

LINUX



LINUX SHELL

Tabla de contenido

1. EL INTÉRPRETE DE ÓRDENES.....	3
1.1. Modos de invocar una orden.....	4
1.2. Histórico de órdenes.....	6
1.3. Variables de entorno.....	6
1.4. Sustitución de órdenes.....	10
1.5. Alias.....	11
2. FICHEROS SCRIPT: PROGRAMACIÓN DE LA SHELL.....	12
2.1. Comentarios.....	15
2.2. Parámetros de un script.....	15
2.3. Más sobre el operador <i>expr</i>	16
2.4. Sentencias de control.....	17
2.5. Exportación de variables.....	27
2.6. Funciones.....	28



Linux Shell

1. El intérprete de órdenes.

Como sabemos, cuando iniciamos una sesión Linux/Unix respondiendo al *login* y dando nuestra clave de acceso, el sistema ejecuta automáticamente un programa denominado *shell* (literalmente caparazón, concha o cáscara), encargado de interpretar todas las órdenes que le indiquemos. La *shell*¹ es, pues, un intérprete de mandatos del Sistema Operativo y actúa de forma parecida a como lo hace el COMMAND.COM en MS-DOS. La *shell* indica que está presente y lista para recibir órdenes mostrándonos una marca o *prompt* del sistema, que en el caso de la *Bourne shell* (**sh**) y *Korn shell* (**ksh**) es el símbolo **\$** y para la *C shell* (**csh**) es el **%**². Linux utiliza por defecto como intérprete de órdenes una variante del *Bourne shell* llamada **bash** (*Bourne another Shell*).

La *shell* envuelve al resto de las capas del sistema (utilidades, núcleo y hardware), sirviéndole de “caparazón”. Es la capa más externa y actúa como interfaz entre el usuario y el resto del sistema. Básicamente es un intérprete de órdenes de línea, como hemos dicho. Su trabajo consiste en leer las instrucciones que le da el usuario, realizar una serie de funciones de análisis y pasar la orden interpretada al S.O. para su ejecución. El mecanismo clásico de ejecución de una orden por parte de la *shell* es hacer una llamada al sistema para crear un proceso hijo (**fork**) seguida por una llamada **exec** que indica el programa que quiere ejecutar. Todas las llamadas al sistema están manejadas por el núcleo, el cual también actúa como interfaz, en este caso entre el hardware y los programas en ejecución o procesos.

Puesto que la *shell* se utiliza como interfaz principal entre el usuario y el sistema operativo, muchos usuarios identifican el intérprete de comandos con Linux. Sin embargo no forma parte del *kernel* del sistema.

Entre las funciones de la *shell* podemos destacar básicamente la sustitución de los valores de las variables de entorno por variables referenciales, generar nombres de archivo a partir de los

¹ No está muy claro el género que debe aplicársele en español.

² En cualquier caso, el símbolo para el administrador del sistema es la almohadilla (#).

metacaracteres (caracteres sustituibles, el “*” y la “?”), manejar la redirección de E/S y las tuberías (*pipes*, “|”), realizar la substitución de órdenes, verificar si una orden es interna o se trata de un programa ejecutable y buscar la imagen binaria de la orden en caso de que sea una orden externa.

Cuando el intérprete de órdenes tiene que ejecutar una orden lo que hace es duplicarse (aparecen dos intérpretes de comandos gracias a la llamada **fork**), una copia se encarga de ejecutar la orden especificada (*proceso hijo*)³ y la otra (*proceso padre*) generalmente se queda esperando a que la orden termine de ejecutarse (**wait**). Aunque esto último no es estrictamente obligatorio, puede ocurrir que el proceso padre (la *shell*) se ejecute concurrentemente con el proceso hijo (no olvidemos que Linux es multitarea). Los procesos por los que la *shell* espera se dice que se están ejecutando en primer plano (*foreground*) y los que se ejecutan a la vez que la *shell* se denominan procesos en segundo plano (*background*). Esto puede ser útil, por ejemplo, para iniciar una orden y después ejecutar otra inmediatamente sin tener que esperar a que la primera finalice (ver capítulo anterior, **Gestión de Procesos**).

El primer intérprete de comandos desarrollado para Unix, se llamó *Bourne Shell*, en honor al hombre que encabezó el equipo que lo desarrolló, **Steve Bourne** (Laboratorios Bell de la AT&T). **William Joy**, de la Universidad de California (Berkeley) mejoró esta *shell* al crear la **cs**h (*California Shell*) que incorporaba nuevas funciones como **history**, **alias**, posibilidad de crear programas de *shell* más versátiles, etc., aunque no es estándar y plantea conflictos con programas de la *Bourne Shell*. Más tarde **David Korn** (Laboratorios Bell) desarrolló una nueva *shell*, el **k**sh (*Korn shell*) que incorpora las mejores funciones de las dos anteriores y es totalmente compatible con la primera (utiliza la misma sintaxis). Por último, la *Free Software Foundation* desarrolló **bash** que se basa en **sh** pero incorpora características de la *Korn* y *C shells*. Red Hat Linux proporciona normalmente en sus distribuciones varios *shell*'s como **sh**, **bash** (activada por defecto), **tcsh**, **cs**h, **pdksh**⁴.

1.1. Modos de invocar una orden.

El intérprete de órdenes es capaz de reconocer distintos modos de invocar una o varias órdenes ofreciéndonos distintas posibilidades para ejecutarlas. Éstas son:

orden &

Ejecuta la orden en segundo plano. De este modo mientras se ejecuta la orden en cuestión, la *shell* nos devuelve el control y mientras podemos ejecutar otros programas. Cuando se ejecuta un proceso en segundo plano aparece un número en la pantalla. Se trata de su **PID** o identificativo del proceso.

