

Programación en shell: Administración de linux

Una guía básica

Pedro Pablo Fábrega Martínez

http://dns.bdat.net/documentos/programacion_shell/

[Aviso Legal](#)

Distribución bajo licencia [Creative Commons](#)

En resumen, se permite la reproducción parcial o total de estos textos en cualquier medio , impreso o electrónico, siempre que no se impongan condiciones adicionales a la reproducción y distribución de las copias o de los trabajos derivados que incorporen este documento. En resumen, cualquier trabajo derivado de ese texto debe mantener esta nota de copyright.

Tabla de contenidos

1. Ejecución y agrupación de órdenes
 - 1.1 Código de terminación de una orden
 - 1.2 Ejecución consecutiva
 - 1.3 Ejecución condicional
 - 1.3.1 Operador &&
 - 1.3.2 Operador ||
 - 1.3.3 Ejecución simultánea
 - 1.3.4 Agrupando con paréntesis
 - 1.3.5 Comillas invertidas `
 - 1.3.6 El operador \$()
2. Programas de shell
 - 2.1 subshell
 - 2.2 Comentarios y continuaciones de línea
 - 2.3 Parámetros posicionales
 - 2.3.1 Modificación de los parámetros posicionales
 - 2.3.2 La sentencia shift
 - 2.3.3 Operador {}
 - 2.4 Variables predefinidas
 - 2.4.1 Variable \$*
 - 2.4.2 Variable \$@
 - 2.4.3 Variable \$#
 - 2.4.4 Variable \$?
 - 2.4.5 Variable \$\$
 - 2.4.6 Variable \$!
 - 2.5 Uso de valores predeterminados de variables
 - 2.5.1 Uso de variable no definida o con valor nulo
 - 2.5.2 Uso de variable no definida
 - 2.5.3 Uso de variable definida o con valor nulo
 - 2.5.4 Uso de variable no definida
 - 2.6 Asignación de valores predeterminados de variables

- 2.6.1 Asignación a variable o definida o con valor nulo
- 2.6.2 Asignación a variable no definida
- 2.6.3 Mostrar un mensaje de error asociado a una variable
- 2.6.4 Variable no definida o con valor nulo
- 2.6.5 Variable no definida
- 2.7 Otras operaciones con variables
 - 2.7.1 Subcadenas de una variable
 - 2.7.2 Cortar texto al principio de una variable
 - 2.7.3 Cortar texto al final de una variable
 - 2.7.4 Reemplazar texto en una variable
- 2.8 Evaluación aritmética
- 2.9 Selección de la shell de ejecución
- 2.10 Lectura desde la entrada estándar: read
- 2.11 Evaluación de expresiones: test
- 2.12 Estructura de control
 - 2.12.1 Sentencia if
 - 2.12.2 Sentencia while
 - 2.12.3 Sentencia until
 - 2.12.4 Sentencia for
 - 2.12.5 Sentencias break y continue
 - 2.12.6 Sentencia case
- 2.13 Terminar un programa de shell (exit)
- 2.14 Opciones en un programa de shell: getopt
- 2.15 Evaluación de variables: eval
- 2.16 Funciones
- 2.17 Trucos de programación en shell
 - 2.17.1 Script con número variable de argumentos:
 - 2.17.2 Script para aplicar una serie de órdenes a cada fichero de un directorio
 - 2.17.3 Leer un fichero de texto línea a línea
 - 2.17.4 Cambiar una secuencia de espacios por un separador de campos
- 2.18 Prácticas Ejercicios propuestos

1.- Ejecución y agrupación de órdenes

Una vez vistas gran parte de las órdenes de usuario, pasamos a ver una de las principales características de un sistema Unix que es la facilidad para agrupar órdenes de distintas formas para realizar tareas complejas. Por un lado en la propia línea de órdenes se pueden especificar órdenes y unir su ejecución de diversas formas.

Hasta ahora hemos visto como combinar órdenes con tubería, como redirigir las salida y como gestionar procesos en primer y segundo planos. Ahora vamos a ver otra serie de mecanismos para ejecutar órdenes y programas y verificar el estado de conclusión de la orden.

1.1 Código de terminación de una orden

Cuando una orden termina le devuelve al sistema un código de finalización, un valor cero en caso de terminar correctamente o un valor uno si se ha producido un error. Este valor se almacena en la variable \$?.

Por ejemplo:

```
[pfabrega@port /home]$ ls -la /home
total 32
drwxr-xr-x  5 root  root   4096 sep 20 14:16 .
drwxr-xr-x 18 root  root   4096 sep 20 03:23 ..
drwxr-xr-x  4 root  root   4096 sep 28 21:16 httpd
drwxr-xr-x  2 root  root  16384 sep 19 18:25 lost+found
drwx--x--x 20 pfabrega pfabrega 4096 nov 24 19:07 pfabrega
[pfabrega@port /home]$ echo $?
0
[pfabrega@port /home]$ ls -la asdfg
ls: asdfg: No existe el fichero o el directorio
[pfabrega@port /home]$ echo $?
1
[pfabrega@port /home]$
```

Podemos observar como en la primera ejecución `ls -la` devuelve un valor 0 de terminación correcta y en la segunda devuelve un error y n código 1.

1.2 Ejecución consecutiva

Podemos agrupar varias órdenes en una misma línea de ordenes separándolas por ";"

La agrupación de órdenes separadas por ";" es útil cuando tenemos que repetir una misma secuencia de órdenes varias veces.

La ejecución de cada una de las órdenes se realiza cuando ha concluido la anterior, e independiente de que el resultado haya sido correcto o no.

Esto nos puede resultar útil para demorar la ejecución de una o varias órdenes un determinado tiempo, por ejemplo

```
$ sleep 300 ; ps axu
```

y la orden `ps axu` se ejecutaría a los 300 segundos.

1.3 Ejecución condicional

Otra situación algo más elaborada que la anterior es ejecutar una orden condicionada a la terminación correcta o no de una orden previa.

Esta funcionalidad nos la proporcionan los operadores "&&" y "||".

1.3.1 Operador &&

El primer operador, "&&" separa dos órdenes de forma que la que tiene a la derecha sólo se ejecuta cuando la de la izquierda termina correctamente, es decir

```
orden1 && orden2
```

orden2 sólo se ejecutará si orden1 terminó sin ningún error.

