Programación BASH

Este manual trata sobre la programación de shellscripts con el intérprete de comandos BASH (*Bourne Again Shell*). Pretende ser un guía para el usuario de Linux, que le permitirá comprender, ejecutar y empezar a programar en la Shell, haciendo referencia especialmente a BASH, ya que es el intérprete de comandos más utilizado en Linux e incluye un completo lenguaje para programación estructurada y gran variedad de funciones internas.

Tabla de contenido

Programación BASH

1 Introducción

El intérprete de comandos o *shell* es la interfaz principal entre el usuario y el sistema, permitiéndole a aquél interactuar con los recursos de éste. El usuario introduce sus órdenes, el intérprete las procesa y genera la salida correspondiente.

El intérprete de comandos puede ser usado en modo interactivo o no interactivo. En modo interactivo el usuario teclea comandos, los cuales se ejecutan y producen una salida, mientras que en modo no interactivo los comandos se leen desde un fichero. En este último caso, el fichero que contiene los comandos a ejecutar se conoce como *script* o *shellscript*. Sin embargo, para construir un *script* no solo se pueden emplear comandos del sistema operativo, sino también otros que se emplean en exclusiva en *scripts* y que no se pueden usar fuera de ellos.

El intérprete de comandos por defecto en los sistemas GNU/Linux es Bash (*Bourne Again Shell*) el cual provee funcionalidad para ejecutar comandos del sistema operativo y para crear nuevos comandos combinando los comandos con estructuras de programación y expresiones. Por lo tanto, un intérprete de comandos de GNU/Linux es tanto una interfaz de ejecución de comandos y utilidades, como un lenguaje de programación, que admite crear nuevos comandos, denominados guiones o *shellscripts*, utilizando combinaciones de comandos y estructuras lógicas de control, que cuentan con características similares a las del sistema.

1.1 Edición y ejecución de un script

Un script o guión interpretado por BASH es un fichero de texto normal que consta de una serie de comandos del sistema operativo, estructuras de control y expresiones que se ejecutan en secuencia. Como cualquier otro programa, el usuario debe tener permiso de ejecución en el fichero del script, y se lanza tecleando su ruta absoluta junto con sus opciones y argumentos. Asimismo, como veremos más adelante,, si el programa se encuentra en un directorio incluido en la variable de entorno PATH , sólo se necesita teclear el nombre, sin necesidad de especificar la ruta.

El proceso completo de edición y ejecución de un script es el siguiente:

1. Crear el script con un editor de texto plano como **vi** o **nano**.
2. Asignarle permisos de ejecución con el comando **chmod** para el usuario, grupo y/o todos los usuarios.
3. Ejecutarlo indicando su ruta completa o con el nombre del script si se encuentra en un directorio incluido en la variable de entorno PATH.
4. Depurar el script corrigiendo los errores detectados.

Existe una manera especial para ejecutar un script, precediéndolo por el signo punto, que se utiliza para exportar todas las variables del programa al entorno de ejecución del usuario.

Un script siempre comienza con la marca **#!** en la primera línea del fichero para especificar el camino completo y los parámetros del intérprete de comandos que ejecutará el programa. En nuestro caso vamos a utilizar el intérprete Bash, así que los scripts tiene que tener la siguiente línea como la primera de todas.

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

Como cualquier otro programa, un guión BASH puede requerir un cierto mantenimiento, que incluya modificaciones, actualizaciones o mejoras del código. Por lo tanto, el programador debe ser precavido y desarrollarlo teniendo en cuenta las recomendaciones de desarrollo típicas para cualquier programa.

Una práctica ordenada permite una verificación y comprensión más cómoda y rápida, para realizar las modificaciones de forma más segura y ayudar al usuario a ejecutar el programa correctamente. Pare ello, seguir las siguientes recomendaciones:

- ✔ El código debe ser fácilmente legible, incluyendo espacios y sangrías que separen claramente los bloques de código.
- ✔ Deben añadirse comentarios claros sobre el funcionamiento general del programa principal y de las funciones, que contengan: autor, descripción, modo de uso del programa, versión y fechas de modificaciones.
- ✔ Incluir comentarios para los bloques o comandos importantes, que requieran cierta aclaración.
- ✔ Agregar comentarios y ayudas sobre la ejecución del programa.
- ✔ Depurar el código para evitar errores, procesando correctamente los parámetros de ejecución.
- ✔ No desarrollar un código excesivamente enrevesado, ni complicado de leer, aunque esto haga ahorrar líneas de programa.
- ✔ Utilizar funciones y las estructuras de programación más adecuadas para evitar repetir código reiterativo.
- ✔ Los nombres de variables, funciones y programas deben ser descriptivos, pero no ha de confundirse con otras, ni con los comandos internos o externos; no deben ser ni muy largos ni muy cortos.

- ✔ Todos los nombres de funciones y de programas suelen escribirse en letras minúsculas, mientras que las variables acostumbran a definirse en mayúsculas.

1.2 Comentarios

En un script todo lo que vaya después del símbolo **#** y hasta el siguiente carácter de nueva línea se toma como comentario y no se ejecuta. Los scripts suelen encabezarse con comentarios que indican el nombre del archivo y lo que hace el script. También se suelen colocar comentarios de documentación en diferentes partes del script para mejorar la comprensión y facilitar el mantenimiento. Un caso especial es el uso de # en la primera línea para indicar el intérprete con que se ejecutará el script.

1.3 El comando interno echo

Cada vez que queramos mostrar en pantalla alguna información tenemos que utilizar el comando interno **echo**. Este comando simplemente escribe en la salida estándar (pantalla) los argumentos que recibe. Un comando **echo** siempre pone un salto de línea al final de la información a mostrar. Si queremos que lo omita lo ejecutaremos con la opción **-n**.

Si vamos a mostrar una línea con espacios en blanco la encerraremos entre comillas dobles. Por ejemplo

```
echo -n "Mostrando la información del usuario"
```

1.4 El primer script

Para crear un script basta con abrir un fichero con cualquier editor de texto (**vi**, **nano** o **gedit**) y, a continuación, escribir los comandos a ejecutar. Supongamos que vamos a escribir un script que emite un listado del directorio activo con la hora en la que se produce. Podría ser algo como lo siguiente:

```
#!/bin/bash

# Script:       listado_fichero
# Descripción:  Lista los archivos del directorio activo
#               en formato largo, posteriormente muestra la hora del
#               sistema.
echo "Listado de archivos del directorio"
ls -l
echo "Este listado se ha emitido en "
date
```

Este script (**listado_fichero**) tiene varias líneas. La primera le indica al sistema qué intérprete va a usar para ejecutar el fichero. A continuación vienen varias líneas de comentarios. Toda línea que comienza por **#**, excepto la primera, es un comentario y el intérprete la ignorará. Mediante comentarios podemos añadir notas aclaratorias sobre fragmentos de código.

A continuación viene el comando **echo** que simplemente muestra en pantalla el argumento. Después se ejecuta un listado en formato largo con **ls -l** y posteriormente se muestra la fecha del sistema con el comando **date**.

1.5 El comando exit

El comando **exit** puede ejecutarse en cualquier sitio del script para dar por terminada su ejecución. Suele usarse para detectar situaciones erróneas que hacen que el programa deba detenerse. Se le puede pasar como argumento un número para indicar el código de terminación, el cual suele ser 0 para indicar que se ha ejecutado correctamente o un número entre 1 y 255 para indicar alguna situación de error que puede ser manejada.

2 Variables

Una variable es un dato con un nombre que puede cambiar de valor durante la ejecución del script. Además, se puede emplear este dato cuando se necesite. Las variables son esencialmente cadenas de caracteres, aunque según el contexto, también pueden usarse con operadores numéricos y condicionales.

Una variable BASH se define o actualiza mediante el operador de asignación **=** y haremos referencia a su valor utilizando el símbolo **$** delante de su nombre. Suele usarse la convención de definir las variables en mayúsculas para distinguirlas fácilmente de los comando y funciones, ya que en GNU/Linux las mayúsculas y minúsculas se consideran caracteres distintos. Veamos un ejemplo

```
#!/bin/bash

# Script:       muestra_nombre
# Descripción:  Ejemplo de uso de variables

# Creamos una variable que se llama NOMBRE mediante el operador de
# asignación =
NOMBRE="Manuel López"

# Ahora hacemos referencia a la variable usando el operador $.
echo "El nombre es $NOMBRE"
```

A la hora de asignar valores a las variables podemos omitir las comillas si solamente tienen una palabra. Veamos los siguientes fragmentos de código donde se ilustra el uso de variables.

```
VAR1="Esto es una prueba ”      # Asignación de una variable
VAR2=35                         # Asignación de un valor numérico
echo $VAR1                      # En pantalla aparece el valor de VAR1
echo "VAR2=$VAR2"               # En pantalla aparece VAR2=35
```

Existen los siguientes tipos de variables:

- ✔ Las variables locales son definidas por el usuario y se utilizan únicamente dentro de un bloque de código, de una función determinada o de un script.
- ✔ Las variables de entorno son las que afectan al comportamiento del intérprete y al de la interfaz del usuario.
- ✔ Los parámetros de posición son los recibidos en la ejecución de cualquier programa o función, y hacen referencia a su orden ocupado en la línea de comandos.
- ✔ Las variables especiales son aquellas que tienen una sintaxis especial y que hacen referencia a

valores internos del proceso. Los parámetros de posición pueden incluirse en esta categoría.

