El shell Bash

Acerca de este documento

En este tutorial pretendemos enseñar el manejo de Bash, el Bourne Again Shell de GNU. Este shell es el que proporcionan por defecto muchos sistemas UNIX entre ellos Mac OS X o Linux. Los ejemplos se explicarán sobre Mac OS X, pero debido a la interoperatividad que caracteriza a Bash, estos ejemplos deberían ser exactamente igual de útiles en otros sistemas UNIX. Cuando existan diferencias las indicaremos para que usuarios de otros sistemas puedan seguir correctamente este documento.

El tutorial asume que el lector conoce los aspectos más básicos de qué es, y para qué sirve un terminal. No pretendemos enseñar cuales son los muchos y útiles comandos a los que podemos acceder, sólo pretendemos centrarnos en el manejo, personalización y programación de scripts con el shell Bash. Aun así, a lo largo del documento comentaremos gran cantidad de comandos que están relacionados con el shell, y que ayudan a hacer que los ejemplos resulten útiles.

Al acabar este tutorial el lector debería de haber aprendido a usar las principales teclas rápidas, personalizar mucho más su terminal para hacerlo más manejable, y modificar o crear los scripts que configuran su sistema.

Nota legal

Madrid, Septiembre 2005
Para cualquier aclaración contacte con:
fernando@macprogramadores.org

Tabla de contenido

TEMA 1: Introducción a Bash

TEMA 2: Combinaciones de teclas

TEMA 3: Personalizar el entorno

TEMA 4: Programación básica del shell

TEMA 5: Control de flujo

TEMA 6: Opciones de línea de comandos, expresiones aritméticas y arrays

TEMA 7: Redicecciones

TEMA 8: Control de procesos

TEMA 9: Depurar scripts

Referencias

Tema 1

Introducción a Bash

Sinopsis:

Como se justifica en el acerca de, este tutorial va a omitir los aspectos más básicos del shell que es normal conocer por parte de cualquier persona que haya usado mínimamente un shell UNIX.

En este primer tema vamos a repasar un conjunto de temas básicos que, aunque en parte puede conocer el lector, creemos que conviene aclarar antes de profundizar.

Por consiguiente, recomendamos empezar leyendo este primer tema, ya que sino pueden quedar ciertos aspectos sin concretar que luego podrían hacer falta para seguir más cómodamente las explicaciones.

Debido a sus objetivos, este tema está escrito avanzando de forma considerablemente más rápida y superficial que en resto de temas.

1. El shell que estamos usando

Mac OS X trae preinstalado el shell Bash desde la versión 10.2, antes traía instalado el shell tcsh, pero debido a que Bash es el shell que GNU eligió para el software libre, Apple decidió dar el salto. Linux lógicamente también usa este shell, con lo cual parece ser que Bash es el shell de los sistemas UNIX más utilizados, y tiene un futuro muy prometedor.

Si queremos saber que versión de shell tenemos instalado podemos usar el comando:

```
$ echo $SHELL
/bin/bash
```

Este comando nos indica que shell estamos usando y en que directorio está instalado.

Si queremos conocer la versión de Bash podemos usar el comando:

```
$ echo $BASH_VERSION
2.05b.0(1)-release
```

También podemos saber donde está instalado Bash con el comando:

```
$ whereis bash
/bin/bash
```

Puede conocer todos los shell de que dispone su máquina con el comando:

```
$ cat /etc/shells
/bin/bash
/bin/csh
/bin/sh
/bin/tcsh
/bin/zsh
```

Si por alguna razón no está usando Bash, pero lo tiene instalado (o lo acaba de instalar) en su máquina, puede hacer que Bash sea el shell por defecto de su cuenta usando el comando:

```
$ chsh -s /bin/bash
```

Si prefiere usar una versión más moderna de shell que la que viene preinstalada con Mac OS X puede bajársela del proyecto Fink:

```
$ fink list bash
Information about 4975 packages read in 12 seconds.
 bash            3.0-2      The GNU Bourne Again Shell
 bash-completion 20041017-1 Command-line completions ...
 bash-doc        3.0-1      Extra documentation for ...

$ fink install bash
```

Y cambiar a este shell con:

```
$ chsh -s /sw/bin/bash
```

Pero antes deberá introducir este shell en `/etc/shells`, o `chsh` no se lo aceptará como un shell válido.

Si ahora nos logamos de nuevo con el comando `login` y preguntamos por la versión de Bash:

```
$ echo $BASH
/sw/bin/bash
$ echo $BASH_VERSION
3.00.0(1)-release
```

Vemos que estamos trabajando con Bash 3.0. En este tutorial supondremos que tenemos la versión 3.0 de Bash, si alguien está usando la versión 2.05, o alguna anterior, puede que no le funcionen todos los ejemplos que hagamos.

2. Expansión de nombres de ficheros y directorios

2.1. Los comodines

Para referirnos a varios ficheros es muy típico usar los comodines de la Tabla 1.1. Un sitio típico donde se usan los comodines es el comando `ls`. Este comando sin argumentos lista todos los ficheros del directorio, pero le podemos pasar como argumentos los nombres de los ficheros que queremos listar:

```
$ ls carta.txt leeme.txt
```

Si lo que le damos son los nombres de uno o más directorios lo que hace es listar su contenido.

Comodín	Descripción
?	Uno y sólo un carácter
*	Cero o más caracteres
[*conjunto*]	Uno los caracteres de *conjunto*
[!*conjunto*]	Un carácter que no este en *conjunto*

Tabla 1.1: Comodines de fichero

Muchas veces queremos referirnos a un conjunto de ficheros para lo cual usamos comandos de la forma:

```
$ ls *.txt
```

Que lista todos los ficheros acabados en `.txt`.

* representa cero o más caracteres, con lo que `*ed` encontraría el fichero `ed`.

Otro comodín menos usado es `?` que sustituye por un sólo carácter, por ejemplo:

```
$ ls carta?.txt
```

Listaría ficheros como `carta1.txt`, `carta2.txt`, pero no `carta10.txt`.

El tercer comodín permite indicar un conjunto de caracteres que son válidos para hacer la sustitución, p.e. `c[ao]sa` encontraría el fichero `casa` y `cosa`, pero no `cesa`. Además podemos indicar un conjunto de caracteres ASCII consecutivos, por ejemplo `[a-z]` serían todas las letras minúsculas, `[!0-9]`

serían todos los caracteres ASCII excepto los dígitos, y `[a-zA-Z0-9]` serían todas las letras mayúsculas, minúsculas y los dígitos.

La razón por la que este comodín no ha sido tan usado como se esperaba es que expande por un, y sólo un dígito, por ejemplo `programa.[co]` encontraría `programa.c` y `programa.o`, pero no `programa.cpp`.

Es importante tener en cuenta que los comandos cuando se ejecutan no ven los comodines sino el resultado de la expansión. Por ejemplo si ejecutamos el comando:

```
$ cp g* /tmp
```

`g*` se expande por todos los ficheros que cumplen el patrón, y esto es lo que se pasa al comando `cp`, pero si no existiera ningún fichero cuyo nombre cumpliese el patrón `g*`, este valor no se expande sino que se pasa tal cual al comando, y éste será el que fallará:

```
$ cp g* /tmp/
cp: g*: No such file or directory
```

Es decir, como podríamos pensar, al fallar no se pasa una lista vacía al comando. Piense por un momento lo que ocurriría con algunos comandos si se hubiese diseñado así.

Este funcionamiento es ligeramente distinto al de tcsh, donde si no se expande el comodín no se ejecuta el comando, en Bash se ejecuta el comando aunque luego éste produzca un error.

2.2. El comodín tilde

El comodín tilde `~` se usa para referirse al directorio home de los usuarios (`/Users` en Mac OS X o `/home` en la mayoría de las máquinas UNIX), por ejemplo si usamos `~carol/carta.txt` nos lo expande por `/Users/carol/carta.txt`.

Además podemos usar el comodín tilde para referirnos a nuestro propio directorio, el cuyo caso debemos de precederlo por una barra, p.e. `~/carta.txt` se expande por el nombre de mi directorio, en mi caso `/Users/fernando/carta.txt`.

Observe la diferencia entre poner la barra y no ponerla, si no la hubiera puesto (hubiera puesto `~carta.txt`), me habría expandido por la ruta `/Users/carta.txt`, y si no existe un usuario con el nombre `carta.txt` hubiera producido un error indicando que no existe el directorio.

2.3. El comodín llaves

El comodín llaves, a diferencia de los anteriores, no estudia el nombre de los ficheros existentes en disco para nada, simplemente expande una palabra por cada una de las cadenas de caracteres que contiene, por ejemplo:

```
$ echo c{ami,ontamina,}on
camion contaminaon con
```

Es posible incluso anidarlo y genera el producto cartesiano de combinaciones:

```
$ echo c{a{min,nt}a,ose}r
caminar cantar coser
```

En el apartado 2.1 comentamos que un problema que tenía el comodín corchete es que expandía por un y sólo un carácter, lo cual era problemático si queríamos referirnos por ejemplo a todos los ficheros de un programa, ya que `*.[coh]` nos permite referirnos a los fichero `.c`, `.o` y `.h`, pero no a los `.cpp`.