Por ejemplo, queremos ejecutar la orden `cat fichero` sólo si existe fichero; entonces tendremos que buscar una orden que termine con un error si no existe fichero, por ejemplo `ls fichero` y condicionar la ejecución de `cat fichero` a esta:

```
$ ls fichero && cat fichero
```

Otro ejemplo, para compilar los controladores de dispositivos de linux e instalarlos, lo podemos hacer como:

```
make module && make modules_install
```

es decir instalará los controladores sólo si ha conseguido compilarlos correctamente.

1.3.2 Operador ||

El segundo operador, "||" tiene un comportamiento similar al anterior, separa dos órdenes de forma que la que tiene a la derecha sólo se ejecuta cuando la de la izquierda termina incorrectamente, es decir

```
orden1 || orden2
```

orden2 sólo se ejecutará si orden1 terminó con algún error.

Por ejemplo si no existe fichero queremos crearlo vacía y si existe no hacemos nada. Igual que en el ejemplo anterior, buscamos una orden que termine con un error si no existe fichero y condicionamos la ejecución de la orden `touch fichero` al error de la orden previa:

```
$ ls fichero || touch fichero
```

También, al igual que en el ejemplo anterior podríamos hacer que si el proceso `make modules` falla, se borran todos los ficheros temporales que se crean:

```
make modules || rm -r *.o
```

1.3.3 Ejecución simultánea

Otra posibilidad de ejecución también posible es lanzar varios procesos simultáneamente en segundo plano; basta escribir uno a continuación de otro en la línea de órdenes separados por "&". Este es el símbolo que se utiliza para indicar que el proceso se tiene que ejecutar en segundo plano, pero también actúa como separador para la ejecución de distintas órdenes.

por ejemplo :

```
[pfabrega@port pfabrega]$ sleep 10 & sleep 20 & sleep 15 &
[pfabrega@port pfabrega]$ ps axu
USER      PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  0.1  0.2  1408  540 ?        S    22:19   0:04 init [3]
...
pfabrega  1263  0.3  0.2  1624  516 pts/4    S    23:24   0:00 sleep 10
pfabrega  1264  0.0  0.2  1624  516 pts/4    S    23:24   0:00 sleep 20
pfabrega  1265  0.0  0.2  1624  516 pts/4    S    23:24   0:00 sleep 15
pfabrega  1266  0.0  0.3  2732  848 pts/4    R    23:24   0:00 ps axu
```

1.3.4 Agrupando con paréntesis

Podemos organizar la ejecución de varias órdenes agrupándolas convenientemente mediante paréntesis para modificar el orden predeterminado de ejecución. En primer lugar actúan los ; después los & y la ejecución es de izquierda a derecha. Para las ejecuciones condicionales se tiene en cuenta el valor de la variable \$? que se fija por la última orden que se ejecuta.

Para alterar esta forma de ejecución podemos utilizar los paréntesis. En realidad los paréntesis fuerzan una nueva subshell para ejecutar las órdenes correspondientes.

Esta característica puede ser interesante en diferentes situaciones:

Quemos enviar una secuencia de órdenes a segundo plano

```
(mkdir copiaseg; cp -r ./original/* ./copiaseg; rm -r ./original~; rm -r ./original.tmp) &
```

Resultado de la ejecución de una orden

Es habitual necesitar almacenar el resultado de la ejecución de una orden en una variable en lugar de que se dirija a la salida estándar o simplemente ejecutar como orden el resultado de otra orden.

1.3.5 Comillas invertidas `

Las comillas invertidas consideran una orden lo que tengan dentro y lo ejecutan devolviendo el resultado como líneas de texto a la shell. Por ejemplo:

```

$ A="ls /bin"
$ `echo $A`
arch  consolechars ed      igawk  mount  rpm    tar
ash   cp      egrep  ipcalc mt      rvi    tcsh
ash.static cpio    ex     kill  mv      rview  touch
awk   csh     false  ln     netstat sed    true
basename date    fgrep  loadkeys nice    setserial umount
bash  dd      gawk   login  nisdomainname sfxload uname
bash2 df      gawk-3.0.6 ls    ping   sh      usleep
bsh   dmesg   grep   mail   ps      sleep  vi
cat   dnsdomainname gtar   mkdir  pwd     sort   view
chgrp doexec  gunzip mknod  red     stty   ypdomainname
chmod domainname gzip   mktemp rm      su     zcat
chown echo    hostname more   rmdir  sync

También podríamos haber puesto:
$ A="ls /bin"
$ B=`$A`
$ echo $B
arch ash ash.static awk basename bash bash2 bsh cat chgrp chmod chown consolechars cp cpio csh date dd df dmesg dnsdomainname
doexec domainname echo ed egrep ex false fgrep gawk gawk-3.0.6 grep gtar gunzip gzip hostname igawk ipcalc kill ln loadkeys login ls
mail mkdir mknod mktemp more mount mt mv netstat nice nisdomainname ping ps pwd red rm rmdir rpm rvi rview sed setserial sfxload sh
sleep sort stty su sync tar tcsh touch true umount uname usleep vi view ypdomainname zcat

```

1.3.6 El operador \$()

La shell bash proporciona el operador \$() similar a las comillas invertidas. Ejecuta como orden los que haya entre paréntesis y devuelve su resultado. El mecanismo de funcionamiento es idéntico, con la ventaja de poder anidar operadores.

2. Programas de shell

Además de las anteriores posibilidades también se pueden agrupar una serie de órdenes en un fichero de texto que se ejecutarán consecutivamente siguiendo el flujo determinado por órdenes de control similares a cualquier lenguaje de programación. Estos ficheros se conocen como scripts, guiones o simplemente programas de shell. A las órdenes agrupadas en ficheros también se le aplican todas las características descritas anteriormente. No olvidemos que para un sistema unix, una línea leída de un fichero es idéntica a una línea leída desde el teclado, una serie de caracteres terminado por un carácter de retorno de carro.

Cualquier forma de ejecución que se pueda dar en la línea de órdenes también se puede incluir en un fichero de texto, con lo que facilitamos su repetición. Y si por último añadimos las estructuras que controlan el flujo de ejecución y ciertas condiciones lógicas, tenemos un perfecto lenguaje de programación para administrar el sistema con toda facilidad. Un administrador que sabe cual es su trabajo cotidiano, realizar copias de seguridad, dar de alta o baja usuarios, comprobar que los servicios están activos, analizar log de incidencias, configurar cortafuegos, lanzar o parar servicios, modificar configuraciones, etc., normalmente se creará sus script personalizados. Algunos los utilizará cuando sea necesario y para otros programará el sistema para que se ejecuten periódicamente.

La programación en shell es imprescindible para poder administrar un sistema Unix de forma cómoda y eficiente.