2.1 Variables locales

Las variables locales son definidas dentro del script para operar en él. Fuera de dicho ámbito de operación, la variable no existe y por tanto no puede usarse.

```
ERR=2                           # Asigna 2 a la variable ERR.
echo ERR                        # Muestra la cadena ERR
echo $ERR                       # Muestra el valor de ERR
echo ${ERR}                     # Muestra el valor de ERR
echo "Error ${ERR}: salir"      # Muestra Error 2: salir
```

El formato **${Variable}** se utiliza en cadenas de caracteres donde se puede prestar a confusión o en procesos de sustitución de valores.

2.2 Variables de entorno

Las variables de entorno sirven para personalizar el entorno en el que se ejecuta la shell y para ejecutar correctamente las órdenes del shell. Cuando damos al shell una comando interactivo, GNU/Linux inicia un nuevo proceso para ejecutar el comando. El nuevo proceso hereda algunos atributos del entorno en el que se creó. Estos atributos se pasan de unos procesos a otros por medio de las variables de entorno, también denominadas variables globales

Al igual que cualquier otro proceso, la shell mantiene un conjunto de variables que informan sobre su propio contexto de operación. El usuario, o un script, puede actualizar y añadir variables exportando sus valores al entorno del intérprete (comando **export**), lo que afectará también a todos los procesos hijos generados por ella. El administrador puede definir variables de entorno estáticas para los usuarios del sistema (como, por ejemplo, en el caso de la variable **IFS**).

La siguiente tabla presenta las principales variables de entorno.

Variable	Descripción	Valor
SHELL	Camino del programa intérprete de comandos	La propia shell
PWD	Directorio de trabajo actual	Lo modifica la shell
OLDPWD	Directorio de trabajo anterior (equivale a ~ -)	Lo modifica la shell
PPID	Identificador del proceso padre	Lo modifica la shell
IFS	Separador de campos de entrada (debe ser de sólo lectura)	ESP, TAB, NL
HOME	Directorio personal del usuario	Lo define root
LOGNAME	Nombre de usuario que ejecuta la shell	Activado por login
PATH	Rutas de búsqueda de comandos	Según el sistema
LANG	Idioma para los mensajes	
EDITOR	Editor usado por defecto	
TERM	Tipo de terminal	
PS1 ... PS4	Puntos indicativos primario, secundario, selectivo y de errores	

FUNCNAME	Nombre de la función que se está ejecutando	Lo modifica la shell
LINENO	Nº de línea actual del guión (para depuración de código)	Lo modifica la shell

Debe hacerse una mención especial a la variable **PATH** , que se encarga de guardar la lista de directorios con ficheros ejecutables. Si no se especifica el camino exacto de un programa, el sistema busca en los directorios especificados por **PATH** , siguiendo el orden de izquierda a derecha. El carácter separador de directorios es dos puntos.

El administrador del sistema debe establecer los caminos por defecto para todos los usuarios del sistema y cada uno de éstos puede personalizar su propio entorno, añadiendo sus propios caminos de búsqueda (si no usa un intérprete restringido). Por ejemplo, se podría modificar la variable de entorno **PATH** como sigue

```
PATH=$PATH:/home/cdc/bin:/opt/oracle/bin
```

2.3 Parámetros de posición

En ocasiones, puede ser útil que nuestros scripts reciban algún tipo de argumento (un directorio sobre el que actuar o un tipo de archivo que buscar) al ser ejecutados. Para hacer referencia a estos argumentos, dentro de los scripts se emplean una serie de variables que siempre estarán disponibles, los parámetros posicionales **$1**, **$2**, **$3**, ..., siendo **$0** el nombre del propio programa. Reciben este nombre porque se les reconoce por su ubicación, es decir el primer argumento es **$1**, el segundo **$2** y así sucesivamente. A partir del décimo tenemos que usar la notación **${Número}**. El comando interno **shift** desplaza la lista de parámetros hacia la izquierda para procesar los parámetros más cómodamente. Por ejemplo imaginemos que creamos el script siguiente:

```
#!/bin/bash
# Ejemplo 4: script que recibe parámetros y los imprime
echo "El script $0"
echo "Recibe los argumentos $1 $2"
```

Si lo hemos guardado en un fichero llamado **ejemplo4** con el permiso de ejecución activado podemos ejecutarlo así:

```
usuario@hostname ~ ./ejemplo4 hola adiós
El script ./ejemplo4
Recibe los argumentos hola adiós
```

No se puede modificar el valor de las variables posicionales, sólo se pueden leer; si se intenta asignarles un valor se produce un error. Si el argumento posicional al que aludimos no se pasa como argumento, entonces tiene el valor nulo. Veamos otro ejemplo

```
#!/bin/bash
# Ejemplo 5: script que recibe parámetros y los imprime
echo "El script $0"
echo "Recibe los argumentos $1 $2 $3 $4"
```

Si lo hemos guardado en un fichero llamado **ejemplo5** con el permiso de ejecución activado podemos ejecutarlo así:

```
usuario@hostname ~ ./ejemplo5 hola adios
El script ./ejemplo5
Recibe los argumentos hola adios
```

2.4 Variables especiales

Las variables especiales son variables que informan sobre el estado del proceso. El intérprete las trata y modifica directamente, por lo tanto, son de sólo lectura. La siguiente tabla describe brevemente estas variables.

Variable	Descripción
`$$`	Identificador del proceso (PID).
`$*`	Cadena con el contenido completo de los parámetros recibidos por el programa.
`$@`	Como en el caso anterior, pero trata cada parámetro como un palabra diferente.
`$#`	Número de parámetros.
`$?`	Código de retorno del último comando (0=normal, >0=error).
`$!`	Último identificador de proceso ejecutado en segundo plano.
`$_`	Valor del último argumento del comando ejecutado previamente.

El siguiente ejemplo es un script que muestra la información de un usuario que se pasa como parámetro.

```
#!/bin/bash

# Script:       info_usuario
# Descripción:  Muestra toda la información de un usuario

# Comprobamos si se ha pasado el parámetro
if [ $# -ne 1 ]; then
     echo “Error. Falta indicar el usuario”
     echo “Sintaxis: $0 usuario”
     exit
fi

# Ahora mostramos la información del usuario con el comando id
id $1

# Comprobamos si ha habido algún error
if [ $? -ne 0 ]; then
     echo “Error al mostrar la información de $1”
fi
```

En uso común de la variable `$$` es el de asignar nombres para ficheros temporales que permiten el uso concurrente del programa, ya que al estar asociada al PID del proceso, éste valor no se repetirá nunca al ejecutar simultáneamente varias instancias del mismo programa.

La variable de entorno `$#` almacena el número total de argumentos o parámetros recibidos por el script sin contar al `$0`. El valor es de tipo cadena de caracteres, pero más adelante veremos como podemos convertir este valor a número para operar con él. La variables `$*` y `$@` contienen, ambas, los valores de todos los argumentos recibidos por el script.

Como ejemplo, el script **`ejemplo4`** lo vamos a modificar tal como muestra el **`ejemplo6`** para

que use estas variables para sacar los argumentos recibidos.

```
#!/bin/bash
# Ejemplo 5: script que recibe parámetros y los imprime
echo "El script $0 recibe $# argumentos:" $*
echo "El script $0 recibe $# argumentos:" $@
```

Al ejecutarlo obtenemos:

```
usuario@hostname ~ ./ejemplo5 hola adios
El script ./ejemplo5 recibe 2 argumentos: hola adios
El script ./ejemplo5 recibe 2 argumentos: hola adios
```

Aunque cuando no entrecomillamos **$*** o **$@** no existen diferencias entre usar uno y otro, cuando los encerramos entre comillas dobles, podemos cambiar el símbolo que usa **$*** para separar los argumentos cuando los muestra por pantalla. Por defecto, cuando se imprimen los argumentos, estos se separan por un espacio, pero podemos utilizar otro separador indicándolo en la variable de entorno **IFS** (*Internal Field Separator*). La variable **$@** siempre usa como separador un espacio y no se puede cambiar. Por ejemplo si hacemos el siguiente script:

```
#!/bin/bash
# Ejemplo 7: script que recibe parámetros y los imprime
IFS=','
echo "El script $0 recibe $# argumentos: $*"
echo "El script $0 recibe $# argumentos: $@"
```

Al ejecutarlo obtenemos:

```
usuario@hostname ~ ./ejemplo7 hola adios
El script ./recibe recibe 2 argumentos: hola,adios
El script ./recibe recibe 2 argumentos: hola adios
```

Vemos como, en el caso de **$***, los argumentos se separan por el símbolo coma (,) (separador establecido en la variable **IFS**), mientras que en el caso de **$@** se siguen separando por espacios (no le afecta el valor de la variable **IFS**).

En GNU/Linux los comandos terminan devolviendo un código numérico al que se llama código de terminación (*exit status*) que indica si el comando tuvo éxito o no. Aunque no es obligatorio que sea así, normalmente un código de terminación 0 significa que el comando terminó correctamente, y un código entre 1 y 255 corresponde a posibles códigos de error. En cualquier caso siempre conviene consultar la documentación del comando para interpretar mejor sus códigos de terminación.

