

Variables de entorno

Ya sabemos que una variable es una zona de memoria reservada para almacenar información que puede ser necesaria durante la ejecución de cualquier programa, incluido el sistema operativo que se está ejecutando.

Las variables pueden ser creadas y manejadas por el propio sistema operativo o pueden ser creadas y manejadas por los programas de aplicación o incluso por los usuarios del sistema.

Las variables de entorno asignadas desde la línea de mandatos son temporales, solo se mantendrán mientras se ejecuta un terminal determinado para hacer que este cambio se mantenga en otras sesiones, hay que agregar la definición en el archivo de recursos, `.bash_profile` de sus directorios home locales.

1.1. Visualización de las variables de entorno

Se pueden visualizar las variables definidas en el sistema junto con los valores que contienen o se puede visualizar el valor de una única variable con el comando `printenv`, que se encuentra en el directorio `/usr/bin`.

Si ejecuto el comando sin parámetros, es decir, sin indicar ninguna variable, visualiza todas las variables definidas y sus valores. Por ejemplo:

```
$ printenv
```

Por ejemplo, puedo utilizar el comando para visualizar el valor de la variable `HOME` ejecutando lo siguiente:

```
$ printenv HOME    Contiene directorio de trabajo
```

```
$ printenv SHELL    /bin/bash
```

También se pueden visualizar las variables definidas en el sistema, así como los valores que contienen con el comando `env`. Este comando a diferencia de `printenv` no permite visualizar el valor de una única variable.

El equivalente a la orden anterior sería:

```
$env
```

1.2. Variables de usuario

Son aquellas variables definidas por el usuario. Son variables que sólo pueden manejar el usuario que las ha creado y no ningún otro del sistema.

Veamos cuales son las características de estas variables y cómo podemos trabajar con ellas, asignándole valores y consultándolos después.

Características de los nombres de variables:

Los caracteres válidos en los nombres de las variables son:

- Los alfabéticos, tanto mayúsculas como minúsculas.
- Los numéricos.
- El subrayado.

Además, las variables no pueden empezar por un número y los números se almacenan como cadenas de texto.

Resumiendo: El nombre de una variable puede estar formado por cualquier conjunto de caracteres alfabéticos, incluyendo el carácter de subrayado. El nombre también podrá incluir números, pero un número no podrá ser el primer carácter del nombre. Los nombres no podrán incluir otros tipos de caracteres como el signo de exclamación el signo \$ o espacios, porque estos símbolos están reservados para su utilización en la shell.

Un mismo nombre de variable en mayúsculas es diferente para el Shell si esta en minúsculas. Ejemplo: VENTAS, Ventas y ventas son tres nombres de variables diferentes. Normalmente para nombrar las variables se utilizan las letras mayúsculas.

Las variables de los Shell scripts son muy simples, ya que no tienen tipo definido ni necesitan ser declaradas antes de poder ser usadas. Para introducir valor en una variable simplemente se usa su nombre.

Para acceder al contenido de una variable se antepone al nombre de dicha variable el símbolo \$. Por ejemplo, para visualizar el contenido de la variable PATH basta con teclear:

```
echo $PATH    Sacara por pantalla el valor del path actual.  
echo PATH     Sacara por pantalla el mensaje PATH.
```

Para asignar un valor a una variable de forma directa se utiliza el carácter = De tal forma que si necesito crear una variable de nombre Prim y con valor 1 deberé teclear:
Prim=1

También puedo crear variables que contengan el carácter separador blanco entre comillas.

```
dos= " palabras separadas"
```

En una asignación nunca puede haber espacios en blanco a ambos lados del signo =.

Hay que tener en cuenta que el valor asignado a una variable sólo es conocido por el Shell en el que se asigna. Si se quiere que ese valor sea conocido por cualquier Shell hijo del Shell en el que se asigna, se tendrá que utilizar el comando **export**.

Formato: export nombre_variable/s

Por ejemplo:

```
$EDITOR="/usr/bin/pico" ; export EDITOR
```

Hay que tener en cuenta que los valores de las variables se pueden exportar de shell padres a hijos, pero no al revés.

Cuando se aplica el comando export a una variable, se ordena al sistema que defina una copia de esa variable para cada nueva subshell generada.

Cada subshell nueva contendrá su propia copia de la variable exportada.

Es erróneo considerar que las variables exportadas son variables globales. Una shell no puede hacer referencia a una variable exterior a ella, sino que se generará una copia de

la variable con el mismo valor para la nueva shell. Se copiarán en todas las shells que deriven de su propia shell.

Todos los scripts de shell a los que se llame directa o indirectamente después de la shell de la variable exportada poseerán una copia de la variable exportada con el valor inicial.

Si utilizamos export sin argumentos nos visualiza la lista de todas las variables exportadas.

También podemos impedir que a una variable se le pueda modificar su contenido, para ello se utiliza el comando **readonly**. Pero hay que tener cuidado porque los efectos de readonly perduran hasta que se termine la sesión.

Por ejemplo:

`$Variable=valor`

`$readonly Variable`

Si después de ejecutar las órdenes anteriores se intenta modificar el valor de la variable se produce un error.

1.3. Variables del sistema

Las **variables internas** o especiales son aquellas a las que el sistema asigna directamente unos valores. Estas variables son:

- \$*** Contiene los argumentos introducidos en la línea de comandos. Con el siguiente texto visualizará todos los parámetros que has introducido → echo el valor * es \$*
- \$#** Contiene el número de argumentos introducidos.
- \$?** Si el último comando se ha ejecutado correctamente contendrá un 0, en caso contrario contendrá otro valor.
- \$\$** Contiene el PID del proceso en ejecución.

Parámetros	
\$1	Devuelve el 1º parámetro pasado al script o función al ser llamado.
\$2	Devuelve el 2º parámetro.
\$3	Devuelve el 3º parámetro. (Podemos usar hasta \$9).
\$*	Devuelve todos los parámetros separados por espacio.
\$#	Devuelve el número de parámetros que se han pasado.
\$0	Devuelve el parámetro 0, es decir, el nombre del script o de la función.

- \$?** Si el último comando se ha ejecutado correctamente contendrá un 0, en caso contrario contendrá otro valor.

1.4. Personalización del inicio de sesión

Cuando un usuario se conecta al sistema, es decir abre una sesión de trabajo, después de comprobarse que existe el usuario en el sistema (fichero `/etc/passwd`) y que la clave introducida por el usuario coincide con la que éste ha tecleado, se ejecuta el shell especificado en el fichero `/etc/passwd`.

Este shell carga varios archivos localizados en diferentes directorios del sistema.

- **.bashrc**: creado de manera predefinida en el directorio HOME de cada usuario.
- **.profile**: creado de manera automática en el directorio HOME de cada usuario.
- **.bash_logout**: creado de manera automática en el directorio HOME de cada usuario.
- **.bash_history**: creado de manera automática en el directorio HOME de cada usuario. Contiene una lista con los mandatos introducidos por el usuario.
- **/etc/profile**: contiene las variables de entorno del sistema.