³ El nuevo *shell* se ejecuta en el directorio actual y hereda muchas de las variables de la *shell* ya existente.

⁴ *Public domain Korn shell*, una copia de la *shell Korn*.

`orden1 ; orden2`

Permite ejecutar varias órdenes invocadas desde una única línea. Las distintas órdenes deben ir separadas por punto y coma (;).

`orden1 | orden2`

Las órdenes se van comunicando mediante una tubería. La salida de `orden1` es la entrada de `orden2`.

`orden1 && orden2`

Ejecuta la primera orden y si finaliza con éxito se ejecutará la segunda. Los símbolos “&&” actúan como el operador lógico AND.

`orden1 || orden2`

Ejecuta la segunda orden sólo si falla la primera, es decir, si `orden1` acaba sin errores jamás se ejecutará `orden2`. Los símbolos “||” actúan como el operador lógico OR.

Ejemplo:

```
# date ; sleep 10 ; date
Wed Mar 15 16:40:55 CET 2000
Wed Mar 15 16:41:05 CET 2000
```

Como cabe suponer, `sleep 10` indica al sistema que espere 10 segundos.

Si se desea ejecutar una orden o un *script* en *background* (segundo plano) bastará con teclear el signo `&` al final del mandato como ya sabemos. En este caso puede resultar interesante redirigir la salida del mandato o proceso a un fichero para que no interfiera en el trabajo que se esté realizando en pantalla. Cuando un proceso se ejecuta en *background* devuelve un identificador de proceso (*PID*) que servirá para controlar su ejecución e incluso para detenerla si fuera necesario. Por ejemplo:

```
# proceso1 > pantalla &
726
```

Cuando un usuario abandona el sistema todos sus procesos son eliminados. Si se quiere que un proceso lanzado continúe su ejecución aunque el usuario se desconecte es necesario utilizar el mandato `nohup` cuando se ejecute.

Ejemplo:

```
# nohup proceso1 > pantalla &
```

De esta forma, aunque el usuario abandone el sistema el proceso continuará su ejecución hasta finalizar y sus salidas serán grabadas en el fichero `pantalla`. Si la salida no está redirigida, el sistema crea un fichero llamado `nohup.out` (normalmente en el directorio de trabajo del usuario) donde deja el resultado de la ejecución del comando. Por otra parte `nohup` no pone al proceso en segundo plano por sí mismo, sino que para ello hay que utilizar el símbolo `&` al final de la invocación como hemos hecho hasta ahora.

1.2. Histórico de órdenes.

Todas las órdenes que vamos indicando en el *prompt* del sistema son almacenadas con objeto de poder recuperarlas en un futuro y volver a ejecutarlas o modificarlas. Para visualizar el registro histórico de órdenes podemos utilizar la orden `history`⁵ que además las muestra numeradas según el orden en que fueron introducidas. Si deseamos repetir una orden y conocemos su posición en el histórico de órdenes bastará pulsar el carácter “!” seguido del número de orden para invocarla de nuevo.

Ejemplo:

```
# !517
```

También podemos repetir la última orden que se ajusta a un determinado patrón tecleando el símbolo “!” seguido del patrón. De modo automático se analiza el histórico de órdenes y si existe alguna que se ajusta al patrón indicado simplemente la ejecuta.

1.3. Variables de entorno.

La *shell* dispone de un mecanismo para definir variables que pueden ser utilizadas para contener información usada por los programas del sistema, aplicaciones o la propia *shell*. Estas variables se pueden usar para personalizar o particularizar la forma en la que los programas interactúan con el usuario. Algunas son prefijadas por el sistema y otras pueden recibir valores según indicación del usuario. Algunas de las más comunes son:

HOME Contiene el nombre de la ruta absoluta del directorio de trabajo. Es definida automáticamente y fijada al directorio de trabajo de un usuario en particular, como

⁵ El número de órdenes que se pueden almacenar en el histórico viene determinado por el valor de la variable de entorno HISTSIZE (en el archivo `/etc/profile` se le asigna el valor 1000 por defecto).

parte del proceso de inicio de sesión. La propia *shell* utiliza esta información para saber a qué directorio debe cambiar cuando usamos la orden `cd` sin argumento.

PS1 Contiene la cadena del identificador del sistema, el signo de petición de orden primaria. Por defecto tiene el valor “\$”. En concreto, define el aspecto que tendrá el indicador del sistema.

PS2 Contiene el signo de petición de orden secundaria. Por defecto es “>”. Cuando la *shell* considere que el mandato especificado está incompleto mostrará este indicador para informarnos de la necesidad de completarlo.

LOGNAME Contiene el nombre de la cuenta de usuario. Es fijada automáticamente por el sistema.

TERM Define el tipo de terminal utilizado. Es utilizado por programas orientados a pantalla para informarse de las características del terminal.

SHELL Contiene la ruta de la *shell* activa. En nuestro caso, por defecto, contendrá `/bin/bash`. Es consultada por algunos programas que admiten órdenes de escape de *shell* (aquellas que interrumpen temporalmente dicho programa para ejecutarse bajo la *shell*).

MAIL Contiene la ruta completa del directorio donde se colocará el correo de un usuario.

TZ Contiene información sobre la hora zonal actual. Es establecida por el sistema y puede ser comprobado por el usuario. Toma como referencia el horario GMT. Esta información es necesaria para órdenes como `date` y para, por ejemplo, cambiar al horario de verano de forma automática.