Con la variable especial **$?** podemos saber el valor del código de terminación del último comando ejecutado por el shell. La variable **$?** debe de ser leída junto después de ejecutar el comando, siendo muy típico guardar su valor en una variable **CT=$?** para su posterior uso. Por ejemplo

```
cd ~/directorio_inexistente
echo -n “Si el valor es cero, se ha ejecutado bien el comando: “
echo $?
```

2.5 Asignar valor de variables desde teclado

BASH proporciona un nivel básico para programar scripts interactivos, soportando instrucciones para solicitar información al usuario. Para ello se emplea el comando **read** el cual lee de la entrada estándar y asigna los valores a las variables indicadas en la orden. Puede mostrarse un mensaje antes

de solicitar los datos. Su sintaxis es:

```
read [-p "Cadena"] [Var1 ...]
```

Por ejemplo,

```
# Las siguientes instrucciones son equivalentes y muestran
# un mensaje y piden un valor
echo –n "Dime tu nombre: "
read NOMBRE
#
read –p "Dime tu nombre: " NOMBRE

# Se muestra el valor introducido
echo “El nombre introducido es $NOMBRE”
```

2.6 Variables con tipo

Hasta ahora todas las variables que hemos usado son de tipo cadena de caracteres. Aunque en los primeros intérpretes de comandos las variables sólo podían contener cadenas de caracteres, después se introdujo la posibilidad de asignar atributos a las variables que indiquen, por ejemplo, que son de tipo entero o de sólo lectura. Para fijar los atributos de las variables, tenemos el comando interno **declare**, el cual tiene la siguiente sintaxis:

Sintaxis

```
declare [opciones] [nombre[=valor]]
```

Parámetros

nombre=valor
: Nombre y valor de la variable que se declara

Opciones

-a
: La variable es de tipo array

-f
: Mostrar el nombre e implementación de las funciones

-F
: Mostrar sólo el nombre de las funciones

-i
: La variable es de tipo entero

-r
: La variable es de sólo lectura

-x
: Exporta la variable. Equivalente a export variable=valor

Una peculiaridad de este comando es que para activar un atributo se precede la opción por un guión **-**, con lo que para desactivar un atributo decidieron preceder la opción por un **+**.

Si escribimos **declare** sin argumentos, nos muestra todas las variables existentes en el sistema. Si usamos la opción **-f**, nos muestra sólo los nombres de funciones (las veremos en la práctica siguiente) y su implementación, y si usamos la opción **-F**, nos muestra sólo los nombres de

las funciones existentes.

La opción **-i** declara la variable de tipo entero, lo cual permite que podamos realizar operaciones aritméticas con ella. Por ejemplo, si usamos variables normales para realizar operaciones aritméticas:

```
var1=5
var2=4
resultado=$var1*$var2
echo $resultado
```

Mostraría lo siguiente

```
5*4
```

Sin embargo, si ahora usamos variables de tipo entero:

```
declare -i var1=5
declare -i var2=4
declare -i resul tado
resultado=$var1*$var2
echo $resultado
```

Ahora muestra lo siguiente

```
20
```

Para que la operación aritmética tenga éxito, no es necesario que declaremos como enteras a **var1** y **var2**, basta con que recojamos el valor en una variable resultado declarada como entera. Es decir, podríamos hacer:

```
declare -i resultado
resultado=4*6
echo $resultado
```

Y en este caso mostraría

```
24
```

E incluso podemos operar con variables inexistentes:

```
$bash resultado=4*var_inexistente
$bash echo $resultado
```

El resultado sería

```
0
```

Podemos saber el tipo de una variable con la opción **-p**. Por ejemplo:

```
$bash declare -p resultado
```

Muestra lo siguiente

```
declare -i resultado="24"
```

La opción **-x** es equivalente a usar el comando **export** sobre la variable. Ambas son formas de exportar una variable.

La opción **-r** declara una variable como de sólo lectura, con lo que a partir de ese momento no podremos modificarla ni ejecutar **unset** sobre ella.

2.7 Matrices

Una matriz (o array) es un conjunto de valores identificados por el mismo nombre de variable, donde se accede a cada uno de sus elementos con un índice que denota el orden del elemento dentro de la matriz. Las matrices deben declararse mediante la cláusula interna **`declare`** , antes de ser utilizadas.

BASH soporta matrices de una única dimensión, conocidas también como vectores, con un único índice numérico, pero sin restricciones de tamaño ni de orden numérico o continuidad.

Los valores de las celdas pueden asignarse de manera individual o compuesta. Esta segunda fórmula permite asignar un conjunto de valores a varias de las celdas del vector. Si no se indica el índice en asignaciones compuestas, el valor para éste por defecto es 0 o sumando 1 al valor previamente usado.

Utilizar los caracteres especiales **`[@]`** o **`[*]`** como índice de la matriz, supone referirse a todos los valores en su conjunto, con un significado similar al expresado en el apartado de variables especiales.

El siguiente ejemplo describe la utilización de matrices.

```
# Declarar la matriz.
declare –a NUMEROS

# Asignación compuesta.
NUMEROS=( cero uno dos tres )

# Muestra el tercer elemento
echo ${NUMEROS[2]}

# Asignación simple. Se asigna al quinto elemento
NUMEROS[4]="cuatro"

# Muestra el quinto elemento
echo ${NUMEROS[4]}

# Asigna celdas 6, 7 y 9.
NUMEROS=( [6]=seis siete [9]=nueve )

# Muestra el octavo elemento
echo ${NUMEROS[7]}

# Muestra uno dos tres cuatro seis siete nueve
echo ${NUMEROS[*]}
```

2.8 Expansión de variables usando llaves

Realmente la forma que usamos para ver acceder al contenido de una variable, **`$variable`**, es una simplificación de la forma más general **`${variable}`**. La simplificación se puede usar siempre que no existan ambigüedades. En este apartado veremos cuando se producen las ambigüedades que nos obligan a usar la forma **`${variable}`**.

La forma `${variable}` se usa siempre que la variable va seguida por una letra, dígito o guión bajo; en caso contrario, podemos usar la forma simplificada `$variable`. Por ejemplo, si queremos escribir nuestro nombre (almacenado en la variable `nombre`) y nuestro apellido (almacenado en la variable `apellido`) separados por un guión podríamos pensar en hacer:

```
nombre=Fernando
apellido=Lopez
echo "$nombre_$apellido"
```

El resultado sería

```
Lopez
```

Pero esto produce una salida incorrecta porque Bash ha intentado buscar la variable `$nombre_` que al no existir la ha expandido por una cadena vacía. Para solucionarlo podemos usar las llaves:

```
echo "${nombre}_$apellido"
```

En este caso la salida sería

```
Fernando_Lopez
```

Otro lugar donde necesitamos usar llaves es cuando vamos a sacar el décimo parámetro posicional. Si usamos `$10`, Bash lo expandirá por `$1` seguido de un `0`, mientras que si usamos `${10}` Bash nos dará el décimo parámetro posicional

También necesitamos usar las llaves cuando queremos acceder a los elementos de una variable de tipo array. Por ejemplo:

```
matriz=(uno  dos  tres)
echo  $matriz
```

El resultado es

```
uno
```

Pero con

```
echo  $matriz[2]
```

El resultado es

```
uno[2]
```

Si usamos las llaves

```
echo  ${matriz[2]}
```

El resultado será

```
tres
```

3 Redirección

En GNU/Linux hay tres ficheros especiales que representan las funciones de entrada y salida de cada programa. Estos son:

- ✔ Entrada estándar: procede del teclado; abre el fichero descriptor 0 (`stdin`) para lectura.
- ✔ Salida estándar: se dirige a la pantalla; abre el fichero descriptor 1 (`stdout`) para escritura.

- ✔ Salida de error: se dirige a la pantalla; abre el fichero descriptor 2 (**stderr**) para escritura y control de mensajes de error.

El proceso de redirección permite hacer una copia de uno de estos ficheros especiales hacia o desde otro fichero normal. También pueden asignarse los descriptores de ficheros del 3 al 9 para abrir otros tantos archivos, tanto de entrada como de salida.

El fichero especial **/dev/null** se utiliza para descartar alguna redirección e ignorar sus datos, como veremos más adelante.

3.1 Redirección de entrada

La redirección de entrada sirve para abrir para lectura el archivo especificado usando un determinado número descriptor de fichero. Se usa la entrada estándar cuando el valor del descriptor es 0 o éste no se especifica.

El siguiente cuadro muestra el formato genérico de la redirección de entrada.

```
[N]<Fichero
```

La redirección de entrada se usa para indicar un fichero que contiene los datos que serán procesados por el programa, en vez de teclearlos directamente por teclado. Por ejemplo:

```
miproceso.sh < fichdatos
```

Sin embargo, conviene recordar que la mayoría de los comandos GNU/Linux tienen como parámetros nombres de ficheros y no es necesario redirigirlos.

3.2 Redirección de salida

De igual forma a los descrito en el apartado anterior, la redirección de salida se utiliza para abrir un fichero, asociado a un determinado número de descriptor, para operaciones de escritura.

Se reservan 2 ficheros especiales para el control de salida de un programa: la salida normal (con número de descriptor 1) y la salida de error (con el descriptor 2).

En la siguiente tabla se muestran los formatos genéricos para las redirecciones de salida.

Redirección	Descripción
`[N]> Fichero`	Abre el fichero de descriptor N para escritura. Por defecto se usa la salida estándar (N=1). Si el fichero existe, se borra; en caso contrario, se crea.
`[N]>| Fichero`	Como en el caso anterior, pero el fichero debe existir previamente.
`[N]>> Fichero`	Como en el primer caso, pero se abre el fichero para añadir datos al final, sin borrar su contenido.
`&> Fichero`	Escribe las salida normal y de error en el mismo fichero.

