vi para crear el fichero saludo

```
echo "Hola mundo"
```

- Damos permisos de ejecución al fichero

```
chmod u+x saludo
```

- Ya podemos ejecutar el script

```
./saludo
```

Recordar diferencia entre ruta
absoluta y relativa

- El resultado es

```
Hola mundo
```

- Bash levanta un sub-shell para la ejecución de nuestro Shell Script.

Terminación y Exit Status

- El comando **exit** sirve para terminar un script y al igual que en C se puede utilizar para devolver un valor de terminación o retorno del script.
- Lo habitual es que un comando que termina sin errores devuelva 0 (éxito), mientras que si da error devuelve cualquier valor distinto de cero. Hay excepciones que dependen de la funcionalidad de los propios scripts.
- Las funciones definidas en nuestros scripts que retornen sin error devolverán 0 y cualquier otro valor para indicar error. Dependerá de la tarea que realice la función.
 - **exit nnn** donde nnn es valor entero en el rango de 0...255.
 - **exit** sin valor se devuelve el valor de retorno del último comando ejecutado en la función o script.
 - Con **\$?** se lee el valor devuelto por el último comando ejecutado.
 - Se puede utilizar también para el valor de retorno de las funciones

Terminación y Exit Status

Ejemplo:

```
echo hola
echo $? # Valor de retorno 0 comando correctamente ejecutado.
wiejd # intentamos ejecutar un comando inexistente.
echo $? # No-cero porque el comando no se ejecuto correctamente.
exit 113 # Termina el script dando como valor de retorno 113
          # Comprobarlo con echo $? al terminar el script.
```

Se pueden utilizar valores de retorno booleanos

```
true
echo "el valor de retorno de \"true\" = $" # 0
! true
echo "el valor de retorno de \"! true\" = $" # 1
false
echo "el valor de retorno de \"fasle\" = $" # 1
! false
echo "el valor de retorno de \"! false\" = $" # 0
```

Aquí hay un espacio

\"
Permite
mostrar las "

Programación con BASH Shell Scripts

Caracteres especiales

Caracteres especiales

- Un carácter es especial si tiene una interpretación para el shell o shell script diferente de su propio significado literal, es decir, si el Bash hace algo con él o lo tiene en cuenta como algo propio, tanto dentro de un script o fuera de él.

Comentarios,

- # comienza una línea de comentario
- En una línea tras # se considera comentario
- Excepción #! que indica con qué shell se ejecuta el script.
Ejemplo:#!/bin/bash
- No hay carácter fin de comentario

```
#Esta línea es un comentario  
echo "A continuación vienen un comentario." #Esto es un comentario  
#Esto también es un comentario
```

- El \ escapea el #. Las comillas anulan #

```
#Esta línea es un comentario  
echo "A continuación vienen un comentario." #Esto es un comentario  
#Esto también es un comentario  
echo "El # no representa un comentario por ir entre comillas dobles"  
echo 'El # no representa un comentario por ir entre comillas simples'  
echo El \# no representa un comentario por ir escapeado.  
echo El # Esto si que es un comentario pues no va ni escapeado ni entre comillas..
```

Otros usos de,

- Otras apariciones del carácter # tampoco son consideradas como comentarios por Bash, ya que dependiendo del contexto lo interpretará correctamente, por ejemplo:

```
echo ${PATH##*:} #Aquí Bash está realizando una sustitución de parámetros,  
#veremos esto más adelante.
```

```
echo $(( 2#101011 )) #Aquí está realizando una conversión numérica entre bases.
```

Separador de comandos, ;

- ; permite poner dos o más comandos en la misma línea

```
echo Hola; echo que tal?  
if [ -x "$fichero" ]; then #if y then necesitan separación  
    echo "El fichero $filename existe."; cp $fichero $fichero.bak  
else  
    echo "El fichero $fichero no existe."; touch $fichero  
fi;
```

Terminador case, ;;

- El punto y coma, ; cuando aparece repetido como aquí, ;; es el terminador de cada una de las opciones dentro de la construcción condicional case.

```
variable=rojo
case "$variable" in
    rojo) echo "\$variable = rojo" ;;
    verde) echo "\$variable = verde" ;;
esac
```

\\$ el \$ está escapeado no realiza la sustitución de variable

Punto, .

- Comando . o “dot command” actúa como comando **source** pero es más portable.
- El comando **source** o . cuando se ejecuta desde la línea de comandos sirve para ejecutar un script.
- Cuando se ejecuta desde dentro de un script realiza la inserción del fichero que le sigue en el propio script, tal como hace la sentencia #include de C.

```
#!/bin/bash
. data?fichero #Incluye un fichero de datos
```

- El fichero a incluir debe estar en el directorio de trabajo, **pwd**.
- Otra interpretación, el fichero cuyo nombre empiece por . será un fichero oculto
- Cuando hablamos de directorios
 - . es el directorio actual
 - .. es el directorio padre del actual
- Cuando hablamos de expresiones regulares
 - . es el sustituto de un único carácter.

Comillas, “ ”

■ Entrecomillado parcial utilizando comillas dobles “ ”

- Con las “ ” se definen cadenas de caracteres
- Entre “ ” se anula el significado de la mayoría de los caracteres especiales, pero no todos

```
echo "El # no representa un comentario por ir entre comillas dobles"  
echo "esto es una cadena de caracteres y $variable es el color elegido"
```

■ Entrecomillado total utilizando comillas simples ’ ’

- También definen cadenas de caracteres pero anulan todos los caracteres especiales

```
echo 'esto es una cadena de caracteres y $variable es simplemente eso, $variable'
```

Coma, ,

- La coma combina una serie de operaciones aritméticas.
- Todas son evaluadas, pero sólo la última es retornada.
- Esto no es muy útil salvo que tenga efectos laterales alguna de las operaciones intermedias como en el ejemplo donde se declara y define el valor de la variable a, pero en general no se usa mucho.

```
$let "j = ((a = 2, 15 * 3))" #Define "a" y calcula "j".  
$echo $j                         #Muestra el valor de j  
$45                                #que es 15*3
```

Escape, \

- El carácter de escape \ es el mecanismo que Bash tiene para anular el significado de un carácter especial, lo que en la mayoría de los casos tiene el mismo efecto que entrecomillar simple un único carácter, ya que como hemos visto el entrecomillado simple anula los caracteres de escape.
- Por ejemplo: \X “escapea” el carácter X. Tiene el mismo efecto que 'X'

Barra, /

- Es el símbolo para la división.
- Pero también actúa como separador de los componentes de un nombre de fichero.

```
/home/alu/fichero.txt
```

Sustitución de comandos, `

- Permite la ejecución embebida de un comando
- Colocando el comando y sus parámetros entre comillas invertidas se permite devolver el resultado de éste para su utilización o para su asignación a una variable.
- `comando`

```
a=`echo "Hola"`
echo $a
```

Dos puntos, :

- Los dos puntos son para el Bash el comando nulo, lo que es equivalente a No-Operation (NOP)
- También se considera como un sinónimo del valor true para el shell.

```
:  
echo $? # 0
```

```
#Bucle sin fin con el comando nulo (true)  
while :  
do  
    operacion-1  
    operacion-2  
    ...  
    operacion-n  
done
```

```
if condicion  
then :           #Asi queda claro que no hay que hacer nada si se cumple  
else  
    comandos   #lo que corresponda en caso contrario.  
fi
```

Dos puntos, :

- Cuando queremos utilizar la sustitución de comandos para asignar un valor a una variable en una línea, sin ellos el bash lo interpretará como un comando y al no existir dará un error:
 : \${minombre='whoami'}
 # \${minombre='whoami'} |sin los : da error
- En la sustitución de parámetros:
 : \${HOSTNAME?} \${USER?} \${MAIL?}
 #Si las variables no están definidas imprime un error y termina el script
- Los : en combinación con la redirección > truncan la longitud de un fichero a cero, sin cambiar sus permisos, y si el fichero no existe lo crea, con los permisos por defecto.
 :> fichero.txt #Borra el contenido del fichero 'fichero.txt'
- Tiene el mismo efecto que **cat /dev/null >fichero.txt** pero es más corto y además no lanza un nuevo proceso ya que los : son un comando embebido del shell.
- Como separador de campos

```
$ echo $PATH  
/usr/local/bin:/bin:/usr/bin:/usr/X11R6/bin:/sbin:/usr/sbin:/usr/games
```

Negador, !