PATH Indica la ruta de búsqueda para los mandatos ejecutables.

Para consultar todas las variables del entorno y sus valores correspondientes podemos utilizar la orden `env` (*environment*).



env

Posiblemente, una de las variables de entorno más importantes para el usuario sea **PATH**. Esta variable contiene una serie de rutas donde Linux buscará los ficheros binarios de los mandatos y las aplicaciones de usuario. Estas rutas están delimitadas por el carácter “:” y el orden en el que están especificadas indicará donde Linux buscará primero. Hay que recordar que los directorios `/bin` y `/usr/bin` contienen todas las órdenes estándares de Linux. El directorio `/usr/local/bin` suele contener órdenes locales añadidas por los usuarios (esta tarea de añadir órdenes locales es normalmente responsabilidad del administrador del sistema). Si se quiere que el sistema incorpore un nuevo directorio de búsqueda habrá que incluirlo en la variable **PATH** de la forma:

```
# PATH=$PATH:nuevo-directorio
```

El símbolo “\$” colocado delante de una variable permite el acceso al contenido de ésta. Podemos utilizar este símbolo conjuntamente con la orden **echo** para ver el contenido de cualquier variable de entorno. La orden **echo** visualiza un mensaje o el contenido de una variable en pantalla.

```
echo [item]
```

Ejemplo:

```
# echo $PATH
/bin:/home/you/bin:/usr/sbin
```

El usuario también puede crear nuevas variables. Estas variables son de tipo alfanumérico y para crearlas basta con asignarles un valor de la siguiente forma:

```
variable=valor
```

Entre el nombre de la variable y el signo igual, así como entre éste y el valor asignado, **no** debe haber espacios en blanco.

```
# MIDIR=$HOME
```

Para evitar confusión cuando se quiere añadir una extensión a una variable deberemos delimitarla con los símbolos “{” y “}”. Por ejemplo, podemos definir una variable que consiste en el valor original de otra variable más una extensión:

```
# MISERRORES=${MIDIR}/errors
# echo $MISERRORES
/home/root/errors
# mkdir $MISERRORES
```

Para visualizar todas las variables activas en la sesión se utiliza el mandato **set**, que muestra las variables de entorno y las creadas por el usuario.

```
set
```

En ocasiones es necesario solicitar información al usuario. Esta información se almacena también en variables y para solicitarla se emplea el mandato **read** seguido del nombre de la variable o variables donde queremos almacenar la/s entrada/salidas.


```
read [lista-variables]
```

Ejemplo:

```
# echo Dame tu nombre

# read nombre

Alexandros

# echo Hola $nombre

Hola Alexandros
```

Si sólo se indica una variable después de **read** en ella se almacenará todo lo que teclee el usuario. Si se utilizan dos variables en un **read** la primera recoge el texto que hay hasta el primer espacio en blanco y la segunda el resto. Si hubiese tres variables, la primera almacenará el texto hasta el primer espacio en blanco, la segunda el texto hasta el siguiente espacio en blanco y la tercera el resto del texto; y así sucesivamente. Es decir, el separador de la entrada es el espacio en blanco y la última variable siempre recoge el resto de la información tecleada por el usuario y que queda por asignar. Por ejemplo:

```
# read uno dos tres

primera segunda y resto del texto

# echo $uno

primera

# echo $dos

segunda

# echo $tres

y resto del texto
```

Las variables que se pueden utilizar son del tipo alfanumérico y por tanto no se pueden realizar operaciones matemáticas con ellas de forma directa. Para realizar cálculos es necesario utilizar el mandato **expr**, con el cual podemos usar los operadores **+**, **-**, ***** (producto), **/** (división) y **%**

(resto)⁶. Este mandato sólo permite trabajar con números enteros y es necesario que los operadores y los operandos vayan separados por un espacio en blanco. Ejemplo:

```
# expr 4 + 5

9
```

Volveremos sobre el operador `expr` cuando tratemos los ficheros *script*.

1.4. Sustitución de órdenes.

La **sustitución de órdenes** es otra característica práctica de la *shell*. Nos permite captar la salida de una orden y asignarla a una variable o bien usar esa salida como un argumento de otra orden. Esto se consigue encerrando la orden con un “mandato especial” denominado **grave** (la comilla simple invertida o acento grave o francés).

Ejemplo, si nos encontramos en el directorio `X11` de `usr`:

```
# directorio=`pwd`

# echo $directorio

/usr/X11
```

También podemos concatenar a un literal el contenido de una variable creada por el usuario, como hicimos antes:

```
# cuenta=`logname`

# echo ${cuenta} es el usuario actual

root es el usuario actual
```

Hay que tener cuidado a la hora de utilizar los siguientes delimitadores:

- **Comillas dobles** (“): Desactivan la creación de nombres de archivos, sin embargo la sustitución de órdenes y variables de *shell* permanece activada.
- **Apóstrofe** ('): Desactiva toda la función de análisis de la *shell*. Todo lo que se delimite con apóstrofes se traspasa como un solo parámetro.
- **Acento grave** (`): Provoca la ejecución de la orden.

⁶ No hay que olvidar que los caracteres `*`, `(` y `)` tienen un significado especial para la *shell*, por lo que habrá que precederlos del carácter `\` (*backslash*).

