CURSO 03-55

PROGRAMACIÓN AVANZADA EN SHELL

Ramón M. Gómez Labrador

(ramon.gomez@eii.us.es)

Octubre de 2.003

03-55 Programación Avanzada en *Shell*

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).

Este curso de formación 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** (*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 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, ppara 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. Modos de operación.

2.1. La línea de mandatos.

La línea de mandatos es el interfaz del usuario con el sistema, que permite personalizar el entorno de operación y ejecutar programas y guiones.

El formato típico de una línea consta de una orden y unos modificadores y parámetros opcionales, aunque puede incluir algunos caracteres especiales, que modifican el comportamiento típico.

```
Mandato [Modificador ...][Parámetro ...]
```

Un caso especial es el de las líneas que comienzan por la almohadilla (**#**), que se consideran comentarios. También puede insertarse un comentario en mitad de una línea, a la derecha de una orden.

```
# Esto es un comentario
ls -al    # lista el contenido del directorio actual
```

Pueden agruparse varias órdenes en la misma línea separadas por el punto y coma (**;**), que se ejecutan siempre en secuencia. Asimismo, si un mandato es muy largo o engorroso, puede usarse el carácter de escape (\) para continuar escribiéndolo en la línea siguiente.

```
cd /var/log; grep –i error *
find /tmp /var/tmp ! –user root –type f \
     -perm +2 –print
```

2.2. Edición y ejecución de un guión.

Un guión interpretado por BASH es un fichero de texto normal que consta de una serie de bloques de código formados por líneas de mandatos que se ejecutan en secuencia. El usuario debe tener los permisos de modificación (escritura) en el directorio –para crear un nuevo programa– o sobre el propio fichero, para modificar uno existente.

Como cualquier otro programa binario, el usuario debe tener permiso de ejecución en el fichero del guión, y se lanza tecleando su camino completo junto con sus opciones y parámetros. Asimismo, como veremos en el Capítulo 4, 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 el camino.

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

```
vi prueba.sh              # o cualquier otro editor de textos
chmod u+x prueba.sh       # activa el permiso de ejecución
./prueba.sh               # ejecuta el guión
prueba.sh                 # si está en un directorio de $PATH
```

Existe una manera especial para ejecutar un guión, precediéndolo por el signo punto, que se utiliza para exportar todas las variables del programa al entorno de ejecución del usuario (para más información sobre las variables de entorno revisar el Capítulo 4). El siguiente ejemplo ilustra el modo de ejecutar

```
apachectl start          # Ejecución normal de un guión.
. miprofile              # Ejecución exportando las variables.
```

Un "*script*" puede –y debe– comenzar con la marca **`#!`** para especificar el camino completo y los parámetros del intérprete de mandatos que ejecutará el programa. Esta marca puede usarse para ejecutar cualquier intérprete instalado en la máquina (BASH, BSH, PERL, AWK, etc.).

El siguiente cuadro muestra un pequeño ejemplo de guión BASH.

```
#!/bin/bash
# ejemplo1: informe de la capacidad de la cuenta

echo "Usuario: $USER"
echo "Capacidad de la cuenta en KB:"
du –ks $HOME           # suma total del directorio del usuario
```

2.3. Recomendaciones de programación.

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 mandatos 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 de ellas, ni con los mandatos 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.

3. Redirecciones.

Unix hereda 3 ficheros especiales del lenguaje de programación C, 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.

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

3.2. 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
```

3.3. Combinación de redirecciones.

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

Como ejercicio, se propone 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 es 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".
```

3.4. 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
```

3.5. 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 **<<**.

3.6. 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.
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
```

Se propone como ejercicio, interpretar la siguiente orden:

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

4. 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
```

4.1. 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.

4.1.1. 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 ${VAR}: 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.

4.1.2. 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 guión– 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).

Como ejercicio, interpretar la siguiente orden:

```
PATH=$PATH:/home/cdc/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".

4.1.3. 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**.