- Niega el sentido de un test o del valor de retorno de una expresión.
- El operador ! Invierte el valor de retorno del comando al que se le aplica y el sentido del operador test.
- Por ejemplo, se puede cambiar el sentido del igual = por distinto !=
- Se trata de una palabra reservada del bash.

Asterisco, *

- El * es un carácter de sustitución, un comodín.
- El solo, sin estar en una expresión regular, significa “*todos los ficheros de un directorio*”.
- Este uso, el de realizar expansión de nombres de fichero, se llama Globbing, se verá más adelante, pero por adelantar, viene a ser la sustitución habitual de nombres de fichero para los que manejan el ya antiguo DOS.
- Así por ejemplo, esto muestra los nombres de todos los ficheros del directorio actual.
- Cuando se utiliza el * dentro de una expresión regular representa cualquier número de caracteres (incluso cero repeticiones)
- También es el operador aritmético del producto y un doble asterisco ** es el operador aritmético de la potenciación.

```
$ echo *
practica.sh script1.sh script2.sh x
```

Operador de test, ?

- Dentro de algunas expresiones el carácter ? representa una condición de test.
- Por ejemplo, dentro del operador (()) el ? funciona como el operador condicional inline de C
- En expresiones regulares sustituye a un único carácter

```
a=8  
(( j = a<1000?10:5 ))      #If in-line estilo C  
echo "j=$j"
```

```
if a < 1000  
then j = 10  
else j = 5
```

Sustitución de variables, \$

```
var1=5  
var2=hola  
echo $var1 # 5  
echo $var2 # hola
```

- Cuando \$ precede al nombre de una variable significa el valor de esa variable.
- Cuando se usa en una expresión regular \$ significa el final de la línea.
- También se utiliza como el operador que da comienzo a la sustitución de parámetros. Se verá más adelante \${}.

```
mivar=casa  
echo El plural de $mivar es $mivars.  
echo El plural de $mivar es ${mivar}s
```

Parámetros posicionales, \$

■ **\$0, \$1**

- Son los parámetros posicionales que se pasan de la línea de comandos al script o a una función.

■ **\$0 es el propio comando o nombre de función.**

■ **\$1 es el primer parámetro, es decir el que ocupa la primera posición. \$2 el que ocupa la segunda ...**

- No siempre se puede asegurar que \$1 valga lo mismo, pues se pueden desplazar los parámetros hacia la izquierda y entonces \$1 pasa a tomar el valor del que antes era \$2

Parámetros posicionales, \$

■ \$#

- Indica el número de caracteres posicionales que se han pasado al script o a la función.
- Aquí hay que tener en cuenta que este número no tiene en cuenta el \$0. “numpar 1 2 3” devolverá el valor 3 puesto que se le han pasado tres parámetros

```
#numpar  
echo $#
```

■ \$*

- Devuelve un string con el valor de todos los parámetros posicionales.

■ \$@

- Lo mismo que \$* pero donde cada parámetro es un string individual.

■ \$_

- Retorna el último comando ejecutado

Parámetros posicionales, \$

■ \$?

- Es el valor de retorno del último comando o función ejecutado. Esto ya lo hemos visto.

■ \$\$

- Contiene el PID del script.
- El PID es el “Process ID” o identificador de proceso.
- Todos los procesos en un entorno unix-like tienen asignado en ejecución un PID por el cual podemos referenciar al proceso para por ejemplo comunicarnos con él

■ \$!

- Contiene el PID del último job lanzado en background.

Paréntesis, ()

- Lo que generan los paréntesis es una lista de comandos.
- La lista puede estar formada incluso por un único comando. `(a=hola; echo $a)`
- Se genera un nuevo proceso Shell que será el que ejecute los comandos de la lista.
- Las variables dentro de un subshell no son visibles fuera
- El proceso padre (el script) no puede leer variables creadas por su proceso hijo (el subshell).
- Cuando se lanza un shell o un subshell éste hereda las variables del shell que lo lanzó.

```
a=abc
( a=xyz; )
echo "a = $a"      #a = abc
                   #la a de dentro de los paréntesis
                   #es una variable local del subshell
                   #que se genera por los parentesis
```

```
a=5
(b=$a; echo "b=$b") #b=5 hereda el valor de la variable a
echo "a=$a"
```

- Paréntesis para la inicialización de arrays

```
array = (elemento1, elemento2, elemento3)
```

Llaves, { }

■ Expansión de llaves

```
grep onofre file*.{txt,htm*}  
# Localiza todas las instancias de la palabra onfore  
# en los ficheros "fileA.txt", "file2.txt", "files.html", "file-67.htm", etc.
```

- El comando actuará para todos los valores de sustitución separados por comas que se encuentren entre las llaves.
- Buscará la palabra en todos los ficheros que empiecen por “file” seguido de lo que sea hasta el punto y tengan cualquiera de las terminaciones que se encuentran entre las llaves, además nos fijamos que la terminación htm* puede ser htm o html gracias al asterisco.
- En el ejemplo, las terminaciones, son valores de sustitución y se separan por comas.

Llaves, { }

■ Bloque de código in-line

- Las llaves encierran un bloque de código también llamado bloque in-line.
- Crea una función anónima, una función sin nombre, sin embargo, a diferencia de una función, las variables declaradas o definidas en un **bloque in-line** son visibles para el resto del script.

```
saludo=hola
{ saludo=adios; }
echo "saludo = $saludo" # saludo=adios que es el valor dado dentro del bloque in-line
```

- { } No lanzan un subshell para ejecutar el código que se encuentra entre ellas, lo ejecuta el shell actual.
- El bloque in-line puede tener redirecciones de entrada y salida de él

```
#!/bin/bash
# Lectura de líneas de /etc/passwd
File=/etc/passwd #asignamos la variable File
{ read linea1
  read linea2
} < $File
echo "La primera línea de $File es: $linea1"
echo "La segunda línea de $File es: $linea2"
exit 0
```

Corchetes, []

■ Operadores de test [] y [[]]

- Los corchetes establecen las condiciones de test.
- [] Los corchetes simples definen dichas condiciones de test.
- Los dobles corchetes [[]] son una versión más robusta del operador de test, ya que permite que dentro de ellos se puedan utilizar los operadores lógicos &&, ||, <,>, etc. . .

```
file=/etc/passwd
if [[ -e $file ]]; then
    echo "El fichero de passwords existe"
fi
```

■ Delimitador de rango []

- Otro uso del carácter corchete es como parte de una expresión regular, donde los corchetes delimitan un rango de caracteres de sustitución. Ejemplos de esto se verán cuando se hable de las expresiones regulares.

Evaluador entero, (())

- Expande y evalúa expresiones enteras en su interior.
- Similar al comando `let`.
- Ya hemos hablado de el cuando poníamos este ejemplo

```
a=8  
(( j = a<1000?10:5 ))      #If in-line estilo C  
echo "j=$j"
```

Redirección,

`scriptfile >filename` redirige la salida del script al fichero filename.
`command &>filename` redirige tanto la salida stdout como la stderr al fichero.
`command >&2` redirige la salida stdout del comando a stderr
`scriptname >> filename` añade la salida del script al filename.
`command << filename` proporciona la entrada al comando.

Comparador ascii, <, >

■ Comparador ascii

- Los caracteres < y > se utilizan para comparar alfabéticamente cadenas de caracteres.
- Para comparar elementos numéricos no se utilizan estos caracteres.

```
veg1=lechuga  
veg2=tomates  
if [[ "$veg1" < "$veg2" ]]; then  
    echo "$veg1 precede a $veg2"  
else  
    echo "$veg2 precede a $veg1"  
fi
```

■ Limitador de palabras.

- Actúan de limitadores de palabras en las expresiones regulares.

```
grep '\<the\>' myfile
```

Tubería, pipe

- Pasa la salida del comando previo a la entrada del comando posterior.

```
cat mifichero.txt | more  
cat *.cli | sort | uniq
```

- También se puede entubar la salida de un comando hacia un script.