Vemos un primer ejemplo:

```

#!/bin/bash

echo Hola Mundo

```

Puestas estas dos líneas en un fichero de texto con permiso de ejecución, al ejecutarlo escribiría en pantalla "Hola Mundo". La primera línea, como veremos con posterioridad, indica qué shell es la que interpreta el programa de shell.

2.1 subshell

Para la ejecución de programas de shell, la shell se encarga de interpretar unas órdenes en unos casos o de lanzar el programa adecuado en otros casos. En general, cuando lanzamos la ejecución de un conjunto de órdenes agrupadas en un programa de shell, se abre una nueva shell (subshell hija de la anterior) que es la encargada de interpretar las órdenes del fichero. Una vez concluida la ejecución esta subshell muere y volvemos a la shell inicial. Esto es importante tenerlo presente para saber el comportamiento de los programas. Por ejemplo, los cambios hechos en las variables de shell dentro de un programa no se conservan una vez concluida la ejecución.

Vamos a ilustrar este comportamiento con nuestro primer ejemplo de programa de shell:

Creamos un programa en un fichero de texto, lo ejecutamos, comprobamos que se crea una nueva shell y que los cambios en las variables hechos dentro del programa no se mantienen una vez concluido. Editamos un fichero llamado "pruebasshell" con el siguiente contenido:

```
echo "***** GUION *****"
echo "el valor previo de VAR es ** $VAR **"
VAR="valor asignado dentro del guion"
echo "Ahora VAR vale ** $VAR **"
ps
echo "***** FIN DEL GUION *****"
```

Con este guion mostramos el valor previo de una variable llamada VAR, le asignamos un valor nuevo y también los mostramos. Para verificar que se lanza una nueva shell mostramos la lista de procesos con la orden ps.

Una vez editado el fichero tendremos que asignarle el permiso de ejecución

```
$ chmod u+x pruebasshell
```

después asignamos un una valor a la variable VAR para comprobar como cambia. Además tendremos que exportarla par que la shell hija pueda heredarla:

```
$ export VAR="valor previo"
```

Ahora mostramos la lista de procesos para ver cuantas shell tenemos abiertas:

```
$ ps
```

o bien

```
$ ps |wc -l
```

y a continuación ejecutamos el guion "pruebasshell":

```
$ ./pruebasshell
```

y volvemos a mostrar el contenido de la variable:

```
$ echo $VAR
```

Podremos observar como aparece una shell más. Si la variable VAR está exportada veremos como muestra el valor que asignamos antes de ejecutar el guion y como muestra el que le asignamos dentro del guion. Y al final, al mostrar la variable VAR, observamos como nos

muestra el valor que tenía antes de ejecutar el guion; el guion no ha modificado la variable.

Este mismo comportamiento se puede aplicar a la orden `cd`. Veamos el siguiente ejemplo, un simple script que cambia de directorio.

Editamos el fichero llamado "cambia" con el siguiente contenido:

```
echo "cambiando de directorio"  
cd /tmp  
echo "estamos en:"  
pwd
```

Es decir, el script simplemente cambia al directorio `/tmp`.

Una vez editado le asignamos el permiso de ejecución

```
$ chmod u+x cambia
```

Ahora mostramos nuestro directorio activo

```
$ pwd
```

ejecutamos el script

```
$ ./cambia
```

y volvemos a comprobar nuestro directorio activo

```
$ pwd
```

y observamos como estamos situados en el mismo directorio que antes de la ejecución del guion.

¿Por qué ocurre todo esto?, Porque todos los cambios se realizan en la subshell que ha interpretado el guion.

2.2 Comentarios y continuaciones de línea

Para añadir un comentario a un programa de shell se utiliza el carácter `#`. Cuando la shell interprete el programa ignorará todo lo que haya desde este carácter hasta el final de la línea.

Por otro lado si nos vemos obligados a partir un línea en dos o más, tendremos que finalizar cada línea de texto inconclusa con el carácter `\`. De esta forma la shell las verá todas ellas como si se tratara de una única línea.

2.3 Parámetros posicionales

En un programa de shell definen unas variables especiales, identificadas por números, que toman los valores de los argumentos que se indican en la línea de órdenes al ejecutarlo. Tras el nombre de un script se pueden añadir valores, cadenas de texto o números separados por espacios, es decir, parámetros posicionales del programa de shell, a los que se puede acceder utilizando estas variables.

La variable `$0` contiene el parámetro 0 que es el nombre del programa de shell.

Las variables `$1`, `$2`, `$3`, `$4`, ... hacen referencia a los argumentos primero, segundo, tercero, cuarto, ... que se le hayan pasado al programa en el momento de la llamada de ejecución.

Por ejemplo, si tenemos un programa de shell llamado `parametros` con el siguiente contenido:

```
echo $0
echo $1
echo $2
echo $3
al ejecutarlo
$ parametros primero segundo tercero
parametros
primero
segundo
tercero
```

2.3.1 Modificación de los parámetros posicionales

Durante la ejecución de un programa de shell podría interesarnos modificar el valor de los parámetros posicionales. Esto no lo podemos hacer directamente, las variables 1, 2, ... no están definidas como tales. Para realizar estos cambios tenemos que utilizar la orden set. Esta orden asigna los valores de los parámetros posicionales a la shell activa de la misma forma que se hace en la línea de órdenes al ejecutara un programa; hay que tener en cuenta que no los asigna individualmente, sino en conjunto.

Por ejemplo

```
$ set primero segundo tercero
$ echo $1
primero
$ echo $2
segundo
$ echo $3
tercero
```

2.3.2 La sentencia shift

La sentencia shift efectúa un desplazamiento de los parámetros posicionales hacia la izquierda un número especificado de posiciones. La sintaxis para la sentencia shift es:

```
$ shift n
```

donde n es el número de posiciones a desplazar. El valor predeterminado para n es 1. Hay que observar que al desplazar los parámetros hacia la izquierda se pierden los primeros valores, tantos como hayamos desplazado, al superponerse los que tiene a la derecha.

Por ejemplo:

```
$ set uno dos tres cuatro
$ echo $1
uno
$ shift
$ echo $1
dos
$ shift
$ echo $1
tres
$ shift
$ echo $1
cuatro
```

2.3.3 Operador {}

Hemos visto la forma de acceder a los diez primeros parámetros posicionales, pero para acceder a parámetros de más de dos dígitos tendremos que usar una pareja {} para englobar el número.

```
$ echo ${10}
$echo ${12}
```

El operador {} también se usa para delimitar el nombre de las variables si se quiere utilizarla incluida dentro de un texto sin separaciones:

```
$DIR=principal
$ DIRUNO=directorio
$ UNO=subdirectorio
$ echo $DIRUNO
directorio
$ echo ${DIR}UNO
principalUNO
```

2.4 Variables predefinidas

Además de las variables de shell propias del entorno, las definidas por el usuario y los parámetros posicionales en un shell existen otra serie de variables cuyo nombre está formado por un carácter especial, precedido por el habitual símbolo \$.

