SEMINARIO 06013

PROGRAMACIÓN AVANZADA EN SHELL (línea de comandos)

Ramón M. Gómez Labrador

(ramon.gomez@eii.us.es)

Septiembre de 2.006

Nota importante: El presente seminario se oferta dentro del plan de formación para personal informático de la Universidad de Sevilla para el año 2006 y toda su documentación asociada está bajo licencia Creative Commons con reconocimiento (http://creativecommons.org/licenses/by/2.5/deed.es).

1ª edición: Curso 03-55 Programación Avanzada en *Shell*, octubre 2.003.
2ª edición: Curso 05-08 Programación Avanzada en *Shell*, abril 2.005.
Esta 3ª edición divide el curso en 2 seminarios:

1. Seminario 06013 Programación Avanzada en *Shell* (línea de comandos), septiembre 2.006.
2. Seminario 06053 Programación Avanzada en *Shell* (*shellscripts*), septiembre 2.006.

06013 Programación Avanzada en *Shell* (línea de comandos)

ÍNDICE

1. Introducción.

El **intérprete de mandatos** 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.

Por lo tanto, un intérprete de mandatos de Unix es tanto una interfaz de ejecución de órdenes y utilidades, como un lenguaje de programación, que admite crear nuevas órdenes –denominadas **guiones** o "*shellscripts*"–, utilizando combinaciones de mandatos y estructuras lógicas de control, que cuentan con características similares a las del sistema y que permiten que los usuarios y grupos de la máquina cuenten con un entorno personalizado [2].

En Unix existen 2 familias principales de intérpretes de mandatos: los basados en el intérprete de Bourne (BSH, KSH o BASH) y los basados en el intérprete C (CSH o TCSH).

Los 2 seminarios de formación sobre programación en *shell* pretenden formar una guía para el usuario de Linux, que le permitirá comprender, ejecutar y empezar a programar en la *Shell*, haciendo referencia especialmente a **BASH** (*Bourne Again Shell*) –evolución de BSH, con características de KSH y CSH–, ya que es el intérprete de mandatos más utilizado en Linux e incluye un completo lenguaje para programación estructurada y gran variedad de funciones internas.

1.1. Características principales de BASH.

Los principales características del intérprete BASH [i] son:

- Ejecución síncrona de órdenes (una tras otra) o asíncrona (en paralelo).
- Distintos tipos de redirecciones de entradas y salidas para el control y filtrado de la información.
- Control del entorno de los procesos.
- Ejecución de mandatos interactiva y desatendida, aceptando entradas desde teclado o desde ficheros..
- Proporciona una serie de órdenes internas para la manipulación directa del intérprete y su entrono de operación.
- Un lenguaje de programación de alto nivel, que incluye distinto tipos de variables, operadores, matrices, estructuras de control de flujo, entrecomillado, sustitución de valores y funciones.
- Control de trabajos en primer y segundo plano.

- Edición del histórico de mandatos ejecutados.
- Posibilidad de usar una "*shell*" para el uso de un entorno controlado.

1.2. Cuándo utilizar el intérprete de mandatos.

Como se ha indicado anteriormente, una "*shell*" de Unix puede utilizarse como interfaz para ejecutar órdenes en la línea de comandos o como intérprete de un lenguaje de programación para la administración del sistema.

El lenguaje de BASH incluye una sintaxis algo engorrosa, pero relativamente fácil de aprender, con una serie de órdenes internas que funcionan de forma similar a la línea de comandos. Un programa o guión puede dividirse en secciones cortas, cómodas de depurar, permitiendo realizar prototipos de aplicaciones más complejas.

Sin embargo, hay ciertas tareas que deben ser resueltas con otros intérpretes más complejos o con lenguajes compilados de alto nivel, tales como [4]:

- Procesos a tiempo real, o donde la velocidad es un factor fundamental.
- Operaciones matemáticas de alta precisión, de lógica difusa o de números complejos.
- Portabilidad de código entre distintas plataformas.
- Aplicaciones complejas que necesiten programación estructurada o proceso multihilvanado.
- Aplicaciones críticas para el funcionamiento del sistema.
- Situaciones donde debe garantizarse la seguridad e integridad del sistema, para protegerlo contra intrusión o vandalismo.
- Proyectos formados por componentes con dependencias de bloqueos.
- Proceso intensivo de ficheros, que requieran accesos directos o indexados.
- Uso de matrices multidimensionales o estructuras de datos (listas, colas, pilas, etc.).
- Proceso de gráficos.
- Manipulación de dispositivos, puertos o "*sockets*".
- Uso de bibliotecas de programación o de código propietario

.

