Programación de scripts bajo shell de LINUX

Que es un script?

Es un tipo de "nueva" programación (interpretada) o estilo de lenguajes que se impone dia a dia en el mundo de la informática, debido principalmente a su facilidad, funcionabilidad. Generalmente los scripts son programados con lenguajes de relativa facilidad pero no por ello son menos funcionales que cualquier otro programa realizado en un lenguaje de alto nivel (C, pascal, delphi, Vbasic, etc).
Los scripts (guiones) son meros ficheros de texto ASCII que presentan una serie de ordenes y estructuras organizadas secuencialmente. Así pues el script se ejecuta de forma lineal-secuencial orden tras orden.
Los lenguajes scripts son lenguajes "interpretados". Lo que quiere decir que sus ordenes secuenciales son interpretadas por un sistema o programa padre.
Hoy en día es muy habitual haber oido hablar de este tipo de programación, dando lugar a una serie de lenguajes-scripts, cada uno con su propia sintaxis y ordenes especiales, pero todos ellos presentan algo en común y es que no dejan de ser guiones.
Algunos lenguajes-scripts son:

Para el desarrollo de webs

- JavaScripT: Lenguaje-script que permite la creación de paginas webs dinámicas locales (en el propio navegador).
- PHP,ASP,etc...:Lenguajes-scripts que permiten la creación de paginas webs dinámicas en servidor.

Para los clientes de IRC:

- mIRC: Presenta su propio lenguaje script (revolucionario donde los halla).
  Podemos aumentar con el, la potencia de nuestro mIRC, claro ejemplo de script sería el OrioN ScripT para mIRC (www.ircorion.cjb.net).

Y un larga lista, que seguirá aumentando día a día.

Centrándonos mas en los scripts para LINUX

El sistema operativo LINUX esconde tras de si una larga lista de utilidades. Tales utilidades se ejecutan a través de comandos bastante complicados, y con muchos parámetros o argumentos. Lo que hace un script es recoger todos esos comandos y presentarlos al usuario de una forma sencilla y de rápido acceso. Básicamente los scripts son pequeñas y potentes aplicaciones creadas por el programador para facilitar la tarea al usuario.
Si alguno conoce el S.O. MSDOS sabra de la existencia de sus bien conocidos bats (ficheros de procesamiento por lotes), bien pues los scripts en linux seria el homologo. Aunque todo sea dicho, los scripts en linux presentan un mayor potencial frente a los bats, como lenguaje de programación; usando estructuras condicionales, bucles iterativos típicos, recursividad, mayor numero de palabras reservadas, etc.

Con que crear un script?

Al ser un lenguaje interpretado, no necesitamos compilar su código para poder ejecutarlo, con lo que solo necesitamos escribir ASCII en un fichero. Para ello podemos usar cualquiera de nuestros editores preferidos. En linux io personalmente uso VI. A la hora de guardar el fichero, bastaría con salir del editor pulsando ":wq" para salir y guardar cambios.
Abriendo un fichero con VI: **vi miScript**
Con este comando crearíamos y abriríamos un fichero con el nombre de miScript. (vease mas información sobre el editor VI en tutorial “Comandos básicos Linux” by Quasi ayuda-internet.net).

Como ejecutar un script?

Bastaría con poner: sh namefile
sh: Era la orden encargada de ejecutar scritps. A continuación el nombre del fichero que contiene el guión.

Que es el shell?
El shell es un programa-interface, que se provee como un elemento comunicacional entre el usuario y el S.O.
El shell presenta dos grandes funciones, que son:

- Servir de interprete de comandos: Acepta los comandos escritos y se encarga de su ejecución. A la hora de interpretar comandos, puede pasar que se trate de comandos internos del propio SO de tal forma que el mismo (shell) se encarga de su propia ejecución, o que se trate de ordenes o comandos realizados por los usuarios (scripts, etc..) en tal caso da paso al KERNEL y es este quien ejecuta y procesa estas ordenes.
- Servir como interprete de programación: Presentando las mismas características de un lenguaje de programación interpretado de alto nivel. Como ya dije el equivalente a MSDOS. Pero muchos mas potente.

Una vez vista esta teoría vamos a pasar a explicar la sintaxis de este lenguaje.