2.4.1 Variable \$*

* La variable \$* contiene una cadena de caracteres con todos los parámetros posicionales de la shell activa excepto el nombre del programa de shell. Cuando se utiliza entre comillas dobles se expande a una sola cadena y cada uno de los componentes está separado de los otros por el valor del carácter separador del sistema indicado en la variable IFS. Es decir si IFS tiene un valor "s" entonces "\$*" es equivalente a "\$1s\$2s...". Si IFS no está definida, los parámetros se separan por espacios en blanco. Si IFS está definida pero tiene un contenido nulo los parámetros se unen sin separación.

2.4.2 Variable \$@

@ La variable \$@ contiene una cadena de caracteres con todos los parámetros posicionales de la shell activa excepto el nombre del programa de shell. La diferencia con \$* se produce cuando se expande entre comillas dobles; \$@ entre comillas dobles se expande en tantas cadenas de caracteres como parámetros posicionales haya. Es decir "\$@" equivale a "\$1" "\$2" ...

2.4.3 Variable \$#

Contiene el número de parámetros posicionales excluido el nombre del programa de shell. Se suele utilizar en un guion de shell para verificar que el número de argumentos es el correcto.

2.4.4 Variable \$?

? Contiene el estado de ejecución de la última orden, 1 para una terminación con error o 0 para una terminación correcta. Se utiliza de forma interna por los operadores || y && que vimos con anterioridad, se utilizar por la orden test que veremos más adelante y también la podremos usar explícitamente.

2.4.5 Variable \$\$

\$ contiene el PID de la shell. En un subshell obtenida por una ejecución con (), se expande al PID de la shell actual, no al de la subshell. Se puede utilizar para crear ficheros con nombre único, por ejemplo \$\$tmp, para datos temporales.

2.4.6 Variable \$!

! Contiene el PID de la orden más recientemente ejecutada en segundo plano. Esta variable no puede ayudar a controlar desde un guion de shell los diferentes procesos que hayamos lanzado en segundo plano.

Ejemplos

Vemos algunos ejemplos a continuación:

```
$ set a b c d e
$ echo $#
5
$ echo $*
a b c d e
$ set "a b" c d
$ echo $#
3
$ echo $*
a b c d
```

Otro ejemplo, si tenemos el script llamado ejvar1 con el siguiente contenido

```
ps
echo " el PID es $$"
```

al ejecutarlo

```
$ ./ejvar1
PID TTY      TIME CMD
 930 pts/3    00:00:00 bash
1011 pts/3    00:00:00 bash
1012 pts/3    00:00:00 ps
el PID es 1011
```

y vemos como muestra el PID de la shell que ejecuta el script.

Como el PID del proceso es único en el sistema, este valor puede utilizarse para construir nombres de ficheros temporales únicos para un proceso. Estos ficheros normalmente se suelen situar en el directorio temporal /tmp.

Por ejemplo:

```
miproctemp=/tmp/miproc.$$
...
rm -f $miproctemp
```

2.5 Uso de valores predeterminados de variables

Además de las asignaciones de valores a variables vista con anterioridad, que consistía en utilizar el operador de asignación (=), podemos asignarle valores dependiendo del estado de la variable.

2.5.1 Uso de variable no definida o con valor nulo

Cuando una variable no está definida, o lo está pero contiene un valor nulo, se puede hacer que se use un valor predeterminado mediante la siguiente la expresión:

```
$ {variable:-valorpredeterminado}
```

Esta expresión devuelve el contenido de variable si está definida y tiene un valor no nulo. Por ejemplo si la variable resultado inicialmente no esta definida:

```
$ echo ${resultado}
$ echo "E1 resultado es: {resultado:-0}"
E1 resultado es: 0
$ resultado=1
$ echo "E1 resultado es: ${resultado:-0}"
E1 resultado es: 1
```

A los parámetros posicionales podemos acceder como:

```
{$1: -0}
```

2.5.2 Uso de variable no definida

En algunas ocasiones interesa utilizar un valor predeterminado sólo en el caso de que la variable no esté definida. La expresión que se utiliza para ello es algo diferente de la anterior:

```
${variable- valorpredeterminado}
```

Consultar el ejemplo anterior.

2.5.3 Uso de variable definida o con valor nulo

Existe una expresión opuesta a la anterior. En este caso, si la variable está definida y contiene un valor no nulo, entonces en vez de usarse dicho valor, se utiliza el que se especifica. En caso contrario, el valor de la expresión es la cadena nula. La expresión para esto es:

```
$ {variable: +valorpredeterminado}
Por ejemplo:
$ resultado=10
$ echo ${resultado:+5}
5
$ resultado=
$ echo ${resultado:+30}
$
```

2.5.4 Uso de variable no definida

En algunas ocasiones puede también interesar que el comportamiento anterior sólo suceda cuando la variable no esté definida. La expresión para ello es:

```
$ {variable+valorpredeterminado}
```

2.6 Asignación de valores predeterminados de variables

Anteriormente veíamos la forma de utilizar un valor predeterminado de las variables en ciertos casos. Ahora, además de usar el valor predeterminado, queremos asignarlo a la variable.

2.6.1 Asignación a variable o definida o con valor nulo

En este caso no sólo utilizamos un valor predeterminado, sino que en la misma operación lo asignamos. La expresión que se utiliza para ello es la siguiente:

```
`${variable:=valorpredeterminado}`
```

Si el contenido de variable es no nulo, esta expresión devuelve dicho valor. Si el valor es nulo o la variable no está definida entonces el valor de la expresión es valorpredeterminado, el cual será también asignado a la variable variable. Veamos un ejemplo en el que se supone que la variable resultado no está definida:

```
$ echo ${resultado}
$ echo "El resultado es: ${resultado:=0}"
E1 resultado es: 0
$ echo ${resultado}
0
```

A los parámetros posicionales no se le pueden asignar valores utilizando este mecanismo.

2.6.2 Asignación a variable no definida

Análogo al caso anterior para el caso de que la variable no esté definida. La expresión ahora es:

```
`${variable=valorpredeterminado}`
```

2.6.3 Mostrar un mensaje de error asociado a una variable

Ahora lo que pretendemos es terminar un script con un mensaje de error asociado al contenido de una variable.