El fichero **.profile**

Las variables de entorno local se definen en el archivo **.profile** del directorio home. Por ejemplo, si quiero modificar mi variable PATH, para que se busque en el directorio “ejecutable” dentro de mi directorio raíz, deberá aparecer la siguiente línea en este fichero:

```
PATH=$PATH:$HOME/ejecutable:$HOME/bin
```

Se puede utilizar el fichero **.profile** para definir todas aquellas variables que se necesite que estén definidas cada vez que el usuario se conecta al sistema.

El fichero **.bashrc**

En este fichero se suelen incluir los alias. Los alias son redefiniciones de mandatos, usualmente es un formato más corto. Las definiciones de alias para todo el sistema las incluye el root en el fichero `/etc/bashrc`, pero también los usuarios pueden definir sus propios alias en el archivo **.bashrc**.

Por ejemplo, incluyendo las siguientes líneas:

```
alias cp="cp -i"  
alias ls="ls -lia"
```

2. Programación shell

La shell (bash) de linux incluye un gran número de herramientas de programación con las que podrá crear programas de la shell. Puede definir variables y asignarles valores, existen estructuras de control de bucle y condicionales que repiten comandos de linux o que deciden los comandos que desea ejecutar. Puede combinar las variables, las estructuras de control, las expresiones y los comandos de Linux para crear un programa de la shell.

Un **guión**, un **shellscripts**, un **fichero bash** o simplemente **script** no son más que ficheros de texto ASCII puro, que pueden ser creados con cualquier editor del que dispongamos (vi, nano, gedit) que contienen una secuencia de ordenes o comandos del sistema operativo Unix/Linux, que el intérprete de bash deberá interpretar y ejecutar.

Versión del bash que tiene tu Linux versión 4.3.

¿Cómo reconocemos un fichero bash?

1) Aunque no es obligatorio, un fichero bash suele tener la extensión **.sh** o **.bash** Para identificar fácilmente los scripts de shell puede agregarse la extensión **.sh** (por ejemplo, hola.sh) aunque no es necesario.

2) Comúnmente, aunque tampoco es obligatorio, las buenas prácticas y un método para

evitar problemas de compatibilidad y de permisos es indicar **#!/bin/bash** en la primera línea del contenido del script.

Documentación: Podemos ver el manual oficial de bash aquí: [bash official manual](#)

Cread un fichero de texto de nombre primero.sh, con el siguiente contenido:

La primera línea que se suele poner en un script sirve para indicar que shell utilizamos (en nuestro caso bash no es obligatoria) y donde puede ser encontrado en nuestro sistema (para saberlo, podemos hacer locate bash). Si no se pone nada utilizará el shell por defecto que hay y que suele ser bash. Esta línea debe ser la primera de todos los scripts que realicemos.

```
#!/bin/bash
```

En este caso se está indicando que el intérprete será bash y que está ubicado (bash) en /bin.

La segunda línea de nuestro script simplemente utiliza el comando para escribir en pantalla (echo) y escribe la línea Hola Mundo

Otro ejemplo de guión es:

```
$cat > Shell1
#!/bin/bash
```

```
# Este es el primer script
echo "este es mi primer shell"
echo
echo "los usuarios conectados al sistema son:"
who
echo mi directorio de trabajo es
pwd
echo "fin del shell"
[ctrl-D]
```

2.1 Ejecución de un shellscript

En principio, antes de ejecutar un guion es necesario escribirlo. Para realizar los guiones utilizaremos cualquier editor (vi, pico, nano, etc.). Tras ello tendremos que asignar los permisos necesarios para que este programa pueda ser ejecutado, mediante la orden **chmod**.

Un shellscripts puede ejecutarse de varias formas diferentes, veamos las particularidades de cada una de ellas:

Nombre del Shell con el que lo vas a ejecutar y después nombre del fichero.

A. sh nombre-shell ó con otra Shell bash nombre-shell

- No es necesario que el fichero tenga permiso de ejecución.

- Se crea un shell hijo del actual que se encarga de ejecutar el shellscripts. Cuando la ejecución ha finalizado, se vuelve al shell anterior.

Bash proviene de sh

Ejemplo:

```
$Sh shell1
```

B. **nombre-shell**

- Es necesario que el shell tenga permiso de ejecución.
- Se crea un shell hijo que será donde se ejecute el shellscript.

Ejemplo:

```
$Shell1
```

Solo cuando la ruta donde esta el fichero se encuentre en la variable PATH

C. **. nombre-shell o ./nombre-shell**

- El shell no necesita permiso de ejecución.
- No se crea un hijo, se ejecuta en el shell actual. Por lo tanto el entorno de trabajo desde el cual nosotros dimos la orden de ejecución puede ser modificado.

Ejemplo:

```
$ . Shell1
```

D. **source nombre-shell**

- Es necesario que el shell tenga permiso de ejecución.
- Se ejecuta en el shell actual. Por lo tanto, el entorno de trabajo desde el cual nosotros dimos la orden de ejecución puede ser modificado.
- Una vez finalizado el shell nos saca al login, es decir cierra la sesión de trabajo.

Ejemplo:

```
$ source Shell1
```

En los tres últimos casos hay que tener en cuenta que los shellscripts se deberán encontrar dentro de alguno de los directorios indicados en la variable PATH.

Para la ejecución de shellscripts y otros ejecutables hay que tener en cuenta el contenido de la variable PATH, que tiene trayectorias de directorios donde buscar los programas a ejecutar si se especifica su trayectoria. El sistema operativo busca primero en los caminos establecidos en Path y si el directorio activo no está entre ellos no busca en él.

Esta es una medida de seguridad para evitar los Caballos de Troya -programas que se "disfrazan" con el nombre de un programa de uso habitual para hacer labores distintas, normalmente nocivas, sin que el usuario se dé cuenta. Por esto en Linux el directorio activo siempre debe ir el último en la secuencia de la variable PATH

Ejemplo:

```
PATH=/bin:/usr/bin:/home/ana
```

¿Como modificar la variable PATH?

Cada ruta de directorio aparece separada por dos puntos (:). En caso de que desees agregar directorios a tu *PATH* puedes hacerlo válido por el tiempo que dura tu sesión. Basta escribir en el terminal:

```
PATH="$PATH:<ruta nueva1>:<ruta nueva2>:...<ruta nuevaN>"
ana@ubuntu:~/linuxayuda$ PATH="$PATH:/home/ana/linuxayuda"
```

Aunque hubiese un Caballo de Troya, llamado, por ejemplo, ls en el directorio activo, el sistema operativo ejecutaría el ls del sistema que esta en el directorio /bin.

Cuando el usuario es root el directorio activo no está en PATH, por eso, cuando root quiere ejecutar un shellscript que está en el directorio activo debe escribir:

./shellXX siendo shellXX el programa que quiere ejecutar.