Para observar el uso de parámetros posicionales y de variables locales, repasemos algunas partes del programa "**usuario**" (Apéndice C.4):

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

1. Muestra la línea del usuario especificado en el 1er parámetro recibido por el programa.
2. Presenta el nombre del grupo cuyo identificador se encuentra en la variable **GID**.

4.1.4. 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 "$@"** puede usarse 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 de minúsculas a mayúsculas. La 2ª línea indica cómo puede utilizarse el programa.

```
#!/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.
```

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.

4.2. Matrices.

Una **matriz** (o "*array*") es un conjunto de valores identificados por el mismo nombre de variable, donde cada una de sus celda 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.

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 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 ) # asigna celdas 6, 7 y 9.
echo ${NUMEROS[7]}                    # ➔ siete
echo ${NUMEROS[*]}    # ==> uno dos tres cuatro seis siete nueve
```

4.3. 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.

5. 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.

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 mandato 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 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 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.

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/($a+3)                      # c=b/(a+3)=4.
let a+=c--                           # a=a+c=9, c=c+1=5.
echo "a=$a, c=$c"                    # ➔ a=9, c=5
let CTE=$b/$c, RESTO=$b%$c           # CTE=b/c, RESTO=resto(b/c).
echo "Cociente=$CTE, Resto=$RESTO"   # ➔ 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 (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).
```

5.2. 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*).

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.

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.

Veamos algunos 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 caminto representado por el contenido de la variable
#    $afile es un directorio; entonces ...
if [ -d "$afile" ]; then
...
```

Se propone como ejercicio, interpretar la siguiente expresión.

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

5.3.2. 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 algunas líneas del fichero de configuración `/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
...
```

5.3.3. Expresiones comparativas de cadenas.

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

Se propone como ejercicio interpretar la siguiente orden.

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

6. Entrecomillado y expansión.

6.1. 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; como último carácter de la línea, sirve para continuar la ejecución de una orden en la línea siguiente. La secuen
- **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\"
```

6.2..Expansión.

Como se lleva observando en todo el curso, 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 topos de expansiones, que según su orden de procesado son [2]:

- **Expansión de llaves**: modifica la expresión para crear cadenas arbitrarias.
- **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 mandato**: procesa el mandato 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 mandatos resultante en palabras usando los caracteres de división incluidos en la variable **`IFS`**.
- **Expansión de fichero**: permite buscar patrones con comodines en los nombres de ficheros.

6.2.1. Expansión de llaves.

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*]	l 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              # ➔ abe ace ade
mkdir $HOME/{bin,lib,doc}   # Se crean los directorios:
                            # $HOME/bin, $HOME/lib y $HOME/doc.
```

2.6.2. 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`**).

Véase este pequeño programa:

```
#!/bin/bash
# capacidad – muestra la capacidad en KB de la cuenta del
#             usuario indicado
# Uso:  capacidad usuario

ls –ld ~$1
du –ks ~$1
```

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 ~-'
```

6.2.3. Expansión de parámetro o de variable.

Permite la sustitución del contenido de la variable siguiendo una amplia variedad de reglas. Los distintos formatos para la expansión de parámetros son [1]:

Formato	Descripción
${!*Var***}**	Se hace referencia a otra variable y se obtiene su valor (**expansión indirecta**).
${*Parám***:-***Val***}**	Se devuelve el parámetro; si éste es nulo, se obtiene el **valor por defecto**.
${*Parám***:=***Val***}**	Si el parámetro es nulo se le **asigna el valor por defecto** y se expande.
${*Parám***:?***Cad***}**	Se obtiene el parámetro; si es nulo se manda un **mensaje de error**.
${*Parám***:+***Val***}**	Se devuelve el **valor alternativo** si el parámetro no es nulo.
${***Parám***:***Inic***}** **${*Parám***:***Inic***:***Long***}**	**Valor de subcadena** del parámetro, desde el punto inicial hasta el final o hasta la longitud indicada.
${!*Pref****}**	Devuelve los nombres de variables que empiezan por el prefijo.
${#***Parám***}** **${#*Matriz***[*]}**	El **tamaño** en caracteres del parámetro o en elementos de una matriz.
${***Parám***#***Patrón***}** **${*Parám***##***Patrón***}**	Se elimina del valor del parámetro la mínima (o la máxima) comparación del patrón, comenzando por el principio del parámetro.
${***Parám***%***Patrón***}** **${*Parám***%%***Patrón***}**	Se elimina del valor del parámetro la mínima (o la máxima) comparación del patrón, buscando por el final del parámetro.
${***Parám/Patrón/Cad***}** **${*Parám//Patrón/Cad***}**	En el valor del parámetro se reemplaza por la cadena indicada la primera comparación del patrón (o todas las comparaciones).

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 89.
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” al camino del fichero.
RPM=${FICHRPM%.rpm}

# Cuenta el nº 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"
```