2. Redirecciones.

Unix hereda 3 ficheros especiales del lenguaje de programación C, que representan las funciones de entrada y salida de cada programa. Éstos 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`** sirve para descartar alguna redirección e ignorar sus datos.

2.1. 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 las utilidades y filtros de Unix soportan los ficheros de entrada como parámetro del programa y no es necesario redirigirlos.

2.2. Redirecciones 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.

El siguiente ejemplo crea un fichero con las salidas generadas para configurar, compilar e instalar una aplicación GNU.

```
configure     > aplic.sal
make         >> aplic.sal
make install >> aplic.sal
```

2.3. Combinación de redirecciones.

Pueden combinarse más de una redirección sobre el mismo mandato o grupo de mandatos, interpretándose siempre de izquierda a derecha.

Ejercicio 2.1: interpretar las siguientes órdenes:

```
ls –al /usr /tmp /noexiste >ls.sal 2>ls.err
find /tmp –print >find.sal 2>/dev/null
```

Otras formas de combinar las redirecciones permiten realizar copias de descriptores de ficheros de entrada o de salida. La siguiente tabla muestra los formatos para duplicar descriptores.

Redirección	Descripción
[*N*]<&*M*	Duplicar descriptor de entrada *M* en *N* (*N*=0, por defecto).
[*N*]<&-	Cerrar descriptor de entrada *N*.
[*N*]<&*M*-	Mover descriptor de entrada *M* en *N*, cerrando *M* (*N*=0, por defecto).
[*N*]>&*M*	Duplicar descriptor de salida *M* en *N* (*N*=1, por defecto).
[*N*]>&-	Cerrar descriptor de salida *N*.
[*N*]>&*M*-	Mover descriptor de salida *M* en *N*, cerrando *M* (*N*=1, por defecto).

Conviene hacer notar, que –siguiendo las normas anteriores– las 2 líneas siguientes son equivalentes y ambas sirven para almacenar las salidas normal y de error en el fichero indicado:

```
ls –al /var/* &>ls.txt
ls –al /var/* >ls.txt 2>&1
```

Sin embargo, el siguiente ejemplo muestra 2 mandatos que no tienen por qué dar el mismo resultado, ya que las redirecciones se procesan de izquierda a derecha, teniendo en cuenta los posibles duplicados de descriptores hechos en líneas anteriores.

```
ls –al * >ls.txt 2>&1  # Salida normal y de error a "ls.txt".
ls –al * 2>&1 >ls.txt  # Asigna la de error a la normal anterior
                       # (puede haberse redirigido) y luego
                       # manda la estándar a "ls.txt".
```

2.4. Redirección de entrada/salida.

La **redirección de entrada y salida** abre el fichero especificada para operaciones de lectura y escritura y le asigna el descriptor indicado (0 por defecto). Se utiliza en operaciones para modificación y actualización de datos. El formato genérico es:

```
[N]<>Fichero
```

El siguiente ejemplo muestra una simple operación de actualización de datos en un determinado lugar del fichero [4].

```
echo 1234567890 > fich   # Genera el contenido de "fich"
exec 3<> fich            # Abrir fich con descriptor 3 en E/S
read -n 4 <&3            # Leer 4 caracteres
echo -n , >&3            # Escribir coma decimal
exec 3>&-                # Cerrar descriptor 3
cat fich                 # ➔ 1234,67890
```

2.5. Documento interno.

La **redirección de documento interno** usa parte del propio programa –hasta encontrar un delimitador de final– como redirección de entrada al comando correspondiente. Suele utilizarse para mostrar o almacenar texto fijo, como por ejemplo un mensaje de ayuda.

El texto del bloque que se utiliza como entrada se trata de forma literal, esto es, no se realizan sustituciones ni expansiones.

El texto interno suele ir tabulado para obtener una lectura más comprensible. El formato **<<** mantiene el formato original, pero en el caso de usar el símbolo **<<-**, el intérprete elimina los caracteres de tabulación antes de redirigir el texto.