Sintaxis del lenguaje-script de programación en shell LINUX

Entrecomillados y caracteres especiales
Existen 3 tipos de entrecomillados:

- Comillas simples: El contenido no es interpretado por el shell. Se suele usar para indicar caracteres literales. Por ejemplo 'a' es tomada como un carácter literal no es procesada por el shell.
- Comillas dobles: Agrupa una cadena de caracteres o string. Por ejemplo: "OrioN ScripT www.ircorion.cjb.net". Se suelen usar para almacenar una string entera en una variable. Por ejemplo: VAR="hola soy Quasi", aqui estaríamos almacenando en la variable VAR el texto hola soy quasi.
  Para ver el resultado basta con hacer: echo $VAR.
- Comillas invertidas: Hacen que se evalue el contenido y que su resultado sea devuelto como el de una variable. Por ejemplo. verfecha=`date`
  Si ejecutamos esto, echo $verfecha, veríamos como el valor devuelto por el comando date es devuelto como una variable y almacenado en otra variable (verfecha).

Algunos caracteres especiales son:

- Separando comandos: Linux interpreta comandos de forma lineal-secuencial de ahi que si en en un script escribimos:
  echo HOLA1
  echo HOLA2
  echo HOLA3
  Se ejecutaría un comando tras otro, pero si pusiéramos:
  echo HOLA1 echo HOLA2
  Estaríamos cometiendo un fallo sintáctico y es que el interprete de linux esta preparado para ejecutar orden tras orden de forma lineal y secuencial y aquí se rompe esa secuencia. Si queremos hacer esto deberemos especificar el fin de un comando para el comienzo de otro con el signo ";"
  echo HOLA1; echo HOLA2; echo HOLA3
  Esto si estaria bien expresado, para linux.
- Insertando comentarios: A la ahora de programar todo buen programador sabe de la importancia de incluir comentarios en su código, esto facilitaría su trabajo de actualización, comprensión, rapidez, etc. Además si el trabajo es en grupo el resto de programadores podrán saber la finalidad del código sin tener que descifrarlo. UN BUEN PROGRAMADOR CREO QUE DEBE PROGRAMAR COMENTANDO SU CODIGO, a no ser excepciones especiales. ;)
  Bien, en linux para insertar un comentario dentro del código del script se debe preceder de ese comentario el signo "#".
  Ejemplo:
  #esta orden muestra la fecha
  date
  En este caso el comentario es totalmente inutil seria una risa :P pero como ejemplo no esta mal. El interprete al llegar a la primera línea de código y empezar por el principio detecta la almohadilla y pasa automáticamente a la segunda línea de código (algo así aria internamente.
- Comodines:
  1) Signo "*": el asterisco es un comodín que funciona como sustitución de cadena de literales.
  2) Signo ?: Sirve como comodín que sustituye un carácter. Si añadimos mas interrogantes sustituirá tantos caracteres como interrogantes haya.
  3) Carácter (~) “alt+126”: Este carácter devuelve la ruta del home del usuario.

Utilización de parámetros : Esta es una utilidad muy preciada en la programación. Se trata de pasar uno o varios parámetros al programa principal para luego operar con ellos.
Se denominan parámetros posesiónales. Esto es porque preceden y son gestionados de forma posiciona.
Mejor verlo con un ejemplo:

sh PasoParametros Quasi pukii MoAsT
Si ejecutamos esto, estaríamos diciendo que le pasamos al script con nombre "PasoParametros" 2 parámetros que son: Quasi, pukii, y moast.
Dentro del script podríamos operar con estos parámetros, puesto que tales son almacenados en variables del tipo: $n
Con lo que la variable $1 devolvería el primer parámetro dado al script (Quasi), $2 daria (pukii), y $3 (moast).
El script podria operar con esto, de esta forma (codigo):

```
echo "Parametro 1: $1"
echo "Parametro 2: $2"
echo "Parametro 3: $3"
echo "Todos los parámetros pasados son: $*"
echo "El numero de parámetros pasados es: $#"
echo "El nombre del script (parámetro 0) es: $0"
```

Variables -parámetros
$1-$9: Parametros posesiónales.
$*: Todos los parametros pasados.
$#: Numero total de parámetros pasados.
$0: Nombre del script.