Usando el comodín llave podemos superar esta dificultad: `*.{h,c,cpp,o}` espadería en todos los ficheros con estas extensiones, aunque ahora surge un nuevo problema y es que, debido a que el comodín llave no mira que ficheros hay en disco, si no existe un fichero con alguna de las extensiones indicadas, la expansión del `*` no se produce, y encontraríamos un mensaje de error. Por ejemplo:

```
$ ls *.{h,c,cpp,o}
ls: *.c: No such file or directory
ls: *.o: No such file or directory
clave.cpp       clave.h
```

Se puede usar `..` para hacer algo similar a lo que hacen los corchetes obteniendo todos lo caracteres ASCII entre dos letras. Por ejemplo:

```
$ echo l{a..e}
la lb lc ld le
```

Obsérvese que, en el caso de los corchetes, lo que obtendríamos no son un conjunto de cinco palabras, sino una expansión por un fichero existente:

```
$ echo cl[a-e]ve.h
clave.h
```

O la cadena sin expandir si no se encuentra el fichero:

```
$ echo cl[e-i]ve.h
```

```
cl[e-i]ve.h
```

Por último comentar que la llave debe contener al menos dos cadenas, sino no se realiza la expansión:

```
$ echo ca{a}sa
ca{a}sa
```

De nuevo este comportamiento difiere con el de tcsh, donde la expansión se realiza aunque haya una sola cadena dentro de las llaves.

2.4. Comodines extendidos

Bash permite usar un conjunto de comodines extendidos, pero para poder usarlos debemos de activar la opción `ext_glob` de Bash (véase el apartado 3 del Tema 3) con el comando:

```
$ shopt -s extglob
```

En este caso se pueden usar uno de estos cinco nuevos tipos de patrones:

`?(`*`pattern-list`*`)`
: Cero o una ocurrencia de *`pattern-list`*

`*(`*`pattern-list`*`)`
: Cero o más ocurrencias de *`pattern-list`*

`+(`*`pattern-list`*`)`
: Una o más ocurrencias de *`pattern-list`*

`@(`*`pattern-list`*`)`
: Exactamente uno de los patrones de la lista

`!(`*`pattern-list`*`)`
: Cualquier cosa excepto uno de los patrones de la lista

`pattern-list` recibe uno o más patrones separados por `|`. Cada patrón de esta lista puede contener comodines, por ejemplo `+([0-9])` busca cadenas formadas por uno o más dígitos.

En el apartado 2.1 vimos que un problema que presentaba el comodín `?` era que `carta?.txt` listaría ficheros como `carta1.txt`, `carta2.txt`, pero no `carta10.txt`. Esto lo podemos solucionar con el comodín extendido `+(`*`pattern-list`*`)` de la forma: `carta+([0..9]).txt`

También vimos en el apartado 2.1 que `*.[cho]` encontraría los ficheros con extensión `.c`, `.o` y `.h`, pero no había forma de encontrar los `.cpp` ya que el corchete sólo aceptaba un carácter. Ahora podemos usar el comodín `@(`*`pattern-list`*`)` para indicar la lista de extensiones a aceptar. Por ejemplo `*.@(c|o|h|cpp)` encontraría correctamente estos ficheros:

```
$ ls *.@(c|o|h|cpp)
clave.cpp  clave.h
```

También hubiera sido equivalente usar `@(*.c|*.o|*.h|*.cpp)` ya que los patrones pueden estar anidados.

Si lo que hubiéramos querido es encontrar todos los ficheros excepto los `.gif`, los `.jpg` y los `.html` podríamos haber usado el patrón `!(*.html|*gif|*jpg)`. Sin embargo, en este caso no podríamos haber usado `*.!(html|gif|jpg)`

Un último ejemplo, si queremos borrar todos los ficheros excepto los que empiezan por `vt` seguido por uno o más dígitos podemos usar el comando:

```
$ rm !(vt+([0-9]))
```

3. Los comandos internos de Bash

Bash busca los comandos a ejecutar en los directorios indicados en la variable de entorno `$PATH`, pero además existen una serie de comandos que no corresponden a ficheros del disco duro, sino que son internos a Bash y están siempre cargados en su memoria.

Ejemplos de estos comandos son `cd`, `chdir`, `alias`, `set` o `export`. Puede obtener una lista completa de estos comandos con su descripción ejecutando:

```
$ man builtin
```

Y puede obtener ayuda de estos comandos usando el comando `help`:

```
$ help alias
alias: alias [-p] [name[=value] ... ]
`alias' with no arguments or with the -p option prints
the list of aliases in the form alias NAME=VALUE on
standard output.
Otherwise, an alias is defined for each NAME whose VALUE
is given.
A trailing space in VALUE causes the next word to be
checked for alias substitution when the alias is
expanded.  Alias returns true unless a NAME is given for
which no alias has been defined.
```

4. Redirecciones y pipes

4.1. Operadores de redirección

UNIX está basado en una idea muy simple pero muy útil: Tratar todos las entrada y salidas como streams (flujos) de bytes.

Cada programa va a tener asociadas siempre una entrada estándar (por defecto el teclado), una salida estándar (por defecto la consola), y una salida de errores estándar (por defecto también la consola).

Si queremos, podemos cambiar la entrada estándar para que el programa reciba datos de un fichero usando el **operador de redirección** <. Por ejemplo el comando `cat`, si no recibe argumentos, lee del teclado por la entrada estándar y lo pasa a la salida estándar:

```
$ cat
Esto es una linea acabada en intro
Esto es una linea acabada en intro
^D
```

Podemos indicar el final de un stream desde el teclado con la combinación de teclas Ctrl+D como se muestra en el ejemplo.

Podemos cambiar la entrada estándar de `cat` para que lea de un fichero con:

```
$ cat < clave.h
#ifndef CLAVE_H_
.....
```

En el caso concreto del comando `cat`, también puede recibir como argumento el nombre del fichero a pasar a la salida estándar, con lo que en el caso del comando `cat` nos podríamos haber ahorrado el operador <:

```
$ cat clave.h
#ifndef CLAVE_H_
.....
```

UNIX dispone de un gran número de comandos que leen de la entrada estándar, realizan una operación con el texto, y escriben en la salida estándar (o en la salida de errores si se produce un error): `cat`, `grep`, `soft`, `cut`, `sed`, `tr`,...

El **operador de redirección de salida** > permite cambiar la salida estándar de un comando, por ejemplo:

```
$ date > ahora
```

Envía el día y hora actuales al fichero `ahora`.

También podemos cambiar a la vez la entrada y salida estándar de un programa usando ambos operadores de redirección. Por ejemplo:

```
$ cat < ficheroa > ficherob
```

También podemos cambiar la salida de errores estándar con el operador de redirección `2>`. Por ejemplo:

```
$ cat < ficheroa > ficherob 2>errores
```

Copia el `ficheroa` en el `ficherob`, y si se produce algún error lo escribe en el fichero `errores`.

Si no queremos sobrescribir un fichero de salida sino añadir el contenido al final podemos usar el operador de redirección `>>` para la salida estándar o `2>>` para los errores estándar. Por ejemplo:

```
$ ls p* >>ficheros 2>>errores
```

Añadiría los ficheros que lista `ls` al fichero `ficheros`, y si se produjesen errores los añadiría al fichero `errores`. El operador de redirección `2>>` es especialmente útil para almacenar los conocidos logs de errores.

Muchas veces no se quiere que un programa muestre mensajes en la consola del usuario, en este caso es muy común redirigir su salida estándar y salida de errores estándar al fichero `/dev/null`:

```
$ gcc *.cpp > /dev/null 2> /dev/null
```

4.2. Pipes

Es posible redirigir la salida estándar de un programa a la entrada estándar de otro usando el operador `|` (pipeline).

`more` es uno de los comandos típicos que lo usan. Este comando lo que hace es recoger la entrada estándar y irla mostrando poco a poco (página a página), luego si por ejemplo tenemos un directorio con muchos ficheros podemos hacer:

```
$ ls -la | more
```

y se irán mostrando página a página los ficheros.

Según vayamos avanzando podremos ir viendo ejemplos más complejos, por ejemplo:

```
$ cut -d: -f1 < /etc/passwd | sort
```

Nos muestra los nombres de todos los usuarios de la máquina ordenados alfabéticamente.

Téngase en cuenta que el operador `|` separa el comando en varios comandos antes de ejecutarlo, con lo que el operador de redirección tiene efecto sólo para el comando `cut`.

5. Ejecución secuencial y concurrente de comandos

Podemos ejecutar un comando que tarde mucho en ejecutarse y dejarlo ejecutando en background precediéndolo por `&`. Por ejemplo para compilar un conjunto de ficheros fuente de un programa C++ podemos hacer:

```
$ gcc *.cpp &
```

Aunque el proceso se sigue ejecutando en background, los mensajes que produce salen en la consola impidiéndonos trabajan cómodamente. Para evitarlo podemos enviar los mensajes a `/dev/null`:

```
$ gcc *.cpp > /dev/null &
```

Aunque si se produce un error, éste irá a la salida de errores estándar, con lo que seguiría saliendo en consola. Podríamos evitarlo redirigiendo también la salida de errores estándar, pero quizá sea mejor que se nos informase del error.

Otras veces lo que queremos es esperar a que se acabe de ejecutar un comando para ejecutar el siguiente, en este caso podemos usar el operador `;` (punto y coma), por ejemplo, podríamos querer compilar el comando `clave` para luego ejecutarlo:

```
$ gcc clave.cpp -o clave ; clave
```

Este comando primero compila el programa, y cuando acaba de compilarlo lo ejecuta.