La siguiente tabla muestra el formato de la redirección de documento interno.

Redirección	Descripción
<<[**-**] *Delimitador* *Texto* ... *Delimitador*	Se usa el propio *shellscript* como entrada estándar, hasta la línea donde se encuentra el delimitador. Los tabuladores se eliminan de la entrada en el caso de usar la redirección **<<-** y se mantienen con **<<**.

Como ejemplo se muestra un trozo de código y su salida correspondiente, que presentan el texto explicativo para el formato de uso de un programa.

```
echo << FIN
Formato:
config OPCION ...

OPCIONES:
    --cflags
    --ldflags
    --libs
    --version
    --help
FIN
```

```
Formato:
config OPCION ...

OPCIONES:
    --cflags
    --ldflags
    --libs
    --version
    --help
```

2.6. 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:

```
Mandato1 | Mandato2 ...
```

Todos los dialectos Unix incluyen gran variedad de filtros de información. La siguiente tabla recuerda algunos de los más utilizados.

Mandato	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.
diff	Muestra las diferencias entre 2 ficheros en un formato compatible con la orden **sed**.
comm	Compara 2 ficheros.
tr	Sustituye grupos de caracteres uno a uno.
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
```

La orden **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.

El siguiente ejemplo muestra y registra el proceso de compilación e instalación de una aplicación GNU.

```
configure   2>&1 | tee aplic.sal
make        2>&1 | tee –a aplic.sal
make instal 2>&1 | tee –a aplic.sal
```

Ejercicio 2.2: interpretar la siguiente orden:

```
ls | tee salida | sort -r
```

3. Variables.

Al contrario que en otros lenguajes de programación, BASH no hace distinción en los tipos de datos de las variables; son esencialmente cadenas de caracteres, aunque –según el contexto– también pueden usarse con operadores de números enteros y condicionales. Esta filosofía de trabajo permite una mayor flexibilidad en la programación de guiones, pero también puede provocar errores difíciles de depurar [4].

Una variable BASH se define o actualiza mediante operaciones de asignación, mientras que se hace referencia a su valor utilizando el símbolo del dólar delante de su nombre.

Suele usarse la convención de definir las variables en mayúsculas para distinguirlas fácilmente de los mandatos y funciones, ya que en Unix las mayúsculas y minúsculas se consideran caracteres distintos.

```
VAR1="Esto es una prueba          # asignación de una variable
VAR2=35                           # asignar valor numérico
echo $VAR1                        # ➔ Esto es una prueba
echo "VAR2=$VAR2"                 # ➔ VAR2=35
```

3.1. Tipos de variables.

Las variables del intérprete BASH pueden considerarse desde los siguientes puntos de vista:

- 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 guión.
- 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 mandatos.
- 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.

3.1.1. Variables locales.

Las variables locales son definidas para operar en un ámbito reducido de trabajo, ya sea en un programa, en una función o en un bloque de código. Fuera de dicho ámbito de operación, la variable no tiene un valor preciso.

Una variable tiene un nombre único en su entorno de operación, sin embargo pueden –aunque no es nada recomendable– usarse variables con el mismo nombre en distintos bloques de código.

El siguiente ejemplo muestra los problemas de comprensión y depuración de código que pueden desatarse en caso de usar variables con el mismo nombre. En la primera fila se presentan 2 programas que usan la misma variable y en la segunda, la ejecución de los programas (nótese que el signo **>** es el punto indicativo del interfaz de la "*shell*" y que lo tecleado por el usuario se representa en letra negrita).

```
#!/bin/bash
# prog1 – variables prueba 1
VAR1=prueba
echo $VAR1
```

```
#!/bin/bash
# prog2 – variables prueba 2
VAR1="otra prueba"
echo $VAR1
```

```
> echo $VAR1

> prog1
prueba
> prog2
otra prueba
> prog1
prueba
```

Por lo tanto, para asignar valores a una variable se utiliza simplemente su nombre, pero para hacer referencia a su valor hay que utilizar el símbolo dólar (**$**). El siguiente ejemplo muestra los modos de referirse a una variable.