Si se diera el caso que pasamos mas de 9 parámetros (no es usual XD), se debería de introducir la orden shift n. Donde "n" seria un numero, esto provocaría un efecto de salto. Con lo que el valor de $1 se perdería, el valor de $2 seria almacenado en $1, el de $3 en $2 y así sucesivamente. Lo mejor para ver esto es probándolo.

Ejemplo:

```
echo "Parametro 1: $1"
shift 1
echo "Parámetro 2: $2"
echo "Parámetro 3: $3"
echo "Todos los parámetros pasados son: $*"
echo "El numero de parámetros pasados es: $#"
echo "El nombre del script (parámetro 0) es: $0"
```

Variables

Definición de variable: Las variables son como una fuente de almacenamiento en la que depositamos un determinado dato. Además, sirven de enlace entre el usuario remoto y el programa, otra de sus utilidades es poder utilizarlas en distintas partes de nuestro código-script. La semejanza que se suele dar a una variable en informática es la de un contenedor en el que almacénanos datos.
Existen dos tipos de variables en linux. A nivel de creación. Por un lado se encuentran las variables de sistema (son las que crea el sistema), y por otro lado están las variables de usuario (son las que creamos nosotros mismos).
Creación de variables: Para crear una variable se usa la sintaxis: namevar=valordatos Donde namevar es el nombre (identificador) de la variable y valordatos (contenido) es el valor o los datos que queremos almacenar.
Así pues la variable namevar devolvería unos valores o datos.
Ejemplo:

```
webOrioN="www.ircorion.cjb.net"
```

La variable se llama webOrioN y en ella se alberga una string o cadena de caracteres (debido al entre comillado doble).
Viendo el contenido de las variables: Para ver lo que contiene una variable, se precede del nombre de la variable un signo "$". Además necesitamos un comando que nos muestre texto por pantalla.
Con lo que podemos usar el comando echo. Así pues con:

```
echo "Este es el contenido de la variable: $webOrioN"
```

Así veríamos el contenido.
Podemos también, hacer que el usuario sea quien almacene un dato en una variable.
Hasta ahora lo que vimos, fue como a nivel de código, le debamos un dato a una variable. Pero si lo que queremos es darle un dato que sea el que quiere el usuario, como lo haríamos?, en este caso no podemos predefinir la variable en el código a un dato concreto, puesto que puede ser esto o no el dato que quiera meter el usuario.
Para solucionar esto, esta la orden (read).
Ejemplo (código):

```
echo "Introduce un numero: "
read numero
echo "El numero introducido es: $numero"
```

En este codigo al usuario le saldría el mensaje (introduce un numero), automáticamente saltaría el interprete a la siguiente línea, y el read provocaría una espera continuada, hasta que el usuario introduzca un dato por teclado una vez pulsado el intro, el usuario vería el mensaje del ultimo echo con el numero que el introdujo, este se grabaría en la variable numero.

Algunas variables de entorno (creadas por el SO), son: $HOME devuelve la ruta del directorio raíz del usuario, $LOGNAME, etc...

Los nombres de las variables se denominan identificadores, puesto que identifican a un elemento en este caso variable. Como tales un identificador debe hacer referencia a un dato. Por ello es buena costumbre a la hora de programar usar nombres de variables acordes al valor o contenido que valla a devolver.

Si por ejemplo queremos crear una variable que almacene la fecha, seria mejor llamarla: verfecha que kk.
Como "programador" que soy son pequeños consejillos, que os doy ;)

Manejo de expresiones

Existen dos tipos de expresiones: expresiones aritméticas y expresiones condicionales.

Expresiones aritméticas: Son evaluadas y por el shell con el comando expr. A la hora de programar un programa siempre se precisan de comandos matemáticos que realicen una serie de operaciones, básicas o complejas. Este comando hace que el shell evalue la expresión matemática y devuelva un resulta (como una variable).
Cada termino especificado en su sintaxis lleva un espacio.
Este comando solo acepta operaciones básicas y con números enteros.
Operadores disponibles:

	Ejemplo
+: suma	expr 2 + 2
-: resta	expr 5 - 2
/: divide	expr 7 / 3
*: multiplica	expr 4 * 2

Metiendo en variable: result=`expr 2 + 1`

En el ultimo caso, el valor devuelto por la expresión es tomado como variable y metido dentro de la variable result.
En el caso del asterisco no se pone solo el *, porque vimos que este era un carácter interpretado por el shell, un comodín.