6. Caracteres especiales y entrecomillado

Los caracteres <, >, |, & *, ? , ~, [,], {, } son ejemplos de caracteres especiales para Bash que ya hemos visto. La Tabla 1.2 muestra todos los caracteres especiales de Bash.

Más adelante veremos otros comandos tienen sus propios caracteres especiales, como puedan ser los comandos que usan expresiones regulares o los operadores de manejo de cadenas.

Carácter	Descripción
~	Directorio home
`	Sustitución de comando
#	Comentario
$	Variable
&	Proceso en background
;	Separador de comandos
*	Comodín 0 a n caracteres
?	Comodín de un sólo carácter
/	Separador de directorios
(	Empezar un subshell
)	Terminar un subshell
\	Carácter de escape
<	Redirigir la entrada
>	Redirigir la salida
|	Pipe
[	Empieza conjunto de caracteres comodín
]	Acaba conjunto de caracteres comodín
{	Empieza un bloque de comando
}	Acaba un bloque de comando
'	Entrecomillado fuerte
"	Entrecomillado débil
!	No lógico de código de terminación

Tabla 1.2: Caracteres especiales de Bash

6.1. Entrecomillado

Aunque los caracteres especiales son muy útiles para Bash, a veces queremos usar un carácter especial literalmente, es decir sin su significado especial, en este caso necesitamos **entrecomillarlo** (quoting).

Por ejemplo si queremos escribir en consola el mensaje: `2*3>5 es una expresión cierta`, podemos usar el comando `echo` así:

```
$ echo 2*3>5 es una expresion cierta
```

Al ejecutarlo podemos observar que no da ninguna salida, pero realmente ha creado el fichero 5 con el texto `2*3 es una expresion cierta`. La razón está en que `>5` ha sido entendido como redirigir al fichero `5`, y además se ha intentado ejecutar el carácter especial `*`, pero al no encontrar ningún fichero que cumpliera el patrón no se ha expandido y se ha pasado el parámetro a `echo` tal cual.

Sin embargo si entrecomillamos usando `'`, el carácter de **entrecomillado fuerte** obtenemos el resultado esperado:

```
$ echo '2*3>5 es una expresion cierta'
2*3>5 es una expresion cierta
```

Un ejemplo más práctico del entrecomillado es el comando `find` que nos permite buscar ficheros por muchos criterios. Por ejemplo para buscar por nombre usa el argumento `-name` *`patron`*. Si por ejemplo queremos buscar todos los ficheros `.c` en nuestra máquina podemos intentar hacer:

```
$ find / -name *.c
```

Pero si tenemos la mala suerte de que en el directorio actual exista algún fichero `.c`, sustituirá el comodín y buscará el nombre de el/los ficheros de nuestro directorio actual en el disco duro. Para evitarlo es recomendable entrecomillarlo:

```
$ find / -name '*.c'
```

En la Tabla 1.2 aparece el **entrecomillado débil** ", el cual pasa sólo por algunos de los pasos del shell, o dicho con otras palabras, interpreta sólo algunos caracteres especiales. Más adelante veremos cuando es preferible usar este tipo de entrecomillado, de momento usaremos sólo el entrecomillado fuerte.

6.2. Caracteres de escape

Otra forma de cambiar el significado de un carácter de escape es precederlo por `\`, que es lo que se llama el **carácter de escape**.

Por ejemplo en el ejemplo anterior podríamos haber hecho:

```
$ echo 2\*3\>5 es una expresion cierta
2*3>5 es una expresion cierta
```

Donde hemos puesto el carácter de escape a los caracteres especiales para que Bash no los interprete.

También este carácter se usa para poder poner espacios a los nombres de ficheros, ya que el espacio es interpretado por Bash como un separador de argumentos de la línea de comandos, si queremos que no considere el espacio como un cambio de argumento podemos preceder el espacio por `\`. Por ejemplo el fichero `Una historia.avi` lo reproduzco en mi ordenador con `mplayer` así:

```
$ mplayer Una\ historia.avi
```

Si queremos que Bash no interprete el carácter de escape podemos entrecomillarlo `'\'`, o bien hacer que él se preceda a si mismo `\\`.

6.3. Entrecomillar los entrecomillados

Podemos usar el carácter de escape para que no se interpreten los entrecomillados simples o dobles, es decir:

```
$ echo \"2\*3\>5\" es una expresion cierta
"2*3>5" es una expresion cierta
$ echo \'2\*3\>5\' es una expresion cierta
'2*3>5' es una expresion cierta
```

6.4. Texto de varias líneas

Otro problema es como escribir un comando que ocupa varias líneas. Bash nos permite utilizar el carácter de escape para ignorar los retornos de carro de la siguiente forma:

```
$ echo En un lugar de la Mancha de cuyo nombre no \
> quiero acordarme no a mucho tiempo vivía un \
> hidalgo caballero.
En un lugar de la Mancha de cuyo nombre no quiero acordar
no a mucho tiempo vivía un hidalgo caballero.
```

Al pulsar intro Bash nos devuelve el segundo prompt, que por defecto es `>`, y nos permite seguir escribiendo.

Una segunda opción es entrecomillar y no cerrar las comillas, con lo que Bash nos pide que sigamos escribiendo sin necesidad de poner un carácter de escape a los retornos de carro:

```
$ echo 'En un lugar de la Mancha de cuyo nombre no
```

```
> quiero acordarme no a mucho tiempo vivía un
> hidalgo caballero.'
En un lugar de la Mancha de cuyo nombre no
quiero acordar no a mucho tiempo vivía un
hidalgo caballero.
```

La diferencia está en que en el primer caso los retornos de carro son eliminados, y en el segundo caso no.

Tema 2

Combinaciones de teclas

Sinopsis:

Nada es más útil a la hora de manejar de forma frecuente un programa que saber manejar un buen conjunto de combinaciones de teclas que nos ayuden a realizar operaciones frecuentes rápidamente.

En el caso de Bash es fácil identificar cuando un usuario conoce estas combinaciones de teclas: Nos sorprende ver con que velocidad escribe en su consola.

En este tema comentaremos las combinaciones de teclas más útiles. Le recomendamos que intente recordarlas y que vuelva a consultar este tema cuantas veces haga falta hasta que las utilice de forma mecánica, verá como aumenta su productividad.

1. El historial de comandos

Bash mantiene un histórico de comandos el cual lee al arrancar del fichero situado en el directorio home `.bash_history`, durante su ejecución va almacenando en memoria los comandos ejecutados, y al acabar la sesión los escribe al final del fichero `.bash_history`.

En principio podemos modificar el fichero que usa Bash para guardar el histórico indicándolo en la variable de entorno `BASH_HISTORY`. Aunque raramente será útil hacer esto.

Puede usar el comando `history` para obtener un listado de su historial.

La forma más cómoda de referirse a comandos previamente utilizados es usar las teclas del cursor flecha arriba y flecha abajo. Si aún no las conocía pruébelas.

En los siguientes subapartados comentaremos más a fondo como aprovechar al máximo su histórico de comandos.

1.1. El comando `fc`

El comando `fc` (fix command) es un clásico del C Shell traído al mundo de Bash. `fc` es un comando tan importante que es un comando interno de Bash que podemos usar para examinar los últimos comandos, y para editar uno de ellos.

Al ser un comando interno no use `man` para obtener información sobre él, utilice el comando:

```
$ help fc
```

Podemos obtener un listado de los últimos comandos usados junto con su numero usando:

```
$ fc -l
```

Vemos que este comando es equivalente a `history`, excepto que lista sólo los últimos comandos del historial y no todos. Para editar uno de ellos indicamos su número a `fc`:

```
$ fc 42
```

`fc` abre el comando en el editor que tengamos predefinido en la variable `EDITOR` y nos permite editarlo.

El principal problema de diseño que tiene este comando es que al acabar de editar el comando (al salir del editor) lo ejecuta, comportamiento que puede ser muy peligroso dependiendo del comando que hayamos editado.

1.2. Ejecutar comandos anteriores

La Tabla 2.1 muestra otro conjunto de comandos internos de Bash (también heredados de C Shell) que permiten ejecutar comandos del histórico.

Comando	Descripción
`!!`	Ejecutar el último comando
`!n`	Ejecutar el comando número *n*
`!cadena`	Ejecutar el último comando que empiece por *cadena*

Tabla 2.1: Comandos internos para ejecutar comandos del histórico

El más útil de ellos es `!cadena`, ya que con dar las primeras letras de un comando que hayamos ejecutado previamente lo busca en el historial y lo ejecuta. Tengase en cuenta que ejecuta el comando más reciente que empiece por *cadena*.

Si hemos consultado el historial, con `history` o `fc -l`, podemos ejecutar un comando por número usando `!n`.

2. Las teclas de control del terminal

Las teclas de control del terminal son combinaciones de teclas que interpreta el terminal (no Bash).