```
ERR=2                                # Asigna 2 a la variable ERR.
echo ERR                             # ➔ ERR    (cadena "ERR").
echo $ERR                            # ➔ 2      (valor de ERR).
echo ${ERR}                          # ➔ 2      (es equivalente).
echo "Error ${ERR}: salir"           # ➔ 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.

3.1.2. Variables de entorno.

Al igual que cualquier otro proceso Unix, la "*shell*" mantiene un conjunto de variables que informan sobre su propio contexto de operación. El usuario –o un *shellscript*– puede actualizar y añadir variables exportando sus valores al entorno del intérprete (mandato **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 de entorno	Descripción	Valor por omisión
SHELL	Camino del programa intérprete de mandatos.	La propia *shell*.
PWD	Directorio de trabajo actual.	Lo modifica la *sheil.*
OLDPWD	Directorio de trabajo anterior (equivale a **~-**).	Lo modifica la *sheil.*
PPID	Identificador del proceso padre (PPID).	Lo modifica la *shell*
IFS	Separador de campos de entrada (debe ser de sólo lectura).	ESP, TAB, NL.
HOME	Directorio personal de la cuenta.	Lo define **root**.
LOGNAME	Nombre de usuario que ejecuta la *shell.*	Activado por **login**
PATH	Camino de búsqueda de mandatos.	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).

Ejercicio 3.1: interpretar la siguiente orden:

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

Recomendaciones de seguridad:
Siempre deben indicarse caminos absolutos en la definición de la variable **`PATH`** y, sobre todo, nunca incluir el directorio actual (`.`), ni el directorio padre (`..`).
Declarar la variable **`IFS`** de sólo lectura, para evitar intrusiones del tipo "caballos de Troya".

3.1.3. Parámetros de posición.

Los parámetros de posición son variables especiales de BASH que contienen los valores de los parámetros que recibe un programa o una función. El número indica la posición de dicho parámetro en la llamada al código.

El 1er parámetro se denota por la variable **`$1`**, el 9º por **`$9`** y a partir del 10º hay que usar la notación **`${`***`Número`***`}`**.

El mandato interno **`shift`** desplaza la lista de parámetros hacia la izquierda para procesar los parámetros más cómodamente. El nombre del programa se denota por la variable **`$0`**.

Observar el siguiente cuadro para observar el uso de parámetros posicionales y de variables locales, donde se muestran algunas líneas de un programa para gestión de usuarios.

```
grep "^$1:" /etc/passwd
grep ":$GID:" /etc/group | cut -f1 -d:
```

1. Imprime la línea del fichero de usuarios para el especificado en el 1er parámetro recibido por el programa.
2. Presenta el nombre del grupo cuyo identificador se encuentra en la variable **`GID`**.

3.1.4. Variables especiales.

Las variables especiales informan sobre el estado del proceso, son tratadas y modificadas directamente por el intérprete, por lo tanto, son de sólo lectura.

La siguiente tabla describe brevemente estas variables.

Variable especial	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 mandato (0=normal, >0=error).
$!	Último identificador de proceso ejecutado en segundo plano.
$_	Valor del último argumento del comando ejecutado previamente.

La construcción **cat "$@"** se utiliza para procesar datos tanto de ficheros como de la entrada estándar [4].

La 1ª fila de la tabla del siguiente ejemplo muestra el código de un programa para convertir minúsculas en mayúsculas; mientras que la 2ª, indica cómo puede utilizarse el programa (el texto tecleado está representado en letra negrita).

Nota: la construcción **^D** representa la combinación de teclas Control-D (carácter fin de texto).

```
#!/bin/bash
# mayusculas – convierte a mayúsculas usando ficheros o stdin
# Uso: mayusculas [ [<]fichero ]

cat "$@" | tr 'a-zñáéíóúü' 'A-ZÑÁÉÍÓÚÜ'
```

```
> mayusculas datos.txt >datos.sal
> mayusculas <datos.txt >datos.sal
> mayusculas
Esto es una prueba de ejecución del programa.
^D
ESTO ES UNA PRUEBA DE EJECUCIÓN DEL PROGRAMA.
```

Un 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.

Ejercicio 3.2: interpretar los siguientes mandatos:

```
echo $0; shift; echo $0
```

3.2. Matrices.

Una **matriz** (o "*array*") es un conjunto de valores identificados por el mismo nombre de variable, donde cada una de sus celdas cuenta con un índice que la identifica. 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.