Expresiones condicionales: Son evaluadas por el shell. Pueden comparar: enteros, strings, así como también puede chequear la existencia de ficheros y directorios, permisos de acceso a ellos, etc. Este tipo de instrucciones de control tienen la finalidad de ramificar la ejecución de sentencias del script, en base a una variable u otro elemento en cualquier otro tipo de lenguaje.
Para evaluar una condición podremos usar:
a) test <expresión>
b) [expresión]
Las dos formas de uso son validas y funcionan efectivamente. No olvidemos situar un espacio entre los corchetes y los caracteres de la expresión en el caso de uso "b".
En caso de que la expresión sea cierta y se cumpla, esta devolverá un cero, en caso contrario devolverá cualquier otro valor distinto de cero. Esto se ha hecho internamente así por convenio.

Posibles expresiones para condicionales

Para ficheros:
-r <fichero> Es Verdadero si el fichero existe y se puede leer
-w <fichero> Es Verdadero si el fichero existe y se puede escribir en el
-x <fichero> Es Verdadero si el fichero existe y es ejecutable
-f <fichero> Es Verdadero si el fichero existe
-d <directorio> Es Verdadero si es un directorio
-s <fichero> Es Verdadero si el fichero existe y tiene un tamaño mayor que cero.
-b fichero - Verdadero si fichero existe y es un block especial.
-c fichero - Verdadero si fichero existe y es un character special.
-e fichero - Verdadero si fichero existe
-g fichero - Verdadero si fichero existe y es un set-group-id.
-k fichero - Verdadero si fichero tiene su ``sticky'' bit set.
-L fichero - Verdadero si fichero existe y es un symbolic link.
-p fichero - Verdadero si fichero existe y es un named pipe.
-S fichero - Verdadero si fichero existe y es un socket.
-t [fd] - Verdadero si fd está abierto en un terminal. Si fd es omitido, su defecto es 1 (standard output).
-u fichero - Verdadero si fichero existe y su set-user-id bit is set.
-O fichero - Verdadero si fichero existe y es un owned by the effective user id.
-G fichero - Verdadero si fichero existe y es un owned by the effective group id. fichero1 -nt fichero2 - Verdadero si fichero1 es mas nuevo (according to modification date) que fichero2.
fichero1 -ot fichero2 - Verdadero si fichero1 is mas viejo que fichero2. fichero1
-ef fichero2 - Verdadero si fichero1 y fichero2 tienen el mismo numero de device and inode.
-z string - Verdadero si la longitud de string es 0.
-n string - Verdadero si la longitud de string no es 0.
string1 = string2 - Verdadero si los strings son iguales
string1 != string2 - Verdadero si los strings no son iguales.
! expr - Verdadero si expr es falso.
expr1 -a expr2 - Verdadero si expr1 y expr2 son verdaderos.
expr1 -o expr2 - Verdadero si expr1 o expr2 es verdadero.
arg1 OP arg2 - OP es uno de -eq, -ne, -lt, -le, -gt, or -ge.

-l string, evalúa la longitud de string.

Para cadenas:
-z cadena Es Verdadero si la longitud de la cadena es cero.
-n cadena Es Verdadero si la longitud de la cadena es distinta de cero.
cadena1 = cadena2 Verdadero si las dos cadenas son iguales (atención a los espacios en blanco).
cadena1 != cadena2 Verdadero si las cadenas son distintas (atención a los espacios en blanco).
cadena Verdadero si la cadena no es nula
Para operaciones aritméticas
-eq Igual. Viene de "equal".
-ne Distinto. Viene de "not equal".
-gt Mayor que. Viene de "bigger than".
-lt Menor que. Viene de "less than".
-ge Mayor o igual. Viene de "bigger equal".
-le Menor o igual . Viene de "less equal".

Operadores lógicos
AND: && ó -a Verdadero si ambas expresiones son ciertas
OR: || ó -o Verdadero si una de las expresiones es cierta.
NOT: !expresión Verdadero si la expresión no es cierta
La orden true devuelve un 0, se emplea en estructuras repetitivas.
La orden false devuelve un valor distinto de 0, igual que true se emplea en estructuras repetitivas.