- Script:

```
tr 'a-z' 'A-Z' # Los rangos de letras deben ir entre comilla simple  
exit 0
```

- Salida:

```
mmartinez@ideafix:~/scripts/intro02$ ls -l | ./mayusculas.sh  
TOTAL 56  
-RW-R--R-- 1 MMARTINE MMARTINE 1058 AUG 6 16:58 LISTA.TXT  
-RWXR--R-- 1 MMARTINE MMARTINE 22 OCT 4 11:11 MAYUSCULAS.SH  
-RWX----- 1 MMARTINE MMARTINE 1994 AUG 6 19:02 MENU  
-RWXR--R-- 1 MMARTINE MMARTINE 0 OCT 4 11:10 MS  
-RW-R--R-- 1 MMARTINE MMARTINE 0 AUG 13 16:41 MSVI
```

Operadores Lógicos

- El operador de test debe ser [[]]
- OR ||
 - En operación de test el operador devuelve 0 (éxito) si alguna de las dos partes de la operación es true.
- AND &&
 - En operación de test el operador devuelve 0 (éxito) cuando los dos operadores implicados son true.

Ejecución en segundo plano, &

- Un comando seguido de & se ejecutará en segundo plano.
- La ejecución en segundo plano se puede extender a los scripts

```
for i in 1 2 3 4 5 6 7 8 9 10
do
    echo -n "$i "
    sleep 1
done &
echo "Esto va en primer plano"
exit 0
```

Operadores, +, -, =, %, ~, ~+

- + Suma
- - Resta
- =
 - Asignación `a=28
echo $a # 28`
 - En una condición de test = es el comparador de strings
- %
 - Módulo aritmético (resto de la división)
 - En expresiones regulares % es operador de concordancia de patrones
- ~ Equivale a \$HOME
- ~+ Equivale a \$PWD

Programación con BASH Shell Scripts

Introducción a variables y parámetros

Variables y parámetros

- El nombre de una variable se puede ver como el lugar donde se almacena su valor.
- Referenciar el valor de una variable se denomina *variable substitution*.
- Clasificaciones de variables
 - Por si son o no heredables por los subshells
 - De entorno (son heredables)
 - Locales (no son heredables)
 - Por quién las define
 - Predefinidas (\$HOME, \$PATH, \$PWD ...)
 - Definidas por el usuario

Variables y parámetros

- El operador \$ realiza la sustitución de variables, obtener su valor.
 - Incluso cuando se encuentra entre comillas dobles “\$HOME”.
 - No entre comillas simples '\$HOME'
- En un shell script una variable aparece sin el \$ cuando
 - Se declara por primera vez
 - Se le asigna un valor
 - Se exporta
 - Representa una señal (comunicación entre procesos, signals).

Variables y parámetros

■ Se puede dar valor a una variable

- Con el operador de asignación =
- Con el comando **read**
- Con la construcción de bucle
 - `for z in 1 2 3`

No tiene porque estar
definida antes del bucle

Variables y parámetros

- *\$variable* es una forma simplificada de *\${variable}`*.

```
mivar=casa  
echo El plural de $mivar es $mivars.  
echo El plural de $mivar es ${mivar}s
```

```
a=375  
hola=$a  
echo hola # No es una variable, sólo la cadena hola.  
echo $hola  
echo ${hola} # Igual que el anterior.  
echo "$hola"  
echo "${hola}" #Igual que el anterior.  
echo  
echo '$hola' # Muestra $hola  
# Referencia a la variable deshabilitada por las comillas simples,
```

Variables y parámetros

■ Varias variables en una línea

```
var1=variable1 var2=variable2 var3=variable3  
echo  
echo "var1=$var1 var2=$var2 var3=$var3"
```

■ Una variable un valor “compuesto”

```
numeros="uno dos tres"  
numericos="1 2 3"  
# Si hay espacios entre variables, entonces las comillas son necesarias.  
echo "numeros = $numeros"
```

Variables y parámetros

- Una variable no inicializada tiene el valor NULL
- Se puede declarar variables pero no asignarles valor
- Se pueden “des-declarar” variables
- Se puede sumar a variables no inicializadas
 - En operaciones aritméticas se trata como 0 (no portable)

```
echo "variable_no_inicializada = $variable_no_inicializada"
variable_no_inicializada=          #declarada, pero no inicializada
echo "variable_no_inicializada = $variable_no_inicializada" # linea en blanco
                                         # Todavía tiene valor null.
variable_no_inicializada=23        # asignada, ya tiene valor
unset variable_no_inicializada   # Ya no tiene valor ni esta declarada
echo "$variable_no_inicializada" # linea en blanco
let "variable_no_inicializada += 5" # le suma 5.
echo "$variable_no_inicializada" # 5
```

Variables y parámetros

■ Asignaciones directas y con let

```
a=879      # Asignacion directa
echo "El valor de \"a\" es $a."
let a=16+5  # Asignacion usando 'let'
echo "El valor de \"a\" es ahora $a."
```

- El comando let nos permite evaluar expresiones aritméticas y realizar las asignaciones de éstas a las variables.

■ En bucle

```
echo -n "Valores de \"a\" en el bucle: "
for a in 7 8 9 11
do
    echo -n "$a "
done
echo
echo "Valor de \"a\" fuera del bucle: $a"
```

Valores de "a" en el bucle: 7 8 9 11

Valor de "a" fuera del bucle: 11

Variables y parámetros

■ Asignaciones mediante **read**

```
echo -n "Introduzca el valor de \"a\" "
read a
echo "El valor de \"a\" es $a."
```

■ Asignación de la ejecución de un comando

```
a='echo Hola!' # Asigna el resultado del comando 'echo' a 'a'
echo $a
a='ls -l'      # Asigna el resultado del comando 'ls -l' a 'a'
echo $a
echo
echo "$a"
```

```
Hola!
total 60
-rw-r--r-- 1 mmartine mmartine 1058 Aug  6 16:58 lista.txt
mmartine mmartine 22 Oct 4 11:11 mayusculas.sh
-rwx----- 1 mmartine
994 Aug  6 19:02 menu
-rwxr--r-- 1 mmartine mmartine 161 Oct  4 12:24
-- 1 mmartine mmartine 0 Aug 13 16:41 msvi
-rwxr--r-- 1 mmartine mmaug 6 17:22 sc1
-mmartine mmartine 253 Aug 6 19:03 sc11
-rwxr--r-- 1 mmartine mmarti6 18:28 sc2
-mmartine mmartine 537 Aug 6 18:29 sc3
-rwxr--r-- 1 mmartine mmartartine 139 Aug 6 18:32 sc4
-mmartine mmartine 8:32 sc5
-rwxr--r-- 1 mmartine mmartine 168 Aug 6 18:33 sc6
-mmartine mmartine 255 Aug 6 18:41 sc7
-rwxr--r-- 1 mmartine mmartine 305 Aug 6 18:34 sc8
4 sc8
-rwxr--r-- 1 mmartine mmaartine 422 Aug 6 18:36 sc9
```

total 60						
-rw-r--r--	1	mmartine	mmartine	1058	Aug	6 16:58 lista.txt
-rwxr--r--	1	mmartine	mmartine	22	Oct	4 11:11 mayusculas.sh
-rwx-----	1	mmartine	mmartine	1994	Aug	6 19:02 menu
-rwxr--r--	1	mmartine	mmartine	161	Oct	4 12:24 ms
-rw-r--r--	1	mmartine	mmartine	0	Aug	13 16:41 msvi
-rwxr--r--	1	mmartine	mmartine	226	Aug	6 17:22 sc1
-rwxr--r--	1	mmartine	mmartine	453	Aug	6 18:37 sc10
-rwxr--r--	1	mmartine	mmartine	253	Aug	6 19:03 sc11
-rwxr--r--	1	mmartine	mmartine	461	Aug	6 18:28 sc2
-rwxr--r--	1	mmartine	mmartine	537	Aug	6 18:29 sc3
-rwxr--r--	1	mmartine	mmartine	139	Aug	6 18:32 sc4
-rwxr--r--	1	mmartine	mmartine	178	Aug	6 18:32 sc5
-rwxr--r--	1	mmartine	mmartine	168	Aug	6 18:33 sc6
-rwxr--r--	1	mmartine	mmartine	255	Aug	6 18:41 sc7
-rwxr--r--	1	mmartine	mmartine	305	Aug	6 18:34 sc8
-rwxr--r--	1	mmartine	mmartine	422	Aug	6 18:36 sc9

Variábles y parámetros

■ Asignaciones mediante sustitución de comandos y variables

```
arquitectura=$(uname -m)
```

- Primero se ejecutan los comandos entre paréntesis
- Despues se asigna el resultado como si fuera una variable