Ejemplo: El comando siguiente muestra las cuatro últimas líneas del archivo de texto especificado:

```
# tail -6 TheHardestPart.txt
```

Este comando podemos almacenarlo en una variable de la forma siguiente:

```
# VER6LINEAS="tail -6 TheHardestPart.txt"
```

Veamos el resultado de ejecutar los siguientes comandos:

```
# echo $VER6LINEAS
tail -6 TheHardestPart.txt

# echo "$VER6LINEAS"
tail -6 TheHardestPart.txt

# echo '$VER6LINEAS'
$VER4LINEAS

# echo ` $VER6LINEAS `
AND THE HARDEST PART
WAS LETTING GO, NOT TAKING PART
WAS THE HARDEST PART
AND THE STRANGEST THING
WAS WAITING FOR THAT BELL TO RING
IT WAS THE STRANGEST START.
```

Si queremos visualizar un mensaje que lleva apóstrofe o cualquier otro carácter interpretable por la *shell* basta con antecederle el símbolo barra invertida ("**"o *backslash*), conocido como carácter escape.

Ejemplo:

```
# echo Today\'s date and time are `date`
Today´s date and time are Mon Mar 20 20:30 CET 2000
```

1.5. Alias.

Los **alias** se utilizan para poder invocar órdenes con un nombre diferente al utilizado normalmente. La orden alias posee el siguiente formato sintáctico:

```
alias identificador="orden"
```

Puede ser interesante para un usuario acostumbrado a utilizar el sistema operativo MS-DOS hacer que `dir` sea equivalente a `ls -ld`, por ejemplo.

```
# alias dir="ls -ld"

# dir *dir

drwxrwxr-x  2 gandalf  lords    30456   Mar 15 21:30 Midir
drwxr-xr-x  3 gandalf  users     3047    Feb  5 22:45 Sudir
```

La orden `alias` define un enlace entre el primer y el segundo argumentos. En el momento en que introducimos el primer argumento en la línea de comandos la shell de Linux lo sustituye por el segundo. Para poder conocer todos los alias que hay definidos en el sistema podemos teclear la orden sin argumentos. Estos alias permanecen activos hasta que empleamos la orden `unalias`.

`unalias identificador`

Por ejemplo, si queremos deshacer el alias que hemos definido antes, tendríamos que teclear:

```
# unalias dir
```

Si queremos que los alias definidos estén disponibles cada vez que iniciemos una sesión de trabajo, tanto en los terminales virtuales como en los *xterm* del entorno gráfico deberemos definirlos en los ficheros *bashrc* que cuelgan del directorio por defecto de cada usuario.

2. Ficheros script: programación de la shell.

Un *script* (*guión de shell*) es un proceso por lotes. En realidad es un fichero de texto⁷ cuyo contenido son mandatos del sistema operativo (los ficheros *batch* de MS-DOS). Cuando se teclee el nombre de un fichero *script* en el indicador del sistema se ejecutarán todos los mandatos que contenga. Los *script's* de Linux no tienen un nombre especial como ocurre en

⁷ Debemos poner especial cuidado en no utilizar tildes ni siquiera en los comentarios. De no hacerlo solo conseguiremos provocar el error `can't execute binary file`.

MS-DOS (.BAT). La ejecución de un *script* puede hacerse de dos formas, la primera ejecutando una nueva capa de la *shell* y la segunda asignando permiso de ejecución al fichero. Por ejemplo, si disponemos de un fichero *script* llamado `proceso1`, podremos ejecutarlo de estas dos formas:

- Con el comando `sh` que ejecuta una nueva capa de la *shell* (o anteponiéndole al nombre del archivo los caracteres `./`):

```
# sh proceso1
```

o

```
# ./proceso1
```

Cuando ejecutamos un script de esta manera, se crea una “sub-shell” para la ejecución del mismo. Esta “sub-shell” hereda todas las variables de entorno de la *shell*, incluido el directorio actual, pero se perderán todos los cambios hechos en las variables de entorno durante la ejecución del script cuando éste finalice.

- Asignándole permiso de ejecución e invocando el archivo en la línea de comando (siempre que la ruta del archivo esté almacenada en la variable de entorno `PATH`) o anteponiéndole un `.` al nombre del archivo.

```
# chmod u+x proceso1
```

```
# proceso1
```

o

```
# .proceso1
```

En este caso el script se ejecuta sin que se lance una nueva capa de *shell*. Esto quiere decir que si modificamos el entorno, las modificaciones perdurarán cuando el script acabe su ejecución.

Podemos ejecutar un script y pedir a la *shell* que nos trace la ejecución para poder comprobar posibles errores y depurarlos. Esto se hace anteponiendo a la invocación del script **`bash -x`**.

Ejemplo: Supongamos el siguiente script:

```
count=1
until [ "$*" = "" ]
do
    echo "Arg $count : $1 "
    shift
    count=$((count+1))
done
```

Si lo ejecutamos nos daremos cuenta de que el contador no se incrementa correctamente. Podemos invocarlo de esta forma:

```
# bash -x listarg abc def
```

y obtendremos su traza:

```
+ count=1
+ [ abc def = ]
+ echo Arg 1 : abc
Arg 1 : abc
+ shift
+ count=1+1
+ [ def = ]
+ echo Arg 1+1 : def
Arg 1+1 : def
+ shift
+ count=1+1+1
+ [ = ]
```

... de esa forma podemos darnos cuenta de que no se incrementa el contador porque hemos olvidado utilizar `expr` y está concatenando la subcadena “+1” al valor anterior de `count` en cada iteración (es decir, ha tomado a `count` como una cadena)