Por último, interpretar los siguientes ejercicios.

```
DATOS=ls –ld $1
TIPO=${DATOS:1}
PERM=${DATOS:2:9}
```

```
if [ ${#OPCIONES} -gt 1 ]; then ...
```

```
f="$f${1%$e}"
```

6.2.4. Sustitución de mandato.

Esta expansión sustituye el mandato ejecutado –incluyendo sus parámetros– por su salida normal, ofreciendo una gran potencia y flexibilidad de ejecución a un “*shellscrpit*”. Los formatos válido son:

Formato	Descripción
$(*Mandato***)**	Sustitución literal del mandato y sus parámetros.
`***Mandato***`	Sustitución de mandatos permitiendo caracteres de escape.

Cuando la sustitución de mandatos va en una cadena entre comillas dobles se evita que posteriormente se ejecute una expansión de ficheros.

El siguiente programa lista información sobre un usuario.

```
#!/bin/bash
# infous – lista información de un usuario.
# Uso:  infous 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:`
```

Se propone como ejercicio interpretar la salida de la siguiente orden:

```
echo "`basename $0`: \"$USER\" sin permiso de ejecución." >&2
```

6.2.5. 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.

Véase el siguiente ejemplo:

```
# Cuenta el nº de espacios para centrar una cadena
#     espacios = ( ancho_pantalla – longitud (cadena) ) / 2.
ESPACIOS=$(( (ANCHO-${#CADENA})/2 ))
```

Y ahora un ejercicio algo complejo de leer; interpretar la siguiente línea:

```
if [ ${#cad} -lt $[${#1}-1] ]; then ...
```

6.2.6. Sustitución de proceso.

La sustitución de proceso permite utilizar un fichero especial de tipo cola para intercambiar información entre 2 procesos, uno que escribe en la cola y el otro que lee de ella en orden (el primero en llegar es el primero en salir). Los formatos válidos para esta expansión son:

Formato	Descripción
Fich **<(***Lista***)** *Descr***<(***Lista***)**	La lista de órdenes escribe en el fichero para que éste pueda ser leído por otro proceso.
Fich **>(***Lista***)** *Descr***>(***Lista***)**	Cuando otro proceso escribe en el fichero, el contenido de éste se pasa como parámetro de entrada a la lista de órdenes.

6.2.7. División en palabras.

Una vez que se hayan realizado las expansiones previas, el intérprete divide la línea de entrada en palabras, utilizando como separadores los caracteres especificados en la variable de entorno **IFS**. Para evitar problemas de seguridad generados por un posible "Caballo de Troya", el administrador debe declarar esta variable como de sólo lectura y establecer unos valores fijos para los separadores de palabra; que por defecto éstos son espacio, tabulador y salto de línea. Una secuencia de varios separadores se considera como un único delimitador.

Por ejemplo, si se ejecuta el siguiente mandato:

```
du –ks $HOME
```

el intérprete realiza las sustituciones y –antes de ejecutar la orden– divide la línea en las siguientes palabras.

```
"du" "-ks" "/home/ramon""du" "-ks" "/home/ramon"
```

6.2.8. Expansión de fichero.

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 [2]. 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 *.??
```

Por último, se propone como ejercicio evaluar la siguiente orden:

```
cd /var/log; tar cvf copialog.tar syslog.[0-9] messages.[0-9]
```

7. 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.

7.1. Listas de mandatos.

Los mandatos 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 órdenes	Descripción
Mandato **&**	Ejecuta el mandato en 2º plano. El proceso tendrá menor prioridad y no debe ser interactivo..
Man1 **|** *Man2*	Tubería. Redirige la salida de la primera orden a la entrada de la segundo. Cada mandato es un proceso separado.
Man1; *Man2*	Ejecuta varios mandatos en la misma línea de código.
Man1 **&&** *Man2*	Ejecuta la 2ª orden si la 1ª lo hace con éxito (su estado de salida es 0).
Man1 **||** *Man2*	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.
Línea1 **** *Línea2*	Posibilita escribir listas de órdenes en más de una línea de pantalla. Se utiliza para ejecutar mandatos largos.

Ya se ha comentado previamente la sintaxis de algunas de estas combinaciones de mandatos. Sin embargo, el siguiente epígrafe presta atención especial a las listas de órdenes condicionales.

7.1.1. Listas condicionales.

Los **mandatos condicionales** son aquellos que se ejecutan si una determinada orden realiza correctamente –lista Y (**&&**)– o si se produce algún error –lista O (**||**)–.

A continuación se comentan algunos ejemplos, sacados del fichero `/etc/rc.d/rc.sysinit`.

```
# Si se ejecuta correctamente “converquota”, se ejecuta “rm”.
/sbin/convertquota -u / && rm -f /quota.user
#
# Si da error la ejecución de “grep”, se ejecuta “modprobe”.
grep 'hid' /proc/bus/usb/drivers || modprobe hid 2> /dev/null
```

Se propone como ejercicio, interpreta la siguiente lista de órdenes.

```
loadkeys $KEYMAP < /dev/tty0 > /dev/tty0 2>/dev/null && \
    success "Loading def keymap" || failure "Loading def keymap"
```

7.2. 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.

7.2.1. 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 de programación	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 situación, incluyendo el código de salida de un mandato o una condición evaluada por la orden interna test. Este último caso se expresa colocando la condición entre corchetes (formato: **[** *Condición* **]**).

Véanse algunos sencillos ejemplos de la estructura condicional simple extraídos del "*script*" `/etc/rc.d/rc.sysinit`. Nótese la diferencia en las condiciones sobre la salida normal de una orden –expresada mediante una sustitución de mandatos– y aquellas referidas al estado de ejecución de un comando (si la orden se ha ejecutado correctamente o si se ha producido un error).

```
# La condición más simple escrita en una línea:
# si RESULT>0; entonces rc=1
if [ $RESULT -gt 0 ]; then rc=1; fi
#
# La condición doble:
# si la variable HOSTNAME es nula o vale "(none)"; entonces ...
if [ -z "$HOSTNAME" -o "$HOSTNAME" = "(none)" ]; then
    HOSTNAME=localhost
fi
#
# Combinación de los 2 tipos de condiciones:
# si existe el fichero /fastboot o la cadena "fastboot" está
#    en el fichero /proc/cmdline; entonces ...
if [ -f /fastboot ] || grep -iq "fastboot" /proc/cmdline \
                              2>/dev/null; then
      fastboot=yes
fi
```

Obsérvense los ejemplos para las estructuras condicionales complejas, basados también en el programa de configuración `/etc/rc.d/rc.sysinit`.

```
# Estructura condicional compleja con 2 bloques:
# si existe el fichero especificado, se ejecuta; si no, se da
#    el valor "no" a la variable NETWORKING
if [ -f /etc/sysconfig/network ];
then
    . /etc/sysconfig/network
else
    NETWORKING=no
fi
# Estructura anidada:
# si rc=0; entonces ...; si no, si rc=1; entonces ...; en caso
#    contrario; no se hace nada.
if [ "$rc" = "0" ]; then
      success "$STRING"
      echo
elif [ "$rc" = "1" ]; then
      passed "$STRING"
      echo
fi
```

Por último, evaluar el siguiente ejercicio:

```
if [ -f /etc/sysconfig/console/default.kmap ]; then
   KEYMAP=/etc/sysconfig/console/default.kmap
else
  if [ -f /etc/sysconfig/keyboard ]; then
    . /etc/sysconfig/keyboard
  fi
  if [ -n "$KEYTABLE" -a -d "/usr/lib/kbd/keymaps" -o -d
"/lib/kbd/keymaps" ]; then
     KEYMAP=$KEYTABLE
  fi
fi
```

7.2.2. 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 de programación	Descripción

case *Variable* **in**
  [**(**]*Patrón1***)** *Bloque1* **;;**
  ...
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.

Obsérvense el siguientes ejemplos:

```
# Según el valor de la variable UTC:
# - si es "yes" o "true", ...
# - si es "no" o "false", ...
case "$UTC" in
   yes|true)   CLOCKFLAGS="$CLOCKFLAGS --utc";
               CLOCKDEF="$CLOCKDEF (utc)";
   ;;
   no|false)   CLOCKFLAGS="$CLOCKFLAGS --localtime";
               CLOCKDEF="$CLOCKDEF (localtime)";
   ;;
esac
```

Y, como en los casos anteriores, describir el modo de ejecución de la siguiente estructura selectiva:

```
case "$SO" in
     AIX)   echo –n "$US: "
            lsuser -ca expires $US|fgrep -v "#"|cut -f2 -d:`"
           ;;
     SunOS) echo "$US: `logins -aol $US|cut -f7 -d:`"
           ;;
     Linux) echo "$US: `chage -l $US|grep Account|cut -f2 -d:`"
           ;;
     *)     echo "Sistema operativo desconocido" ;;
esac
```

7.3. Bucles.

Los bucles son estructuras reiterativas que se ejecutan repetitivamente, para no tener que teclear varias veces un mismo bloque de código. Un bucle debe tener siempre una condición de salida para evitar errores provocados por bucles infinitos.

La siguiente tabla describe las 2 órdenes especiales que pueden utilizarse para romper el modo de operación típico de un bucle.

Orden	Descripción
`break`	**Ruptura inmediata de un bucle** (debe evitarse en programación estructurada para impedir errores de lectura del código).
`continue`	**Salto a la condición del bucle** (debe evitarse en programación estructurada para impedir errores de lectura del código).

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

7.3.1. Bucles genéricos.

Los bucles genéricos de tipos “para cada” ejecutan el bloque de código para cada valor asignado a la variable usada como índice del bucle o a su expresión de control. Cada iteración debe realizar una serie de operaciones donde dicho índice varíe, hasta la llegar a la condición de salida.

El tipo de bucle **`for`** más utilizado es aquél que realiza una iteración por cada palabra (o cadena) de una lista. Si se omite dicha lista de valores, se realiza una iteración por cada parámetro posicional.

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 los formatos de los bucles iterativos genéricos (de tipo "para cada") interpretados por BASH.

Bucle	Descripción
for *Var* [**in** *Lista*]; **do** *Bloque* **done**	**Bucle iterativo**: se ejecuta el bloque de mandatos del bucle sustituyendo la variable de evaluación por cada una de las palabras incluidas en la lista.
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 mandatos y *Exp3* (las 3 expresiones deben ser aritméticas).

Véanse algunos ejemplos:

```
# Se asigna a la variable "library" el camino de cada uno de
#    los archivos indicados en la expansión de ficheros y se
#    realizan las operaciones indicadas en el bloque de código.
for library in /lib/kernel/$(uname -r)/libredhat-kernel.so* ; do
        ln -s -f $library /lib/
        ldconfig -n /lib/
done
...
# Se establece un contador de hasta 20 iteraciones para
#    ejecutar el bucle.
for (( times = 1; times < 20; times++ )); do
    /usr/sbin/rpcinfo -p | grep ypbind > /dev/null 2>&1 && \
            ypwhich > /dev/null 2>&1
done
```

Y el ejercicio a evaluar:

```
for US in `cut -sf2 -d: /home/cdc/*.lista`; do
    grep "^$US:" /etc/shadow | cut -sf1-2 -d: >>$FICHTEMP
done
```

7.3.2. Bucles condicionales "mientras" y "hasta".

Los bucles condicionales evalúan la expresión en cada iteración del bucle y dependiendo del resultado se vuelve a realizar otra iteración o se sale a la instrucción siguiente.

La siguiente tabla describe los formatos para los 2 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 y 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
        processdir "$1"
        shift
done
```