Suele ser habitual durante la ejecución de un script evitar que aparezcan los mensajes de error de los comandos y manejar nosotros mismos estos comandos con la variable especial **$?**. Para anular la salida de error de cualquier comando hay que redireccionar la salida de error al fichero **/dev/null**. Por ejemplo, si ejecutamos un listado de las particiones del disco y que no aparezca el error que pudiera haber escribiríamos la siguiente línea.

```
fdisk -l /dev/sda 2> /dev/null
```

3.3 Tuberías

La tubería es una herramienta que permite utilizar la salida normal de un programa como entrada de otro, por lo que suele usarse en el filtrado y depuración de la información, siendo una de las herramientas más potentes de la programación con intérpretes Unix.

Pueden combinarse más de una tubería en la misma línea de órdenes, usando el siguiente formato:

```
Comando1 | Comando2 ...
```

A continuación se muestran los comandos más utilizados con redirecciones y tuberías.

Comando	Descripción
`head`	Corta las primeras líneas de un fichero.
`tail`	Extrae las últimas líneas de un fichero.
`grep`	Muestra las líneas que contienen una determinada cadena de caracteres o cumplen un cierto patrón
`cut`	Corta columnas agrupadas por campos o caracteres.
`uniq`	Muestra o quita las líneas repetidas.
`sort`	Lista el contenido del fichero ordenado alfabética o numéricamente.
`wc`	Cuenta líneas, palabras y caracteres de ficheros.
`find`	Busca ficheros que cumplan ciertas condiciones y posibilita ejecutar operaciones con los archivos localizados
`sed`	Edita automáticamente un fichero.
`awk`	Procesa el fichero de entrada según las reglas de dicho lenguaje.

El siguiente ejemplo muestra una orden compuesta que ordena todos los ficheros con extensión `.txt`, elimina las líneas duplicadas y guarda los datos en el fichero `resultado.sal`.

```
cat *.txt | sort | uniq > resultado.sal
```

El comando `tee` es un filtro especial que recoge datos de la entrada estándar y lo redirige a la salida normal y a un fichero especificado, tanto en operaciones de escritura como de añadidura. Esta es una orden muy útil que suele usarse en procesos largos para observar y registrar la evolución de los resultados los cuales se van mostrando por pantalla también.

4 Entrecomillado y expansión

Todos los shells poseen un grupo de caracteres que en diferentes contextos tienen diferentes significados. Estos caracteres se les conoce como **metacaracteres**. Juegan un papel importante cuando el shell está analizando la línea de órdenes, antes de ejecutarla.

4.1 Entrecomillado

El entrecomillado es el procedimiento utilizado para modificar o eliminar el uso normal de dichos metacaracteres. Obsérvese el siguiente ejemplo.

```
# El “;” se usa normalmente para separar comandos.
echo Hola; echo que tal          # Muestra  → Hola
                                 # que tal

# Usando entrecomillado pierde su función normal.
echo "Hola; echo que tal"        Muestra → Hola; echo que tal
```

Los 3 tipos básicos de entrecomillado definidos en BASH son :

- ✔ Carácter de escape (`\Carácter`).- Mantiene el valor literal del carácter que lo precede; como último carácter de la línea, sirve para continuar la ejecución de una orden en la línea siguiente.
- ✔ Comillas simples (`'Cadena'`).- Siempre conserva el valor literal de cada uno de los caracteres de la cadena.
- ✔ Comillas dobles (`"Cadena"`).- Conserva el valor de literal de la cadena, excepto para los caracteres dólar (`$`), comilla simple (`'`) y de escape (`\$` , `\\` , `\'` , `\"` , ante el fin de línea y secuencia de escape del tipo ANSI-C).

Veamos unos ejemplos:

```
echo "Sólo con permiso \"root\"" # Muestra →  Sólo con permiso "root"
echo 'Sólo con permiso \"root\"' # Muestra →  Sólo con permiso \"root\"
```

4.2 Expansión

La línea de comandos se divide en una serie de elementos que representan cierto significado en la semántica del intérprete. La expansión es un procedimiento especial que se realiza sobre dichos elementos individuales. BASH dispone de 8 tipos de expansiones, que según su orden de procesado son

- ✔ Expansión de llaves.- Modifica la expresión para crear cadenas arbitrarias.
- ✔ Expansión de fichero.- Permite buscar patrones con comodines en los nombres de ficheros.
- ✔ Expansión de tilde.- Realiza sustituciones de directorios.
- ✔ Expansión de parámetro y variable.- Tratamiento general de variables y parámetros, incluyendo la sustitución de prefijos, sufijos, valores por defecto y otros operaciones con cadenas.
- ✔ Sustitución de comando.- Procesa el comando y devuelve su salida normal.
- ✔ Expansión aritmética.- Sustituye la expresión por su valor numérico.
- ✔ Sustitución de proceso.- Comunicación de procesos mediante tuberías con nombre de tipo cola (FIFO).
- ✔ División en palabras.- Separa la línea de comandos resultante en palabras usando los

caracteres de división incluidos en la variable `IFS` .

4.2.1 Expansión de llaves en nombres de fichero

La expansión de llaves es el preprocesado de la línea de comandos que se ejecuta en primer lugar y se procesan de izquierda a derecha. Se utiliza para generar cadenas arbitrarias de nombre de ficheros, los cuales pueden o no existir, por lo tanto puede modificarse el número de palabras que se obtienen tras ejecutar la expansión. El formato general es el siguiente:

Formato	Descripción
`[Pre]{C1,C2[,...]}[Suf]`	El resultado es una lista de palabras donde se le añade a cada una de las cadenas de las llaves, y separadas por comas, un prefijo y un sufijo opcionales.

Para ilustrarlo, véanse los siguientes ejemplo.

```
echo a{b,c,d}e              # Muestra → abe ace ade
mkdir $HOME/{bin,lib,doc}   # Se crean los directorios:
                            # $HOME/bin, $HOME/lib y $HOME/doc.
```

4.2.2 Expansión de nombre de archivo con metaracteres

Si algunas de las palabras obtenidas tras la división anterior contiene algún caracteres especial conocido como comodín (* , ? o [ ), ésta se trata como un patrón que se sustituye por la lista de nombres de ficheros que cumplen dicho patrón, ordena da alfabéticamente. El resto de caracteres del patrón se tratan normalmente. Los patrones válidos son:

Formato	Descripción
`*`	Equivale a cualquier cadena de caracteres, incluida una cadena nula.
`?`	Equivale a cualquier carácter único.
`[Lista]`	Equivale a cualquier carácter que aparezca en la lista. Pueden incluirse rangos de caracteres separados por guión (-). Si el primer carácter de la lista es ^ , se comparan los caracteres que no formen parte de ella.

La siguiente tabla describe algunos ejemplos.

```
# Listar los ficheros terminados en .rpm
ls *.rpm

# Listar los ficheros que empiecen por letra minúscula y tengan
# extensión .rpm
ls [a-z]*.rpm

# Listar los ficheros que empiezan por ".b", ".x" y ".X"
ls .[bxX]*

# Listar los ficheros cuya extensión tenga 2 caracteres
ls *.??
```

4.2.3 Expansión de tilde

Este tipo de expansión obtiene el valor de un directorio, tanto de las cuentas de usuarios, como

de la pila de directorios accedidos. Los formatos válidos de la expansión de tilde son:

Formato	Descripción
`~[Usuario]`	Directorio personal del usuario indicado. Si no se expresa nada `$HOME` .
`~+`	Directorio actual (`$PWD`).
`~-`	Directorio anterior (`$OLDPWD`).

Es recomendable definir un alias en el perfil de entrada del usuario para cambiar al directorio anterior, ya que la sintaxis del comando es algo engorrosa. Para ello, añadir la siguiente línea al fichero de configuración `~/.bashrc` .

```
alias cda='cd ~-'
```

Véase este pequeño script:

```
#!/bin/bash

# Script:       capacidad
# Descripción:  muestra la capacidad en KB de la cuenta del
#               usuario indicado
ls –ld ~$1
du –ks ~$1
```

4.2.4 Expansión de parámetro o de variable

Permite la sustitución del contenido de la variable siguiendo una amplia variedad de reglas. En un apartado anterior hemos visto ya la expansión de variables empleando las llaves. Aquí ampliamos este concepto. Los distintos formatos para la expansión de parámetros son:

Formato	Descripción
`${!Var}`	Se hace referencia a otra variable y se obtiene su valor (expansión indirecta).
`${Var_Par:-Val}`	Se devuelve el parámetro o variable; si éste es nulo, se obtiene el valor por defecto.
`${Var_Par:=Val}`	Si el parámetro o variable es nulo se le asigna el valor por defecto y se expande.
`${Var_Par:?Cadena}`	Se obtiene el parámetro o variable; si es nulo se manda un mensaje de error.
`${Var_Par:+Val}`	Se devuelve el valor alternativo si el parámetro o variable no es nulo.
`${Var_Par:Inicio}` `${Var_Par:Inicio:Longitud}`	Valor de subcadena del parámetro o variable, desde el punto inicial hasta el final o hasta la longitud indicada.
`${!Prefijo*}`	Devuelve los nombres de variables que empiezan por el prefijo.
`${#Var_Par}`	El tamaño en caracteres del parámetro o variable.
`${#Matriz[*]}`	El tamaño en número de elementos de una matriz.