Sentencia condicional if-else: La sintaxis que siguen este tipo de estructuras de control es:

```
if <condicion>
        hacer
        instrucciones
else
        instrucciones
fi
```

En español ;)

```
Si CondicionEsVerdadera
        hacer
        instruccion1
        instruccion2
                ...
SiNO
        instruccion1
        instruccion2
...
fin
```

Si se cumple la condición de la primera línea. Ósea que sea cierta la condición, que se cumpla, entonces se ejecutarían las instrucciones que se dan a continuación del then. Luego saldría al final de la estructura. Si por el contrario no se cumple, se ejecutarían las instrucciones del bloque del else y saldría al final.

Ejemplos:
algoritmo1:

```
if test $num -eq 1
        then
                echo "La variable num contiene el valor 1"
else
        echo "La variable num no es uno"
fi
```

algoritmo2:

```
if  [ $num -eq 1 ]
        then
                echo "La variable num contiene el valor 1"
else
        echo "La variable num no es uno"
fi
```

Es el mismo caso que el anterior pero sin test, al poner los corchetes obtenemos el mismo resultado.

Sentencia CASE: Este tipo de sentencia, seria al equivalente al swicht en lenguaje C. Como otro tipo de estructura seria el equivalente a una serie de if anidados, con estos podría obtener el mismo resultado, la diferencia es que usando la sentencia case, el algoritmo es mas ordenado y completo.
Su funcionamiento es: Toma un valor (en este caso numérico) y lo alberga en la variable $num. Una vez que entre en el case, procesara: Si la variable es el caso 1 (osea si la variable "num" contiene el valor "1" <comparara la variable con todos los patrones>) realizara las instrucciones de esa linea. Y así sucesivame nte. Si se da el caso que no existe ningún caso que sea igual al valor de la variable, entonces se ejecutaría el case "*" que es por defecto (no ha introducido una opción valida).
Podemos especificar dos patrones o mas en una sola línea, con el signo "|" (alt gr + 1).

```
case $num in
        1) instruccion1;insruccion2; ...;;
        2) instruccion1; ...;;
        3) instruccion1; ...;;
        *) echo "Opcion no valida";instruccion23;....;;
        5)echo "Saliendo del CASE";exit;;
esac
```

Instrucciones iterativas (looping)

Bucles: Lleguemos ante los denominados bucles, (procesos de repetición iterativa).
Los bucles son un tipo de estructuras de programación (algoritmos) tremendamente útiles, que realizan una serie de comportamientos repetitivos (iterativos) mientras o hasta que se cumpla la condición. Esto hace que ahorremos muchas líneas de código, entre otras cosas: orden, estructuración, etc.
Esta técnica también es denominada rizo o looping. Su homologo en MSDOS seria el GOTO y sus etiquetas, que cumpliría la misma función, con la única diferencia es que el GOTO por lo normal esta mal visto en la programación estructurada, debido a que se "pierde" la estructuración secuencial del algoritmo del programa. En Linux contamos con verdaderas sentencias de control para el uso de bucles , son las siguientes:

Sentencias while y until: En el caso del while. Las instrucciones se realizaran mientras (la condición se cumpla). Cuando se deje de cumplir, el bucle se saldrá y finalizaría.
Por contrario en el caso del until, las instrucciones s e ejecutan hasta (until) que la conducción se cumpla. Seria el bucle inverso al while.

Sintaxis:

```
while <condición>
 do
    instrucción1
    instrucción2
    .....
done

until <condición>
 do
     instrucción1
     instrucción2
     ....
done
```

Ejemplo:

```
opcion=0
while [ $opcion -ne 4 ] || [ $opcion -ne 0 ]
        do
                clear
                echo "Menu (selecciona un editor)"
                echo "1-vi"
                echo "2-emacs"
                echo "3-joe"
                echo "4-Salir"
                echo "Introduzca una opcion: "
                case opcion in
                        1)echo "Ejecutando editor VI";Vi;;
                        2)echo "Ejecutando editor emacs";emacs;;
                        3)echo "Ejecutando editor joe";joe;;
                        4|0)echo "Fin de programa";exit;;
                        *)echo "Opcion no valida";read;;
                esac
        done
```