Variables y parámetros

■ Tipificación de variables

- No están tipadas
- Para el Shell las variables son cadenas de caracteres
- En función del contexto operará con ellas de una forma u otra

```
a=2334      # Entero.  
let "a += 1"  
echo "a = $a" # a = 2335  
echo # Entero.  
b=${a/23/BB} # Substituye 23 por "BB".  
              # Esto transforma $b en una cadena.  
echo "b = $b" # b = BB35  
declare -i b  # Aunque se declare como entero  
              # la seguirá tratando como alfanumérica  
echo "b = $b" # b = BB35  
let "b += 1" # BB35 + 1 =  
echo "b = $b" # b = 1  
echo  
  
c=BB34  
echo "c = $c" # c = BB34  
d=${c/BB/23} # substituye "BB" por "23".  
              # Esto hace $d un entero.  
echo "d = $d" # d = 2334  
let "d += 1" # 2334 + 1 =  
echo "d = $d" # d = 2335  
echo
```

Variables y parámetros

■ Tipificación de variables

- ¿Qué pasa con las variables NULL?

```
e=""  
echo "e = $e" # e =  
let "e += 1" # Operaciones aritmeticas permitidas sobre una variable null?  
echo "e = $e" # e = 1  
echo          # Variable null transformada en un entero.  
              # Que pasa con las variables no declaradas?  
echo "f = $f" # f =  
let  "f += 1" # Operaciones aritmeticas permitidas?  
echo "f = $f" # f = 1  
echo          # Variable no declarada transformada en un entero.
```

Variables y parámetros

- Las variables son de tipo string por defecto
 - Con **declare** o **typeset** se puede forzar el tipo de la variable.
(**declare +v2.0**) (**typeset ksh**)
 - Readonly
 - **declare -r var**
 - Integer
 - **declare -i var**
 - Array
 - **declare -a array**
 - Functions
 - **declare -f**
 - Export
 - **declare -x var**
 - **declare -x var=\$value**
- ```
#!/bin/bash
saludo()
{
 echo Hola estoy dentro de la función
}
declare -f #Lista el código de la función, la única que hay

declare -i v1 #declara v1 como entera
v1=1234 #le asigna un valor

v1=var1+1 #No hace falta let
v1=2367.1 #Devuelve un error por intentar asignar un numero
 #en punto flotante a la variable. La deja inalterada.

declare -r v2=22.13 #Declara la constante con un valor
v2=18.23 #Da error al intentar modificar el valor
 #y además sale del script
```

# Variables y parámetros

## ■ Variables de Entorno

- Definidas para el conjunto del Shell
- Afectan a la ejecución de comandos y scripts.
- Cada proceso tiene su entorno de ejecución con acceso a las variables a las que puede acceder de forma global.
- Las variables pueden ser modificadas por comandos, scripts o funciones.
- Un script hereda las variables del shell que lo lanza.
- Para colocar una variable en el entorno se debe exportar.
  - Si un script llama a otros que deben ver sus variables, éstas deben estar en su entorno, por lo que tiene que exportarlas para que sean heredadas y por tanto utilizables.
  - Lo mismo ocurre a nivel de shell.

# Variábles y parámetros

## ■ Variables de Entorno

- Para definir una variable en el entorno del shell que llama a un script hay que utilizar la salida estandar del script en un export
- Sea el script ms.sh
- Se puede declarar a nivel del shell una variable así:

```
export a='ms'
```

```
a='uname'
echo "$a"
```

# Variables y parámetros

## ■ Variables de Entorno

- **BASH**

- Contiene el path del shell actual

```
$ echo $BASH
/bin/bash
```

- **BASH\_ENV**

- Variable de entorno que sirve para apuntar a un script de arranque del bash que se leerá cuando se lance un script.
  - Se lanzará con la ejecución de cada nuevo shell y con la ejecución de un script.

- **BASH\_VERSION**

- La versión del Bash instalada en el sistema

```
$ echo $BASH_VERSION
2.05a.0(1)-release
```

# Variables y parámetros

## ■ Variables de Entorno

- **BASH\_VERSINFO [n]**

- Un array con información de la versión del shell instalada en cada uno de sus elementos.
- Es similar a la anterior pero más detallada.

```
Bash version info:
for n in 0 1 2 3 4 5
do
echo "BASH_VERSINFO[$n] = ${BASH_VERSINFO[$n]}"
done
```

```
BASH_VERSINFO[0] = 2 # Major version no.
BASH_VERSINFO[1] = 05a # Minor version no.
BASH_VERSINFO[2] = 0 # Patch level.
BASH_VERSINFO[3] = 1 # Build version.
BASH_VERSINFO[4] = release # Release status.
BASH_VERSINFO[5] = i386-pc-linux-gnu # Architecture
(same as $MACHTYPE).
```

# Variables y parámetros

## ■ Variables de Entorno

- EDITOR

- El editor invocado por un script, usualmente vi, vim, emacs.

- EUID

- El identificador efectivo del usuario.
  - El UID (User ID) es el identificador real del usuario y sirve para determinar el usuario responsable de la creación del proceso (comando, o script que se lanza, quién lo lanza).
  - El EUID (Efective User ID) es el identificador efectivo de usuario, que en principio es igual que el UID, pero que puede ser cambiado si el programa está marcado con el bit SUID.
  - El EUID es el que sirve para determinar los derechos en ejecución que tiene el proceso, independientemente de los derechos que tenga el usuario que lanzó el proceso.

# Variábles y parámetros

## ■ Variables de Entorno

- FUNCNAME

- Nombre de la función que actualmente está en ejecución.

```
xyz23 ()
{
 echo "Función $FUNCNAME en ejecución."
}
```

```
xyz23
```

```
echo "FUNCNAME = $FUNCNAME" # FUNCNAME =
Retorna Null fuera de una función.
```

# Variables y parámetros

## ■ Variables de Entorno

- HOME

- Directorio home, raíz del usuario actual, habitualmente suele ser /home/usuario.

- HOSTNAME

- Contiene el nombre del sistema.

- HOSTTYPE

- Contiene el tipo de host, arquitectura del procesador

```
$ echo $HOSTTYPE
i686
```

# Variábles y parámetros

## ■ Variables de Entorno

- IFS

- Internal Field Separator.
- Determina como el Bash reconoce los campos, o los límites entre las palabras cuando interpreta las cadenas de caracteres.
- \$IFS por defecto es un espacio en blanco, pero puede ser cambiado por ejemplo para procesar un fichero separado por comas.
- El parámetro \${\*} utiliza el primer carácter definido en \$IFS.

- LINENO

- Esta variable contiene el número de línea actual que está ejecutando el script.
- Sólo es útil para propósitos de debug pues únicamente tiene valor en ejecución del propio script.

```
*** BEGIN DEBUG BLOCK ***
last_cmd_arg=$_ # Save it.
echo "At line number $LINENO, variable \"v1\" = $v1"
echo "Last command argument processed = $last_cmd_arg"
*** END DEBUG BLOCK ***
```

# Variables y parámetros

## ■ Variables de Entorno

- OLDPWD
  - Anterior directorio pwd. Es el directorio en el que se ha estado anteriormente, del que se viene.
- OSTYPE
  - Tipo de sistema operativo.
- PATH
  - Contiene las rutas a los ficheros binarios, ejecutables.
  - Cuando se introduce un comando el shell automáticamente hace una búsqueda en los directorios listados en el path para localizar el ejecutable.
  - El Path está guardado formando una lista de directorios separados por el carácter :
  - Normalmente el sistema define el PATH en los ficheros /etc/profile o en >>/.bashrc
  - Con PATH=\${PATH}:/opt/bin se añade el directorio /opt/bin al path, en la última posición.

```
mmartinez@ideafix:~$ echo $OSTYPE
linux-gnu
```

# Variables y parámetros

## ■ Variables de Entorno

- PPID
  - El \$PPID de un proceso es el process ID (pid) del padre del proceso.
- PWD
  - Directorio actual, working directory (directorio en el que se encuentra el usuario).
  - En el siguiente ejemplo se muestra un fragmento de script que puede servir para comprobar que se encuentra en el directorio actual antes de proceder al borrado de ficheros.

```
E_NOPWD=66
MIDIR=/home/usuario1/proyecto
cd $MIDIR
echo "Borrando ficheros temporales de $MIDIR"
if ["$PWD" != "$MIDIR"] then # evitamos que se cambie de directorio accidentalmente
 echo "Directorio erróneo!"
 echo "En $PWD, en vez de en $MIDIR !"
 echo "Abandonando script"
 exit $E_NOPWD
fi
```