`${Var_Par#Patrón}`	Se elimina del valor del parámetro o variable la mínima comparación del patrón, comenzando por el principio del parámetro o variable.
`${Var_Par##Patrón}`	Se elimina del valor del parámetro o variable la máxima comparación del patrón, comenzando por el principio del parámetro o variable.
`${Var_Par%Patrón}`	Se elimina del valor del parámetro o variable la mínima comparación del patrón, buscando por el final del parámetro o variable.
`${Var_Par%%Patrón}`	Se elimina del valor del parámetro o variable la máxima comparación del patrón, buscando por el final del parámetro o variable.
`${Var_Par/Patrón/Cadena}` `${Var_Par//Patrón/Cadena}`	La parte más grande de **`Patrón`** que coincide en **`Var`** es reemplazada por **`Cadena`**. La primera forma sólo reemplaza la primera ocurrencia, y la segunda forma reemplaza todas las ocurrencias. Si **`Patrón`** empieza por **`#`** debe producirse la coincidencia al principio de **`Var`**, y si empieza por **`%`** debe producirse la coincidencia al final de **`Var`**. Si **`Cadena`** es nula se borran las ocurrencias. En ningún caso **`Var`** se modifica, sólo se retorna su valor con modificaciones.

BASH proporciona unas potentes herramientas para el tratamiento de cadenas, sin embargo la sintaxis puede resultar engorrosa y requiere de experiencia para depurar el código. Por lo tanto, se recomienda crear guiones que resulten fáciles de comprender, documentando claramente las órdenes más complejas. Unos ejemplos para estudiar:

```
# Si el 1er parámetro es nulo, asigna el usuario que lo ejecuta.
USUARIO=${1:-`whoami`}

# Si no está definida la variable COLUMNS, el ancho es de 80.
ANCHO=${COLUMNS:-80}

# Si no existe el 1er parámetro, pone mensaje de error y sale.
: ${1:?"Error: $0 fichero"}

# Obtiene la extensión de un fichero (quita hasta el punto).
EXT=${FICHERO##*.}

# Quita la extensión “rpm” a la ruta del fichero.
RPM=${FICHRPM%.rpm}

# Cuenta el número de caracteres de la variable CLAVE.
CARACTERES=${#CLAVE}

# Renombra el fichero de enero a Febrero.
NUEVO=${ANTIGUO/enero/febrero}

# Añade nuevo elemento a la matriz (matriz[tamaño]=elemento).
```

```
matriz[${#matriz[*]}]="nuevo"
```

Veamos otro ejemplo para los operadores de búsqueda de patrón. Supongamos que tenemos la variable `RUTA` cuyo valor es `/usr/local/share/qemu/bios.core.bin` . Si ejecutamos los siguientes operadores de búsqueda de patrones, entonces los resultados serían los siguientes:

```
${RUTA##/*/}     # Resultado → bios.core.bin
${RUTA#/*/}      # Resultado → local/share/qemu/bios.core.bin
${RUTA%.*}       # Resultado → /usr/local/share/qemu/bios.core
${RUTA%%.*}      # Resultado → /usr/local/share/qemu/bios
```

4.2.5 Sustitución de comando

Esta expansión sustituye el comando ejecutado, incluyendo sus parámetros, por su salida normal, ofreciendo una gran potencia y flexibilidad de ejecución a un scrpit. Los formatos válido son:

Formato	Descripción
`$(Comando)`	Sustitución literal del comando y sus parámetros.
`` `Comando` ``	Sustitución de comandos permitiendo caracteres de escape.

Cuando la sustitución de comandos va en una cadena entre comillas dobles se evita que posteriormente se ejecute una expansión de ficheros. El siguiente script lista información sobre un usuario.

```
#!/bin/bash

# Script:       infous
# Descripción:  Lista la información de un usuario

TEMPORAL=`grep "^$1:" /etc/passwd 2>/dev/null`
USUARIO=`echo $TEMPORAL | cut -f1 -d:`
echo "Nombre de usuario: $USUARIO"
echo -n "Identifidor (UID): "
echo $TEMPORAL | cut -f3 -d:
echo -n "Nombre del grupo primario: "
GID=`echo $TEMPORAL | cut -f4 -d:`
grep ":$GID:" /etc/group | cut -f1 -d:
echo "Directorio personal: "
ls -ld `echo $TEMPORAL | cut -f6 -d:`
```

4.2.6 Expansión aritmética

La expansión aritmética calcula el valor de la expresión indicada y la sustituye por el resultado de la operación. El formato de esta expansión es:

Formato	Descripción
`$((Expresión)` `$[Expresión]`	Sustituye la expresión por su resultado.

Veamos los siguientes ejemplos:

```
SEGUNDOS=$((60*60))
echo "Número de segundos en una hora: $SEGUNDOS"
```

5 Expresiones

Una expresión es una combinación de operandos y operadores que al ejecutarse las operaciones indicadas ofrece un resultado final que puede almacenarse en una variable o utilizarse en otra expresión. El intérprete BASH permite utilizar una gran variedad de expresiones en el desarrollo de programas y en la línea de comandos. Las distintas expresiones soportadas por el intérprete pueden englobarse en las siguientes categorías:

- ✔ Expresiones aritméticas : las que dan como resultado un número entero o binario.
- ✔ Expresiones condicionales: utilizadas por comandos internos de BASH para su evaluar indicando si ésta es cierta o falsa.
- ✔ Expresiones de cadenas: aquellas que tratan cadenas de caracteres.

Las expresiones complejas cuentan con varios operandos y operadores, se evalúan de izquierda a derecha. Sin embargo, si una operación está encerrada entre paréntesis se considera de mayor prioridad y se ejecuta antes.

A modo de resumen, la siguiente tabla presenta los operadores utilizados en los distintos tipos de expresiones BASH.

Operadores aritméticos	`+ - * / % ++ --`
Operadores de comparación	`== != < <= > >= -eq –nt –lt -le –gt –ge`
Operadores lógicos	`! && ||`
Operadores binarios	`& | ^ << >>`
Operadores de asignación	`= *= /= %= += -= <<= >>= &= ^= |=`
Operadores de tipos de ficheros	`-e –b –c –d –f -h -L -p -S –t`
Operadores de permisos	`-r -w -x -g -u -k -O -G –N`
Operadores de fechas	`-nt –ot –et`
Operadores de cadenas	`-z -n`

5.1 Expresiones aritméticas

Las expresiones aritméticas representan operaciones números enteros o binarios (booleanos) evaluadas mediante el comando interno `let` . BASH no efectúa instrucciones algebraicas con números reales ni con complejos. La valoración de expresiones aritméticas enteras sigue las reglas:

- ✔ Se realiza con números enteros de longitud fija sin comprobación de desbordamiento, esto es, ignorando los valores que sobrepasen el máximo permitido.
- ✔ La división por 0 genera un error que puede ser procesado.
- ✔ La prioridad y asociatividad de los operadores sigue las reglas del lengua je C.

La siguiente tabla describe las operaciones aritméticas enteras y binarias agrupadas en orden de prioridad.

Operación	Descripción	Comentarios
`Var++` `Var--`	Post-incremento de variable Post-decremento de variable	La variable se incrementa o decrementa en 1 después de evaluarse su expresión.
`++Var` `--Var`	Pre-incremento de variable Pre-decremento de variable	La variable se incrementa o decrementa en 1 antes de evaluarse su expresión.
`+Expr` `-Expr`	Más unario Menos unario	Signo positivo o negativo de la expresión (por defecto se considera positivo).
`! Expr` `~ Expr`	Negación lógica Negación binaria	Negación de la expresión lógica o negación bit a bit.
`E1 ** E2`	Exponenciación	E1 elevado a E2 ($E1^{E2}$).

`E1 * E2`
`E1 / E2`
`E1 % E2`
	Multiplicación
	División
	Resto
		Operaciones de multiplicación y división entre números enteros.

`E1 + E2` `E1 – E2`	Suma Resta	Suma y resta de enteros.
`Expr << N` `Expr >> N`	Desplazamiento binario a la izquierda Desplazamiento binario a la derecha	Desplazamiento de los bits un número indicado de veces

`E1 < E2`
`E1 <= E2`
`E1 > E2`
`E1 >= E2`
	Comparaciones (menor, menor o igual, mayor, mayor o igual).	

`E1 = E2` `E1 != E2`	Igualdad Desigualdad	
`E1 & E2`	Operación binaria Y	
`E1 ^ E2`	Operación binaria O Exclusivo.	
`E1 | E2`	Operación binaria O	
`E1 && E2`	Operación lógica Y	
`E1 || E2`	Operación lógica O	
`E1 ? E2 : E3`	Evaluación lógica	Si E1=cierto, se devuelve E2; si no, E3.
`E1 = E2` `E1 Op= E2`	Asignación normal y con preoperación (operadores válidos: `*=` , `/=` , `%=` , `+=` , `-=` , `<<=` , `>>=` , `&=` , `^=` , `|=`)	Asigna el valor de E2 a E1. Si se especifica un operador, primero se realiza la operación entre las 2 expresiones y se asigna el resultado (E1 = E1 Op E2).
`E1, E2`	Operaciones independientes.	Se ejecutan en orden.

A continuación se muestran algunos ejemplos en el uso de las expresiones aritméticas.

```
let a=5                                 # Se asigna 5 a la variable a
let b=$a+3*9                            # Se asigna a b a+(3*9)=32.
echo "a=$a, b=$b"                       # Se muestra a=5, b=32
let c=$b/($a+3)                         # Se asigna a c b/(a+3)=4
```

```
let a+=c--                              # a=a+c=9, c=c+1=5.
echo "a=$a, c=$c"                       # Se muestra a=9, c=5
let CTE=$b/$c, RESTO=$b%$c              # CTE=b/c, RESTO=resto(b/c)
echo "Cociente=$CTE, Resto=$RESTO"      # Se muestra Cociente=6, Resto=2
```

Los números enteros pueden expresarse en bases numéricas distintas a la decimal (base por defecto). El siguiente ejemplo muestra los formatos de cambio de base.

```
let N=59                    # Base decimal
let N=034                   # Base octal, empieza por 0
let N=0x34AF                # Base hexadecimal, empieza por 0x
```

5.2 Expresiones condicionales

Las expresiones condicionales son evaluadas por los comandos internos del tipo **`test`** , dando como resultado un valor de cierto o de falso. Suelen emplearse en operaciones condicionales y bucles, aunque también pueden ser empleadas en órdenes compuestas.