En resumen, la siguiente tabla muestra las características de las diferentes formas de ejecución de los shell:

	Sh	nombre	.	source
Permiso de ejecución	No	Si	No	Si
En el Shell actual	No	No	Si	Si

Cat ejemplo1.sh Este ejemplo es para ejecutarlo con Shell bash no con Shell sh

```
#!/bin/bash
```

```
# Este script displaya la fecha, hora, nombre de
# usuario y directorio actual
echo "Fecha y hora:"
date
echo
echo "Tu usuario es: `whoami` \n" # comillas contrarias
#ejecutan el comando y sustituirlo en la salida
echo "Tu directorio actual es: \c"
pwd
```

Las primeras dos líneas que comienzan con una almohadilla (#) son **comentarios** y no son interpretadas por el shell.

Usa comentarios para documentar tu shell script, te sorprenderá saber lo fácil que es olvidar lo que hacen tus propios programas.

Las backquotes (`) entre el comando whoami ilustran el uso de la **sustitución de comandos**.

- **sustitución de comandos** : para incluir la salida de un comando dentro de una línea de comandos de otro comando, encierra el comando cuya salida quieres incluir, entre backquotes (`)

- **whoami** : displaya el usuario actual

3. echo

El echo con la **opción -n** permite no hacer el intro final, podemos verlo en el siguiente ejemplo:

```
$ echo ejemplo.....; echo ok
ejemplo.....
```


ok

Añadiendo el **-n** al primer **echo** tenemos la siguiente salida:

```
$ echo -n ejemplo.....; echo ok
ejemplo.....ok
```

Por defecto el echo después de mostrar el argumento produce un salto de línea. Para eliminarlo hay que usar la opción **-n**.

```
# estas dos son equivalentes
echo "Hola mundo"
echo Hola mundo
# estas dos no son equivalentes
echo /etc/sh*
echo "/etc/sh*"
# con -n se elimina el salto de linea por defecto
echo -n "Introduzca un numero: "
read num
echo El numero es $num
```

Resultado

```
hola mundo
hola mundo
/etc/shadow /etc/shadow- /etc/shells
/etc/sh*
introduzca un numero5
El numero es 5
```

Por otro lado, tenemos el conjunto de opciones **-e** y **-E**. Con la **-e** indicamos que interprete los caracteres escapados con una **contrabarra**, por el contrario, con **-E** indicamos que no se interpreten (este es el comportamiento por defecto)

-e Activa la interpretación de caracteres precedidos por el carácter de escape.

\n = salto de línea

- Incorrecta, sin el parámetro **-e**, ofrecería este resultado:

```
$ echo "Hola\nAdios"
```

```
Hola\nAdios
```

- **Correcta, necesita -e para interpretar los caracteres especiales:**

```
$ echo -e "Hola\nAdios"
```

```
Hola
```

Adios

```
echo -e "\n linea 1 \n linea 2 \n"
```

```
echo -e "\n linea 1 \n linea 2 \n" > salida.txt
```

- **\t** para un tabulador:

```
$ echo -e 'tab\ttab'    O   echo -e "tab\ttab"
tab    tab
```

- **\v** para un “tabulador vertical”, lo que también se conoce como un salto de línea (sin retorno de carro):

```
$ echo -e 'lol\vlol'
lol
  lol
```

- **\b** para **backspace**, se desplaza al carácter anterior:

```
$ echo -e 'a\b'
b
```

se ve solo b

Resulta muy útil para hacer dibujos, como la siguiente barra giratoria ASCII:

```
$ while true; do for i in / - \\\ '|'; do echo -n $i; sleep 1; echo -ne '\b'; done; done
\
```



4. Uso de las comillas.

Es importante resaltar que podemos usar distintos métodos de entrecomillado que harán que el significado del contenido varíe.

- ☐ Las **comillas simples** se suelen emplear para indicar que el contenido se va a interpretar de forma literal.

- ❑ Las **comillas dobles** se usan para asignar una cadena a una variable, pero si encierran otras variables, el contenido de estas se expande.
- ❑ Las **comillas invertidas** y también la expresión de expansión **\$()** se utilizan para evaluar comandos y el resultado se devuelve como el de una variable.

Veamos estas diferencias con un ejemplo:

```
1  #!/bin/bash
2  X=100
3  Y='500$X'
4  echo $Y
5  Y="500$X"
6  echo $Y
7  Z=`who | wc -l` # equivalente a Z=$(who | wc -l)
8  echo $Z
```

Línea: 10 Col: 1 INS LÍNEA UTF-8 ejem.sh

```
masote@masote-virtual-machine:~/scripts$ ./ejem.sh
500$X
500100
3
```

La línea 4 muestra la variable Y como un literal (tal cual estaba inicializada), la línea 6 sin embargo muestra la variable Y, pero expandiendo la parte que corresponde a la variable X. Por último, la línea 8 lo que muestra es el resultado de expandir lo que habíamos asignado a Z entre comillas invertidas, es decir el número de usuarios conectados.

Aunque los valores de las variables pueden estar formados por caracteres de cualquier tipo, surgirán problemas cuando se incluyan caracteres que la shell utiliza como operadores. Si desea utilizar estos caracteres como parte del valor de una variable deberá entrecomillarlos.

Al entrecomillar un carácter especial en la línea de comandos, el carácter será igual a cualquier otro.

- El espacio se utiliza para separar los argumentos de la línea de comandos.
- El asterisco, el signo de interrogación y los corchetes son caracteres especiales que se utilizan para generar listas de nombres de archivo.
- El punto representa el directorio actual.
- El signo & se utiliza para ejecutar comandos en segundo plano

Las comillas simples y dobles le permiten entrecomillar varios caracteres especiales al mismo tiempo. Una barra invertida modifica el significado de un único carácter, el expresado a continuación de la misma.

Ejemplos:

```
Aviso="La reunión será mañana"
```

```
echo $Aviso
```

```
La reunión será mañana
```

En este ejemplo las comillas dobles engloban palabras separadas por blancos, como los espacios están dentro de las comillas serán tratados como caracteres y no como delimitadores para analizar los argumentos de la línea de comandos.

```
Mensaje='El proyecto se ha terminado a tiempo.'
```

```
echo $mensaje
```

```
El proyecto se ha terminado a tiempo.
```

En este ejemplo las comillas simples engloban un punto y lo tratan, como un carácter.

Ejemplo:

```
Aviso="Puede obtener una lista de archivos con . ls *.c "
```

```
echo $Aviso
```

```
Puede obtener una lista de archivos con . ls *.c
```

En este ejemplo el asterisco será considerado como cualquier otro carácter de la cadena y no será evaluado.

Sin embargo, las comillas dobles no alteran el significado del signo de dólar el operador que evalúa las variables. Los signos & próximos a un nombre de variable encerrados entre comillas dobles seguirán siendo evaluados y remplazarán el nombre de la variable con su valor. El valor de la variable formará parte de la cadena, y no el nombre de la variable.