Se proponen como ejercicios convertir este bucle **`while`** en un bucle **`until`** y en uno **`for`** de estilo C.

```
i=5
while [ $i -ge 0 ] ; do
        if [ -f /var/log/ksyms.$i ] ; then
                mv /var/log/ksyms.$i /var/log/ksyms.$(($i+1))
        fi
        i=$(($i-1))
done
```

7.3.3. Bucle de selección interactiva.

La estructura **`select`** no es propiamente dicho un bucle de programación estructurada, ya que 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. Este tipo de bucles no suele utilizarse.

La siguiente tabla describe el formato del bucle interactivo.

Bucle	Descripción
select *Var* [**in** *Lista*]; **do** *Bloque1* ... **done**	**Bucle de 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.

8. 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 llamada de una función..

Declaración	LLamada
[**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 del siguiente cuadro 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
# comprus - comprueba la existencia de usuarios en listas y en
#           el archivo de claves (normal y NIS).
#           Uso: comprus ? | cadena
#                     ?: ayuda.
# Ramón Gómez - Septiembre 1.998.

# 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
```

Para terminar el capítulo se propone desarrollar en BASH:

- Una función que reciba un único parámetro, una cadena de caracteres, y muestre el último carácter de dicha cadena. Debe tenerse en cuenta que el primer carácter de una cadena tiene como índice el valor 0.
- Una función que reciba una cadena de caracteres y que la imprima centrada en la pantalla. Tener en cuenta que el ancho del terminal puede estar definido en la variable de entorno `COLUMNS`, en caso contrario usar un valor por defecto de 80 columnas.

9. Características especiales

En última instancia se van a describir algunas de las características especiales que el intérprete BASH añade a su lenguaje para mejorar la funcionalidad de su propia interfaz y de los programas que interpreta.

Seguidamente se enumeran algunas de las características especiales de BASH y en el resto de puntos de este capítulo se tratan en mayor profundidad las que resultan más interesantes:

- Posibilidad de llamar al intérprete BASH con una serie de opciones que modifican su comportamiento normal.
- Mandatos para creación de programas interactivos.
- Control de trabajos y gestión de señales.
- Manipulación y personalización del punto indicativo.
- Soporte de alias de comandos.
- Gestión del histórico de órdenes ejecutadas.
- Edición de la línea de mandatos.
- Manipulación de la pila de últimos directorios visitados.
- Intérprete de uso restringido, con características limitadas.
- Posibilidad de trabajar en modo compatible con la norma POSIX 1003.2 [2].

9.1. Programas interactivos.

BASH proporciona un nivel básico para programar "*shellscripts*" interactivos, soportando instrucciones para solicitar y mostrar información al usuario.

Las órdenes internas para dialogar con el usuario se describen en la siguiente tabla.

Mandato	Descripción
read [**-p** *"Cadena"*] [*Var1 ...*]	**Asigna la entrada a variables**: lee de la entrada estándar y asigna los valoras a las variables indicadas en la orden. Puede mostrarse un mensaje antes de solicitar los datos. Si no se especifica ninguna variable, **REPLY** contiene la línea de entrada.
echo [**-n**] *Cadena*	**Muestra un mensaje**: manda el valor de la cadena a la salida estándar; con la opción `-n` no se hace un salto de línea.

printf *Formato Parám1 ...*	**Muestra un mensaje formateado**: equivalente a la función `printf` del lenguaje C, manda un mensaje formateado a la salida normal, permitiendo mostrar cadena y números con una longitud determinada.

Véanse los siguientes ejemplos.