Existen varios tipos de expresiones condicionales según el tipo de parámetros utilizado o su modo de operación:

- ✔ Expresiones con ficheros, que comparan la existencia , el tipo, los permisos o la fecha de ficheros o directorios.
- ✔ Expresiones comparativas numéricas, que evalúan la relación de orden numérico entre los parámetros.
- ✔ Expresiones comparativas de cadenas, que establecen la relación de orden alfabético entre los parámetros.

Todas las expresiones condicionales pueden usar el modificador de negación (**`!Expr`**) para indicar la operación inversa. Asimismo, pueden combinarse varias expresiones en una expresión compleja usando los operadores lógicos Y (**`Expr1 && Expr2`**) y O (**`Expr1 || Expr2`**), aunque podemos evaluar varias condiciones dentro de una sola expresión con el operador **`-a`** (Y) y **`-o`** (O)

5.2.1 Expresiones de ficheros

Son expresiones condicionales que devuelven el valor de cierto si se cumple la condición especificada; en caso contrario da un valor de falso. Hay una gran variedad de expresiones relacionadas con ficheros y pueden agruparse en operaciones de tipos, de permisos y de comparación de fechas.

Hay bastantes operadores de tipos de ficheros. La siguiente tabla lista los formatos de estas expresiones.

Operador	Condición (cierto si...)
`-e Fich`	El fichero (de cualquier tipo) existe
`-s Fich`	El fichero no está vacío.
`-f Fich`	Es un fichero normal.
`-d Fich`	Es un directorio.

`-b Fich`	Es un dispositivo de bloques.
`-c Fich`	Es un dispositivo de caracteres.
`-p Fich`	Es una tubería.
`-h Fich` `-L Fich`	Es un enlace simbólico.
`-S Fich`	Es una "socket" de comunicaciones.
`-t Desc`	El descriptor está asociado a una terminal.
`F1 -ef F2`	Los 2 ficheros son enlaces hacia el mismo archivo.

Las condiciones sobre permisos establecen si el usuario que realiza la comprobación puede ejecutar o no la operación deseada sobre un determinado fichero. La tabla describe estas condiciones.

Operador	Condición (cierto si...)
`-r Fich`	Tiene permiso de lectura.
`-w Fich`	Tiene permiso de escritura.
`-x Fich`	Tiene permiso de ejecución/acceso.
`-u Fich`	Tiene el permiso SUID.
`-g Fich`	Tiene el permiso SGID.
`-t Fich`	Tiene permiso de directorio compartido o fichero en caché.
`-O Fich`	Es el propietario del archivo.
`-G Fich`	El usuario pertenece al grupo con el GID del fichero.

Las operaciones sobre fechas, descritas en la siguiente tabla, establecen comparaciones entre las correspondientes a 2 ficheros.

Operador	Condición (cierto si...)
`-N Fich`	El fichero ha sido modificado desde al última lectura.
`F1 -nt F2`	El fichero F1 es más nuevo que el F2.
`F1 -ot F2`	El fichero F1 es más antiguo que el F2.

5.2.2 Expresiones comparativas numéricas

Aunque los operadores de comparación para números ya se han comentado en el apartado anterior, la siguiente tabla describe los formatos para este tipo de expresiones.

Operador	Condición (cierto si...)

`N1 -eq N2`
`N1 -ne N2`
`N1 -lt N2`
`N1 -gt N2`
	Se cumple la condición de comparación numérica (respectivamente igual, distinto, menor y mayor).

5.2.3 Expresiones comparativas de cadenas

También pueden realizarse comparaciones entre cadenas de caracteres. La tabla indica el formato de las expresiones.

Operador	Condición (cierto si...)
`Cad1 = Cad2` `Cad1 != Cad2`	Se cumple la condición de comparación de cadenas (respectivamente igual y distinto).
`[-n] Cad`	La cadena no está vacío (su longitud no es 0).
`-z Cad`	La longitud de la cadena es 0.

6 Programación estructurada

Las estructuras de programación se utilizan para generar un código más legible y fiable. Son válidas para englobar bloques de código que cumplen un cometido común, para realizar comparaciones, selecciones, bucles repetitivos y llamadas a subprogramas.

6.1 Lista de comandos

Los comandos BASH pueden agruparse en bloques de código que mantienen un mismo ámbito de ejecución. La siguiente tabla describe brevemente los aspectos fundamentales de cada lista de órdenes.

Lista de comandos	Descripción
`Comando &`	Ejecuta el comando en 2º plano. El proceso tendrá menor prioridad y no debe ser interactivo..
`Com1 | Com2`	Tubería. Redirige la salida de la primera orden a la entrada de la segundo. Cada comando es un proceso separado.
`Com1; Com2`	Ejecuta varios comandos en la misma línea de código.
`Com1 && Com2`	Ejecuta la 2a orden si la 1a lo hace con éxito (su estado de salida es 0).
`Com1 || Com2`	Ejecuta la 2a orden si falla la ejecución de la anterior (su código de salida no es 0).
`(Lista)`	Ejecuta la lista de órdenes en un subproceso con un entorno común.
`{ Lista; }`	Bloque de código ejecutado en el propio intérprete.
`Linea1 \` `Linea2`	Posibilita escribir listas de órdenes en más de una línea de pantalla. Se utiliza para ejecutar comandos largos.

6.2 Estructuras condicionales y selectivas

Ambas estructuras de programación se utilizan para escoger un bloque de código que debe ser ejecutado tras evaluar una determinada condición. En la estructura condicional se opta entre 2 posibilidades, mientras que en la selectiva pueden existir un número variable de opciones.

6.2.1 Estructuras condicionales

La estructura condicional sirve para comprobar si se ejecuta un bloque de código cuando se cumple una cierta condición. Pueden anidarse varias estructuras dentro del mismo bloques de código, pero siempre existe una única palabra **fi** para cada bloque condicional. La tabla muestra cómo se representan ambas estructuras en BASH.

Estructura	Descripción

`if Expresión; then`
`  Bloque;`
`fi`
	Estructura condicional simple.- Se ejecuta la lista de órdenes si se cumple la expresión. En caso contrario, este código se ignora.

`if Expresión1; then`
`  Bloque1;`
`[ elif Expresión2; then`
`  Bloque2;`
`  …`
`]`
`[else BloqueN; ]`
`fi`
	Estructura condicional compleja.- El formato completo condicional permite anidar varias órdenes, además de poder ejecutar distintos bloques de código, tanto si la condición de la expresión es cierta, como si es falsa.

Aunque el formato de codificación permite incluir toda la estructura en una línea, cuando ésta es compleja se debe escribir en varias, para mejorar la comprensión del programa. En caso de teclear la orden compleja en una sola línea debe tenerse en cuenta que el carácter separador (**;**) debe colocarse antes de las palabras reservadas: **then** , **else** , **elif** y **fi** .

Hay que resaltar la versatilidad para teclear el código de la estructura condicional, ya que la palabra reservada **then** puede ir en la misma línea que la palabra **if** , en la línea siguiente sola o conjuntamente con la primera orden del bloque de código, lo que puede aplicarse también a la palabra **else**).

Puede utilizarse cualquier expresión condicional para evaluar la condición, incluyendo el código de salida de un comando o una condición evaluada por el comando interna **test**. Este último caso se expresa colocando la condición entre corchetes (**[Condición]**) .

Veamos algunos ejemplos en los que combinamos el comando interno **test** con los corchetes.

```
#!/bin/bash

# Script:       prueba_if
# Descripción:  Diferentes ejemplos del uso de la estructura
#               condicional if
#
# La condición más simple escrita en una línea:
# si RESULT>0; entonces visualiza un mensaje en
RESULT=1
if [ $RESULT -gt 0 ]; then echo "Es mayor que cero"; fi
#
# La condición doble:
# si la variable HOSTNAME es nula o vale “(none)”
# Primera forma. Usando operador condicional || para unir
# dos expresiones lógicas y empleando corchetes en lugar del
```

```
# comando test
if [ -z "$HOSTNAME" ] || [ "$HOSTNAME" = "(none)" ]; then
        HOSTNAME=localhost
fi

# Segunda forma. Usando operador condicional || para unir
# dos expresiones lógicas y el comando test
if test -z "$HOSTNAME" || test "$HOSTNAME" = "(none)" ; then
        HOSTNAME=localhost
fi

# Tercera forma. Usando operador condicional -o para unir
# dos condiciones en una sola expresión lógica
# y empleando corchetes en lugar del comando test
if [ -z "$HOSTNAME" -o  "$HOSTNAME" = "(none)" ] ; then
        HOSTNAME=localhost
fi

# Cuarta forma. Usando operador condicional -o para unir
# dos condiciones en una sola expresión lógica
# y empleando el comando test
if test  -z "$HOST" -o  "$HOST" = "(none)" ; then
        HOSTNAME=localhost
fi

#
# Combinación de los 2 tipos de condiciones:
# si existe el fichero /etc/hosts y tiene permiso de escritura
# se establece la variable HOSTS a 1
if [ -f /etc/hosts ] && [ -w /etc/hosts ] ; then
     HOSTS=1
fi
```

Veamos ahora algunos ejemplos más complejos en el mismo script

```
# Estructura condicional compleja con 2 bloques:
# si existe el fichero especificado, se muestra; si no, se da
# el valor “no” a la variable NETWORKING
if [ -f /etc/network/interfaces ]; then
     more /etc/network/interfaces
else
     NETWORKING=no
fi

# Estructura anidada:
# si ~/documentos es un directorio y $RC es 0, lista su contenido,
# si no, si /etc/hosts no está vacío lo muestra, en caso
# contrario; muestra el archivo /etc/hostname.
if [ "$RC" = "0" -a -d ~/documentos ]; then
     ls -l ~/documentos
elif [ -z /etc/hosts ]; then
     cat /etc/hosts
```

```
else
      cat /etc/hostname
fi
```

6.2.2 Estructura selectiva

La estructura selectiva evalúa la condición de control y, dependiendo del resultado, ejecuta un bloque de código determinado. La siguiente tabla muestra el formato genérico de esta estructura.