Podemos preguntar por las teclas de control del terminal usando el comando:

```
$ stty all
speed 9600 baud; 24 rows; 80 columns;
lflags: icanon isig iexten echo echoe -echok echoke
        -echonl echoctl -echoprt -altwerase -noflsh
        -tostop -flusho pendin -nokerninfo -extproc
iflags: -istrip icrnl -inlcr -igncr ixon -ixoff ixany
imaxbel -ignbrk brkint -inpck -ignpar -parmrk
oflags: opost onlcr -oxtabs -onocr -onlret
cflags: cread cs8 -parenb -parodd hupcl -clocal -cstopb
        -crtscts -dsrflow -dtrflow -mdmbuf
discard dsusp   eof     eol     eol2    erase   intr
^O      ^Y      ^D      <undef> <undef> ^?      ^C
kill    lnext min     quit    reprint start   status
^U      ^V    1       ^\      ^R      ^Q      <undef>
stop    susp    time    werase
^S      ^Z      0       ^W
```

El comando, aparte de darnos ciertos flags sobre distintos aspectos del comportamiento del terminal, nos muestra las combinaciones de teclas que tenemos asignadas a las tareas más frecuentes (recuérdese que ^X significa Ctrl+X). Algunas de estas combinaciones de teclas están obsoletas, y no merece mucho la pena recordarlas, pero otras, las que se muestran en la Tabla 2.2, si que son muy útiles:

Combinación de tecla	**Nombre** `stty`	**Descripción**
Ctrl+C	intr	Para el comando actual
Ctrl+\	quit	Fuerza la parada del comando actual (usar si Ctrl+C no responde)
Ctrl+D	eof	Final del flujo de entrada
Ctrl+U	kill	Borra desde la posición actual al principio de la línea
Ctrl+W	werase	Borra desde la posición actual al principio de la palabra

Tabla 2.2: Teclas de control del terminal

Para detener un comando es muy típico usar la combinación de teclas Ctrl+C, ésta manda un mensaje al programa para que pueda terminar liberando

correctamente los recursos asignados. Si este programa por alguna razón no termina (ignora el mensaje que le envía Ctrl+C) siempre podemos usar Ctrl+\, el cual termina el programa sin esperar a que este libere recursos, con lo que esta combinación de teclas sólo debe ser usada cuando Ctrl+C no ha funcionado.

Otra combinación de teclas que ya hemos visto en el apartado 4.1 es Ctrl+D, que nos permitía indicar un fin de flujo, cuando el flujo procedía del teclado.

Las otras combinaciones de teclas que vamos a estudiar en esta sección son las que nos permiten borrar "hacia tras" una palabra Ctrl+W, o toda la frase Ctrl+U. Pruébelas en su consola y procure recordar estas combinaciones de teclas, son muy útiles cuando se equivoca o quiere cambiar el comando que está usando.

3. Modos de edición en la línea de comandos

¿Por qué no puedo editar mis comandos Bash de la misma forma que edito texto en un editor de texto? Esta es una idea introducida por Bash. En concreto existen los modos de edición de los dos editores más conocidos en el mundo UNIX: vi y emacs.

Por defecto Bash utiliza las teclas del modo de edición de emacs, pero puede cambiar a las teclas del modo de edición de vi usando el comando:

```
$ set -o vi
```

Y volver a la combinación de teclas por defecto con:

```
$ set -o emacs
```

Nosotros en está sección veremos sólo las teclas de edición de emacs que se pueden usar también en Bash, si no conoce emacs le recomendamos que aprenda a usarlo, porque es, y será durante muchos años, un clásico del mundo de UNIX. Si es usted un amante de vi sólo le diremos que tiene que preceder las teclas rápidas por la tecla de escape, y pulsar `i` para volver a escribir.

3.1. Moverse por la línea

Algo que se hace muy a menudo es moverse por la línea de edición, puede ir al principio y final de la línea usando las combinaciones de tecla Ctrl+A (ir al principio) y Ctrl+E (ir al final), si lo que quiere es moverse una palabra hacía atrás o una palabra hacia adelante utilice Esc+B y Esc+F respectivamente. También puede moverse una letra atrás o adelante con Ctrl+B y Ctrl+F, pero desde que los teclados tienen cursores estas teclas han perdido su utilidad real. Estas teclas se resumen en la Tabla 2.3.

Combinación de teclas	Descripción
Ctrl+A	Ir al principio de la línea
Ctrl+E	Ir al final de la línea
Esc+B	Ir una palabra hacia atrás (backward)
Esc+F	Ir una palabra hacia adelante (forward)
Ctrl+B	Ir una letra hacia atrás
Ctrl+F	Ir una letra hacia adelante

Tabla 2.3: Combinaciones de teclas para moverse por la línea

3.2. Borrar partes de la línea

Otra operación que es muy común es querer borrar partes de la línea. Las combinaciones de teclas que vamos a comentar se resumen en la Tabla 2.4.

En el apartado 2 ya vimos que combinaciones de tecla proporcionaba el terminal, y vimos que con Ctrl+U podíamos borrar desde la posición del cursor al principio de la línea. Si lo que queremos es borrar hasta el final de la línea podemos usar la combinación de tecla emacs Ctrl+K.

Si lo que queremos es borrar sólo desde a posición actual del cursor hasta el principio de la palabra donde estemos situados, vimos que podíamos usar Ctrl+W. Si lo que queremos es borrar de la posición del cursor actual hasta el final de la palabra donde estemos situados podemos usar Esc+D.

Para borrar una sola letra es muy típico que la tecla de borrar hacía atrás ← sí que funcione, pero normalmente la tecla DEL no suele funcionar, y en su lugar podemos usar Ctrl+D.

La última operación de borrado se puede deshacer siempre con la tecla Ctrl+Y.

Combinación de teclas	Descripción
Ctrl+U	Borra de la posición actual al principio de la línea
Ctrl+K	Borra de la posición actual al final de la línea
Ctrl+W	Borra de la posición actual al principio de la palabra
Esc+D	Borra de la posición actual al final de la palabra
Ctrl+D	Borra el carácter actual hacia adelante
Ctrl+Y	Deshace el último borrado (Yank)

Tabla 2.4: Combinaciones de teclas para borrar

3.3. Buscar en el historial

Ya comentamos que las teclas del cursor nos permitían movernos por el histórico de comandos, pero si queremos buscar un determinado comando podemos usar Ctrl+R, que nos permite buscar hacía atrás en el historial un comando que contenga un determinado texto. Al pulsar esta combinación de teclas el prompt cambia de forma, y según vamos escribiendo nos va indicando el comando del histórico que cumple el patrón dado:

```
(reverse-i-search)`yuvs': yuvscaler -O SIZE_600x400 < result.yuv > result2.yuv
```

Si pulsamos intro se ejecutará el comando actualmente encontrado. Si por contra pulsamos Esc pasaremos a poder editar el comando encontrado. Por último si pulsamos Ctrl+G se abortará la búsqueda y la línea de edición quedará limpia. Estas combinaciones de teclas se resumen en la Tabla 2.5.

Combinación de teclas	Descripción
Ctrl+R	Realiza una búsqueda hacia atrás (Reverse)
Intro	Ejecuta el comando encontrado
Esc	Pasa a editar el comando encontrado
Ctrl+G	Cancela la búsqueda y deja limpia la línea de edición

Tabla 2.5: Combinaciones de teclas para búsquedas

3.4. Autocompletar con el tabulador

Una tecla muy conocida en Bash es el tabulador que nos ayuda a terminar de rellenar un comando con el nombre de un comando, de una variable, de un fichero o directorio, o con el nombre de una función Bash. Para ello se siguen las siguientes regla cuando se pulsa el tabulador:

1. Si no hay nada que empiece por el texto de la palabra que precede al cursor se produce un pitido que informa del problema.

2. Si hay un comando (en el `PATH`), una variable (siempre precedida por `$`), un nombre de fichero o función Bash que comienza por el texto escrito, Bash completa la palabra.

3. Si hay un directorio que comienza por el nombre escrito, Bash completa el nombre de directorio seguido por una barra de separación de nombres de directorios `/`.

4. Si hay más de una forma de completar la palabra el shell completa lo más que puede y emite un pitido informando de que no la pudo terminar de completar.

5. Cuando Bash no puede completar una cadena (por haber varias posibles que empiezan igual), podemos pulsar dos veces el tabulador y se nos mostrará una lista con las posibles cadenas candidatas.

4. La librería readline

La librería readline es una librería GNU que permite editar líneas de texto usando los modos de emacs y vi. Aunque originariamente creada por Bash, actualmente la utilizar multitud de comandos GNU. Esto permite estandarizar las combinaciones de teclas entre muchos comandos.

readline se puede personalizar usando ficheros de configuración o bien asignando combinaciones de teclas para la sesión. En los siguientes subapartados estudiaremos cada una de estas formas de configurarla.

4.1. El fichero de configuración

readline tiene un fichero de configuración global que debe ser colocado en `/etc/inputrc`, y otro para cada usuario que por defecto se llama `.inputrc` y que se debe colocar en el directorio home de cada usuario. Podemos cambiar este fichero usando la variable de entorno `INPUTRC` para indicar una ubicación distinta.

Cuando Bash arranca lee estos ficheros (si existen) y carga sus valores. Dentro del fichero puede haber comentarios (precedidos por `#`), combinaciones de teclas y asignaciones a variables de readline.