2.6.4 Variable no definida o con valor nulo

En otras ocasiones no interesa utilizar ningún valor por defecto, sino comprobar que la variable está definida y contiene un valor no nulo. En este último caso interesa avisar con un mensaje y que el programa de shell termine. La expresión para hacer esto es:

```
`${variable} : ?Mensaje`
```

Por ejemplo:

```
$ res=${resultado:? "variable no válida"}
resultado variable no válida
```

En el caso de que la variable resultado no esté definida o contenga un valor nulo, se mostrará el mensaje especificado en pantalla, y si esta instrucción se ejecuta desde un programa de shell, éste finalizará.

2.6.5 Variable no definida

Análogo al caso anterior para el caso de que la variable no esté definida. La expresión para ello es:

```
${variable?mensaje}
```

2.7 Otras operaciones con variables

Ciertas shell proporcionan unas facilidades que pueden ser útiles para ahorrar código en la programación de guiones de shell, como son la eliminación o extracción de subcadenas de una variable.

2.7.1 Subcadenas de una variable

```
${variable:inicio:longitud}
```

Extrae una subcadena de variable, partiendo de inicio y de tamaño indicado por longitud. Si se omite longitud toma hasta el fin de la cadena original.

```
$ A=abcdef  
$ echo ${A:3:2}  
de  
$ echo ${A:1:4}  
bcde  
$ echo ${A:2}  
cdef
```

2.7.2 Cortar texto al principio de una variable

```
${variable#texto}
```

Corta texto de variable si variable comienza por texto. Si variable no comienza por texto variable se usa inalterada. El siguiente ejemplo muestra el mecanismo de funcionamiento:

```
$ A=abcdef  
$ echo ${A#ab}  
cdef  
$ echo ${A#$B}  
cdef  
$ B=abc  
$ echo ${A#$B}  
def  
$ echo ${A#cd}  
abcdef
```

2.7.3 Cortar texto al final de una variable

```
${variable%texto}
```

Corta texto de variable si variable termina por texto. Si variable no termina por texto variable se usa inalterada.

Vemos un ejemplo:

```
$ PS1=$
$PS1="$ "
$ A=abcdef
$ echo ${A%def}
abc
$ B=cdef
$ echo ${A%$B}
ab
```

2.7.4 Reemplazar texto en una variable

```
${variable/texto1/texto2}
${variable//texto1/texto2}
```

Sustituye texto1 por texto2 en variable. En la primera forma, sólo se reemplaza la primera aparición. La segunda forma hace que se sustituyan todas las apariciones de texto1 por texto2.

```
$ A=abcdef
$ echo ${A/abc/x}
xdef
$ echo ${A/de/x}
abcxf
```

2.8 Evaluación aritmética

Es habitual tener que efectuar evaluaciones de expresiones aritméticas enteras durante la ejecución de un script de shell; por ejemplo para tener contadores o acumuladores o en otros casos.

Hasta ahora habíamos visto que esto lo podíamos hacer con `expr`, pero hay otra forma más cómoda: `let`

La sintaxis de `let` es la siguiente:

```
let variable=expresión aritmética
```

por ejemplo :

```
let A=A+1
```

En algunas shell incluso podremos omitir la palabra `let`, aunque por motivos de compatibilidad esto no es aconsejable.

Para evaluar expresiones reales, es decir con coma decimal, tendremos que usar otros mecanismos y utilidades que pueda proporcionar el sistema. En linux disponemos de la orden `bc`.

2.9 Selección de la shell de ejecución

Si queremos que nuestro programa sea interpretado por una shell concreta lo podemos indicar en la primera línea del fichero de la siguiente forma

```
#!/ruta/shell
```

Por ejemplo, si queremos que sea la shell bash la que interprete nuestro script tendríamos que comenzarlo por

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

En ciertas ocasiones es interesante forzar que sea una shell concreta la que interprete el script, por ejemplo si el script es simple podemos seleccionar una shell que ocupe pocos recursos como sh. Si por el contrario el script hace uso de características avanzadas puede que nos interese seleccionar bash como shell.

2.10 Lectura desde la entrada estándar: read

La orden read permite leer valores desde la entrada estándar y asignarlos a variables de shell.

La forma más simple de leer una variable es la siguiente:

```
read variable
```

Después de esto, el programa quedará esperando hasta que se le proporcione una cadena de caracteres terminada por un salto de línea. El valor que se le asigna a variable es esta cadena (sin el salto de línea final).

La instrucción read también puede leer simultáneamente varias variables:

```
read var1 var2 var3 var4
```

La cadena suministrada por el usuario empieza por el primer carácter tecleado hasta el salto de línea final. Esta línea se supone dividida en campos por el separador de campos definido por la variable de shell IFS (Internal Field Separator) que de forma predeterminada es una secuencia de espacios y tabuladores.

El primer campo tecleado por el usuario será asignado a var1, el segundo a var2, etc. Si el número de campos es mayor que el de variables, entonces la última variable contiene los campos que sobran. Si por el contrario el número de campos es mayor que el de variables las variables que sobran tendrán un valor nulo.

La segunda característica es la posibilidad incluir un mensaje informativo previo a la lectura. Si en la primera variable de esta instrucción aparece un carácter "?", todo lo que quede hasta el final de este primer argumento de read, se considerará el mensaje que se quiere enviar a la salida estándar.

Ejemplo:

```
read nombre?"Nombre y dos apellidos? " apl ap2
```

2.11 Evaluación de expresiones: test

La orden test evalúa una expresión para obtener una condición lógica que posteriormente se utilizará para determinar el flujo de ejecución del programa de shell con las sentencias correspondientes. La sintaxis de la instrucción es:

```
test expr
```

Para construir la expresión disponemos una serie de facilidades proporcionadas por la shell. Estas expresiones evalúan una determinada condición y devuelven una condición que puede ser verdadera o falsa. El valor de la condición actualiza la variable \$? con los valores cero o uno para los resultados verdadero o falso.

Pasamos a describir a continuación estas condiciones, y tenemos que tener en cuenta que es importante respetar todos los espacios que aquí aparecen.