Ejemplo

```
Ganador =David
```

```
Aviso="La persona que gano fue $Ganador"
```

```
echo $Aviso
```

```
La persona que gano fue David
```

Se puede cambiar el significado de los caracteres especiales como el operador \$ precediéndolos con una barra invertida. La barra invertida es útil cuando se desea evaluar variables dentro de una cadena e incluir caracteres \$.

Ejemplo:

```
Ganador =David
```

```
Aviso="$Ganador gano \"$100.00"
```

```
echo $Aviso
```

```
David gano $100.00
```

Sustitución de comandos

La *sustitución de comandos* o *expansión de comandos*, permite que la salida de un comando sustituya al propio comando. El formato utilizado es:

`$(comando)`

La sustitución de comandos puede ser anidada.

```
$ fecha=$(date)
$ echo $fecha
lun sep 5 12:27:33 CEST 2013
```

Un formato más antiguo de esta expansión es:

``comando``

```
$ fecha=`date`
$ echo $fecha
lun sep 5 12:27:33 CEST 2013
```

Nosotros usaremos el primero.

Comillas contrarias o invertidas :

Las comillas contrarias significan ejecutar el comando y sustituir por la salida.

`TODAY=`(set `date`; echo $1)``

Las comillas invertidas les permiten ejecutar un comando de Linux y utilizar los resultados como argumentos en la línea de comandos.

Ejemplo: `LISTC = `ls -l`` después si se realiza `echo LISTC` nos dará el resultado de realizar el comando `ls -l`.

```
# Este script displaya la fecha, hora, nombre de
# usuario y directorio actual
echo "Fecha y hora:"
date
echo
echo "Tu usuario es: `whoami` \n"
echo "Tu directorio actual es: `pwd`"
pwd
LISTC=`ls -l`
echo $LISTC
```

A una variable le podemos asignar un número, una cadena, el resultado de una operación, el resultado de un comando, entre otros

```
1  #!/bin/bash
2  CAD="Hola mundo"
3  NUM=1234
4  SUM=$(( $NUM+$NUM ))
5  TEXT="$CAD somos $(who | wc -l) usuarios conectados y $NUM+$NUM=$SUM"
6  echo $TEXT
```

Línea: 10 Col: 1 INS LÍNEA UTF-8 ejem.sh

masote@masote-virtual-machine:~/scripts\$./ejem.sh
Hola mundo somos 3 usuarios conectados y 1234+1234=2468

C02.PNG

5. READ

Podemos definir variables y asignarles un valor en tiempo ejecución pidiendo al usuario que introduzca el valor de dicha variable desde teclado.

La asignación de variables desde teclado se hace con el comando **read**. Por ejemplo, durante la ejecución de un guion se puede solicitar al usuario que responda a una pregunta.

```
echo Por favor escriba un saludo;
read saludo
echo "El saludo que escribió fue $saludo"
```

Por favor escriba un saludo:

Buenos días

El saludo que escribió fue **Buenos días**

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)

```
1  #!/bin/bash
2  # recogiendo un variable por teclado
3  echo -n "Introduzca número1: "
4  read NUM1
5
6  # igual que lo anterior pero en un solo paso
7  read -p "Introduzca número2: " NUM2
8
9  # recogiendo dos variables
10 read -p "Introduzca nombre y edad: " NOM ED
11
12 # recoge la variable sin darle a enter
13 echo -n "Año de nacimiento: "
14 read -n4 AN
15
16 # recoge variable sin mostrar caracteres
17 echo -ne "\nIntroduce password: "|
18 read -s PASS
19
20 echo -e "\n\nNum1: $NUM1 - Num2: $NUM2"
21 echo -e "Nombre: $NOM - Edad:$ED\nAño: $AN"
22 echo "Password: $PASS"
```

Línea: 17 Col: 34 INS LÍNEA UTF-8 ejem.sh

```
masote@masote-virtual-machine:~/scripts$ ./ejem.sh
Introduzca número1: 18
Introduzca número2: 25
Introduzca nombre y edad: Juan 30
Año de nacimiento: 1983
Introduce password:

Num1: 18 - Num2: 25
Nombre: Juan - Edad:30
Año: 1983
Password: q1w2e3
```

Ejemplo

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



```
Read -s -n1 -p "¿si (S) o no (N)?" respuesta
Echo -e "\n la respuesta es:" $respuesta
```

6. EVALUAR EXPRESIONES, TEST Y CORCHETES

El comando test permite evaluar si una expresión es verdadera o falsa. Los tests no sólo operan sobre los valores de las variables, también permiten conocer, por ejemplo, las propiedades de un fichero.

Principalmente se usan en la estructura if/then/else/fi para determinar qué parte del script se va a ejecutar. ¿ Se pueden evaluar otras expresiones, como una lista de órdenes (usando su valor de retorno), una variable o una expresión aritmética, básicamente cualquier orden que devuelva un código en \$?

EL comando test es un comando importante dentro de los guiones, su función es la de realizar una evaluación lógica de expresiones. El comando test realiza entre otras funciones: chequeo de tipos de ficheros, comparación de cadenas y comparación de enteros.

test evalúa la expresión que se le pasa como parámetro y si su valor es cierto retorna un código o estatus 0, en caso contrario retorna un estatus distinto de cero. ¿ El estatus está contenido en la variable interna "\$?".

La sintaxis para evaluar puede ser una de las dos que se muestran a continuación:

```
EDAD1=10
EDAD2=20
# Es necesario dejar espacios entre operador y operandos
if test $EDAD1 -lt $EDAD2;then
    echo Se cumple
fi
# Si utilizamos corchetes hay que dejar espacios antes y despues del corchete
if [ $EDAD1 -lt $EDAD2 ];then
    echo Se cumple
fi
```

```
if test $EDAD1 -lt $EDAD2
then
    echo "Se cumple"
fi
```

La palabra test puede ser sustituida por corchetes "[]", en este caso entre los corchetes y la expresión a evaluar se deben dejar espacios en blanco de forma que el shell reconozca las palabras [y].

De esta forma test \$var -eq 2 es equivalente a [\$var -eq 2].

```
If [ $EDAD1 -lt $EDAD2 ]
Then
    Echo "Se cumple"
Fi
```


Comparación de cadenas de caracteres

Operador	Significado
<code>cadena1 = cadena2</code>	Cierto si las cadenas de caracteres son iguales.
<code>cadena1 != cadena2</code>	Cierto si las cadenas de caracteres no son distintas.
<code>cadena</code>	Cierto si la cadena de caracteres no es nula.