4.1.1. Combinaciones de teclas

El primer tipo de entradas que puede haber en este fichero son asignaciones de combinaciones de teclas a funciones de readline. Readline tiene un conjunto grande de funciones que podemos encontrar documentadas en `info readline` o en `man bash`. Estas funciones realizan multitud de operaciones comunes. Para listar estas variables podemos crear el fichero `.inputrc` en nuestro directorio home y añadir la siguiente entrada:

```
# Lista las funciones y variables de readline
"\C-x\C-f": dump-functions
"\C-x\C-v": dump-variables
```

Como el fichero `.inputrc` no se lee hasta logarnos de nuevo en Bash podemos usar la combinación de teclas Ctrl+X Ctrl+R para que readline vuelva a leer este fichero.

Ahora hemos indicado que la combinación de teclas Ctrl+X Ctrl+F muestre las funciones de readline (ejecute la función `dump-variables()` de readline), y que Ctrl+X Ctrl+V muestre las variables de readline.

Secuencia	Descripción
\C-	Tecla Ctrl
\M-	Tecla de escape (Meta)
\e	Secuencia de escape
\\	El carácter \ literal
\"	El carácter " literal
\'	El carácter ' literal

Tabla 2.6: Secuencias usadas en el fichero de configuración de readline

La Tabla 2.6 muestra como representar las combinaciones de teclas: Ctrl se representa con `\C-` como hemos visto, la tecla de escape (Meta) con `\M-`, y por último si queremos referirnos a teclas especiales podemos usar secuencias de escape que se representan con `\e`.

Por ejemplo, vimos que para borrar una línea de la posición del cursor al principio de la línea usábamos Ctrl-U, y para borrarlo de la posición actual al final de la línea Ctrl+K, pero desafortunadamente por defecto no hay ninguna combinación de teclas que borre toda la línea (adelante y atrás). Podemos inventarnos la combinación de teclas Esc+K, que llama a la función `kill-whole-line()` de readline, modificando el fichero de configuración de readline como se muestra más abajo. Otro problema que tiene el terminal es que normalmente la teclas DEL no funciona correctamente, en este caso necesitamos usar una secuencia de escape para hacer que esta tecla llame a la función `delete-char()` de readline.

```
# Borra toda la linea
"\M-k": kill-whole-line

# Borra con la tecla DEL
"\e[3~": delete-char
```

4.1.2. Variables de readline

readline tiene una serie de variables, que aparecen documentadas junto con las funciones de readline (y que también puede consultar usando Ctrl+X Ctrl+V del ejemplo del apartado anterior), a las cuales podemos asignar un valor en le fichero de configuración de readline con `set`. Por ejemplo, una opción que algunas personas encuentran interesante en Mac OS X es evitar que al completar con tabulador distingan minúsculas de mayúsculas. Es decir, si escribimos:

```
$ cd mus <TAB>
```

Nos expanda a:

```
$ cd Music/
```

Para ello podemos añadir la siguiente línea al fichero de configuración de readline:

```
# Ignora diferencias de mayusculas/minusculas al
# completar con tabulador
set completion-ignore-case on
```

4.2. Asignación de teclas de sesión

Si lo que queremos es probar combinaciones de teclas que no queremos almacenar de forma permanente podemos usar el comando `bind`, cuya sintaxis es similar a la del fichero de configuración, pero tenemos que encerrar cada asignación entre comillas blandas o fuertes. Por ejemplo:

```
$ bind "\"\C-x\C-g\": dump-functions"
```

Si queremos ver las asignaciones de teclas que hay hechas a cada función de readline podemos usar el comando:

```
$ bind -P
```

Este comando es muy útil para que no asignemos una combinación de teclas a otra que ya existe. Por ejemplo, en el apartado anterior para sacar las funciones de readline creamos la combinación de teclas Ctrl+X Ctrl+F, pero al usar este comando descubrimos que ya existía la combinación Ctrl+G que hace lo mismo:

```
$ bind -P
..................
dump-functions can be found on "\C-g", "\C-x\C-f".
..................
```

De hecho también podríamos haber sacado las funciones de readline usando el comando:

```
$ bind -l
```

Podemos sacar todas las asignaciones de teclas en formato `inputrc` usando el comando:

```
$ bind -p
```

En este caso sería muy útil enviar esta salida a un fichero para posterior edición:

```
$ bind -p > .inputrc
```

Por último comentar que podemos asignar combinaciones de teclas que ejecutan comandos comunes con la opción `-x`:

```
$ bind -x "\"\C-l\": ls -la"
```

Ejecutará `ls -la` al pulsar Ctrl+L.

Tema 3

Personalizar el entorno

Sinopsis:

En este tema veremos como hacer el entorno más agradable personalizando distintos aspectos de Bash.

Empezaremos viendo los ficheros que configuran Bash siempre que nos logamos. También veremos como personalizar distintos aspectos del comportamiento de Bash, y que variables de entornos participan en esta personalización.

Este es el último tema que dedicamos a aprender a manejar el terminal, el resto del tutorial enseñara aspectos más relativos a la programación de scripts con Bash.

1. Los ficheros de configuración de Bash

Cada vez que nos logamos en la cuenta se ejecuta el contenido del fichero `/etc/profile`, y luego se mira a ver si en el directorio home existe el fichero `.bash_profile`, de ser así se ejecuta su contenido para personalizar aspectos de nuestra cuenta.

Cualquier configuración que añadamos a `.bash_profile` no será efectiva hasta que salgamos de la cuenta y volvamos a logarnos, si hacemos cambios en este fichero y queremos verlos sin salir de la cuenta podemos usar el comando `source`, el cual ejecuta el contenido del fichero que le digamos:

```
$ source .bash_profile
```

Alternativamente al comando `source` está el comando punto (`.`), con lo que el contenido de `.bash_profile` también se podría hacer ejecutado así:

```
$ . .bash_profile
```

Bash permite usar dos nombres alternativos para `.bash_profile` por razones de compatibilidad histórica: `.bash_login`, nombre derivado del fichero `.login` del C Shell, y `.profile` nombre usado por el Bourne Shell y el Korn Shell. En cualquier caso, sólo uno de estos ficheros será ejecutado: Si `.bash_profile` existe los demás serán ignorados, sino Bash comprueba si existe `.bash_login` y, sólo si éste tampoco existe, comprueba si existe `.profile`. La razón por la que se eligió este orden de búsqueda es que podemos almacenar en `.profile` opciones propias del Bourne Shell, y añadir opciones exclusivas de Bash en el fichero `.bash_profile` seguido del comando `source .profile` para que Bash también cargue las opciones del fichero `.profile`.

`.bash_profile` se ejecuta sólo al logarnos, si abrimos otro shell (ejecutando `bash` o `su`) desde la línea de comandos de Bash lo que se intenta ejecutar es el contenido de `.bashrc`. Si `.bashrc` no existe no se ejecutan configuraciones adicionales al abrir un nuevo shell. Este esquema nos permite separar configuraciones que se hacen una sola vez, al logarnos, de configuraciones que se cambian cada vez que se abre un nuevo shell. Si hay configuraciones en `.bashrc` que también queremos ejecutar al logarnos podemos poner `source .bashrc` dentro del fichero `.bash_profile`.

Por último, el fichero `.bash_logout` es un fichero que, de existir, contiene ordenes que se ejecutarán al abandonar la cuenta, por ejemplo eliminar ficheros temporales o almacenar datos de actividad de usuarios en un fichero de log.

2. Los alias

Un alias es un nombre más corto para un comando que usamos muy a menudo.

Un ejemplo clásico de alias que se suele encontrar muchas veces en `/etc/profile` es:

```
$ alias ll='ls -laF'
```

Ahora podemos usar el alias `ll` para hacer un listado detallado del directorio.

Los alias son recursivos con lo que por ejemplo podemos hacer:

```
$ alias l='ll'
```

Aunque los alias sólo pueden ser usados para representar comandos, si por ejemplo intentamos representar un directorio largo con un alias, para luego meternos en el con `cd`, el comando fallará:

```
$ alias ASCII='/usr/share/misc/ascii'
$ cat ASCII
cat: ASCII: No such file or directory
```

Existe un oscuro truco en Bash que permite hacer esto. Si la parte derecha de la asignación del alias acaba con un espacio, entonces Bash intenta hacer sustitución de alias en la siguiente palabra del comando. Por ejemplo si hacemos:

```
$ alias cat='cat '
$ cat ASCII
```

Obtendremos la tabla ASCII en pantalla.

Si usamos el comando `alias` sin parámetros, obtenemos un listado de todos los alias definidos. Si queremos conocer el valor de un determinado alias podemos usar el comando `alias` seguido del nombre del alias sin símbolo `=` ni valor.

```
$ alias ll
alias ll='ls -laF'
```

Por último, si queremos borrar la definición de un alias podemos usar el comando `unalias`:

```
$ unalias ll
```

3. Las opciones de Bash

Las opciones del shell es una forma de poder modificar el comportamiento de éste.

Para fijar estas opciones usamos el comando `shopt` (shell option). Si ejecutamos este comando sin argumentos obtendremos un listado de todas las opciones del shell y su valor actual.

Todas estas variables son booleanas y tienen valor de on/off. Podemos activar estas opciones con `shopt -s` *opcion* y desactivarlas con `shopt -u` *opcion*.

La Tabla 3.1 muestra las principales opciones de `shopt` junto con una descripción de su propósito.