# Variables y parámetros

## ■ Variables de Entorno

### ● SECONDS

- El número de segundos que el script ha estado ejecutándose.
- Puede no ser muy exacto si la máquina está muy sobrecargada o se trata de una máquina lenta. No usar para controlar tareas en tiempo real.

```
TIME_LIMIT=10
INTERVAL=1
echo
echo "Pulsa Control-C para salir antes de $TIME_LIMIT segundos."
echo
while ["$SECONDS" -le "$TIME_LIMIT"]
do
 if ["$SECONDS" -eq 1]; then
 unidades=segundo
 else
 unidades=segundos
 fi
 echo "El script se ha ejecutado durante $SECONDS $unidades."
 sleep $INTERVAL
done
```

```
mmartinez@ideafix:~/scripts/intro02$ ms
Pulsa Control-C para salir antes de 10 segundos.
El script se ha ejecutado durante 0 segundos.
El script se ha ejecutado durante 1 segundo.
El script se ha ejecutado durante 2 segundos.
El script se ha ejecutado durante 3 segundos.
El script se ha ejecutado durante 4 segundos.
El script se ha ejecutado durante 5 segundos.
El script se ha ejecutado durante 6 segundos.
El script se ha ejecutado durante 7 segundos.
El script se ha ejecutado durante 8 segundos.
El script se ha ejecutado durante 9 segundos.
El script se ha ejecutado durante 10 segundos.
mmartinez@ideafix:~/scripts/intro02$
```

# Variables y parámetros

## ■ Variables de Entorno

### ● SHLVL

- El nivel de ejecución del Shell.
- Cada subshell incrementa en uno esta variable.
- Si a nivel de prompt (línea de comandos) el \$SHLVL es 1, entonces en un script el nivel se incrementa a 2.
- Veamos esto en un ejemplo, supongamos que tenemos un script que se llame level.sh con el siguiente contenido

```
echo "Nivel $SHLVL"
exit 0
```

- Ahora si desde la linea de comandos del shell tecleamos el siguiente comando echo  
`$ echo "Nivel fuera del script es $SHLVL y dentro del script tenemos el $(level.sh)"`
- Obtendremos la siguiente salida en la que vemos como dentro del script el nivel se incrementa.  
`Nivel fuera del script es 1 y dentro del script tenemos el Nivel 2`
- Esto sucede de igual forma en los subshells que se creen dentro del script, incrementando en cada uno el nivel de anidamiento.

# Variables y parámetros

## ■ Variables de Entorno

### ● TMOUT

- Si !=0 entonces el shell termina transcurrido el numero de segundos
- Sólo a partir de la versión 2.05b del shell.

```
Funciona solo en Bash, versiones 2.05b y posteriores
TMOUT=5 # time out en 5 segundos
echo "Cual es tu bebida favorita?"
echo "Rápido, solo tienes $TMOUT segundos para contestar!"
read bebida
if [-z "$bebida"]
then
 bebida="(eres un lento o estas borracho)"
fi
echo "Tu bebida favorita es $bebida."
```

# Variables y parámetros

## ■ Variables de Entorno

- TMOUT

- Para versiones anteriores se puede usar -t del comando read

```
TIMELIMIT=5
read -t $TIMELIMIT variable
echo
if [-z "$variable"]
then
 echo "Timed out, la variable sigue indefinida."
else
 echo "variable = $variable"
fi
```

# Variables y parámetros

## ■ Variables de Entorno

- TMOUT

- ... o trabajando con señales y capturando interrupciones

```
TIMELIMIT=5 #se puede pasar por parámetro

VerRespuesta()
{
 if ["$respuesta" = "TIMEOUT"]
 then
 echo $respuesta
 else
 echo "Tu bebida favorita es $respuesta"
 kill $! # Ya no se necesita la función TimerOn en background
 # $! es el PID del último proceso corriendo en background
 fi
}

TimerOn()
{
 sleep $TIMELIMIT && kill -s 14 $$ & #espera 5 segundos
 #luego envia la señal sigalarm al proceso $$

}

Int14Func()
{
 respuesta="TIMEOUT"
 VerRespuesta
 exit 14
}

trap Int14Func 14 #Definimos el procedimiento de tratamiento
 #para la señal 14

#Comienza el código
echo "Cuál es tu bebida favorita?"
TimerOn #Establece el temporizador
read respuesta #Lee la respuesta del teclado
VerRespuesta #Muestra la respuesta
exit 0
```

# Variables y parámetros

## ■ Variables de Entorno

### • UID

- Es el identificador del usuario
- Es una variable de solo lectura no modificable en ningún script
- Detectar si se es root o no

```
ROOT_UID=0 # Root has $UID 0.
if ["$UID" -eq "$ROOT_UID"]
then
 echo "Eres root."
else
 echo "Eres un usuario cualquiera"
fi
```

```
ROOTUSER_NAME=root
username='id -nu' # o también username='whoami'
if ["$username" = "$ROOTUSER_NAME"]
then
 echo "Eres root."
else
 echo "Eres un usuario cualquiera"
fi
```

# Parámetros posicionales

\$0 es el nombre del propio script

\$1, \$2, ... son los parámetros primero, segundo, etc.

Después del \$9 los parámetros deben ir entre llaves, \${10}, \${11} ...

\$\* devuelve en un único string todos los parámetros que se pasan.

\$@ devuelve cada parámetro en un string independiente.

\$\_ contiene el último comando ejecutado desde el shell.

- Partimos de esto, iremos analizando lo que se puede hacer con los parámetros posicionales

```
MINPARAMS=10
echo
echo "El nombre de este script es \"\$0\""
echo "El nombre de este script es \"`basename \$0`\""
echo
```

mmartinez@ideafix:~/scripts/intro02\$ ms

El nombre de este script es "./ms"  
El nombre de este script es "ms"

# Parámetros posicionales

```
if [-n "$1"]; then # La variable esta entrecomillada.
 echo "Parametro #1 is $1" #Son necesarias las comillas para mostrar el #
fi
if [-n "$2"]; then
 echo "Parametro #2 is $2"
fi
if [-n "$3"]; then
 echo "Parametro #3 is $3"
fi

...

if [-n "${10}"] # Parametros > $9 deben ser encerradas en {llaves}.
then
 echo "Parámetro #10 is ${10}"
fi

echo "-----"
echo "Todos los parámetros son: \"$*\""
if [$# -lt "$MINPARAMS"]; then
 echo
 echo "El script necesita al menos $MINPARAMS parámetros"
fi
```

# Parámetros posicionales

- Con shift se pueden desplazar los parámetros

```
$1 <- $2,
$2 <- $3,
$3 <- $4, etc.
```

```
until [-z "$1"] # Hasta que todos los parametros sean usados...
do
 echo -n "$1 "
 shift
done
exit 0
```

# Parámetros posicionales

- Analizaremos \$\* y \$@ y cómo IFS influye en la forma en la que se interpretan éstos

```
#!/bin/bash
E_BADARGS=65 #Valor de retorno si error por argumentos
if [! -n "$1"]; then
 echo "Uso: .basename $0. argumento1 argumento2 etc."
 exit $E_BADARGS
fi

echo
index=1 # Inicializamos el contador
echo "Listando argumentos con \"\$*\":"
for arg in "$*" #No funciona correctamente sin las comillas
do
 echo "Arg #$index = $arg"
 let "index+=1"
done # $* ve todos los argumentos como una única palabra
echo "Toda la cadena de argumentos como una única palabra."

echo
index=1 # Reseteamos el contador
echo "Listando argumentos con \"\$@\":"
for arg in "$@"
do
 echo "Arg #$index = $arg"
 let "index+=1"
done # $@ ve los argumentos como palabras separadas.
echo "La cadena de argumentos como palabras separadas."

echo
index=1 # Reseteamos el contador.
echo "Listando argumentos con \$* (sin comillas):"
for arg in $*
do
 echo "Arg #$index = $arg"
 let "index+=1"
done # $* sin comillas ve los argumentos como palabras separadas.
echo "Lista de argumentos como palabras separadas."
exit 0
```

```
mmartinez@ideafix:~/scripts/intro02$ ms
Uso: ms argumento1 argumento2 etc.

mmartinez@ideafix:~/scripts/intro02$ ms uno dos tres
Listando argumentos con "$*":
Arg #1 = uno dos tres
Toda la cadena de argumentos como una única palabra.

mmartinez@ideafix:~/scripts/intro02$ ms uno dos tres
Listando argumentos con "$@":
Arg #1 = uno
Arg #2 = dos
Arg #3 = tres
Toda la cadena de argumentos como una única palabra.

mmartinez@ideafix:~/scripts/intro02$ ms uno dos tres
Listando argumentos con "$@":
Arg #1 = uno
Arg #2 = dos
Arg #3 = tres
La cadena de argumentos como palabras separadas.

mmartinez@ideafix:~/scripts/intro02$ ms uno dos tres
Listando argumentos con $* (sin comillas):
Arg #1 = uno
Arg #2 = dos
Arg #3 = tres
Lista de argumentos como palabras separadas.
```