Estructura	Descripción

`case Variable in`
`  [(]Patrón1)`
`  Bloque1`
`  ;;`
`  [(]Patrón2)`
`  Bloque2`
`  ;;`
`  [(]Patrón_N)`
`  Bloque_N`
`  ;;`
 	
`...`
`esac`
	Estructura selectiva múltiple: si la variable cumple un determinado patrón, se ejecuta el bloque de código correspondiente. Cada bloque de código acaba con " `;;` ".
	La comprobación de patrones se realiza en secuencia.

Las posibles opciones soportadas por la estructura selectiva múltiple se expresan mediante patrones, donde puede aparecer caracteres comodines, evaluándose como una expansión de ficheros, por lo tanto el patrón para representar la opción por defecto es el asterisco (`*`).

Dentro de una misma opción pueden aparecer varios patrones separados por la barra vertical (`|`), como en una expresión lógica O.

Si la expresión que se comprueba cumple varios patrones de la lista, sólo se ejecuta el bloque de código correspondiente al primero de ellos, ya que la evaluación de la estructura se realiza en secuencia.

Un ejemplo habitual de esta estructura lo encontramos en los scripts de inicio de servicio que hay en **`/etc/init.d`**. Estos scripts reciben un parámetro para iniciar el servicio (**`start`**), reiniciarlo (**`restart`**), pararlo (**`stop`**), volver cargar a cargar la configuración (**`force-reload`**) o mostrar el estado (**`status`**). Para controlar el valor del parámetro, y las acciones a realizar en cada caso, es habitual utilizar una estructura **`case`**. Vamos a ver un ejemplo.

```
case "$1" in
      start)
            echo “Se ha requerido el inicio del servicio”
            ;;

      stop)
            echo “Se ha requerido la parada del servicio”
            ;;

      restart|force-reload)
            echo “Se ha requerido reiniciar o cargar la configuración
```

```
            ;;

      status)
            echo "Se ha requerido mostrar el estado del servicio"
            ;;

      *)
            # Error de sintaxis
            echo "No se ha pasado el parámetro correcto"
            echo "Sintaxis: $0 {start|stop|restart|force-reload|status}"
            ;;
esac
```

Vemos como cada la estructura **case** evalúa el valor del parámetro pasado al script. Para cada opción hemos omitido el primer paréntesis. Si el valor del parámetro coincide con alguna de las opciones se ejecuta el bloque de código asociado a la opción. La última opción ***** representa un patrón que coincide con cualquier cadena de caracteres y el bloque de código asociado se ejecuta con cualquier valor del parámetro **$1** que no coincida con alguno de los anteriores. Cada bloque de código acaba en **;;**. Este ejemplo está muy simplificado para ayudar a su compresión, pero en una situación real los bloques de código suelen incluir más comandos.

6.3 Bucles

Un bucle es la ejecución repetida de un bloque de código varias veces. El número de veces que se ejecuta el bloque de código dependerá del tipo de bucle, pero todos ellos se basan en el uso de una expresión lógica (una condición) para limitar este número de ejecuciones, es decir, el bloque de código continúa ejecutándose de forma repetitiva cuando la condición del bucle se cumple. La forma de especificar esta condición en el bucle determina el tipo.

Hay dos comandos especiales que pueden utilizarse dentro del bloque de código para romper la secuencia normal de ejecución de comandos en determinadas situaciones. Ambos se emplean dentro del bloque de código del bucle para alterar la ejecución del mismo.

La siguiente tabla describe estos dos comandos especiales.

Comando	Descripción
break	Ruptura inmediata de un bucle.- Se interrumpe la ejecución del bloque de código y la ejecución del script continua en la línea inmediata siguiente al bucle.
continue	Salto a la condición del bucle.- Se interrumpe la ejecución del bloque de código del bucle para volver a evaluar la condición.

Los siguientes puntos describen los distintos bucles que pueden usarse tanto en un script como en la línea de comandos de BASH.

6.3.1 Bucle for

El bucle **for** ejecuta el bloque de código que contiene un número de veces determinado de antemano. El bloque de código se ejecuta para cada valor de una lista de valores que puede ser un conjunto de ficheros cuyo nombre se ajusta a un patrón, un conjunto de cadenas de texto en una lista o cada parámetro posicional, siendo este último por defecto si no se indica ninguna lista de valores.

Por otra parte, BASH soporta otro tipo de bucle iterativo genérico similar al usado en el lenguaje de programación C, usando expresiones aritméticas. El modo de operación es el siguiente:

- ✔ Se evalúa la primera expresión aritmética antes de ejecutar el bucle para dar un valor inicial al índice.
- ✔ Se realiza una comprobación de la segunda expresión aritmética, si ésta es falsa se ejecutan las iteraciones del bucle. Siempre debe existir una condición de salida para evitar que el bucle sea infinito.
- ✔ Como última instrucción del bloque se ejecuta la tercera expresión aritmética, que debe modificar el valor del índice, y se vuelve al paso anterior.

La siguiente tabla describe la sintaxis del bucle **`for`** en las dos formas posibles.

Bucle	Descripción

`for Var [in Lista]; do`
`  Bloque`
`done`
	Bucle iterativo.- Se ejecuta el bloque de comandos del bucle sustituyendo la variable de evaluación por cada una de las palabras incluidas en la lista. Si se omite la lista se toma por defecto la lista de parámetros posicionales.

`for ((Exp1; Exp2; Exp3)) do`
`  Bloque`
`done`
	Bucle iterativo de estilo C.- Se evalúa **`Exp1`** , mientras **`Exp2`** sea cierta se ejecutan en cada iteración del bucle el bloque de comandos y **`Exp3`** (las 3 expresiones deben ser aritméticas).

Veamos los siguientes ejemplos. En este primer script vamos a mostrar por pantalla la el tipo todos los archivos de un directorio pasado como parámetro. Para obtener el tipo de fichero emplearemos el comando **`file`**.

```
#!/bin/bash

# Script:       tipo_fichero
# Descripción:  Muestra el tipo de los ficheros de un directorio
#
# Comprobamos si se ha pasado el parámetro
if [ $# -ne 1 ]; then
     echo “Error. Falta indicar el directorio”
     echo “Sintaxis: $0 directorio”
     exit 1
fi

# Comprobamos si el directorio existen
if [ ! -d $HOME/$1 ]; then
        echo "Error. $1 no es un directorio de $HOME"
        exit 2
fi

# Ahora montamos el bucle for
for ARCHIVO in  ~/$1/* ; do
        echo “Información del archivo $ARCHIVO:”
```

```
        file "$ARCHIVO"
done
```

Vemos que la lista del bucle **for** es **~/$1/*** que son todos los archivos del directorio pasado como parámetro y que se encuentra en el directorio personal. La variable **ARCHIVO** toma el valor de cada archivo de la lista en cada iteración del bucle.

El siguiente ejemplo muestra cuanto ocupan en disco los directorios del directorio personal del usuario que se pasan como parámetros.

```
#!/bin/bash

# Script:       ocupacion_directorio
# Descripción:  Muestra lo que ocupan los directorios
#               pasados como parámetros

clear

for CARPETA; do
        if [ -d $HOME/$CARPETA ]; then
                echo -n "El directorio $CARPETA ocupa: "
                du -sh "$HOME/$CARPETA"
        fi
done
```

Como podemos apreciar, en el bucle **for** no hemos puesto ninguna lista, ya que toma por defecto la lista de parámetros posicionales.

El siguiente ejemplo utiliza un bucle **for** estilo lenguaje C el cual no suele ser muy utilizado.

```
#!/bin/bash

# Script:       for_C
# Descripción:  Muestra por pantalla números del 0 al 9

clear

for ((i=0; $i<10; i++)); do
     echo $i
done
```

Aquí la variable índice o contador es i. La primera expresión inicializa i a cero y la tercera expresión la incrementa en uno. La segunda expresión comprueba que la variable sea menor que 10. La salida del script muestra los números del 0 al 9 por pantalla.

La secuencia de números anterior la podemos obtener con el comando **seq**. En este caso el bucle **for** quedaría de la siguiente manera.

```
for i in `seq 0 9`; do
     echo $i
done
```

6.3.2 Bucle while y until

En este caso los bucles condicionales tienen una expresión lógica que determinan si se ejecuta

o no el bloque de código asociado. En cada iteración del bucle evalúan la expresión lógica y dependiendo del resultado se vuelve a realizar otra iteración o se sale del bucle para continuar la ejecución del script en la instrucción siguiente al bucle. La siguiente tabla describe los formatos para los dos tipos de bucles condicionales soportados por el intérprete BASH.