Aquí vemos de forma clara el funcionamiento de la sentencia while (bucle) y el case (ifs anidados).
Si no se comprende el algoritmo en su primer vistazo, seria bueno que lo probase en su ordenador y comprobar su funcionamiento.
Lo que hace este programilla, es preguntar por el editor que se desea usar en LINUX. Usted introduce un numero y este es almacenado en la variable "opción". Si introduce un 1, ejecuta el VI, si es un 2 será el emacs, y si es un 3 se tratara del joe. Si el usuario pulsa el 4 o el numero 0, automáticamente sale del case y llegara al done puesto que la condición no se cumplirá.

Sentencia for: Este es el ultimo tipo de bucle a ver. Este bucle se repite tantos elementos hay en la cadena de elementos
Sintaxis:

```
for variables in var1 var2 var3 var4 ....
        do
                instrucción1
                instrucción2
                ......
        done
```

Su traduccion seria:

```
<para> <elemento> <en> <cadenaElementos>
        hacer
                instrucciones
final
```

Ejemplos:

```
for num in `seq 1 10`
        do
                echo "Contador: $num"
        done
```

En este ejemplo la variable num toma los valores del 1 al 10 cuando llega a este sale del bucle for. Hasta entonces se ejecuta el comando echo con el que visualizamos el valor de la variable "num".

Funciones

Las funciones son pequeños algoritmos o trozos de código que cumplen una determinada función. Son creados por nosotros mismos para su posterior utilización en distintas partes de nuestro script, mediante llamadas. Y se sitúan en el propio script.
Las funciones son independientes, solo se ejecutaran si en el código principal se hace una llamada a ellas. Un buen programador debe cuidar que sus funciones realicen tareas bien determinadas, y que podamos usarlas con un margen de actuación elevado, ósea hacer funciones lo mas genéricas posibles. También es recomendable darles un nombre que identifique su función o lo que realizan.
A menudo los nuevos programadores suelen cometer el error (para mi), de no dar la suficiente importancia a las funciones y no realizarlas e incluirlas dentro del código del script.
Sintaxis:

```
function NombreFuncion (){
instrucciones
}
```

A las funciones se les pueden pasar parámetros de la misma forma que al programa principal, luego podríamos operar con ellos con las variables $1, $2, $3, etc.
Si deseamos forzar la salida en la función, nos bastara con poner la palabra "return". En las funciones se usa return en vez de exit tal y como se hace en el script principal.

Ejemplo:

```
#inicio del programa
function saludo () {
echo "HOLA MUNDO (la variable conter vale: $1)"
}

conter=1
while test $conter -le 10
        do
                saludo $conter
                conter=`expr $conter + 1`
        done
#fin del programa
```

El programa se inicia dando el valor 1 a la variable conter. Luego pasa a evaluar el while y mientras la variable conter sea menor o igual a 10, hacer; una llamada a la funcion saludo, esta a su vez muestra por pantalla el texto HOLA MUNDO y el valor de $conter que es contenido en $1 que seria el parametro posicional 1 pasado a la hora de llamar a la funcion (esto se ha visto al principio del tutorial). A continuación sale de la función y sigue en el while y así sucesivamente, cuando conter ia valga 11 saldra del bucle y finalizara el programa.

Recursividad: Una buena técnica de programación es la llamada recursividad. A menudo se emplean funciones recursivas. Estas se denominan así, porque son llamadas así mismas.
Bien es cierto, que todo aquello que podemos realizar con una función recursiva podemos hacerlo con un bucle (iteratividad). Aunque en el caso de la recursividad la optimización es mayor y el tamaño del algoritmo decrece, es mucho mas pequeño. Lo cual es mejor siempre que su función habilidad no se vea dañada.
A continuación expongo el típico ejemplo de recursividad.

```
El factorial
function factorial () {
if  [ $1 -eq 1 ]
        then
        echo 1
else
        echo `expr $(factorial $(expr $1 - 1)) \* $1`
fi
}
echo "El factorial de $1 es `factorial $1`
```

Instrucciones varias

El comando select

Las instrucciones break y continue: La instrucción break, finaliza la iteratividad o repetición del bucle mas interior saliendo por su done, aunque la condición de este bucle se cumpla, este se romperá y saldrá por ser forzado por el comando break. El comando continue causa la ejecución (iteratividad) del bucle. Realiza lo contrario del break.