Operadores de comparación de cadenas alfanuméricas	
<code>Cadena1 = Cadena2</code>	Verdadero si Cadena1 es IGUAL a Cadena2
<code>Cadena1 != Cadena2</code>	Verdadero si Cadena1 NO es IGUAL a Cadena2
<code>Cadena1 < Cadena2</code>	Verdadero si Cadena1 es MENOR a Cadena2
<code>Cadena1 > Cadena2</code>	Verdadero si Cadena1 es MAYOR que Cadena2

Expresiones

Una expresión puede ser: [comparación de cadenas](#), [comparación numérica](#), [operadores de fichero](#) y [operadores lógicos](#) y se representa mediante [\[expresión\]](#):

Comparación de cadenas:

- `=`
- `!=`
- `-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)

Comparación de enteros

Operador	Significado
<code>-eq</code>	igual a
<code>-ne</code>	distinto de
<code>-lt</code>	menor que
<code>-le</code>	menor que o igual a
<code>-gt</code>	mayor que
<code>-ge</code>	mayor que o igual a

Operadores de comparación de valores numéricos.	
Numero1 -eq Numero2	Verdadero si Numero1 es IGUAL a Numero2. (equal)
Numero1 -ne Numero2	Verdadero si Numero1 NO es IGUAL a Variable2. (not equal)
Numero1 -lt Numero2	Verdadero si Numero1 es MENOR a Variable2. (less that)
Numero1 -gt Numero2	Verdadero si Numero1 es MAYOR que Variable2. (greater that)
Numero1 -le Numero2	Verdadero si Numero1 es MENOR O IGUAL que Numero2. (less or equal).
Numero1 -ge Numero2	Verdadero si Numero1 es MAYOR O IGUAL que Numero2 . (greater or equal).

Por ejemplo, si tecleo:

```
$var=2
$test $var -eq 2
$echo $?
0
```

Operadores aritméticos

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

Ejemplo

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

Operadores lógicos

- **Negación (!)**

Formato: test ! (expresión)

retorna un estatus 1 si la expresión es verdadera, en caso contrario un 0.

- **AND (-a)**

Formato: `test (expresión1) -a (expresión2)`
retorna un estatus 0 si ambas expresiones son verdaderas.

- **OR (-o)**

Formato: `test (expresión1) -o (expresión2)`
retorna un estatus 0 , tanto en el caso de que una de las expresiones sea verdadera o ambas lo sean.

Expresiones

Operadores lógicos:

- **!** NOT
- **-a** AND
- **-o** OR

Ejemplo (`comparacion_logical.sh`)

```
#!/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
```

Expresiones

Operadores lógicos:


- **&&** AND
- **||** OR

Ejemplo (**comparacion_logica2.sh**)

```
#!/bin/bash
echo -n "Introduzca un número 1 < x < 10: "
read num
if[ "$number" -gt 1 ] && [ "$number" -lt 10 ];
then
    echo "$num*$num=$(( $num*$num ))"
else
    echo "Número introducido incorrecto !"
fi
```

La instrucción **let** se puede utilizar

```
- $ let X=10+2*7
$ echo $X
24
$ let Y=X+2*4
$ echo $Y
32
```



Un expresión aritmética se puede evaluar con **`$[expression]`** o **`$((expression))`**

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

Ejemplo (`operaciones.sh`)

```
#!/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"
```

Comprobación de ficheros:

Opción	Significado
-d	comprueba si es un directorio
-f	comprueba si es un fichero de datos
-e	comprueba si existe

7. ESTRUCTURAS DE CONTROL

7.1. ESTRUCTURA IF

Se trata de una estructura de control condicional que permite redirigir el curso de la acción según la evaluación de una condición. La condición puede ser múltiple y estas estructuras se pueden anidar.

La estructura de control if examina el estado que devuelve la condición y si es cero ejecuta las órdenes que siguen al then, en caso de que devuelva un valor distinto de cero, no se ejecuta nada.

El comando “if” se puede utilizar con tres formatos diferentes, que son los siguientes:

Formato 1:

if condición		if condición ; then
then	==	mandatos
mandato/s		fi
fi		

La instrucción “fi” marca el fin del “if” y es necesario que cada “if” lleve su “fi”, en caso contrario nos dará un error de sintaxis en la ejecución del programa.

La estructura de control if examina el estado que devuelve la condición y si es cero ejecuta las órdenes que siguen al then, en caso de que devuelva un valor distinto de cero, se ejecuta las órdenes que hay a continuación del else. .

Formato 2:

```
if condición  
then  
    mandato/s  
else  
    mandato/s  
fi
```

La introducción de la instrucción “else” convierte a la estructura “if” en una rama doble. Si la condición que se examina devuelve un código de retorno 0, la estructura if ejecuta los mandatos que están entre las instrucciones “then” y “else” y pasa el control a la instrucción que sigue a “fi”. Si la condición retorna un estatus falso, se ejecutan los mandatos que siguen a la instrucción “else”.

Formato3:

```
if condición  
then  
    mandato/s  
elif condición  
    then  
        mandato/s  
    else  
        mandato/s  
fi
```

La instrucción “elif” combina las instrucciones “else” e “if” y permite construir un conjunto anidado de estructuras “if-then-else”.

- La condición se encierra normalmente entre corchetes
- Es imprescindible cerrar el **if**
- Puede haber múltiples **elif** pero un solo **else**
- Sólo es necesario colocar el punto y coma detrás de la condición si vamos a colocar en la misma línea el **then**.

Un ejemplo sencillo

```

1  #!/bin/bash
2  clear
3  read -p "Introduzca la edad: " EDAD
4  if [ $EDAD -gt 80 ];then
5      echo Anciano
6  elif [ $EDAD -ge 65 ];then
7      echo Jubilado
8  elif [ $EDAD -ge 18 ];then
9      echo Adulto
10 elif [ $EDAD -ge 12 ];then
11     echo Juvenil
12 else echo infantil
13 fi

```

Por ejemplo:

\c no hace salto de línea, sino que permanece en la misma línea

\n salto de línea

\$cat if3

echo “introduce valor 1 \c”; read var1

echo “introduce valor 2 \c”; read var2

echo “introduce valor 3 \c”; read var3

if test “\$var1” = “\$var2” -a “\$var1” = “\$var3” # if [“\$var1” = “\$var2” -a “\$var1” = “\$var3”]

then

echo “los tres valores son iguales”

else

if [“\$var1” = “\$var2”]

then

echo “los dos primeros valores son iguales”

else

if [“\$var1” = “\$var3”]

then

echo “el primer y tercer valor son iguales”

else

if [“\$var2” = “\$var3”]

then

echo “los dos últimos valores son iguales”


```

        else
            echo "ningún valor es igual"
        fi
    fi
fi

```

7.2. ESTRUCTURA CASE

El comando case permite tomar una decisión entre múltiples casos. Es más adecuada que una secuencia de if anidados cuando se saben con antelación los posibles valores que puede tomar una variable.

Formato:

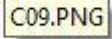
```

case $variable in
    patron-1) mandato1 ;;
    patron-2) mandato2 ;;
    patron-3) mandato3 ;;
    .....
    patron-n) mandaton ;;
    *)      mandato  ;;
esac

```

ejemplo: echo elige un número
 read numero
 case \$numero in
 1) Echo el número es el uno;;
 2) Echo el número es el dos;;
 esac

```

case $VARIABLE in
    VALOR1)
        orden
        orden
        ordenfinal;;
    VALOR2)
        orden
        orden
        ordenfinal;;
    *) 
        ordendefault
        ordendefault
        ordendefaultfinal;;
esac

```


La sentencia case compara la variable que hay delante de la palabra “in”, con cada uno de los valores o variables que hay delante del paréntesis. En caso de que alguno de los valores sea igual al de la variable que hay delante de la palabra “in”, ejecuta las órdenes que hay a continuación del paréntesis y al encontrar “;;” bifurca al “esac”, desde donde sigue la ejecución del programa en secuencia.