Opción	Descripción
`cdable_vars`	Permite que `cd` use los valores de las variables como nombres de directorios.
`cdspell`	Ignora pequeños errores en los cambios de directorio con `cd`. Sólo en la ejecución interactiva.
`cmdhist`	Guarda los comandos que hemos escrito en varias líneas en una sola línea del historial.
`dotglob`	Incluye en la expansión de comodines los ficheros que empiezan por punto (`.`).
`expand_aliases`	Expande un alias cuando lo ejecuta.
`extglob`	Utiliza extensiones de los comodines. Véase apartado 2.4 del Tema 1. El término "glob" también se usa a veces para referirse a los comodines.
`failglob`	Si falla la expansión de un comodín porque no encuentra nada falla el comando (como hace el C Shell).
`force_fignore`	Los sufijos especificados en la variable de entorno `FIGNORE` no se usan para completar palabras con tabulador.
`hostcomplete`	Se intenta completar nombres de host al pulsar tabulador cuando la palabra contiene una `@`.
`interactive_comments`	Permite que dentro de un comando de sesión interactiva haya comentarios (precedidos por `#`)
`login_shell`	Variable de sólo lectura que indica si Bash ha sido lanzado como un shell de login
`nocaseglob`	Indica si los comodines expanden sin sensibilidad a mayúsculas/minúsculas. No confundir con la variable `completion-`

	`ignore-case` de `inputrc` que lo que hacía era expandir con tabulador.
`nullglob`	Hace que cuando un patrón no encuentra ficheros, se expandan por la cadena vacía en vez de por el patrón sin expandir.
`sourcepath`	Hace que el comando interno `source` busque el argumento en los directorios que indique `PATH`

Tabla 3.1: Principales opciones de `shopt`

Una opción que resulta muy útil es `cdable_vars`, la cual permite que el comando `cd` use los valores de las variables como nombres de directorios. Esto se usa cuando hay un directorio al que solemos movernos mucho. Por ejemplo si tenemos el directorio `/usr/local/mldonkey/incomming` al que solemos ir muy a menudo desde la consola podemos definir en los ficheros de configuración de Bash:

```
# Para ir a mldonkey
export DONKEY=/usr/local/mldonkey/incomming
shopt -s cdable_vars
```

Y ahora para movernos a este directorio basta con hacer:

```
$ cd DONKEY
```

Obsérvese que `DONKEY` no va precedida por `$`.

Otra opción muy útil es `cdspell` que permite que se ignoren pequeños errores en los directorios que indicamos a `cd`. Por ejemplo este pequeño baile de letras es corregido por `cd`:

```
$ cd /usr/locla/share/
/usr/local/share/
```

Esta opción sólo funciona en modo interactivo.

4. Las variables de entorno

Las variables de entorno nos ayudan a configurar el entorno en el que se ejecutan los programas.

Para declararlas es importante recordar que las variables de entorno no pueden tener espacios, y si queremos asignar un valor con espacio debemos de entrecomillarlo, o bien preceder el espacio por un carácter de escape:

```
$ NOMBRE='Fernando Lopez'
$ echo $NOMBRE
Fernando Lopez
$ NOMBRE=Fernando\ Lopez
$ echo $NOMBRE
Fernando Lopez
```

Como sabemos, para obtener el valor de una variable debemos de precederla por `$`. Obsérvese que las comillas o el carácter de escape no forman parte del valor de la variable una vez que se realiza la asignación.

Para eliminar una variable de entorno podemos usar el comando `unset`. Por ejemplo:

```
$ unset NOMBRE
```

4.1. Variables y entrecomillado

En el apartado 6.1 del Tema 1 comentamos que las comillas dobles (las débiles) interpretaban algunos de los caracteres que contenían. Uno de ellos es el carácter `$`, el cual nos permite expandir el valor de una variable dentro de una cadena. Por ejemplo:

```
$ articulo='Zapatos negros'
$ precio=20.5
$ echo "El articulo $articulo cuesta $precio euros"
El articulo Zapatos negros cuesta 20.5 euros
```

4.2. Personalizar el prompt

Podemos personalizar el prompt de Bash usando las variables de entorno `PS1`, `PS2`, `PS3` y `PS4`. La Tabla 3.2 muestra las opciones de personalización que podemos usar en cualquiera de los prompt.

En el prompt podemos encontrar mucha opciones que nos ayudan a trabajar, quizá la más usada es `\w` que muestra le directorio donde estamos situados en cada momento.

```
$ PS1='\w->'
~->
```

Donde `~` significa que estamos en el directorio home.

Es muy típico querer saber si somos root o cualquier otro usuario de la máquina, para lo cual se usa la opción `\$` que muestra una `#` si somos root, o un `$` en cualquier otro caso.

```
~->PS1='\w\$'
~$
```

Muchas veces el prompt nos ayuda a saber en que máquina estamos situados, esto es útil por ejemplo cuando tenemos la costumbre de hacer muchos telnet o ssh a otras máquinas. En este caso podemos usar la opción `\H` que nos da el nombre de la máquina donde estamos situados, o la opción `\h` que nos da el nombre de la máquina sólo hasta el primer punto:

```
~$ PS1='\h\w\$'
macbox~$ PS1='\H\w\$'
macbox.macprogramadores.org~$
```

Aunque en Mac OS X no es muy útil, cuando trabajamos con una máquina UNIX en modo sólo consola solemos poder movernos por un conjunto de 6 terminales virtuales usando las teclas Ctrl+F1 hasta Ctrl+F6 (o Alf+F1 hasta Alf+F6 en algunas configuraciones). En estos casos sí que es recomendable conocer el nombre del terminal con la opción `\l`. Por ejemplo en mi FreeBSD tengo este prompt:

```
[root@freebsd@ttyv0]~# echo $PS1
[\u@\h@\l]\w\$
```

Donde me indica el usuario con el que estoy logado (opción `\u`), la maquina donde estoy, y el terminal virtual donde estoy trabajando.

En mi Mac OS X tengo este otro prompt:

```
$ echo $PS1
[\u@\h]\w\$
[fernando@macbox]~$
```

Donde me indica el usuario con el que estoy logado y el host, pero no el terminal virtual.

Si estamos trabajando con el comando `fc` (véase apartado 1.1 del Tema 2) podría ser útil usar `\!` que nos va dando el número histórico del comando que vamos a ejecutar.

Opción	Descripción
`\a`	El carácter ASCII bell (007)
`\A`	La hora en formato *HH*:*MM*
`\d`	La fecha en formato *semana mes dia*
`\D {`*formato*`}`	Nos permite personalizar más la fecha
`\e`	El carácter de escape ASCII (033)
`\H`	Hostname
`\h`	El nombre de la máquina hasta el primer punto
`\j`	Número de jobs hijos del shell
`\l`	Nombre del terminal en el que estamos trabajando (p.e. `ttyp2`)
`\n`	Retorno de carro y nueva línea
`\r`	Retorno de carro
`\s`	Nombre del shell
`\t`	Hora en formato *HH*:*MM*:*SS* con 12 horas
`\T`	Hora en el formato anterior pero con 24 horas
`\u`	Nombre de usuario
`\v`	Versión de Bash
`\w`	Ruta del directorio actual
`\W`	Nombre del directorio actual (sin ruta)
`\#`	Número de comandos ejecutados
`\!`	Número histórico del comando
`\$`	`#` si somos root, o `$` en caso contrario
`\nnn`	Código de carácter a mostrar en octal
`\\`	La barra hacía atrás
`\[`	Empieza una secuencia de caracteres no imprimibles, como los caracteres de control de secuencias del terminal
`\]`	Acaba una secuencia de caracteres no imprimibles

Tabla 3.2: Opciones de personalización del prompt

`PS2` es el prompt secundario y por defecto vale `>`. Se usa cuando escribimos un comando inacabado, por ejemplo vimos:

```
$ echo En un lugar de la Mancha de cuyo nombre no \
> quiero acordarme no a mucho tiempo vivía un \
> hidalgo caballero.
En un lugar de la Mancha de cuyo nombre no quiero acordar
no a mucho tiempo vivía un hidalgo caballero.
```

Por último, `PS3` y `PS4` son prompts de programación y depuración que estudiaremos en temas posteriores.

4.3. Variables de entorno internas

Existen una serie de variables de entorno, las cuales se resumen en la Tabla 3.3, que no las fijamos nosotros sino que las fija el shell, y cuyo valor es de sólo lectura.

Variable	Descripción
SHELL	Path del fichero que estamos usando
LINES	Líneas del terminal en caracteres
COLUMNS	Columnas del terminal en caracteres
HOME	Directorio home del usuario
PWD	Directorio actual
OLDPWD	Anterior directorio actual
USER	Nombre de usuario en la cuenta donde estamos logados

Tabla 3.3: Variables de entorno internas

Algunas variables se actualizan dinámicamente, por ejemplo `LINES` y `COLUMNS` en todo momento almacenan en número de filas y columnas de la pantalla, y las usan algunos programas como vi o emacs para modificar su apariencia.

4.4. Exportar variables

Las variables de entorno que definimos dentro de una sesión de Bash no están disponibles para los subprocesos que lanza Bash (sólo las puede usar Bash y nuestros comandos interactivos y scripts) a no ser que las exportemos con el comando `export`. Por ejemplo:

```
$ EDITOR=/sw/bin/joe
$ export EDITOR
```

También podemos exportar una variable de entorno a la vez que la asignamos un valor así:

```
$ export EDITOR=/sw/bin/joe
```

Podemos obtener un listado de las variables exportadas usando el comando `env` o el comando `export` sin argumentos. Para obtener las variables tanto exportadas como no exportadas debemos usar el comando `set` sin argumentos.