Forzar la evaluación de comandos
Comando: eval
Etimologia: Evaluar
Sintaxis: eval <comando>
Donde comando es la variable o comando que debe ser evaluado por el comando eval. Una vez evaluado el comando es ejecutado como tal.

Ejecutar un comando sin crear un nuevo proceso: Se realiza con el comando exec. Si introducimos: exec who
Se ejecutara el comando who sin crear un nuevo proceso, cuando el proceso del comando who se termine, dará paso al shell.

Finalizar un script
Comando: exit
Etimología: salida
Sintaxis: exit
Para finalizar un script podemos usar el comando exit.

Manejo de interrupciones: Cuando un script se está ejecutando este puede ser interrumpido si llega una señal del SO o del usuario por ej CTRL.+C. Esto puede producir efectos indeseados. El comando trap captura las señales permitiendo que se realicen los comandos especificados en una lista. Existen varias señales de interrupción las cuales se pueden visualizar a trabes del comando trap -l. Las más frecuentes son:
0->Salida del shell
2->Interrupción ctrl + C
9->Muerte de un proceso
15->Terminación generada por kill
Con el comando trap sin opciones se puede averiguar que señal se está produciendo en el sistema en caso de que la haya.

Ej. Pedir por teclado que se introduzca un comando. Permite que se ejecute un comando sin ser interrumpido al pulsar ctrl. +C

```
while true
        do
                trap exit 2
                echo Introduzca un comando
                read comando
                trap “2
                $comando
        done
```

Ejercicio: Busca ficheros según una mascara y desvia la lista a ficheros temporales de manera que si pulso Ctrl. + C limpia del subdirectorio esos ficheros temporales.

```
Tempo=/tmp/Temp.?
Trap ´rm $temps;exit ‘2
Echo buscando *ss* .....
Find /* -name * ss * > /Temp. /temp1 2> /tmp/tempe
Echo buscando * jj *.....
Find /* -name * jj *> /tmp/temp2 2>/tmp/tempe
```

Depuración de los scripts: Como los buenos compiladores, en LINUX podemos depurar nuestros códigos y obtener los errores que se producen en el proceso de ejecución de un script.
Sintaxis: sh [opciones] <nameScript>
-x Para saber donde se produce un error en nuestro script.
-n Chequea la sintaxis de nuestro script, (no lo ejecuta).
-e Obliga a finalizar el script en caso de que algun comando falle.
-v Imprime las líneas de entrada según se leen.