El uso de 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 anterior.

El siguiente ejemplo describe la utilización de matrices.

```
declare –a NUMEROS                          # Declarar la matriz.
NUMEROS=( cero uno dos tres )               # Asignación compuesta.
echo ${NUMEROS[2]}                          # ➔ dos
NUMEROS[4]="cuatro"                         # Asignación simple.
echo ${NUMEROS[4]}                          # ➔ cuatro
NUMEROS=( [6]=seis siete [9]=nueve ) #  celdas 6, 7 y 9.
echo ${NUMEROS[7]}                          # ➔ siete
```

Ejercicio 3.3: según los datos de la matriz del ejemplo anterior, cuál es la salida de la siguiente orden:

```
echo ${NUMEROS[*]}
```

3.3. Configuración del entorno.

El intérprete de mandados 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 (archivo **`/etc/bashrc`** en el caso de BASH),

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 mandatos 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 (archivo **`~/.bash_profile`** para BASH). Aunque el administrador debe suministrar un perfil válido, el usuario puede retocarlo a su conveniencia. En el siguiente capítulo se presentan las variables de entorno más importantes usadas por BASH.

4. Expresiones.

El intérprete BASH permite utilizar una gran variedad de expresiones en el desarrollo de programas y en la línea de mandatos. 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 mandatos internos de BASH para su evaluar indicando si ésta es cierta o falsa.
- **Expresiones de cadenas**: aquellas que tratan cadenas de caracteres (se tratarán a fondo en el Capítulo 6).

Las expresiones complejas cuentan con varios parámetros 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.

La tabla lista 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`

4.1. Expresiones aritméticas.

Las expresiones aritméticas representan operaciones números enteros o binarios (booleanos) evaluadas mediante el mandato interno **`let`** (no se permiten número reales ni 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 lenguaje 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 pre-operación (operadores válidos: *=, /=, %=, +=, -=, <<=, >>=, &=, ^=, |=).	Asigna el valor de E2 a E1. Si es indica 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.

El cuadro que se muestra a continuación ilustra el uso de las expresiones aritméticas.

```
let a=5                                     # Asignación a=5.
let b=$a+3*9                                # b=a+(3*9)=32.
echo "a=$a, b=$b"                           # ➔ a=5, b=32
let c=$b/($a+3)                             # c=b/(a+3)=4.
let a+=c--                                  # a=a+c=9, c=c-1=3.
echo "a=$a, c=$c"                           # ➔ a=9, c=3
let CTE=$b/$c, RESTO=$b%$c                  # CTE=b/c, RESTO=resto(b/c).
echo "Cociente=$CTE, Resto=$RESTO" # ➔ Cociente=10, 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 (0-9).
let N=034        # Base octal (0-7), empieza por 0.
let N=0x34AF     # Base hexadecimal (0-9A-F), empieza por 0x.
let N=[20#]G4H2  # Base 20 (especificada entre 2 y 64).
```

Ejercicio 4.1: explicar la siguiente expresión.

```
let a=(b>c)?b:c
```

4.2. Expresiones condicionales.

Las **expresiones condicionales** son evaluadas por los mandatos 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 de ellas en una expresión compleja usando los operadores lógicos Y (*Expr1* **`&&`** *Expr2*) y O (*Expr1* `||` *Expr2*).

4.2.1. 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.

Conviene recordar que “todo en Unix es un fichero”, por eso hay bastantes operadores de tipos de ficheros. La siguiente tabla lista los formatos de estas expresiones.

Formato	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.

Formato	Condición (cierto si...)
-r *Fich*	Tiene permiso de lectura.
-w *Fich*	Tiene permiso de escritura (modificación).
-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.

Formato	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.

Véanse ejemplos extraídos del fichero de configuración `/etc/rc.d/rc.sysinit`.

```
# Si /proa/mdstat y /etc/raidtab son ficheros; entonces ...
if [ -f /proc/mdstat -a -f /etc/raidtab ]; then
...
# Si el camino representado por el contenido de la variable
#    $afile es un directorio; entonces ...
if [ -d "$afile" ]; then
...
```

Ejercicio 4.2: interpretar la siguiente expresión.

```
if [ -e /proc/lvm -a -x /sbin/vgchange -a -f /etc/lvmtab ];
```

4.3.2. 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.