Tema 4

Programación básica del shell

Sinopsis:

Este tema introduce los conceptos más básicos de la programación de scripts y funciones con Bash.

En este momento debería de ser capaz de manejarse con el terminal (el modo interactivo) con más soltura que antes de leer este tutorial. Ahora pretendemos empezar a entender los scripts que configuran su máquina, y a que sea capaz de crear sus propios scripts.

El tema supone que el lector conoce los conceptos básicos de programación en algún lenguaje estructurado (C, Pascal, Basic). Bash no usa los conceptos de programación orientada a objetos, pero si conoce este tipo de programación (p.e. Java) también le servirá aquí, ya que la programación estructurada es un subconjunto de la orientada a objetos.

Bash es un lenguaje muy críptico, con lo que sería recomendable que se fijara bien en la sintaxis, y que intentara hacer los ejercicios que proponemos en su ordenador con el fin de consolidad las ideas básicas que vamos a desarrollar en este tema.

1.Scripts y funciones

1.1. Scripts

Un **script** es un fichero que contiene comandos Bash ejecutables. Los ficheros de configuración de Bash como `.bash_profile` o `.bashrc`, vistos en el apartado 1 del Tema 3, son ejemplos de scripts.

Para crear un script use su editor de texto favorito, y guardelo en un fichero. Una vez creado el script existen dos formas de ejecutarlo:

La primera consiste en ejecutarlo con el comando `source` *`fichero`*, el cual carga el fichero en la memoria de Bash y lo ejecuta.

La segunda forma implica poner al fichero permiso de ejecución (con el comando `chmod +x` *`fichero`*). Una vez puesto este permiso, podremos ejecutarlo siempre que esté en alguno de los directorios indicados en la variable de entorno `PATH`. De hecho muchos comandos comunes (p.e. `spell` o `man`) están implementados como scripts Bash, y no como programas C ejecutables. Es muy típico en este caso empezar el script poniendo en la primera línea `#! /bin/bash`, de esta forma indicamos que el script se debe ejecutar con Bash, a pesar de que Bash no fuera el shell por defecto. Otros scripts para otros lenguajes como `awk`, `ksh` o `perl` también se escriben empezando la primera línea por el path del shell a utilizar.

Hay otra diferencia entre estas dos formas de ejecutarlo: Cuando ejecutamos un script con `source`, éste se ejecuta dentro del proceso del shell, mientras que si lo ejecutamos como un fichero ejecutable se ejecuta en un subshell. Esto implica que sólo las variables de entorno exportadas son conocidas por el nuevo subshell que ejecuta el programa.

De hecho, cada vez que un script llama a un comando también se ejecuta en un proceso aparte, excepto con los comandos internos que se ejecutan dentro del proceso del script, lo cual les hace más rápidos y les permite acceder a las variables de entorno no exportadas.

1.2. Funciones

Las funciones de Bash son una extensión de las funciones que existen desde el Bourne Shell.

Éstas, a diferencia de los scripts, se ejecutan dentro de la memoria del propio proceso de Bash, con lo que son más eficientes que ejecutar scripts aparte, pero tienen el inconveniente de que tienen que estar siempre cargadas en la

memoria del proceso Bash para poder usarse. Actualmente, debido a la gran cantidad de memoria que tienen los ordenadores, el tener funciones cargadas en la memoria de Bash tiene un coste insignificante, con lo que no debería de preocuparle el tener cargadas gran cantidad de estas en su entorno.

Para definir una función existen dos formatos:

El estilo del Bourne Shell;

```
function nombrefn
{
 ...
 comandos bash
 ...
}
```

O el estilo del C Shell:

```
nombrefn()
{
 ...
 comandos bash
 ...
}
```

No existe diferencia entre ellos, y usaremos ambos indistintamente. Para borrar una función podemos usar el comando `unset -f nombrefn`.

Cuando definimos una función se almacena como una variable de entorno (con su valor almacenando la implementación de la función). Para ejecutar la función simplemente escribimos su nombre seguido de argumentos, como cuando ejecutamos un comando. Los argumentos actúan como parámetros de la función.

Podemos ver que funciones tenemos definidas en una sesión usando el comando `declare -f`. El shell imprime las funciones, y su definición, ordenadas alfabéticamente. Si preferimos obtener sólo el nombre de las funciones podemos usar el comando `declare -F`.

Si una función tiene el mismo nombre que un script o ejecutable, la función tiene preferencia: Esta regla se ha usado muchas veces para, engañando a los usuarios, violar la integridad de su sistema.

1.3. Orden de preferencia de los símbolos de Bash

A continuación se enumera el orden de preferencia que sigue Bash a la hora de resolver un símbolo:

1. Alias.
2. Palabras clave (keywords) como `function`, `if` o `for`.
3. Funciones
4. Comandos internos (p.e. `cd` o `type`).
5. Scripts y programas ejecutables, para los cuales se sigue el orden en que se dan sus directorios en la variable de entorno `PATH`.

Aunque no es muy común, a veces conviene cambiar estos ordenes usando los comandos internos `command`, `builtin` y `enable`. Esto nos permite tener alias y scripts con el mismo nombre y elegir cual de ellos ejecutar. En posteriores temas veremos su utilidad.

Si nos interesa conocer la procedencia de un comando podemos usar el comando interno `type`. Si le damos un comando nos indica cual es su fuente, si le damos varios nos indica la fuente de cada no de ellos. Por ejemplo:

```
$ type ls
ls is /bin/ls
$ type ls ll
ls is /bin/ls
ll is aliased to `ls -laF'
```

Si tenemos un script, una función y un alias llamados `hazlo`, `type` nos dirá que está usando `hazlo` como alias, ya que los alias tienen preferencia sobre el resto de símbolos:

```
$ type hazlo
hazlo is aliased to `rm *'
```

`type` también tiene opciones que nos permiten obtener detalles de un símbolo. Si queremos conocer todas las definiciones de un símbolo podemos usar `type -a`. Por ejemplo:

```
$ type -a hazlo
hazlo is aliased to `rm *'
hazlo is a function
hazlo ()
{
    rm *
}
hazlo is ./hazlo
```

Podemos conocer el tipo de un símbolo con la opción `-t`: Nos devolverá `alias`, `keyword`, `function`, `builtin` o `file`. Por ejemplo:

```
$ type -t cd
builtin
$ type -t cp
file
```

2. Variables del shell

Principalmente Bash utiliza variables de tipo cadena de caracteres. Esto le diferencia de otros lenguajes de programación donde las variables tienen distintos tipos. Aunque éste es el tipo por defecto de las variables de Bash, más adelante veremos que las variables en Bash también pueden tener otros tipos, como por ejemplo enteros con los que realizar operaciones aritméticas.

Por convenio las variables de entorno exportadas (las que pasamos a los subprocesos) se escriben en mayúsculas, y las que no exportamos en minúsculas. Esta regla, más que por el modo interactivo, es especialmente seguida por los scripts y funciones que vamos a empezar a programar ahora.

2.1. Los parámetros posiciónales

Los **parámetros posiciónales** son los encargados de recibir los argumentos de un script y los parámetros de una función. Sus nombres son `1`, `2`, `3`, etc., con lo que para acceder a ellos utilizaremos, como normalmente, el símbolo `$` de la forma `$1`, `$2`, `$3`, etc. Además tenemos el parámetro posicional `0` que almacena el nombre del script donde se ejecuta.

Por ejemplo imaginemos que creamos el script del Listado 4.1:

```
#!/bin/bash
# Ejemplo de script que recibe parametros y los imprime

echo "El script $0"
echo "Recibe los argumentos $1 $2 $3 $4"
```

Listado 4.1: Script que recibe argumentos

Si lo hemos guardado en un fichero llamado `recibe` con el permiso `x` activado podemos ejecutarlo así:

```
$ recibe hola adios
El script ./recibe
Recibe los argumentos hola adios
```

Como los argumentos `$3` y `$4` no los hemos pasado son nulos que dan lugar a una cadena vacía que no se imprime.

Para definir una función podemos escribirla en un fichero y cargarla en el shell usando el comando `source`, o bien definirla directamente en modo interactivo:

```
$ function recibe
> {
>  echo "La funcion $0"
>  echo "Recibe los argumentos $1 $2 $3 $4"
> }
$ recibe buenos dias Bash
La funcion -bash
Recibe los argumentos buenos dias Bash
```

Como los nombres de funciones tienen preferencia sobre los de los scripts ahora se ejecuta la función `recibe()` y no el script `recibe`. Vemos que ahora `$0` no se ha sustituido por el nombre de la función sino por `-bash`, esto es así porque `$0` siempre se sustituye por el nombre de script (no el de la función), o `-bash` si lo ejecutamos directamente desde la línea de comandos.

Por último comentar que no podemos modificar el valor de las variables posiciónales, sólo se pueden leer, si intentamos asignarlas un valor se produce un error.