```
# 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
...
# Muestra los usuarios y nombres completos
#   (la modificación de la variable IFS sólo afecta al bucle)
while IFS=: read usuario clave uid gid nombre ignorar
do
  printf "%8s (%s)\n" $usuario $nombre
done </etc/passwd
```

Como ejercicio se propone explicar paso a paso el funcionamiento del bucle anterior y los valores que se van asignando a las distintas variables en cada iteración.

9.2. Control de trabajos.

Las órdenes para el control de trabajos permiten manipular los procesos ejecutados por el usuario o por un guión BASH. El usuario puede manejar varios procesos y subprocesos simultáneamente, ya que el sistema asocia un número identificador único a cada uno de ellos.

Los procesos asociados a una tubería tienen identificadores (PID) propios, pero forman parte del mismo trabajo independiente.

La siguiente tabla describe las instrucciones asociadas con el tratamiento de trabajos.

Mandato	Descripción
Mandato **&**	**Ejecución en 2º plano**: lanza el proceso de forma desatendida, con menor prioridad y con la posibilidad de continuar su ejecución tras la desconexión del usuario.
bg *%NºTrabajo*	**Retorna a ejecutar en 2º plano**: continúa la ejecución desatendida de un procesos suspendido.
fg *%NºTrabajo*	**Retorna a ejecutar en 1er plano**: vuelve a ejecutar el proceso asociado al trabajo indicado de forma interactiva.
jobs	**Muestro los trabajos en ejecución**, indicando el nº de trabajo y los PID de sus procesos.

kill *Señal PID1|%Trab1 ...*	**Manda una señal a procesos o trabajos**, para indicar una excepción o un error. Puede especificarse tanto el nº de señal como su nombre.
suspend	**Para la ejecución del proceso**, hasta que se recibe una señal de continuación.
trap [*Comando*] [*Señal1 ...*]	**Captura de señal**: cuando se produce una determinada señal (interrupción) en el proceso, se ejecuta el mandato asociado. Las señales especiales `EXIT` y `ERR` se capturan respectivamente al finalizar el "*script*" y cuando se produce un error en una orden simple.
wait [*PID1|%Trab1*]	**Espera hasta que termine un proceso o trabajo**: detiene la ejecución del proceso hasta que el proceso hijo indicado hay finalizado su ejecución.

Revisar los siguientes ejemplos.

```
# Se elimina el fichero temporal si en el programa aparecen las
3    señales 0 (fin), 1, 2, 3 y 15.
trap 'rm -f ${FICHTEMP} ; exit' 0 1 2 3 15
...
# Ordena de forma independiente las listas de ficheros de
#   usuarios, espera a finalizar ambos procesos y compara
#   los resultados.
(cat alum03.sal prof03.sal pas03.sal | sort | uniq > usu03) &
(cat alum04.sal prof04.sal pas04.sal | sort | uniq > usu04) &
wait
diff usu03 usu04
```

9.3. Intérprete de uso restringido.

Cuando el administrador del sistema asigna al usuario un intérprete de uso restringido, éste último utiliza un entorno de operación más controlado y con algunas características limitadas o eliminadas.

Existen 3 formas de ejecutar BASH en modo restringido:

```
rbash
bash –r
bash --restricted
```

BASH restringido modifica las siguientes características [2]:

- No se puede usar la orden **cd**.
- No se pueden modificar las variables `SHELL`, `PATH`, `ENV` ni `BASH_ENV`.

- No se pueden ejecutar mandatos indicando su camino, sólo se ejecutarán aquellos que se encuentren en los directorios especificados por el administrador..
- No se pueden especificar caminos de ficheros como argumento del mandato “**.**”.
- No se pueden añadir funciones nuevas en los ficheros de inicio.
- No se admiten redirecciones de salida.
- No se puede reemplazar el intérprete RBASH (no se puede ejecutar la orden **exec**).
- No se pueden añadir o quitar mandatos internos.
- No se puede modificar el modo de operación restringido.

10. Referencias.

1. B. Fox, C. Ramey: *"BASH(1)" (páginas de manuales de BASH v2.5a)"*. 2.001.
2. C. Ramey, B. Fox: "*Bash Reference Manual, v2.5a*". Free Software Foundation, 2.001.
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 de Sistemas Linux Red Hat"*. Secretariado de Formación Permanente del PAS (Universidad de Sevilla), 2.002.

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/