-f (fichero existe el fichero y es un fichero regular)

-r (fichero existe el fichero y es de lectura)
-w (fichero existe el fichero y es de escritura)
-x (fichero existe el fichero y es ejecutable)
-h (fichero existe el fichero y es un enlace simbólico.)
-d (fichero existe el fichero y es un directorio)
-p (fichero existe el fichero y es una tubería con nombre)
-c (fichero existe el fichero y es un dispositivo de carácter)
-b (fichero existe el fichero y es un dispositivo de bloques)
-u f (fichero existe el fichero y está puesto el bit SUID)
-g (fichero existe el fichero y está puesto el bit SGID)
-s (fichero existe el fichero y su longitud es mayor que 0)
-z s1 (la longitud de la cadena s1 es cero)
-n s1 (la longitud de la cadena s1 es distinta de cero)
s1=s2 (la cadena s1 y la s2 son iguales)
s1!=s2 (la cadena s1 y la s2 son distintas)
n1 -eq n2 (los enteros n1 y n2 son iguales.)
n1 -ne n2 (los enteros n1 y n2 no son iguales.)
n1 -gt n2 n1 (es estrictamente mayor que n2.)
n1 -ge n2 n1 (es mayor o igual que n2.)
n1 -lt n2 n1 (es menor estricto que n2.)
n1 -le n2 n1 (es menor o igual que n2.)

Estas expresiones las podemos combinar con:

! (operador unario de negación)
-a (operador AND binario)
-o (operador OR binario)

Ejemplos:

```
$ test -f /etc/profile  
$ echo $?  
$ test -f /etc/profile -a -w /etc/profile  
$ echo $?  
$ test 0 -lt 0 -o -n "No nula"  
$ echo $?
```

La orden test se puede sustituir por unos corchetes abiertos y cerrados. Es necesario dejar espacios antes y después de los corchetes; este suele ser uno de los errores más frecuentes.

2.12 Estructura de control

La programación en shell dispone de las sentencias de control del flujo de instrucciones necesarias para poder controlar perfectamente la ejecución de las órdenes necesarias.

2.12.1 Sentencia if

La shell dispone de la sentencia if de bifurcación del flujo de ejecución de un programa similar a cualquier otro lenguaje de programación. La forma más simple de esta sentencia es:

```
if lista_órdenes
then
  lista_órdenes
fi
```

fi, que es if alrevés, indica donde termina el if.

En la parte reservada para la condición de la sentencia if, aparece una lista de órdenes separados por ";". Cada uno de estos mandatos es ejecutado en el orden en el que aparecen. La condición para evaluar por if tomará el valor de salida del último mandato ejecutado. Si la última orden ha terminado correctamente, sin condición de error, se ejecutará la lista de órdenes que hay tras then. Si esta orden ha fallado debido a un error, la ejecución continúa tras el if.

Como condición se puede poner cualquier mandato que interese, pero lo más habitual es utilizar diferentes formas de la orden test. Por ejemplo:

```
if [ -f mifichero ]
then
echo "mifichero existe"
fi
Pero también podemos poner:
if grep body index.html
then
echo "he encontrado la cadena body en index.html"
fi
```

Como en cualquier lenguaje de programación también podemos definir las acciones que se tienen que ejecutar en el caso de que la condición resulte falsa:

```
if lista_órdenes
then
  lista_órdenes
else
  lista_órdenes
fi
```

Por ejemplo:

```
if [ -f "$1" ] then
pr $1
else
echo "$1 no es un fichero regular"
fi
```

Cuando queremos comprobar una condición cuando entramos en el else, es decir, si tenemos else if es posible utilizar elif. Vemos un ejemplo:

```
if [ -f "$1" ]
then
cat $1
elif [ -d "$1" ]
then
ls $1/*
else
echo "$1 no es ni fichero ni directorio"
fi
```

2.12.2 Sentencia while

La sentencia while tiene la siguiente sintaxis:

```
while lista_órdenes
do
lista órdenes
done
```

La lista de órdenes que se especifican en el interior del bucle while se ejecutará mientras que lista_órdenes devuelva un valor verdadero, lo que significa que la última orden de esta lista termina correctamente.

Vemos un ejemplo:

```
I=0
while [ ${resp:=s} = s ]
do
I=\{ }expr $I + 1\{ }
echo $I
read resp?"Quiere usted continuar(s/n)? "
done
```

2.12.3 Sentencia until

La sentencia until similar a while, es otro bucle que se ejecutará hasta que se cumpla la condición, es decir, hasta que la lista de órdenes termina correctamente. Su formato es el siguiente:

```
until lista_órdenes
do
lista órdenes
done
```

2.12.4 Sentencia for

La sentencia for repite una serie de órdenes a la vez que una variable de control va tomando los sucesivos valores indicado por una lista de cadenas de texto. Para cada iteración la variable de control toma el valor de uno de los elementos de la lista. La sintaxis de for es la siguientes

```
for variabl in lista
do
lista mandatos
done
```

lista es una serie de cadenas de texto separadas por espacios y tabuladores. En cada iteración del bucle la variable de control variable toma el valor del siguiente campo y se ejecuta la secuencia de mandatos lista_mandatos.

Ejemplo:

```
for i in $*
do
echo $i
done
```

y mostraríamos todos los parámetros posicionales.

```
for i in *
do
echo $i
done
```

y mostraríamos la lista de ficheros del directorio activo.

2.12.5 Sentencias break y continue

La sentencia break se utiliza para terminar la ejecución de un bucle while o for. Es decir el control continuará por la siguiente instrucción del bucle. Si existen varios bucles anidados, break terminará la ejecución del último que se ha abierto.

Es posible salir de n niveles de bucle mediante la instrucción break n.

La instrucción continue reinicia el bucle con la próxima iteración dejando de ejecutar cualquier orden posterior en la iteración en curso.

2.12.6 Sentencia case

La sentencia case proporciona un if múltiple similar a la sentencia switch de C. El formato básico de esta sentencia es el siguiente:

```
case variable in
patrón1)
lista_órdenes1
;;
patrón2)
lista_órdenes2
;;
...
patrónN)
lista_órdenesN;;
esac
```

La shell comprueba si variable coincide con alguno de los patrones especificados. La comprobación se realiza en orden, es decir empezando por patrón1 terminando por patrónN. En el momento en que se detecte que la cadena cumple algún patrón, se ejecutará la secuencia de mandatos correspondiente hasta llegar a ";;". Estos dos puntos y comas fuerzan a salir de la sentencia case y a continuar por la siguiente sentencia después de esac (esac es case alrevés).