# Parámetros posicionales

- Analizaremos \$\* y \$@ y cómo IFS influye en la forma en la que se interpretan éstos

```
set -- "Primer argumento" "segundo" "tercer:argumento" "" "quinto: :argumento"
```

```
set -- "Primer argumento" "segundo" "tercer:argumento" "" "quinto: :argumento"
echo "$@"
echo $@
```

La salida del ejemplo anterior sería la siguiente, donde se pone de manifiesto como el primer mantiene un espacio en el lugar que ocupa el cuarto argumento que va vacío.

```
Primer argumento segundo tercer:argumento quinto: :argumento
Primer argumento segundo tercer:argumento quinto: :argumento
```

# Parámetros posicionales

- Analizaremos \$\* y \${@} y cómo IFS influye en la forma en la que se interpretan éstos
- \$\* y \${@} sólo son diferentes cuando van entre comillas dobles

```
set -- "Primer argumento" "segundo" "tercer:argumento" "" "quinto: :argumento"
echo
echo 'IFS sin cambios, usando "$*"'
c=0
for i in "$*" # entrecomillado
do
 echo "${((c+=1))}: [$i]" # Permanece igual cada iteración
done
```

```
IFS sin cambios, usando "$*"
1: [Primer argumento segundo tercer:argumento quinto: :argumento]
```

- \$\* devuelve todas los parámetros como palabras separadas, pero al ir entre comillas se tratan como un único string

# Parámetros posicionales

- Analizaremos \$\* y \${@} y cómo IFS influye en la forma en la que se interpretan éstos
- Si no ponemos las “”

```
set -- "Primer argumento" "segundo" "tercer:argumento" "" "quinto: :argumento"
echo 'IFS sin cambios, usando $*'
c=0
for i in $* # sin comillas
 do echo "$((c+=1)): [$i]"
done
IFS sin cambios, usando $*
1: [Primer]
2: [argumento]
3: [segundo]
4: [tercer:argumento]
5: [quinto:]
6: [:argumento]
```

- Cada parámetro es un elemento para el iterador
- No hemos modificado el IFS que sigue siendo espacio
- \$\* devuelve la cadena completa de argumentos como palabras sin entrecomillar

# Parámetros posicionales

- Analizaremos \$\* y \$@ y cómo IFS influye en la forma en la que se interpretan éstos
- \$@ devuelve la cadena completa de argumentos entrecomillando las palabras de dicha cadena. Para respetar los espacios hay que entrecomillar, “\$@”

```
set -- "Primer argumento" "segundo" "tercer:argumento" "" "quinto: :argumento"
echo 'IFS sin cambios, usando "$@"'
c=0
for i in "$@"
do echo "$(c+=1)): [$i]"
done
```

IFS sin cambios, usando "\$@"

1: [Primer argumento]  
2: [segundo]  
3: [tercer:argumento]  
4: []  
5: [quinto: :argumento]

- Las comillas actúan entrecomillando cada uno de los strings individuales, respetando por tanto los espacios que lleven dentro.
- No hemos modificado el IFS que sigue siendo espacio

# Parámetros posicionales

- Analizaremos \$\* y \$@ y cómo IFS influye en la forma en la que se interpretan éstos
- \$@ devuelve la cadena completa de argumentos
- Ahora no la entrecomillamos

```
set -- "Primer argumento" "segundo" "tercer:argumento" "" "quinto: :argumento"
echo 'IFS sin cambios, usando $@'
c=0
for i in $@
do echo "$((c+=1)): [$i]"
done
```

IFS sin cambios, usando \$@  
1: [Primer]  
2: [argumento]  
3: [segundo]  
4: [tercer:argumento]  
5: [quinto:]  
6: [:argumento]

- Se devuelve cada parámetro como un string individual, pero al no ir entrecomillado no se protegen al efecto del IFS
- Por tanto, sin cambiar el IFS podemos asegurar que \$\* y \$@ se comportan igual, solo presentan diferencias cuando van entrecomillados.

# Parámetros posicionales

- Realizar el mismo análisis pero modificando el IFS de la siguiente forma.

IFS=:

- ¿Que conclusiones obtenéis?

# Quoting

- Entrecomillar cadenas de caracteres para proteger la interpretación de caracteres especiales.

```
mmartinez@ideafix:~/scripts/intro02$ ls -l ms*
-rwxr--r-- 1 mmartine mmartine 1005 Nov 16 16:23 ms
-rwxr--r-- 1 mmartine mmartine 25 Nov 2 19:48 ms2
-rw-r--r-- 1 mmartine mmartine 0 Aug 13 16:41 msvi
.

mmartinez@ideafix:~/scripts/intro02$ ls -l [ms]*
-rwxr--r-- 1 mmartine mmartine 22 Oct 4 11:11 mayusculas.sh
-rwx----- 1 mmartine mmartine 1994 Aug 6 19:02 menu
-rwxr--r-- 1 mmartine mmartine 1005 Nov 16 16:23 ms
-rwxr--r-- 1 mmartine mmartine 25 Nov 2 19:48 ms2
-rw-r--r-- 1 mmartine mmartine 0 Aug 13 16:41 msvi
-rwxr--r-- 1 mmartine mmartine 226 Aug 6 17:22 sc1
-rwxr--r-- 1 mmartine mmartine 453 Aug 6 18:37 sc10
-rwxr--r-- 1 mmartine mmartine 253 Aug 6 19:03 sc11
-rwxr--r-- 1 mmartine mmartine 461 Aug 6 18:28 sc2
-rwxr--r-- 1 mmartine mmartine 537 Aug 6 18:29 sc3
-rwxr--r-- 1 mmartine mmartine 139 Aug 6 18:32 sc4
-rwxr--r-- 1 mmartine mmartine 178 Aug 6 18:32 sc5
-rwxr--r-- 1 mmartine mmartine 168 Aug 6 18:33 sc6
-rwxr--r-- 1 mmartine mmartine 255 Aug 6 18:41 sc7
-rwxr--r-- 1 mmartine mmartine 305 Aug 6 18:34 sc8
-rwxr--r-- 1 mmartine mmartine 422 Aug 6 18:36 sc9

mmartinez@ideafix:~/scripts/intro02$ ls -l '[ms]*'
ls: [ms]*: No such file or directory
```

# Quoting

- Algunas utilidades reinterpretan ciertos caracteres a pesar de estar entrecomillados, y por tanto hay que referirse a la sintaxis de dichos programas para poder utilizarlos correctamente.
- Un ejemplo de esto es la utilidad grep.

```
$ grep 'l[oa]s' quijote.txt
no ha mucho tiempo que vivía un hidalgo de los de lanza en
Una olla de algo más vaca que carnero, salpicón las más noches,
duelos y quebrantos los sábados, lentejas los viernes, algún
palomino de añadidura los domingos, consumían las tres partes
calzas de velludo para las fiestas con sus pantuflas de lo mismo,
los días de entre semana se honraba con su vellori de lo más fino.
Tenía en su casa una ama que pasaba de los cuarenta, y una sobrina
que no llegaba a los veinte, y un mozo de campo y plaza, que así
nuestro hidalgo con los cincuenta años, era de compleción
Quesada (que en esto hay alguna diferencia en los autores que
```