- Cada una de las órdenes en cada caso se termina directamente con un salto de línea y cada sentencia final se termina con un “;;”
- Si no pusiéramos “;;” se seguiría ejecutando hasta encontrar el siguiente “;;” o el final del case.
- El *) representa el valor por defecto y se ejecutaría si el Valor de la variable no se corresponde con ninguno de los valores

El patrón de la estructura case es análogo al de una referencia mediante patrones a un fichero, pueden usarse los siguientes caracteres especiales:

- * Un asterisco equivale a cualquier cadena de caracteres y también se utiliza como opción por defecto del case, es decir que si la variable no coincide con ninguno de los patrones se ejecutará la sentencia o sentencias que tengamos a continuación del “*”.
- ? Una interrogación equivale a cualquier carácter simple.
- | Permite poner dos opciones en la misma línea para un mismo resultado.

Ejemplo. Calcular el horóscopo chino según el año de nacimiento

```
1  #!/bin/bash
2  clear
3  read -p "Año de nacimiento (4 cifras):" AN
4  let RESTO=$AN%12
5  case $RESTO in
6    0)HOROS="Mono";;
7    1)HOROS="Gallo";;
8    2)HOROS="Perro";;
9    3)HOROS="Cerdo";;
10   4)HOROS="Rata";;
11   5)HOROS="Buey";;
12   6)HOROS="Tigre";;
13   7)HOROS="Conejo";;
14   8)HOROS="Dragon";;
15   9)HOROS="Serpiente";;
16   10)HOROS="Caballo";;
17   11)HOROS="Cabra";;
18   *)HOROS="Error";;
19  esac
20  echo "Tu horóscopo chino es $HOROS"
```

```
Línea: 24 Col: 1      INS LÍNEA UTF-8 ejem.sh
Año de nacimiento (4 cifras):1973
Tu horóscopo chino es Buey
```

BUCLES

Las estructuras until y while son muy parecidas. Sólo difieren en el sentido de la prueba al inicio del ciclo

Estructura UNTIL

until condición

do

mandato/s

done

Estructura WHILE

while condición

do

mandato/s

done

7.1. ESTRUCTURA WHILE

Se trata de una estructura de control de tipo repetitivo en la que al inicio se evalúa la condición y si resulta verdadera se ejecutan las instrucciones que encierra.

```
while [ condicion ];do  
    # CODIGO A EJECUTAR  
done
```

while [expresión]; do
estas líneas se repiten MIENTRAS la expresión sea verdadera
done

La ejecución de la construcción de un bucle while es como sigue:

- Se evalúa la condición.
- Si el código devuelto por la condición es 0 (verdadero), se ejecuta la orden u órdenes y se vuelve a iterar.
- Si el código de retorno de la condición es falso, se salta a la primera orden que haya después de done.

La ejecución de la construcción de un bucle until es como sigue:

- Se evalúa la condición.
- Si el código devuelto por la condición es distinto de 0 (falso), se ejecuta la orden u órdenes y se vuelve a iterar.
- Si el código de retorno de la condición es 0 (verdadero), se salta a la primera orden que haya después de done.

Para crear bucles infinitos se puede utilizar el operador true, que devuelve un código de 0.

Ejemplo de validación usando while

```
1  #!/bin/bash  
2  clear  
3  read -p "Introduce una nota (de 0 a 10): " NOTA  
4  while [ $NOTA -lt 0 -o $NOTA -gt 10 ];do  
5      clear  
6      echo "$NOTA está fuera de los límites !!"  
7      read -p "Introduce una nota (de 0 a 10): " NOTA  
8  done  
9  echo Test de validación pasado
```

Comando break

El comando break **rompe** la ejecución de un **bucle** y continúa la ejecución del programa en secuencia. Sólo se aplica a los comandos for, while, until .

Comando exit

La sentencia exit permite **abandonar** la ejecución de un **shellscript** en el momento que nosotros deseemos. En caso de que lo utilicemos dentro de un bucle rompe el bucle y abandona el shellscript.

7.2. Estructura UNTIL

until [expresión]; do

estas líneas se repiten HASTA que la expresión sea verdadera

done

```
1  #!/bin/bash
2  clear
3  read -p "Introduce una nota (de 0 a 10): " NOTA
4  until [ $NOTA -ge 0 -a $NOTA -le 10 ];do
5      clear
6      echo "$NOTA está fuera de los límites !!"
7      read -p "Introduce una nota (de 0 a 10): " NOTA
8  done
9  echo Test de validación pasado
```

Bucle infinito.

```
1  #!/bin/bash
2  while true; do
3      clear
4      echo 1-Sumar
5      echo 2-Restar
6      echo 3-Multiplicar
7      echo 4-Dividir
8      echo 5-Salir
9      echo
10     read -p "Introduce una opción: " OP
11     case $OP in
12     1)echo "$1 + $2 = $((($1+$2))"
13         read pause;;
14     2)echo "$1 - $2 = $((($1-$2))"
15         read pause;;
16     3)echo "$1 * $2 = $((($1*$2))"
17         read pause;;
18     4)echo "$1 / $2 = $((($1/$2))"
19         read pause;;
20     5)echo "No has decidido nada"
21         break;;
22     esac
23 done
```

```
#!/bin/bash
uno=1
dos=2
while true
do

    clear
    echo 1- Sumar
    echo 2- Restar
    echo 3- Multiplica
    echo 4- Dividir
    echo 5- Salir
    echo
    read -p "Introduce una opción:" OP
    case $OP in
        1) echo "$uno + $dos = $[ $uno+$dos ]"
            read pause;;
        2) echo "$uno - $dos = $[ $uno-$dos ]"
            read pause;;
        3) echo "$uno * $dos = $[ $uno*$dos ]"
            read pause;;
        4) echo "$uno / $dos = $[ $uno/$dos ]"
            read pause;;
        5) echo " No has decidido ninguna opcion correcta"
            break;;
    esac
done
```

Línea: 30 Col: 1 INS LÍNEA

```
1-Sumar
2-Restar
3-Multiplicar
4-Dividir
5-Salir
```

Introduce una opción:

7.3. Estructura for

La instrucción `for` permite ejecutar un bloque de instrucciones un número determinado de veces. Se evalúa la condición y mientras sea cierta se ejecuta el bloque de instrucciones; el proceso continuará hasta que la evaluación de la condición sea falsa

Formato:

```
for variable in lista
do
    orden
done
```

Variable puede ser cualquier variable del shell, y lista es una lista compuesta de cadenas separadas por blancos o tabuladores. La construcción funciona como sigue:

- Se asigna a variable la primera cadena de la lista.
- Se ejecuta la orden u órdenes del bucle.
- Se asigna a variable la siguiente cadena de la lista. Se devuelve al segundo paso.
- Se repite hasta que se hayan usado todas las cadenas.