Ejemplos variados de scripts

```
#script que muestra la fecha, la ruta del home del usuario, y hace un listado de los ficheros y directorios del directorio actual.
date
echo $HOME
ls

#script que muestra el uso de parámetros, entrecomillados y variables.
nombre=Pedro
hola1='Hola $nombre'
hola2="Hola $nombre"
hola3=`Hola $nombre`
comando=`ls`
echo $nombre
echo $hola1
echo $hola2
echo $hola3
echo $comando
echo "Guion: $0"
echo "Parametro1: $1"
echo "Todos los parametros: $*"
echo "Numero de parametros: $#"
echo $*
shift 1
echo $1 $2 $3
shift 2
echo $1 $2

#script que muestra un ejemplo sobre el uso de sentencias condicionales
i=20
j=40
if test $i -le $j
then
echo VERDADERO
else
echo FALSO
fi

#script que muestra el uso de parametros pasados al script (a la hora de ejecutarlo)
echo $0
echo $*
echo Parametro 1: $1
echo Parametro 2: $2
echo Parametro 3: $3
echo Parametro 4: $4
echo Parametro 5: $5
```

```
echo Parametro 6: $6
echo Parametro 7: $7
echo Parametro 7: $7
echo Parametro 8: $8
echo Todos los parametros pasados son: $*
shift 3
echo Tras eliminar los 3 parametros quedan: $*

#script que muestra un ejemplo sobre la concatenacion (union) de variables.
nombre="Javi"
apellidos="Fernandez Rivera"
nameyape="$nombre $apellidos"
echo $nameyape

#script que muestra el uso del comando expr.
v1=`expr 4 + 4`
v2=`expr 4 - 4`
v3=`expr 4 \* 4`
v4=`expr 4 / 4`
v5="resultado es: "
v5="resultado es: "
echo "Valor de v1 --> $v5 $v1"
echo "Valor de v2 --> $v5 $v2"
echo "Valor de v3 --> $v5 $v3"
echo "Valor de v4 --> $v5 $v4"

#script que le pasas 3 parámetros (directorios) y testea si son 3 en caso de serlo, mira lo que hay dentro de ellos con el comando ls
clear
if test $# -eq 3
then
echo "Ha pasado 3 parametros"
echo "======================"
echo " "
echo " "
echo "Visualizando contenido de $1"
echo " "
ls $1
echo " "
echo "Visualizando contenido de $2"
echo " "
ls $2
echo " "
echo "Visualizando contenido de $3"
echo " "
ls $3
echo " "
echo " "
echo "Fin del proceso. ;)"
else
echo "No ha pasado los 3 parametros"
fi



#Ejercicio que muetsra la tabla de multiplicar del parametro pasado
        n=0
        while test $n -le 10
         do
         result=`expr $n \* $1`
         echo $1*$n = [$result]
         n=`expr $n + 1`
        done

#Ejercicio que muestra los numeros del 1 al 10 con un until
        x=0
          until test $x -eq 10
           do
            x=`expr $x + 1`
            echo $x
```

```
        done

#Ejercicio que muestra los numros del 1 al 10.
       for x in `seq 1 10`
        do
         echo Numero: [$x]
       done

#Suma todos los ficheros de un directorio
       result=0
       total=0
        for result in $(ls -l|grep ^-|cut -c 39-42)
         do
          total=`expr $total + $result`
          echo $result $total
        done
       echo total = $total

#Script que pide un numero como argumento y devuelve el factorial
       x=$1
       cont=`expr $x - 1`
        while test $cont -ge 1
         do
           x=`expr $x \* $cont`
           cont=`expr $cont - 1`
         done
       echo El numero factorial es: [$x]

#Script q traslade todos los ficheros q empiezan por letra minuscula al subdirectorio minusculas
#y los que empiezan por mayusculas al subdirectorio mayusculas
       if test ! -d minusculas
        then mkdir minusculas
             cp `ls -l [a-z]* | grep ^- |cut -c 56-90` minusculas
       fi

       if test ! -d mayusculas
        then mkdir mayusculas
             cp `ls -l [A-Z]* | grep ^- |cut -c 56-90` mayusculas
       fi
       fi

#Script que lea texto (lo coje) y lo pase al file  temp hasta que se introduzca un punto o el enter.
clear
txt="x"
echo "Gabando desde (stdin) al fichero log: temp"
echo "Para finalizar teclea un punto"
echo "==========================" >> temp
echo "Comienzo del fichero [log]." > temp
echo "==========================" >> temp
       while test -n "$txt" -a "$txt" != "."
        do
         read txt
          if test $txt != "."
           then
           echo $txt >> temp
          fi
       done
echo "==========================" >> temp
echo "Fin del fichero [log]." >> temp
echo "==========================" >> temp
#script q pase a mayusculas el contenido de todos los ficheros pasados como para
metros
metros
x=0
for x in $*
do
cat $x | tr "[a-z]" "[A-Z]" > $x
done

#script q coja como parametros 3 cosas lo primero una p o una s en base a esto pondra un sufijo o un prefijo
```

```
#luego poner el sufijo o el prefijo a añadir y al final pasar como parametro la mascara de ficheros que queremos q afecte.
clear
        if test $# -eq 3
                then
                        echo "Ha pasado 3 parametros"
                        echo "====================="
                        echo " "

                if test -f $1
                        then cat $1;echo Fin del fichero [$1].
                else echo ERROR [$1] no es un fichero.
                fi

                if test -f $2
                        then cat $2;echo Fin del fichero [$2].
                 then cat $2;echo Fin del fichero [$2].
                else echo ERROR [$2] no es un fichero.
                fi

                if test -f $3;echo Fin del fichero [$3].
                        then cat $3
                else echo ERROR [$3] no es un fichero.
                fi
         else
                echo "No ha pasado los 3 parametros"
        fi
```