# Quoting

## ■ Atención a “” y al IFS en la llamada a comandos

```
variable1="una variable con cinco palabras"
```

```
COMANDO Esta es $variable1 #Ejecuta COMANDO con 7 argumentos:
 # "Esta" "es" "una" "variable" "con" "cinco" "palabras"
```

```
COMANDO "Esta es $variable1" # Ejecuta COMANDO con 1 argumento:
 # "Esta es una variable con cinco palabras"
```

```
variable2="" # Vacía
```

```
COMANDO $variable2 $variable2 $variable2
 # Ejecuta COMANDO sin argumentos.
```

```
COMANDO "$variable2" "$variable2" "$variable2"
 # Ejecuta COMANDO con 3 argumentos vacios.
```

```
COMANDO "$variable2 $variable2 $variable2"
 # Ejecuta COMANDO con 1 argumento (2 espacios).
```

# Quoting

## ■ \ no siempre es una secuencia de escape

- \n significa newline
- \r significa return
- \t significa tabulador
- \v significa tabulador vertical
- \b significa backspace
- \a significa "alert" (beep or flash)
- \0xx traduce al valor octal del ASCII 0xx (hexadecimal)

```
echo "\v\v\v\v" # muestra \v\v\v\v literalmente.
La opción -e de 'echo' imprime los caracteres de escape
echo "===== "
echo "VERTICAL TABS"
echo -e "\v\v\v\v" # muestra 4 tabuladores verticales.
echo "===== "
```

mmartinez@ideafix:~/scripts/intro02\$ ms

```
\v\v\v\v
=====
VERTICAL TABS
=====
```

# Quoting

## ■ Algunos ejemplos más

```
echo "COMILLAS DOBLES"
echo -e "\042" # Imprime " (comillas dobles, octal ASCII 42).
```

```
mmartinez@ideafix:~/scripts/intro02$ ms
COMILLAS DOBLES
"
```

```
echo; echo "NUEVA LIENA Y BEEP"
echo $'\n' # Nueva linea.
echo $'\a' # Beep.
```

```
mmartinez@ideafix:~/scripts/intro02$ ms
```

```
NUEVA LIENA Y BEEP
```



# Quoting

## ■ Una técnica que puede ser muy útil

```
comillas=$'\042' # " asignada a la variable.
echo "$comillas Texto entre comillas$comillas y este fuera de ellas"
```

" Texto entre comillas" y este fuera de ellas

Aquí va un espacio  
necesariamente

# Quoting

## ■ Una técnica que puede ser muy útil

```
comillas=$'\042' # " asignada a la variable.
echo "$comillas Texto entre comillas$comillas y este fuera de ellas"
```

" Texto entre comillas" y este fuera de ellas

## ■ ¿Y si estoy obligado a no dejar un espacio?

“\${comillas}Texto ....

## ■ Otro ejemplo de la misma técnica

```
Concatenar caracteres ASCII en una variable.
triple_subrrayado=$'\137\137\137'
echo "${triple_subrrayado}ATENCION${triple_subrrayado}"
```

\_\_\_ATENCION\_\_\_

# Quoting

## ■ \" da a las " su valor literal

```
echo "Hola Maria" #-Hola Maria
echo "-\"Hola\"", dijo ella, "#-\"Hola\"", dijo ella,
```

## ■ \\$ da al \$ su valor literal

```
echo "Con el caracter \$ obtenemos el valor de la variable A de esta forma \$A"
```

Con el caracter \$ obtenemos el valor de la variable A de esta forma \$A

## ■ \\ da al \ su significado literal

```
echo "\\\" # muestra \\\\"
```

# Tests

- El comando **test** permite realizar test de condición utilizables en las sentencias condicionales
- [ es un comando equivalente a **test**.
  - [ existe como fichero en /etc/bin/test
  - **test condicion** equivalente a [ **condicion** ]
- Con test o [ se pueden utilizar
  - Operadores de condición
  - Test de fichero
  - Nada. Si no se utilizan las anteriores [ comprueba si el contenido de los [] es null.
- [[ es una versión extendida de [
  - [[ es una palabra reservada, no un comando.

# Tests y condicionales

■ **if test condicion es equivalente a if [ condicion ]**

■ **test condicion o [ condicion ] o [[ condicion ]]**

- Se evalúan retornando un valor de verdad, verdadero o falso.
- La **condicion** debe ir separada por espacios de los corchetes.

■ Las siguientes condicionales son equivalentes

```
if test condicion
then
...
else
...
fi
```

```
if [condicion]
then
...
else
...
fi
```

```
if [[condicion]]
then
...
else
...
fi
```

■ La diferencia entre [ y [[ es la sintaxis de los operadores relacionales y las condiciones múltiples

- [ utiliza operadores tipo fortran, -eq, -lt, -gt
- [[ utiliza operadores tipo C, ==, <=, >=
- AND

- [ condicion1 ] && [ condicion2 ]
- [ condicion1 -a condicion2 ]
- [[ condicion1 && condicion2 ]]
- test condicion1 -a condicion2

• OR

- [ condicion1 ] || [ condicion2 ]]
- [ condicion1 -o condicion2 ]
- [[ condicion1 || condicion2 ]]
- test condicion1 -o condicion2

• NEGACION

- ! Condicion
- ! [ condicion ]
- [[ ! Condicion ]]

# Tests y condicionales

- **let “expresión” y (( expresión ))** también permiten realizar comparaciones

- Devuelven el resultado de la comparación

```
if let "1 < 2"
then
...
else
...
fi
```

  

```
if ((1 < 2))
then
...
else
...
fi
```

- **if** puede testear cualquier comando

```
if cmp a b &> /dev/null #Suprimimos la salida del comando
then
 echo "Los ficheros a y b son idénticos."
else
 echo "Los ficheros a y b son diferentes."
fi

Esta construcción "if-grep" suele ser muy útil:

if grep -q Bash MiFichero #-q modo silent-quiet. Para tras la primera ocurrencia
then
 echo "MiFichero contiene al menos una ocurrencia de \"Bash\"."
fi
```

# Tests y condicionales

- Tras un **if** puede ir cualquier comando cuyo valor de retorno sea 0 (éxito-true) o distinto de 0 (fallo-false)
- Estructura de un bloque **if then else**

```
if [condition-true]
then
 command 1
 command 2
 ...
else
 # Opcional. Se puede omitir si no es necesario
 command 3
 command 4
 ...
fi
```

# Tests y condicionales

## ■ if anidados y else if o elif

```
if [condicion1]
then
 if [condicion2]
 then
 comandos
 else
 comandos
 fi
else
 comandos
fi
```

```
if [condicion1]
then
 comandos
else if [condicion 2]
 then
 comandos
 else
 comandos
 fi
fi
```

```
if [condicion1]
then
 comandos
elif [condicion 2]
 then
 comandos
elif [condicion 3]
 then
 comandos
else
 comandos
fi
```

# Tests y condicionales. TRUE o FALSE

- Hay que llevar cuidado con que entendemos que resulta TRUE o FALSE en las comparaciones. Si no se usan operadores se evalua TRUE si el contenido no es NULL

“0” es true

```
echo "Comprobando \"0\""
if [0] # cero
then
 echo "0 es true."
else
 echo "0 es false."
fi # 0 es true.
```

“1” es true

```
echo "Comprobando \"1\""
if [1] # uno
then
 echo "1 es true."
else
 echo "1 es false."
fi # 1 es true.
```

“-1” es true

```
echo "Comprobando \"-1\""
if [-1] # menos uno
then
 echo "-1 es true."
else
 echo "-1 es false."
fi # -1 es true.
```

“null” es false

```
echo "Comprobando \"NULL\""
if [] # NULL (condición vacía)
then
 echo "NULL es true."
else
 echo "NULL es false."
fi # NULL es false.
```

“Cualquier cadena de caracteres” es true

```
echo "Comprobando \"xyz\""
if [xyz] # cadena
then
 echo "Cadena es true"
else
 echo "Cadena es false"
fi # Cadena es true.
```

# Tests y condicionales. TRUE o FALSE

- Cuando se evaluan variables, las condiciones cambian.
- Una variable no inicializada es false aunque utilicemos -n

```
echo "Comprobando \"\$xyz\" variable no inicializada"
if [\$xyz] # Comprobando si $xyz es null, pero...
 # solo es una variable no inicializada.
then
 echo "Variable no inicializada es true."
else
 echo "Variable no inicializada es false."
fi # Variable no inicializada es false.
```

- la construcción -n xyz devuelve true si el string xyz no es vacío, en otras palabras, si su longitud no es 0 (pero si está inicializada, claro)

```
echo "Comprobando \"-n \$xyz\""
if [-n "$xyz"]
then
 echo "Variable no inicializada es true."
else
 echo "Variable no inicializada es false."
fi # Variable no inicializada es false.
```

¡CUIDADO!