Después de que haya acabado el bucle, la ejecución continúa en la primera línea que sigue a la palabra clave done.

```
for VARIABLE in conjunto
```

```
do
```

Estas líneas se repiten una vez por cada elemento del conjunto,

Y variable va tomando los valores del conjunto uno por uno.

```
done
```

Ese conjunto que aparece en la estructura del for, es normalmente un conjunto de valores cualesquiera, separados por espacios en blanco o retornos de línea. Así, si queremos mostrar los días de la semana por pantalla podríamos hacerlo mediante este script:

```
For DIA in lunes martes miércoles jueves viernes sábado domingo
```

```
do
```

```
    echo "día de la semana: $DIA"
```

```
done
```

Así, por ejemplo, si queremos obtener por pantalla los números del 1 al 10 podríamos hacerlo de la siguiente forma:

```
For NUM in 1 2 3 4 5 6 7 8 9 10
```

```
do
```

```
    echo "NUM vale $NUM en este paso"
```

```
    if [ $NUM -eq 5 ]
```

```
    then
```

```
        break
```

```
    fi
```

```
done
```

La salida sera 1 2 3 4 5

La potencia del comando for viene de la flexibilidad de valores que admite el conjunto de valores, ya que podemos crear dicho conjunto con una orden del sistema operativo. En el siguiente ejemplo vamos a usar como conjunto los nombres de los ficheros con extensión sh del directorio actual:

```
#!/bin/bash
for NOMBRE in $( ls *.sh ); do
    echo "Un script es $NOMBRE"
done
```

La sentencia continue se salta la iteración.

La sentencia “continue” suspende la secuencia de ejecución y vuelve al principio del bucle. Se utiliza con el comando “for” para suspender la secuencia de ejecución y volver al principio del bucle conservando los valores de las variables.

Por ejemplo:

```
for var in 1 2 3 4 5 6 7
do
    if test "$var" = "2"
    then
        continue
    else
        echo "el valor de la variable \ $var es $var"
    fi
done
```

La salida será: 1 3 4 5 6 7

```
for ((inicio;mientras;incremento));do
    # CODIGO A EJECUTAR
done
```

- **inicio:** es el valor inicial del que parte la variable
- **mientras:** es la condición que se evalúa en cada repetición
- **incremento:** es como se modifica la variable en cada repetición


```
1  #!/bin/bash
2  clear
3  for ((i=1;i<=5;i++));do
4      subtotal=$((i*i))
5      acumulador=$((acumulador + subtotal))
6      echo "Rep_$i --> subtotal: $subtotal "
7  done
8  echo -e "\t\t\t ---"
9  echo -e "\t\t\t Total: $acumulador "
```

Línea: 12 Col: 1 INS LÍNEA UTF-8 ejem.sh

```
Rep_1 --> subtotal: 1
Rep_2 --> subtotal: 4
Rep_3 --> subtotal: 9
Rep_4 --> subtotal: 16
Rep_5 --> subtotal: 25
          ---
          Total: 55
```

8. PARAMETROS

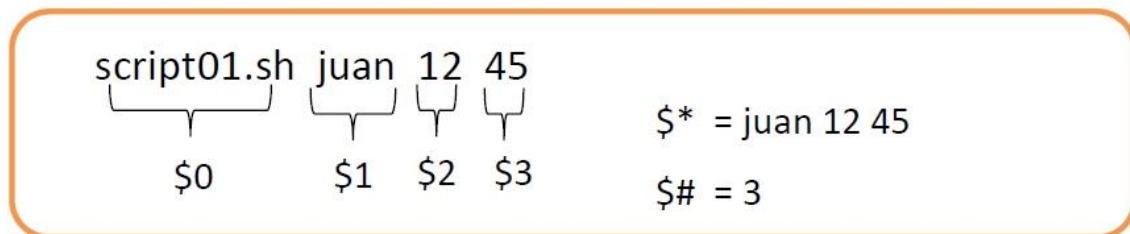
En muchas ocasiones necesitamos que nuestros scripts reciban información de fuera a la hora de ejecutarlos, y para esto vamos a utilizar los parámetros de posición. Podemos pasar parámetros tanto a los scripts como a las funciones. Los parámetros \$0,\$1....\$9,\$*,\$#.

script01.sh

```
#!/bin/bash
#Parametros
Echo "El primer parámetro que se ha pasado es $1"
Echo "El segundo parámetro que se ha pasado es $2"
Echo "El tercer parámetro que se ha pasado es $3"
Echo "El conjunto de todos los parámetros es $*"
Echo "Me has pasado un total de $# parámetros"
Echo "El parámetro 0 es el nombre del script $0"
```

Ejecutamos sh script01.sh Caballo Perro 675 Madrid
Entonces el resultado por pantalla es
El primer parámetro que se ha pasado es Caballo
El segundo parámetro que se ha pasado es Perro
El tercer parámetro que se ha pasado es 675

El conjunto de todos los parámetros es Caballo Perro 675 Madrid
 Me has pasado un total de 4 parámetros
 El parámetro 0 es el nombre del script script01.sh



Es decir, después de ejecutar cualquier orden o comando del sistema (o casi cualquier orden, mejor dicho) podemos comprobar el valor de \$? que tendrá un 0 si todo ha ido bien, y otro valor cualquiera en caso de que haya fallado. Comprobarlo es muy simple:

Desde la línea de comandos, haced un cd a un directorio que no existe, por ejemplo cd /juegos/faluyah y luego mirad el contenido de \$? con un echo \$?

Comprobareis como \$? vale 1, es decir, indica que la última orden no funcionó correctamente.

Ahora haced un cd a un directorio que si exista cd /etc/network y luego mirad el contenido de \$? con un echo \$?

Comprobareis como vale 0, es decir, indica que la última orden funciono sin problemas.

Ejemplo:

Internas

```
Echo "El valor de \$1 es $1"
Echo "El valor de \$2 es $2"
Echo "El valor de \$3 es $3"
Echo "El número de argumentos es $#"
```

```
Echo " los argumentos introducido son :\c"
```

```
Echo $*
```

Internas a b c

```
El valor de $1 es a
El valor de $2 es b
El valor de $3 es c
El número de argumentos es 3
los argumentos introducidos son : a b c
```

Comando eval: el comando eval ejecuta el contenido de una variable cada vez que hacemos referencia a ella.

Formato: eval nombre-variable

Ejemplo:

```
B=date
```

```
Eval $B
```

Sun Mar 12 18:21:46 ESP 1995

2.3 Parámetros y Variables

Un **parámetro posicional** o parámetro de posición es una variable dentro de un programa shell cuyo valor se establece a partir de un argumento especificado en la línea de comandos que invoca al programa.