Formato	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).

Continuando con el ejemplo, se comentan unas líneas de `/etc/rc.d/rc.sysinit`.

```
# Si la variable RESULT es > 0 y
#    /sbin/raidstart es ejecutable; entonces ...
if [ $RESULT -gt 0 -a -x /sbin/raidstart ]; then
...
# Si el código de la ejecución del 1er mandato es 0 y
#    el del 2º es distinto de 0; entonces ...
if grep -q /initrd /proc/mounts && \
   ! grep -q /initrd/loopfs /proc/mounts ; then
...
# Si la expresión de que existe el fichero /fastboot es cierta o
#    el código de salida del mandato es correcto; entonces ...
if [ -f /fastboot ] || \
   grep -iq "fastboot" /proc/cmdline 2>/dev/null ; then
...
```

4.3.3. Expresiones comparativas de cadenas.

De forma similar, también pueden realizarse comparaciones entre cadenas de caracteres. Conviene destacar que es recomendable usar comillas dobles para delimitar las cadenas a comparar.

La tabla siguiente indica el formato para este tipo de expresiones.

Formato	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.

Como en los párrafos previos, se revisa parte del código del fichero `/etc/rc.d/rc.sysinit`.

```
# Si LOGNAME es una variable vacía o
#    tiene el valor "(none)"; entonces ...
if [ -z "$HOSTNAME" -o "$HOSTNAME" = "(none)" ]; then
...
# Si la variable $fastboot no tiene contenido y
#    la variable $ROOTFSTYPE no es "nfs"; entonces ...
if [ -z "$fastboot" -a "$ROOTFSTYPE" != "nfs" ]; then
...
```

Ejercicio 4.3: interpretar la siguiente orden.

```
if [ -n "$PNP" -a -f /proc/isapnp -a -x /sbin/sndconfig ]; then
```

5. Entrecomillado.

Cada uno de los caracteres especiales –**metacaracteres**– usados en BASH tienen un comportamiento especial, según la sintaxis del lenguaje. El **entrecomillado** es el procedimiento utilizado para modificar o eliminar el uso normal de dicho metacaracteres. Obsérvese el siguiente ejemplo.

```
# El “;” se usa normalmente para separar comandos.
echo Hola; echo que tal                    # ➔ Hola
                                           #    que tal
# Usando entrecomillado pierde su función normal.
echo "Hola; echo que tal"                  # ➔ Hola; echo que tal
```

Los 3 tipos básicos de entrecomillado definidos en BASH son [2]:

- **Carácter de escape (*Carácter*)**: mantiene el valor literal del carácter que lo precede; la secuencia `\\` equivale a presentar el carácter `\`. Cuando aparece 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).

El entrecomillado con formato **`$"`***`Cadena`***`"`** se utiliza para procesos de traducción según el idioma expresado por la variable **`LANG`**. Si se utiliza el valor de idioma por defecto (C o POSIX), la cadena se trata normalmente con comillas dobles.

Veamos unos ejemplos:

```
echo "Sólo con permiso \"root\"" # ➔ Sólo con permiso "root"
echo 'Sólo con permiso \"root\"' # ➔ Sólo con permiso \"root\"
```

Ejecrcicia 5.1: indicar la salida del siguiente mandato.

```
echo "Ejecutar 'smbmount \\\\servidor\\documentos \$HOME/doc'"
```

6. Referencias.

1. B. Fox, C. Ramey: *"BASH(1)" (páginas de manuales de BASH v2.5b)"*. 2.002.
2. C. Ramey, B. Fox: "*Bash Reference Manual, v2.5b*". Free Software Foundation, 2.002.
3. Mike G, trad. G. Rodríguez Alborich: *"Programación en BASH – COMO de Introducción"*. 2.000.
4. M. Cooper: *"Advanced Bash-Scripting Guide, v2.1"*. Linux Documentation Project, 2.003.
5. R. M. Gómez Labrador: *"Administración Avanzada de Sistemas Linux (3ª edición)"*. Secretariado de Formación del PAS (Universidad de Sevilla), 2.004.

i. Proyecto GNU.: http://www.gnu.org/
ii. The Linux Documentation Project (TLDP): http://www.tldp.org/
iii. Proyecto HispaLinux (LDP-ES): http://www.hispalinux.es/