False porque no  
está inicializada

# Tests y condicionales. TRUE o FALSE

- Si la variable está inicializada pero es null entonces...

```
xyz= # Inicializada pero null.
echo "Comprobando \"-n \$xyz\""
if [-n "$xyz"]
then
 echo "Variable null es true."
else
 echo "Variable null es false."
fi # Variable null es false.
```

- A veces es más cómodo utilizar ; para separar el **if** del **then** y ponerlos en la misma línea
  - **if [ condicion ]; then**
  - **elif [ condicion ]; then**

# Tests

## ■ Operadores de comparación simple entre enteros

n1 -eq n2 cierto si los enteros n1 y n2 son iguales

```
if ["$a" -eq "$b"]
```

n1 -ne n2 cierto si los enteros n1 y n2 no son iguales

```
if ["$a" -ne "$b"]
```

n1 -gt n2 cierto si el entero n1 es mayor que n2

```
if ["$a" -gt "$b"]
```

n1 -ge n2 cierto si los enteros n1 y n2 son iguales o n1 es mayor que n2

```
if ["$a" -ge "$b"]
```

n1 -lt n2 cierto si el enteros n1 es menor que n2

```
if ["$a" -lt "$b"]
```

n1 -le n2 cierto si los enteros n1 y n2 son iguales o n1 es menor que n2

```
if ["$a" -le "$b"]
```

< menor que

```
(("$a" < "$b"))
```

<= menor o igual que

```
(("$a" <= "$b"))
```

> mayor que

```
(("$a" > "$b"))
```

>= mayor o igual que

```
(("$a" >= "$b"))
```

# Tests

## ■ Operadores de ficheros

- d fichero** cierto si fichero existe y es un directorio
- e fichero** cierto si fichero existe, independientemente del tipo que sea
- f fichero** cierto si fichero existe y es un fichero normal (no es directorio ni fichero especial)
- b fichero** cierto si fichero existe y es un dispositivo de bloques (floppy, cd-rom, hd, etc)
- c fichero** cierto si fichero existe y es un dispositivo de caracteres (keyboard, modem, sound card, etc.)
- p fichero** cierto si fichero existe y es un pipe
- h fichero** cierto si fichero existe y es un hard link
- L fichero** cierto si fichero existe y es un soft link
- S fichero** cierto si fichero existe y es un socket
- t fichero** cierto si fichero existe y es un descriptor de fichero.
- r fichero** cierto si fichero existe y se puede leer (tiene permiso de lectura para el usuario que ejecuta test)
- s fichero** cierto si fichero existe y tiene tamaño mayor que cero
- w fichero** cierto si fichero existe y es se puede escribir sobre él
- x fichero** cierto si fichero existe y es ejecutable
- O fichero** cierto si fichero existe y el que ejecuta el script es el propietario
- G fichero** cierto si fichero existe y el que ejecuta el script pertenece al grupo del fichero
- N fichero** cierto si fichero existe y se ha modificado desde su última lectura.
- f1 -nt f2** cierto si el fichero f1 es mas nuevo que el f2
- f1 -ot f2** cierto si el fichero f1 es mas antiguo que el f2
- f1 -ef f2** cierto si los ficheros f1 y f2 son hard links al mismo fichero

# Tests

## ■ Operadores de cadenas de caracteres

**s1 = s2** cierto si las cadenas de texto s1 y s2 son idénticas

```
if ["$a" = "$b"]
if ["$a" == "$b"]
```

**s1 != s2** cierto si las cadenas de texto s1 y s2 no son idénticas

```
if ["$a" != "$b"]
```

**s1 < s2** cierto si la cadena de texto s1 es menor que s2

```
if [["$a" < "$b"]]
if ["$a" \< "$b"]
```

**s1 > s2** cierto si la cadena de texto s1 es mayor que s2

```
if [["$a" > "$b"]]
if ["$a" \> "$b"]
```

**-n cadena** cierto si la longitud de la cadena de texto es distinta de cero

**-z cadena** cierto si la cadena es null, es decir tiene longitud 0

## ■ Uso de los valores de retorno de comandos para decidir

```
dir=/home/mmartinez
if cd "$dir" 2>/dev/null; then # 2>/dev/null oculta el mensaje de error.
 echo "Ahora estamos en $dir."
else
 echo "No he podido cambiar a $dir."
fi
```

# && y || como operadores de lista

- Se ejecuta el siguiente comando si el anterior devuelve true

```
var1=20
var2=22
["$var1" -ne "$var2"] && echo "$var1 no es igual que $var2"
```

El resultado de la ejecución del ejemplo es:

```
20 no es igual que 22
```

- Se ejecuta el siguiente comando si el anterior devuelve false. Este ejemplo es mucho más interesante, ya que sin la necesidad de un bloque if podemos determinar si un directorio existe o no y en su defecto crearlo y además mostrar si se ha podido crear o no.

```
dir=/home/mmartinez/datos
[-d "$datos"] || (mkdir $dir && echo "El directorio $dir se ha creado con éxito")
```

# Case

- Es equivalente al switch del C
- En función de la comparación salta a un bloque de código.
- La comparación es la concordancia de patrones

```
case "$variable" in
 "$condicion1")
 comandos...
 ;;
 "$condicion2")
 commandos...
 ;;
esac

echo "Elige una de las siguientes opciones:"
echo A B C
read opcion
case $opcion in
 A | a)
 echo "Ha seleccionado la opción A"
 ;;
 B | b)
 echo "Ha seleccionado la opción B"
 ;;
 C | c)
 echo "Ha seleccionado la opción C"
 ;;
esac
```

# Case

- La condición puede ser la ejecución de un comando
- Espacios entre el comando y los paréntesis

```
case $(arch) in
 i386) echo "Arquitectura 80386";;
 i486) echo "Arquitectura 80486";;
 i586) echo "Arquitectura Pentium";;
 i686) echo "Arquitectura Pentium2+";;
 *) echo "Otro tipo de máquina";;
esac
exit 0
match_string () {
 IGUALES=0
 DISTINTAS=90
 NUMPARAM=2 #Número de parametros que acepta la función
 BAD_PARAMS=91 #Error a devolver por falta de parámetros

 [$# -eq $NUMPARAM] || return $BAD_PARAMS

 case "$1" in
 "$2") return $IGUALES;;
 *) return $DISTINTAS;;
 esac
}
```

Otro ejemplo

# Case

## Más ejemplos con case y expresiones regulares

```
EXIT0=0
FALLO=-1
isalpha (){ #Comprueba que el primer carácter de la cadena
 #que se pasa es alfabético.
 if [-z "$1"]; then #No se pasan parametros
 return $FALLO
 fi

 case "$1" in
 [a-zA-Z]*) return $EXIT0;;
 *) return $FALLO;;
 esac
}

isalpha2 (){ #Comprueba que toda la cadena es alfabetica
 [$# -eq 1] || return $FALLO

 case $1 in
 [!a-zA-Z]|"") return $FALLO;;
 *) return $EXIT0;;
 esac
}
```

¿Y toda la  
cadena  
numérica?

# Select

- Permite crear menus

```
select variable [in list]
do
```

```
 command...
 break
done
```

La variabel \$PS3 almacena la pregunta

```
#!/bin/bash
PS3='Selecciona un mes:' #Establecemos la pregunta
echo
```

```
select mes in "Enero" "Marzo" "Junio" "Noviembre"
do
```

```
 echo
 echo "Has elegido $mes"
```

```
 echo
 break mmartinez@ideafix:~/scripts/intro02$ ms
done
exit 0
```

```
1) Enero
2) Marzo
3) Junio
4) Noviembre
Selecciona un mes:3
```

```
Has elegido Junio
```

# Select

- Permite crear menus

```
select variable [in list]
do
 command...
 break
done
```

- [ in list ] es opcional. Si se omite la lista la componen los parametros posicionales

```
PS3='Elige un mes'
echo

seleccion(){
 select mes
 do
 echo
 echo "El mes elegido es $mes"
 break
 done
}

seleccion enero febrero marzo abril mayo junio
$1 $2 $3 $4 $5 $6
#Parametros pasados a la funcion
exit 0
```