Los parámetros posicionales de los guiones se referencian de forma consecutiva desde el 0 hasta el 9. Pero ya sabemos que para acceder al valor de una variable se debe anteponer al nombre el carácter \$, (entonces si los parámetros son variables) de forma que se puede acceder a dichos parámetros desde \$0 a \$9.

Parámetros	
\$1	Devuelve el 1º parámetro pasado al script o función al ser llamado.
\$2	Devuelve el 2º parámetro.
\$3	Devuelve el 3º parámetro. (Podemos usar hasta \$9).
\$*	Devuelve todos los parámetros separados por espacio.
\$#	Devuelve el número de parámetros que se han pasado.
\$0	Devuelve el parámetro 0, es decir, el nombre del script o de la función.

- \$0 almacena el nombre del propio script.
- \$# almacena el Nº de parámetros que se han recibido.
- \$* guarda una lista completa de todos los parámetros separados por espacios.
- @\$ igual que el anterior.
- \$? almacena el código de error de la orden inmediatamente anterior que se ha ejecutado (0 si ha ido bien, distinto de cero en caso de error).

El parámetro posicional **\$0** hace referencia al nombre del comando o nombre del fichero ejecutable que contiene el shell script que se está ejecutando y desde \$1 hasta \$9 son los parámetros que se han ejecutado desde la línea de comandos cuando se invocó el shell script.

Ejemplo: sh prog1 luis pepe juan ana, entonces dentro del programa "prog1" al parámetro de posición \$1 se le asignara el valor "luis" al parámetro \$2 se le asignara el valor "pepe" y así sucesivamente.

Ejemplo de paso de argumentos :

```
#!/bin/bash
# Este script lista los cinco primeros argumentos
echo "Los cinco primeros argumentos de la línea"
echo "de comandos son" $1 $2 $3 $4 $5

nombre_shell_Script a pint john o.k.
```

Veamos esto a través de un ejemplo:

```

1  #!/bin/bash
2  ▼ if [ $# -eq 0 ];then
3      echo "Introduzca parametros a $0"
4      exit
5  fi
6  AC=0
7  echo "El nombre del fichero es: " $0
8  echo "Se han pasado $# parámetros al programa"
9  echo "La lista de parametros es --->[ $* ]<---"
10 ▼ for i in $*;do
11     AC=$((AC+$i))
12 done
13 echo "La suma de todos los parametros es: $AC"

```

Línea: 16 Col: 1 INS LÍNEA UTF-8 ejem.sh

```

masote@masote-virtual-machine:~/scripts$ ./ejem.sh 10 15 20 25
El nombre del fichero es: ./ejem.sh
Se han pasado 4 parámetros al programa
La lista de parametros es --->[ 10 15 20 25 ]<---
La suma de todos los parametros es: 70

```

Comando set:

- Set sin parametros visualiza la lista de variables en curso.
- Se puede cambiar el valor de los parámetros posicionales con el comando set seguido de los nuevos valores que se deseen dar. **Con el comando set podemos asignar nuevos valores a los parámetros de posición.**

Formato: set valor1 valor2 ...valorn donde, valor1... valorn pueden ser cadenas o expresiones set.

Ejemplo: cat ejem_set1

```

Echo "El valor de $1 es ...$1"
Echo "El valor de $2 es ...$2 "
Echo "El valor de $3 es ...$3"
Set `date`
Echo "El valor de $1 es ...$1"
Echo "El valor de $2 es ...$2 "
Echo "El valor de $3 es ...$3"
Set 33 "es una cadena" "fin"
Echo "El valor de $1 es ...$1"
Echo "El valor de $2 es ...$2 "
Echo "El valor de $3 es ...$3"

```

Ejem_set1 Juan Luis Jorge

```

El valor de $1 es ...Juan
El valor de $2 es ...Luis

```

El valor de \$3 es ...Jorge
El valor de \$1 es ...Sat
El valor de \$2 es ...Mar
El valor de \$3 es ...11
El valor de \$1 es ...33
El valor de \$2 es ...es una cadena
El valor de \$3 es ...fin

DESPLAZAMIENTO CON SHIFT

Cuando trabajamos con parámetros, podemos pasar al script tantos como necesitemos, pero una vez que lo estemos ejecutando, sólo podemos acceder a los parámetros `$1...$9`. Para poder acceder al resto de parámetros a través de las variables implícitas se usa la orden **shift**, que sirve para desplazar (eliminar) el primer parámetro y dejar hueco para el siguiente. Con **shift n** desplazaríamos n parámetros de una sola vez.

También podemos trasladar los valores de los parámetros posicionales en orden decreciente con el comando **shift**. El formato es **shift número**, donde número indica el número de posiciones que se desplazarán los parámetros. El comando **shift** sin argumentos traslada una posición los valores de los parámetros posicionales, el que era antes `$2` ahora es `$1` y así sucesivamente.

```
1  #!/bin/bash
2  clear
3  NP=$#
4  for ((i=1;i<=$NP;i++));do
5      echo $1 $2 $3 $4 $5 $6 $7 $8 $9
6      shift
7  done
8  # razona que ocurre si se usa $# directamente
```

Línea: 10 Col: 1 INS LÍNEA UTF-8 ejem.sh

```
1 2 3 4 5 6 7 8 9
2 3 4 5 6 7 8 9 10
3 4 5 6 7 8 9 10 11
4 5 6 7 8 9 10 11 12
5 6 7 8 9 10 11 12 13
6 7 8 9 10 11 12 13
7 8 9 10 11 12 13
8 9 10 11 12 13
9 10 11 12 13
10 11 12 13
11 12 13
12 13
13
```

Solo se pueden pasar nueve argumentos, pero se puede acceder a mas de nueve usando el comando shift.

Ejemplo: cat ejem_set2

```
Echo "El valor de $1 es ...$1"
Echo "El valor de $2 es ...$2 "
Echo "El valor de $3 es ...$3"
Shift 3
Echo "El valor de $1 es ...$1"
Echo "El valor de $2 es ...$2 "
Echo "El valor de $3 es ...$3"
```

Ejem_set2 a bb c d e f

```
El valor de $1 es ...a
El valor de $2 es ...b
El valor de $3 es ...c
El valor de $1 es ...d
El valor de $2 es ...e
El valor de $3 es ...f
```

Cuando hemos realizado la llamada al shellscript, hemos introducido seis valores (a,b,c,d,e,f), los cuales hemos visualizado mediante tres parámetros posicionales \$1, \$2, \$3, haciendo un traslado a la izquierda de tres posiciones mediante la utilización del comando shift.

Además de las definiciones de variables de forma directa que hemos visto en el primer apartado de esta Unidad de Trabajo, los valores asignados a las variables estarán formados por cualquier conjunto de caracteres alfabéticos y numéricos. En la mayoría de los casos deberá entrecomillar los valores utilizando comillas simples, comillas dobles, barras invertidas o comillas invertidas.

VER COMO MINIMO HASTA AQUÍ.