Es recomendable que un usuario coloque todos sus ficheros *script* en un mismo directorio y que añada su ruta a la variable `PATH`. De esta forma estarán disponibles independientemente del directorio en el que se encuentre el usuario cuando los invoque. Este directorio suele denominarse también `/bin` y depender del directorio propiedad del usuario. Por ejemplo, podríamos hacer:

```
# mkdir $HOME/bin

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

La sentencia de inclusión del directorio `/bin` del usuario en la variable `PATH` se debe especificar en el fichero *profile* si queremos que el cambio tenga efecto cada vez que se arranque el sistema.

2.1. Comentarios.

Podemos utilizar líneas de comentario en nuestros ficheros *script* para facilitar la legibilidad de los mismos. Estos comentarios han de ir precedidos por el carácter almohadilla (#). Cuando la *shell* se encuentra con una línea que comienza con una almohadilla hace caso omiso de la misma (excepto si le sigue el carácter “!” en cuyo caso será considerada una directiva).

Ejemplo:

```
# MiPrimerScript
# Esto es un comentario en un script
echo Hola
```

Por ejemplo se suelen comenzar con los scripts con una directiva que indica la shell con la que se va a ejecutar el script, comúnmente:

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

2.2. Parámetros de un script.

Linux permite (al igual que UNIX) pasar parámetros a un fichero *script*. Para ello utiliza las variables \$0 hasta \$9. La variable \$0 recoge el nombre del *script* que se está ejecutando, la variable \$1 el primer argumento que se pasa al *script*, \$2 es el segundo y así sucesivamente⁸. El número de argumentos que recibe el *script*, sin contar su nombre, se almacena en la variable \$#. El conjunto de todos los parámetros pasados, sin el nombre del *script*, se encuentra almacenado en la variable \$*. La variable \$? contiene el código de retorno del último mandato ejecutado en el *script*. Si el mandato se ha ejecutado correctamente devolverá un 0. Esta variable permitirá comprobar si se ha ejecutado correctamente o no un mandato. El siguiente ejemplo muestra un fichero *script* que recibe parámetros y los muestra en pantalla.

Ejemplo:

```
# Fichero ejemplo1 que muestra el paso de parámetros.

clear

echo Se han pasado $# parámetros al script

echo Los parámetros pasados son:

echo $*

echo El primer parámetro es $1
```

⁸ A partir del parámetro 10 debemos delimitar el número de parámetro entre llaves. Ejemplo: \${10}, \${11}, etc.

En este fichero se ha utilizado el mandato `echo` que, recordemos, permite mostrar un mensaje en pantalla. Si el mensaje contiene una variable mostrará su contenido (siempre que antepongamos el carácter `$` al nombre de la variable). Si una línea comienza con el carácter almohadilla (`#`) el sistema lo toma como un comentario del mismo y no le presta la menor atención (salvo que sea una directiva).

Así al ejecutar:

```
# sh ejemplo1 uno dos tres cuatro
```

... obtendríamos en pantalla la siguiente salida:

```
Se han pasado 4 parámetros al script
```

```
Los parámetros pasados son:
```

```
uno dos tres cuatro
```

```
El primer parámetro es uno
```



```
shift [n]
```

La orden `shift` nos permite desplazar los argumentos `n` posiciones. Si no se especifica `n` se supone que se desplazarán una posición. Esto quiere decir que a partir de la ejecución de la sentencia el identificador `$1` recogerá el segundo valor introducido como parámetro en la línea de comandos, `$2` el tercer valor y así sucesivamente.

2.3. Más sobre el operador `expr`.

Ya hemos visto como utilizar el operador `expr` para evaluar expresiones matemáticas con enteros.

Ejemplo:

```
# Programa que pide dos valores por teclado
# y visualiza el producto de ambos

echo "Multiplicación de dos variables"
echo
echo -n "Introduce un valor: "
read A
echo -n "Introduce otro valor: "
read B
RESULTADO= `expr $A \* $B`
echo Resultado = $RESULTADO
```


Utilizar **echo** o **read** con el modificador **-n** provoca que no se produzca un salto de línea cuando el comando termine de visualizar su argumento.

El operador **expr** también se puede utilizar para comparar dos argumentos. Los argumentos pueden ser, en este caso, también palabras. Si el resultado de la comparación es cierto, el resultado devuelto es 1 y si es falso 0. Los distintos operadores relacionales utilizables son **=**, **!=** (distinto), **>**⁹, **>=**, **<**, **<=**.

Ejemplo:

```
# Programa que pide dos valores por teclado y comprueba
# si son iguales (devuelve un 1) o no (devuelve un 0)

echo "¿ Serán iguales ?"

echo

echo -n "Introduce un valor: " 10

read A

echo -n "Introduce otro valor: "