Bucle	Descripción

`while Expresión; do`
`  Bloque`
`done`
	Bucle iterativo "mientras".- Se ejecuta el bloque de órdenes mientras que la condición sea cierta. La expresión de evaluación debe ser modificada en algún momento del bucle para poder salir.

`until Expresión; do`
`  Bloque`
`done`
	Bucle iterativo "hasta".- Se ejecuta el bloque de código hasta que la condición sea cierta. La expresión de evaluación debe ser modificada en algún momento del bucle para poder salir.

Ambos bucles realizan comparaciones inversas, es decir, el bloque de código del bucle while se ejecuta si la condición es cierta mientras que el bucle until ejecuta el bloque de código si la condición es falsa. Pueden usarse indistintamente, aunque se recomienda usar aquél que necesite una condición más sencilla o legible, intentando no crear expresiones negativas. Véase el siguiente ejemplo:

```
# Mientras haya parámetros que procesar, ...
while [ $# != 0 ] ; do
      echo "$1"
      shift
done
```

En este ejemplo estamos recorriendo la lista de parámetros posicionales para mostrarlos en pantalla. Nótese el uso del comando **shift** para desplazar los parámetros posicionales.

Veamos ahora este ejemplo. El siguiente bucle **while** va a mostrar por pantalla todos los directorios que hay en la variable de entorno **$PATH**.

```
# Script que muestra los directorios de PATH
echo "La variable path es $PATH"
path=$PATH
while [ "$path" != "" ]; do     # La condición podría ser -n "$path"

        # Visualiza el primer directorio de la cadena restante
        echo ${path%%:*}

        # Se suprime el directorio visualizado
        if [ ${path#*:} = $path ]; then
                path=
        else
                path=${path#*:}
        fi
done
```

El mismo bucle anterior podríamos haberlo hecho con **until**. Sería lo siguiente:

```
path=$PATH
```

```
until [ "$path" == "" ]; do     # La condición podría ser -z “$path”

        # Visualiza el primer directorio de la cadena restante
        echo ${path%%:*}

        # Se suprime el directorio visualizado
        if [ ${path#*:} = $path ]; then
                path=
        else
                path=${path#*:}
        fi
done
```

Vemos que el sentido de la condición del bucle ha habido que cambiarlo.

6.3.3 La estructura select

La estructura **select** se utiliza para mostrar un menú de selección de opciones y ejecutar el bloque de código correspondiente a la selección escogida. En caso de omitir la lista de palabras, el sistema presenta los parámetros posicionales del programa o función. La siguiente tabla describe el formato de esta estructura.

Bucle	Descripción

`select VAR [in Lista]; do`
`  Bloque1`
`  ...`
`done`
	Selección interactiva: se presenta un menú de selección y se ejecuta el bloque de código correspondiente a la opción elegida. El bucle se termina cuando se ejecuta una orden **break** . La variable **PS3** se usa como punto indicativo.

La sentencia genera un menú con los elementos de lista, donde asigna un número a cada elemento, y pide al usuario que introduzca un número. El valor elegido se almacena en variable, y el número elegido en la variable **REPLY**. Una vez elegida una opción por parte del usuario, se ejecuta el cuerpo de la sentencia y el proceso se repite en un bucle infinito.

Aunque el bucle de **select** es infinito (lo cual nos permite volver a pedir una opción cuantas veces haga falta), el bucle se puede abandonar usando la sentencia **break**. La sentencia **break** se usa para abandonar un bucle.

El prompt que usa la función es el definido en la variable de entorno **PS3**, y es habitual cambiar este prompt antes de ejecutar **select** para que muestre al usuario un mensaje más descriptivo. Por defecto el valor de **PS3** es **#?**, lo cual no es un prompt muy descriptivo.

Veamos el siguiente ejemplo simple en el que se muestra un menú al usuario para elegir la ejecución de un listado del directorio personal, que muestre la fecha o la información del usuario.

```
#!/bin/bash

# Script:       select
# Descripción:  Muestra menú de seleccion al usuario

clear

# Cambiamos el prompt que se muestra al usuario
```

```
PS3="Elegir la acción a realizar: "

select OPCION in Directorio Fecha Usuario Terminar ; do
        case $OPCION in
                "Directorio")
                        ls
                        ;;
                "Fecha")
                        date
                        ;;
                "Usuario")
                        id
                        ;;
                "Terminar")
                        break
                        ;;
                *)
                        echo "Error. Tiene que elegir entre 1 y 4"
                        ;;
        esac
done
```

Al ejecutar este script vemos que se muestra un menú con las cuatro opciones. El usuario tiene que pulsar un número entre 1 y 4. Sin embargo, en la estructura case que controla la opción elegida debemos contemplar la cadena que hemos puesto como opción de menú, no el número de opción. Podemos incluir opciones con espacios en blanco si la entrecomillamos. El bucle es infinito y solamente al elegir la opción 4 es cuando sale por efecto del comando `break`.

7 Funciones

Una función en BASH es una porción de código declarada al principio del programa, que puede recoger parámetro de entrada y que puede ser llamada desde cualquier punto del programa principal o desde otra función, tantas veces como sea necesario.

El uso de funciones permite crear un código más comprensible y que puede ser depurado más fácilmente, ya que evita posibles errores tipográficos y repeticiones innecesarias.

Los parámetros recibidos por la función se tratan dentro de ella del mismo modo que los del programa principal, o sea los parámetros posicionales de la función se corresponden con las variables internas $0 , $1 , etc.

El siguiente cuadro muestra los formatos de declaración y de invocación de una función..

Declaración	Invocación

`[function] NombreFunción () {`
`  Bloque`
`  ...`
`[ return [Valor] ]`
`  ...`
`}`
	`NombreFunción [Parámetro1 ...]`

La función ejecuta el bloque de código encerrado entre sus llaves y, al igual que un programa, devuelve un valor numérico. En cualquier punto del código de la función, y normalmente al final, puede usarse la cláusula **return** para terminar la ejecución y opcionalmente indicar un código de salida.

Las variables declaradas con la cláusula local tienen un ámbito de operación interno a la función. El resto de variables pueden utilizarse en cualquier punto de todo el programa. Esta característica permite crear funciones recursivas sin que los valores de las variables de una llamada interfieran en los de las demás.

En el ejemplo siguiente se define una función de nombre **salida** , que recibe 3 parámetros. El principio del código es la definición de la función (la palabra **function** es opcional) y ésta no se ejecuta hasta que no se llama desde el programa principal. Asimismo, la variable **TMPGREP** se declara en el programa principal y se utiliza en la función manteniendo su valor correcto.

```
#!/bin/bash

# Script:       comprus
# Descripción:  Comprueba la existencia de usuarios en listas y en
                el archivo de claves (normal y NIS).

# Rutina de impresión.
# Parámetros:
#     1 - texto de cabecera.
#     2 - cadena a buscar.
#     3 - archivo de búsqueda.

salida () {
     if egrep "$2" $3 >$TMPGREP 2>/dev/null; then
          echo "     $1:"
          cat $TMPGREP
     fi
}

## PROGRAMA PRINCIPAL ##
TMPGREP=/tmp/grep$$
DIRLISTAS=/home/cdc/listas
if [ "x$*" = "x?" ] then
     echo "
          Uso: `basename $0` ? | cadena
          Propósito: `basename $0`: Búsqueda de usuarios.
          cadena: expresión regular a buscar.
          "
     exit 0
fi

if [ $# -ne 1 ]; then
     echo "
          `basename $0`: Parámetro incorrecto.
          Uso: `basename $0` ? | cadena
          ?: ayuda" >&2
```

```
     exit 1
     fi

echo
for i in $DIRLISTAS/*.lista; do
     salida "$1" "`basename $i | sed 's/.lista//'`" "$i"
done

salida "$1" "passwd" "/etc/passwd"
[ -e "$TMPGREP" ] && rm -f $TMPGREP
```

8 Configuración del entorno

El intérprete de comandos de cada cuenta de usuario tiene un entorno de operación propio, en el que se incluyen una serie de variables de configuración.

El administrador del sistema asignará unas variables para el entorno de ejecución comunes a cada grupo de usuarios, o a todos ellos, mientras que cada usuario puede personalizar algunas de estas características en su perfil de entrada, añadiendo o modificando las variables.

Para crear el entorno global, el administrador crea un perfil de entrada común para todos los usuarios en el archivo **`/etc/profile`**, donde, entre otros cometidos, se definen las variables del sistema y se ejecutan los ficheros de configuración propios para cada aplicación.

Estos pequeños programas se sitúan en el subdirectorio **`/etc/profile.d`**; debiendo existir ficheros propios de los intérpretes de comandos basados en el de Bourne (BSH, BASH, PDKSH, etc.), con extensión **`.sh`**, y otros para los basados en el intérprete C (CSH, TCSH, etc.), con extensión **`.csh`** .

El proceso de conexión del usuario se completa con la ejecución del perfil de entrada personal del usuario en el archivo **`~/.bashrc`** para BASH). Aunque el administrador debe suministrar un perfil válido, el usuario puede retocarlo a su conveniencia.

9 Bibliografía

HERNANDEZ López, F., *El Shell Bash* [acceso febrero 2015]. Disponible en Disponible <http://macprogramadores.org/documentacion/bash.pdf>

LUQUE Rodríguez, M, *Comandos shell y programación en la shell del Bash* [acceso febrero 2015]. Disponible en <http://www.uco.es/~in1lurom/materialDocente/apuntesSO.pdf>

GOMEZ Labrador, R.M., *Programación avanzada en shell,* [acceso febrero 2015]. Disponible en Disponible en <http://www.forpas.us.es/documentacion/programacion_avanzada_en_shell.pdf>