Las reglas para componer patrones son las mismas que para formar nombres de ficheros, así por ejemplo, el carácter "*" es cumplido por cualquier cadena, por lo que suele colocarse este patrón en el último lugar, actuando como acción predeterminada para el caso de que no se cumpla ninguna de las anteriores. Ejemplo:

```
case "$1" in
  start)
    echo -n "Ha seleccionado start "
    ;;
  stop)
    echo -n "Ha seleccionado stop "
    ;;
  status)
    echo -n "Ha seleccionado stop "
    ;;
  restart)
    echo -n " Ha seleccionado restart "
    ;;
  *)
    echo "No es una opción válida"
    exit 1
esac
```

2.13 Terminar un programa de shell (exit)

Como hemos visto, todas las órdenes órdenes tienen un estado de finalización, que por convenio es 0 cuando la ejecución terminó correctamente, y un valor distinto de 0 cuando lo incorrectamente o con error.

Si un programa de shell termina sin errores devolverá un valor cero, pero también es posible devolver explícitamente un valor mediante la sentencia exit. La ejecución de esta sentencia finaliza en ese instante el programa de shell, devolviendo el valor que se le pose como argumento, o el estado de la última orden ejecutada si no se le pasa ningún valor.

Ejemplo:

```
if grep "$1" /var/log/messages
then
  exit 0
else
  exit 10
fi
```

2.14 Opciones en un programa de shell: getopt

En los sistema Unix es habitual poner las opciones para ejecutar una orden siguiendo unas normas:

Las opciones están formadas por una letra precedida de un guión.

Cada opción puede llevar uno o más argumentos.

Las opciones sin argumentos pueden agruparse después de un único guión

Para facilitar el análisis de las opciones la shell dispone de la orden getopt. La sintaxis de getopt es la siguiente:

```
getopts cadenaopciones variable [args ...]
```

En cadenaopciones se sitúan las opciones válidas del programa. Cada letra en esta cadena significa una opción válida. El carácter ":"

después de una letra indica que esa opción lleva un argumento asociado o grupo de argumentos separados por una secuencia de espacios y tabuladores.

Esta orden se combina con la sentencia `while` para iterar por cada opción que aparezca en la línea de órdenes en la llamada al programa. En variable se almacena el valor de cada una de las opciones en las sucesivas llamadas. En la variable `OPTIND` se guarda el índice del siguiente argumento que se va a procesar. En cada llamada a un programa de shell esta variable se inicializa a 1.

Además:

Cuando a una opción le acompaña un argumento, `getopts` lo sitúa en la variable `OPTARG`.

Si en la línea de mandatos aparece una opción no válida entonces asignará "?" a variable.

Veamos un ejemplo:

```
while getopts ab:cd opcion
do
  case $opcion in
    a) echo "Opción a"
      ;;
    b) echo "Opción b con argumento $OPTARG"
      ;;
    c) echo "Opción c"
      ;;
    d) echo "Opción d"
      ;;
    ?) echo "Uso: $0 -acd -b opcion "
      ;;
  esac
done
shift `expr $OPTIND - 1`
echo "resto de argumentos: $@"
```

Observamos como hemos desplazados todos los argumentos con `shift` para hacerlos disponibles con los parámetros posicionales y descartando las opciones previamente analizadas.

2.15 Evaluación de variables: eval

La orden `eval` toma una serie de cadenas como argumentos:

```
eval [arg1 [arg2] ...]
```

los expande siguiendo las normas de expansión de la shell, separándolos por espacios y trata de ejecutar la cadena resultante como si fuera cualquier orden.

Esta instrucción se debería utilizar cuando:

Pretendemos examinar el resultado de una expansión realizada por la shell.

Para encontrar el valor de una variable cuyo nombre es el valor de otra variable.

Ejecutar una línea que se ha leído o compuesto internamente en el programa de shell.

2.16 Funciones

Ciertas shell como `bash` permiten la declaración de funciones para agrupar bloques código como en un lenguaje de programación convencional.

La forma de declarar un función es

```
function mi_funcion
{
código de la función
}
```

Para realizar la llamada a la función sólo tenemos que usar su nombre. Además las funciones pueden tener argumentos en su llamada. No es necesario declarar los parámetros en la declaración de la función, basta usar las variables \$1, \$2, etc. dentro de la definición de las instrucciones y serán los parámetros en su orden correspondiente. Para llamar a una función con argumentos no se usan los habituales paréntesis.

Ejemplos:

```
#!/bin/bash
function terminar {
    exit 0
}
function saludo {
    echo ¡Hola Mundo!
}
saludo
terminar
```

Otro ejemplo:

```
#!/bin/bash
function terminar {
    exit 0
}
function saludo {
    echo $1
}
saludo Hola
saludo Mundo
terminar
```

En este último ejemplo podemos observar el uso de una función con argumentos, tanto en la declaración como en la llamada.

2.17 Trucos de programación en shell

Vemos a continuación una serie de ejemplos genéricos para resolver cuestiones que se presentan en la programación de shell con cierta frecuencia

2.17.1 Script con número variable de argumentos:

Este programa de shell pretende ejecutar una serie de instrucciones para cada uno de los argumentos que se le pasen desde la línea de órdenes.

```
for i in $*
do
instrucciones
done
```

Por ejemplo, hacemos un script que mueva todos los ficheros pasado como argumento al directorio ./papelera:

```
for i in $*
do
if [ -f $i ]
then
mv $i ./papelera
fi
done
```

2.17.2 Script para aplicar una serie de órdenes a cada fichero de un directorio

Muy similar al ejemplo anterior, pero sustituimos \$* por simplemente * que equivale a todos los ficheros del directorio activo.

```
for i in *
do
instrucciones
done
```

2.17.3 Leer un fichero de texto línea a línea

Es muy habitual tener que procesar todas las línea de un fichero de texto para realizar diferentes operaciones. Vemos una primera forma:

```
while read LINEA
do
instucciones por línea
done < fichero
```

En este caso estamos redirigiendo la entrada estándar de la orden read, que es el teclado, por un fichero. Al igual que en el caso del teclado, la lectura se realizará hasta que se encuentre un salto de línea. Observamos como la redirección se realiza tras el final de la sentencia while.

Otra forma posible para hacer esto mismo sería:

```
cat fichero|while read LINEA
do
instrucciones por línea
done
```

Este método difiere ligeramente del anterior, ya que al utilizar una tubería creamos una nueva shell con lo cual puede ocurrir que no se conserven ciertos valores de las variables de shell.

2.17.4 Cambiar una secuencia de espacios por un separador de campos

La salida de ciertas órdenes y ciertos ficheros separan los datos por espacios en blanco. Por ejemplo las órdenes ps o ls -la, o ifconfig.