read B

echo `expr $A = $B`
```

Como operadores lógicos tenemos el carácter *pipe* ("**|**") que funciona como un OR y el "**&**" como AND. Ambos son caracteres especiales y deben ir precedidos por "****" o entre apostrofes.

2.4. Sentencias de control.

El sistema Linux suministra una serie de mandatos que se emplean en los *scripts* y proporcionan al usuario un completo lenguaje de programación. Para terminar la ejecución de un *script* se utiliza el mandato **exit** acompañado de un valor que quedará almacenado en la variable **\$?** para indicar el código de terminación del *script*.

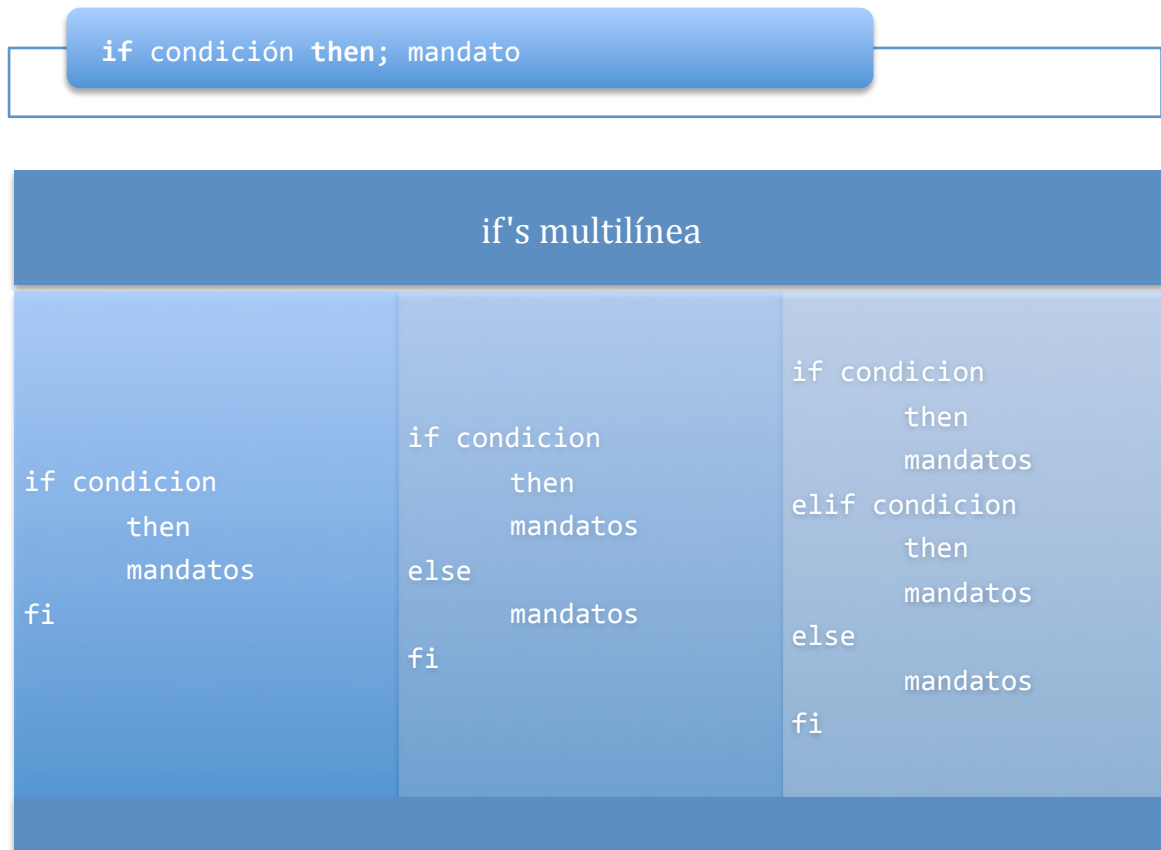
⁹ No olvidar que los símbolos "**>**" y "**<**" tienen significado especial para la *shell* y por tanto habrá que utilizarlos precedidos del *backslash* ("****") o entrecomillados (con apóstrofes).

¹⁰ Recordad que el parámetro **-n** de la sentencia **echo** hace que no se pase el cursor a la línea siguiente una vez visualizado el mensaje.

Ejemplo:

`exit 5`

Para comprobar condiciones en un *script* se utiliza el mandato **if**, cuya sintaxis posee las siguientes variaciones.



La condición indicada en el mandato puede ser una expresión lógica o un mandato que devuelva un valor. Si el resultado es cero, se considerará que la condición es verdadera y se ejecutarán los mandatos que haya después del **then**. La sentencia **if -then-else** también se puede escribir en una sola línea. Para ello es necesario separar cada uno de sus elementos por **(;)**. Por ejemplo:

`If condicion;then mandato;mandato;else mandato;mandato;fi`

Para indicar expresiones lógicas tales como comparaciones de números, de cadenas alfanuméricas, comprobaciones de ficheros, etc., se utiliza el mandato `test`¹¹.

`test valor operador valor`

Para comprobaciones de enteros se utilizan los siguientes operadores:

<code>test n1 -eq n2</code>	cierto si <code>n1</code> y <code>n2</code> son iguales
<code>test n1 -ne n2</code>	cierto si <code>n1</code> y <code>n2</code> son distintos
<code>test n1 -ge n2</code>	cierto si <code>n1</code> es mayor o igual que <code>n2</code>
<code>test n1 -gt n2</code>	cierto si <code>n1</code> es mayor que <code>n2</code>
<code>test n1 -le n2</code>	cierto si <code>n1</code> es menor o igual que <code>n2</code>
<code>test n1 -lt n2</code>	cierto si <code>n1</code> es menor que <code>n2</code>

Por ejemplo si se crea el *script* siguiente llamado *creadir*:

```
# Script que crea un directorio pasado como parámetro
# e informa si la acción ha tenido éxito

mkdir $1
if test $? -eq 0
then
    echo El subdirectorio $1 ha sido creado
fi
```

Y se ejecuta con la orden:

```
# sh creadir Prueba
```

Se obtendrá la salida:

```
El subdirectorio Prueba ha sido creado
```

También se puede utilizar la orden `test` con archivos:

`test opcion fichero`

¹¹ Existe una forma alternativa en UNIX y algunas distribuciones de Linux de especificar el operador de comprobación y es delimitando la expresión entre corchetes. De esta forma `test $# -eq 0` equivale a `[$# -eq 0]`.

En este caso las opciones son:

- f Verdadero si el archivo existe y es un archivo ordinario.
- d Verdadero si el archivo es un directorio
- r Verdadero si el archivo existe y es legible
- s Verdadero si el archivo existe y no está vacío.
- w Verdadero si el archivo existe y puede escribirse en él.
- x Verdadero si el archivo existe y es ejecutable.

Para comparar cadenas de caracteres, las constantes se deben delimitar con comillas simples y las variables irán con el signo \$. En este caso `test` se utiliza de esta forma:

```
test -z cadena          cierto si la longitud de la cadena es 0
test -n cadena          Idem. si no es 0
test cadena1 = cadena2  cierto si las cadenas son iguales
test cadena1 != cadena2 cierto si las cadenas son distintas
```

En general se puede utilizar el símbolo “!” como operador NOT. Por ejemplo es válida la expresión:

```
if test ! -r $1 ...
```

Ejemplo:

```
# Comprobar si estamos en nuestro directorio por defecto
midirectorio=`pwd`
if test $midirectorio = $HOME; then
    echo Estoy en mi directorio por defecto
fi
```

Las condiciones pueden agruparse con los operadores lógicos OR y AND (`-o`, `-a` en Linux). Por ejemplo:

```
if test -f fichero1 -a -f fichero2; then
    echo EXISTEN AMBOS FICHEROS
fi
```

También existe una sentencia de selección múltiple. Este mandato es una instrucción **case** que compara el contenido de una variable o expresión con una serie de patrones y, cuando coincide, ejecuta los mandatos que acompañan a dicho valor. Su sintaxis es:

```
case <exp> in
    patron1)
        mandatos
        ;;
    patrón2)
        mandatos
        ;;
    *)
        mandatos
        ;;
esac
```

Donde:

- **exp** Puede ser una variable o una expresión.
- **patrón** Representa el valor o conjunto de valores que puede tener **<exp>**. Se pueden indicar varios valores separándolos por el signo *pipe* ("**|**") que es tomado como un operador lógico OR. El final de los mandatos a ejecutar cuando se cumple un patrón se especifica con dos signos de punto y coma ("**;;**").
- ***** Es la opción por defecto. Se ejecutarán los mandatos indicados cuando **exp** no cumpla ningún patrón de los especificados.

Como ejemplo veamos un *script* que pagina o visualiza el contenido de un fichero según el modificador que se indique (**-p**, **-v**) respectivamente.

Ejemplo:

```
# Script que admite dos parametros: un nombre de fichero y un
# modificador que puede ser -p o -v y visualiza
# respectivamente paginado o no el fichero especificado

if test $# -ne 2; then
    echo Error de Sintaxis. Uso: $0 -modificador fichero
    echo Modificadores: -p (paginación) -v (visualiza)
    exit 1
fi
```

```
case $1 in
    -p|-P) cat $2 | pg;;
    -v|-V) cat $2;;
    *) echo Modificador no válido ...
esac
```

El mandato **for** nos permite crear un bucle en el que una variable toma todos los valores de una lista indicada. Por cada valor que haya en la lista se ejecutarán una vez todos los mandatos que se especifiquen entre las palabras **do** y **done**. Su formato es:



```
for variable in lista-de-valores
do
    mandatos
done
```

Donde:

- **variable** Es la variable que irá tomando los valores de la lista.
- **lista-de-valores** Contiene los valores separados por un carácter espacio o patrones como el asterisco (*) que indica todos los ficheros del directorio actual, o **\$*** que representa todos los argumentos pasados a un *script*. Se puede usar la instrucción **seq** para generar esos valores (ver página de manual).

Ejemplo:

```
# script que cuenta el numero de ficheros
# que contiene el directorio actual
cuenta=0
for fichero in *
do
    cuenta=`expr $cuenta + 1`
done
echo Hay $cuenta ficheros ...
```

```
# OTRA VERSIÓN: cuenta el numero de ficheros del directorio
# que le sirve como parámetro de entrada


if test $# -eq 1
then
    echo Hay
    ls -l $1|wc -l
    echo archivo\ /s en $1
else
    Formato correcto:  cuenta \


---



```

Linux también permite crear un bucle de mandatos que se ejecutarán mientras se cumpla una condición especificada. Su sintaxis es:



```
while condicion
do
    mandatos
done
```

Se pueden utilizar como condición las palabras reservadas **true** y **false**, creando así bucles infinitos que pueden romperse con **break**.

Ejemplo: El siguiente fichero crea cinco ficheros vacíos llamados *file1*, *file2*, ..., *file5*.

```
# Script que crea cinco ficheros vacíos "file<n>"
NUM=1
while test $NUM -le 5
do
    touch file$NUM
    NUM=`expr $NUM + 1`
done
```

Pero también se puede realizar bucles con mandatos que se ejecutarán hasta que la condición especificada sea verdadera. El formato de esta sentencia es:



Ejemplo: El mismo *script* anterior pero realizado con un `until` sería:

```

# Script que crea cinco ficheros vacíos "file<n>"

NUM=1
until test $NUM -gt 5
do
    touch file$NUM
    NUM=`expr $NUM + 1`
done
    
```

Normalmente, cuando se establece un bucle utilizando las órdenes que hemos visto, la ejecución de las órdenes englobadas por el lazo continúa hasta o mientras se de la condición especificada. La *shell* proporciona dos formas de alterar este funcionamiento: `break`¹² y `continue`. La primera provoca la salida del bucle que la engloba. Si se utiliza `break` con un argumento numérico, el programa salta ese número de bucles. Por ejemplo, en una serie de bucles anidados `break 2` saltaría del bucle que lo engloba inmediatamente y del bucle siguiente. En todo caso, se sigue ejecutando la orden siguiente al bucle del que se sale. La orden opuesta a `break` es `continue`. Controla la vuelta al principio del bucle más pequeño que la engloba. Si se le da un argumento, por ejemplo `continue 2`, el control va al inicio del segundo bucle que la engloba.

¹² `break` nos permite deshacer bucles infinitos creados con `true` y `false` utilizados como condiciones en los bucles condicionales.

Existe una sentencia específica para hacer selecciones de una lista. Se trata de `select`¹³, que posee el siguiente formato:

```
select variable in lista-de-valores
do
    mandatos
done
```

`select` visualiza los elementos indicados en la lista, numerados en el orden en que aparecen en ésta. A continuación se visualiza el *prompt* contenido en la variable de entorno `PS3` que nos indica que debemos elegir entre una de las opciones presentadas introduciendo su número. Estas opciones pueden ser controladas con una sentencia `case` dentro del bloque `do ... done`.

Ejemplo:

```
# Programa que ilustra el uso de select
```

```
PS3="Opcion:"
select i in Listado Quien Salir
do
    case $i in
        Listado) ls -l ;;
        Quien)   who ;;
        Salir)   exit 0;;
        *)       echo Ehhh?;;
    esac
done
```

Ejercicio: Sabiendo que el sistema devuelve la hora con el comando `date` y el modificador `+%H`, hacer un *script* que devuelva un saludo al usuario dependiendo de la hora (Buenos días, Buenas tardes o Buenas noches):

¹³ Sólo es válida en los shell Korn y bash.

```
# Obtener la hora del sistema
hora=`date +%H`
#comprobar hora
if test $hora -lt 12; then
    echo "Buenos días!, $LOGNAME"
else
    if test $hora -lt 20; then
        echo "Buenos tardes!, $LOGNAME"
    else
        echo "Buenos noches!, $LOGNAME"
    fi
fi
fi
```

Hacer un *script* que efectúe la copia segura del fichero que le sirve como primer parámetro de entrada en el fichero que le sirve como segundo parámetro de entrada. Observar todas las opciones posibles.

```
# Copia segura
case $# in
    2) if test ! -r $1 # No se puede leer el primer archivo
        then echo No tiene permiso de lectura para el primer archivo
        fi
        if test -f $2 # Existe el segundo archivo ?
        then
            if test -w $2 # Tiene permiso de escritura ?
            then
                echo "$2 existe, sobrescribir (S/N) ?"
                read resp
                case $resp in
                    s|S) cp $1 $2
                        echo Copia efectuada ...
                        ;;
                    *) exit
                        ;;
                esac
            else
                echo El archivo destino existe y
                echo no tiene permiso de escritura
            fi
        else # El archivo destino no existe, se crea
            cp $1 $2
            echo Copia efectuada ...
        fi
    ;;
    *) echo "Formato: safecopy <origen> <destino>"
    ;;

```

esac

2.5. Exportación de variables.

Cuando se crean variables de *shell* o se asignan valores a variables ya existentes éstas conservan su valor sólo dentro de la *shell* en la que han sido definidas. El valor desaparece o vuelve a fijarse al salir de la *shell*.

Por ejemplo, supongamos la variable **hoy** y el siguiente fichero llamado **quedia**:

```
# Mostrar el valor actual de la variable hoy
echo Hoy es $hoy
# Establecer el valor de la variable hoy
hoy=Sabado
# Mostrar el valor actual de la variable hoy
echo Hoy es $hoy
```

Si escribimos las siguientes órdenes en la línea de comando:

```
# chmod +x quedia
# hoy=Viernes
# quedia
# echo $hoy
```

Obtendremos la siguiente salida:

```
Hoy es
Hoy es Sabado
Viernes
```

Esto es así porque la variable **hoy** cuando se ejecuta el guión de *shell* no está definida aún (carece de valor) en esta capa de la *shell*. Después visualiza el valor que se le asigna dentro del *script* (Sábado) y por último aparece el valor de la variable en la *shell* de entrada (Viernes).

Para dar a una variable el mismo valor que tiene en la *shell* de entrada es necesario utilizar la orden **export** que exporta o transfiere las variables de una *shell* a *shells* posteriores. Si hacemos:

```
export hoy
```

Cualquier *shell* puesto en marcha desde el de entrada heredará el valor de la variable **hoy**. Así, con la siguiente secuencia:

```
# hoy=Viernes
# export hoy
# quedia
```

```
# echo $hoy
```

... obtendríamos:

```
Hoy es Viernes
```

```
Hoy es Sabado
```

```
Viernes
```

Se observa que la transferencia de valores de una *shell* a otra ocurre sólo en una dirección: desde la *shell* en ejecución hacia abajo.

2.6. Funciones

En la *shell* bash se pueden definir funciones. Una función en bash (denominada subrutina o procedimiento en otros lenguajes de programación) se podría definir como un script dentro de un script. Sirve para organizar un script en unidades lógicas de manera que sea más fácil mantenerlo y programarlo. En bash las funciones se pueden definir de la siguiente manera:

```
function nombre-de-funcion() {
    mandatos
}
```

Ejemplo:

```
#!/bin/bash
let A=100
let B=200

# Funcion suma()
# Suma las variables A y B
function suma(){
    let C=$A+$B
    echo "Suma: $C"
}
# Funcion resta()
# Resta las variables A y B
function resta(){
    let C=$A-$B
    echo "Resta: $C"
}

suma
resta
```