Si queremos utilizar parte de los datos de las salidas de estas órdenes tendremos que contar las columnas en las que aparece cada dato y

cortarlos con cut usando la opción -c. Pero otra opción sería sustituir toda una serie de espacios en blanco por un separador, por ejemplo ":" o ";".

Por ejemplo vamos a ver como sustituir los espacios de la orden ifconfig por ";".

La salida normal sería:

```
$ /sbin/ifconfig
eth0  Link encap:Ethernet HWaddr 00:90:F5:08:37:E4
      inet addr:192.168.1.5 Bcast:192.168.1.255 Mask:255.255.255.0
      UP BROADCAST MULTICAST MTU:1500 Metric:1
      RX packets:0 errors:0 dropped:0 overruns:0 frame:0
      TX packets:5103 errors:0 dropped:0 overruns:0 carrier:0
      collisions:0 txqueuelen:100
      Interrupt:10 Base address:0x3200

lo    Link encap:Local Loopback
      inet addr:127.0.0.1 Mask:255.0.0.0
      UP LOOPBACK RUNNING MTU:16436 Metric:1
      RX packets:1726 errors:0 dropped:0 overruns:0 frame:0
      TX packets:1726 errors:0 dropped:0 overruns:0 carrier:0
      collisions:0 txqueuelen:0
```

Ahora usamos sed para sustituir cualquier secuencia de espacios en blanco ([]*) por un separador ":":

```
$ /sbin/ifconfig | sed "s/[ ]*/;/g"
eth0;Link;encap:Ethernet;HWaddr;00:90:F5:08:37:E4;
;inet;addr:192.168.1.5;Bcast:192.168.1.255;Mask:255.255.255.0
;UP;BROADCAST;MULTICAST;MTU:1500;Metric:1
;RX;packets:0;errors:0;dropped:0;overruns:0;frame:0
;TX;packets:5241;errors:0;dropped:0;overruns:0;carrier:0
;collisions:0;txqueuelen:100;
;Interrupt:10;Base;address:0x3200;

lo;Link;encap:Local;Loopback;
;inet;addr:127.0.0.1;Mask:255.0.0.0
;UP;LOOPBACK;RUNNING;MTU:16436;Metric:1
;RX;packets:1773;errors:0;dropped:0;overruns:0;frame:0
;TX;packets:1773;errors:0;dropped:0;overruns:0;carrier:0
;collisions:0;txqueuelen:0;
```

Ahora podríamos cortar de forma exacta el campo que nos interese, por ejemplo:

```
$ /sbin/ifconfig | sed "s/[ ]*/;/g" | grep inet | cut -f4 -d:
192.168.1.5
127.0.0.1
```

Vemos paso a paso la anterior orden compuesta:

Primero ejecutamos la orden ifconfig

Sustituimos los espacios en blanco por ":"

Buscamos la línea que contenga la palabra inet
Cortamos el campo 4 usando ":" como separador.

2.18 Prácticas Ejercicios propuestos

¿Que salida ocasionaría cada una de las siguientes órdenes si la ejecutamos consecutivamente?

```
$ set a b c d e f g h i j k l m n  
$ echo $10  
$ echo $15  
$ echo $*  
$ echo $#  
$ echo $?  
$ echo ${11}
```

Explica que realizaría cada una de las siguientes órdenes ejecutadas en secuencia

```
$ A= $B  
$ B=ls  
$ echo $A  
$ eval $A
```

Mostrar el último parámetro posicional

Pista: \$#

Asignar el último parámetro posicional a la variable ULT

Realizar un programa que escriba los 20 primeros números enteros.

Realizar un programa que numere las líneas de un fichero

Realizar un programa que tomando como base el contenido de un directorio escriba cada elemento contenido indicando si es fichero o directorio.

Realizar un programa que muestre todos los ficheros ejecutables del directorio activo.

Modificar el programa anterior para que indique el tipo de cada elemento contenido en el directorio activo: fichero, directorio, ejecutable,...

Ejercicios resueltos sobre ficheros y directorios

Guion de shell que genere un fichero llamado listaetc que contenga los ficheros con permiso de lectura que haya en el directorio /etc:

```
for F in /etc/*  
do  
if [ -f $F -a -r $F ]  
then  
echo $F >> listaetc  
fi  
done
```

Hacer un guión de shell que, partiendo del fichero generado en el ejercicio anterior, muestre todos los ficheros del directorio /etc que contengan la palabra "procmal":

```
while read LINEA
do
if grep procmail $L >/dev/null 2>&1
then
echo $L
fi
done <listaetc
```

Hacer un guion de shell que cuente cuantos ficheros y cuantos directorios hay en el directorio pasado como argumento:

```
DI=0
FI=0
for I in $1/*
do
if [ -f $I ]
then
let FI=FI+1
fi
if [ -d $I ]
then
let DI=DI+1
fi
done
```

Hacer un guion de shell que compruebe si existe el directorio pasado como argumento dentro del directorio activo. En caso de que exista, que diga si no está vacío.

```
if [ -d $1 ]
then
echo "$1 existe"
N=$(ls | wc -l)
if [ $N -gt 0 ]
then
echo "S1 no está vacío, contiene $N ficheros no ocultos"
fi
fi
```

Hacer un guion de shell que copie todos los ficheros del directorio actual en un directorio llamado csg. Si el directorio no existe el guion lo debe de crear.

```
if [ ! -d csg ]
then
mkdir csg
fi
cp * csg
```

Hacer un script que muestre el fichero del directorio activo con más líneas:

```

NLIN=0
for I in *
do
if [ -f $I ]
then
N=$(wc -l $I)
if [ $N -gt $NLIN ]
then
NOMBRE=$I
NLIN=$N
fi
fi
done
echo "$NOMBRE tiene $NLIN lineas"

```

¿Qué haría el siguiente script?

```

ORIGINAL="mejor"
NUEVA="óptimo"
for f in $(find . -type f)
do
    sed s/$ORIGINAL/$NUEVA/g $f >$f.tr
    mv $f.tr $f
done

```

¿Qué haría el siguiente script?

```

ORIGINAL="mejor"
NUEVA="óptimo"
for f in $(find . -type d)
do
    chmod a+rx $f
done

```

¿Qué haría el siguiente script?

```

find . -type d -exec chmod a+rx {} \;

```

¿Qué haría el siguiente script?

```

#!/bin/bash
fecha=`date +%Y%m%d`
mkdir /var/copia/$fecha
tar cfz /var/copia/$fecha/etc-$fecha.mod.tar.gz $(find /etc -newer /var/copia/ultimo ! -type d)
touch /var/copia/ultimo

```