2.2. Variables locales y globales

Por defecto los parámetros posiciónales son locales al script o función y no se pueden acceder o modificar desde otra función. Por ejemplo en el Listado 4.2 vemos un script, que guardaremos en el fichero `saluda`, que llama a una función para que salude al nombre que pasamos al script como argumento en `$1`:

```
# Script que llama a una funcion para saludar

function DiHola
{
 echo "Hola $1"
}

DiHola
```

Listado 4.2: Script que llama a una función para saludar

Al ejecutarlo obtenemos:

```
$ saluda Fernando
Hola
```

Vemos que el argumento pasado en `$1` no a sido cogido por la función, eso es porque `$1` es local al script, y si queremos que lo reciba la función tendremos que pasárselo como muestra el Listado 4.3.

```
# Script que llama a una funcion para saludar

function DiHola
{
 echo "Hola $1"
}

DiHola $1
```

Listado 4.3: Script que pasa el argumento a una función para que salude

A diferencia de los parámetros posiciónales, el resto de variables que definimos en un script o función son globales, es decir una vez definidas en el script son accesibles (y modificables) desde cualquier función. Por ejemplo, en el Listado 4.4 se muestra un script llamado `dondeestoy`, en el que la variable `donde` está definida por el script y modificada por la función.

```
# Ejemplo de variable global

function EstasAqui
{
 donde='Dentro de la funcion'
}

donde='En el script'
echo $donde
EstasAqui
echo $donde
```

Listado 4.4: Ejemplo de variable global

Al ejecutarlo obtenemos:

```
$ dondeestoy
En el script
Dentro de la funcion
```

Si queremos que una variable no posicional sea local debemos de ponerla el modificador `local`, el cual sólo se puede usar dentro de las funciones (no en los scripts). Por ejemplo en el Listado 4.5 ahora la función se crea su propia variable local `donde` y no modifica la global del script.

```
# Ejemplo de variable global

function EstasAqui
{
 local donde='Dentro de la funcion'
}
```

```
donde='En el script'
echo $donde
EstasAqui
echo $donde
```

Listado 4.5: Ejemplo de variable local y global con el mismo nombre

Al ejecutarlo obtenemos:

```
$ dondeestoy
En el script
En el script
```

Por último comentar que las variables globales también se pueden definir dentro de una función. Por ejemplo, en el Listado 4.6 se muestra como definimos la variable global `quiensoy` dentro de la función `EstasAqui()` y la usamos más tarde desde el script.

```
# Ejemplo de variable global

function EstasAqui
{
 local donde='Dentro de la funcion'
 quiensoy=Fernando
}

donde='En el script'
echo $donde
EstasAqui
echo $donde
echo "Soy $quiensoy"
```

Listado 4.6: Ejemplo de variable global definida dentro de una función

Al ejecutar el script del Listado 4.6 obtenemos:

```
$ dondeestoy
En el script
En el script
Soy Fernando
```

2.3. Las variables $*, $@ y $#

La variable `$#` almacena el número de argumentos o parámetros recibidos por el script o la función. El valor es de tipo cadena de caracteres, pero más adelante veremos como podemos convertir este valor a número para operar con él.

Tanto `$*` como `$@` nos devuelven los argumentos recibidos por el script o función.

Como ejemplo, el script `recibe` del Listado 4.1 lo vamos a modificar tal como muestra el Listado 4.7 para que use esta variable para sacar los argumentos recibidos.

```
Ejemplo de script que recibe parametros y los imprime

echo "El script $0 recibe $# argumentos:" $*
echo "El script $0 recibe $# argumentos:" $@
```

Listado 4.7: Ejemplo de uso de `$*` y `$@`

Al ejecutarlo obtenemos:

```
$ recibe hola adios
El script ./recibe recibe 2 argumentos: hola adios
El script ./recibe recibe 2 argumentos: hola adios
```

Aunque cuando no entrecomillamos `$*` o `$@` no existen diferencias entre usar uno y otro, cuando los encerramos entre comillas débiles existen dos diferencias que conviene resaltar:

La primera es que podemos cambiar el símbolo que usa `$*` para separar los argumentos indicándolo en la variable de entorno `IFS` (Internal Field Separator), mientras que `$@` siempre usa como separador un espacio. Por ejemplo si hacemos el siguiente script:

```
IFS=','
echo "El script $0 recibe $# argumentos: $*"
echo "El script $0 recibe $# argumentos: $@"
```

Al ejecutarlo obtenemos:

```
$ recibe hola adios
El script ./recibe recibe 2 argumentos: hola,adios
El script ./recibe recibe 2 argumentos: hola adios
```

La segunda diferencia se produce al recoger los argumentos para pasárselos a una función. Imaginemos que tenemos definida la siguiente función:

```
function CuentaArgumentos
{
 echo "Recibidos $# argumentos"
}
```

Ahora podemos pasar los argumentos recibidos por el script a la función usando cualquiera de las dos variables:

```
CuentaArgumentos $*
CuentaArgumentos $@
```

Y al ejecutarlo obtenemos:

```
$ recibe hola adios
Recibidos 2 argumentos
Recibidos 2 argumentos
```

Pero el no encerrar tanto `$*` como `$@` entre comillas al llamar a una función tiene un efecto lateral por el que no se recomienda. Este efecto se produce cuando recibimos argumentos que contienen espacios:

```
$ recibe "El perro" "La casa"
Recibidos 4 argumentos
Recibidos 4 argumentos
```

El problema está en que se pierden las comillas de las variables y se interpreta cada palabra como un parámetro de la función.

Sin embargo, si entrecomillamos `$*` y `$@`:

```
CuentaArgumentos "$*"
CuentaArgumentos "$@"
```

Obtenemos el siguiente resultado:

```
$ recibe "El perro" "La casa"
Recibidos 1 argumentos
Recibidos 2 argumentos
```

Es decir, entrecomillar `$*` tiene el efecto de que se convierte en una sola palabra, mientras que si entrecomillamos `$@` cada argumento es una palabra.

En consecuencia obtenemos las siguentes reglas generales:

1. Siempre conviene entrecomillar las variables `$*` y `$@` para evitar que los argumentos que contengan espacios sean mal interpretados.
2. Si queremos cambiar el delimitador que separa los argumentos (usando `IFS`) debemos utilizar `@*` entrecomillado.
3. Si lo que queremos es pasar los argumentos a una función debemos de usar `$@` entrecomillado.

2.4. Expansión de variables usando llaves

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

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

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

Pero esto produce una salida incorrecta porque Bash ha intentado buscar la variable `$nombre_` que al no existir la ha expandido por una cadena vacía.

Para solucionarlo podemos usar las llaves:

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

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

3. Operadores de cadena

Los **operadores de cadena** nos permiten manipular cadenas sin necesidad de escribir complicadas rutinas de procesamiento de texto. En particular los operadores de cadena nos permiten realizar las siguientes operaciones:

- Asegurarnos de que una variable existe (que está definida y que no es nula).
- Asignar valores por defecto a las variables.
- Tratar errores debidos a que una variable no tiene un valor asignado
- Coger o eliminar partes de la cadena que cumplen un patrón

Vamos a empezar viendo los operadores de sustitución, para luego ver los de búsqueda de cadenas.

3.1. Operadores de sustitución

La Tabla 4.1 muestra los operadores de sustitución.

Operador	Descripción
`$ {var:-valor}` [1]	Si *var* existe y no es nula retorna su valor, sino retorna *valor*.
`$ {var:+valor}` [1]	Si *var* existe y no es nula retorna *valor*, sino retorna una cadena nula.
`$ {var:=valor}` [1]	Si *var* existe y no es nula retorna su valor, sino asigna *valor* a *variable* y retorna su valor.
`$ {var:?mensaje}` [1]	Si *var* existe y no es nula retorna su valor, sino imprime *var:mensaje* y aborta el script que se esté ejecutando (sólo en shells no interactivos). Si omitimos *mensaje* imprime el mensaje por defecto `parameter null or not set`.
`$ {var:offset:longitud}`	Retorna una subcadena de *var* que empieza en *offset* y tiene *longitud* caracteres. El primer carácter de *var* empieza en la posición 0. Si se omite *longitud* la subcadena empieza en *offset* y continua hasta el final de *var*.

Tabla 4.1: Operadores de sustitución

[1] Los dos puntos (:) en este operador son opcionales. Si se omiten en vez de comprobar si existe y no es nulo, sólo comprueba que exista (aunque sea nulo).

`$ {var:-valor}` se utiliza para retornar un valor por defecto cuando el valor de la variable *var* está indefinido. Por ejemplo `${veces:-1}` devuelve 1 si el valor de veces está indefinido o es nulo.

`$ {var:+valor}` por contra se utiliza para comprobar que una variable tenga asignado un valor no nulo. Por ejemplo `${veces:+1}` retorna 1 (que se puede interpretar como verdadero) si `veces` tiene un valor asignado.

Los dos operadores de cadena anteriores no modifican el valor de la variable *var*, simplemente devuelven un valor, si queremos modificar *var* podemos usar `$ {var:=valor}` que asigna un valor a la variable si ésta está indefinida. Por ejemplo `$ {veces:=1}` asigna 1 a `veces` si ésta no tiene valor.

También podemos querer detectar errores producidos porque una variable no tenga valor asignado, en este caso usamos `$ {var:?mensaje}` que detecta si *var* no tiene valor asignado y produce un mensaje de error. Por ejemplo `$ {veces:?'Debe tener un valor asignado'}` imprime `veces: Debe tener un valor asignado` si `veces` no tiene valor asignado.

Por último podemos coger partes de una cadena usando `$ {var:offset:longitud}`. Por ejemplo si `nombre` vale `Fernando Lopez`, `$ {nombre:0:8}` devuelve `Fernando` y `$ {nombre:9}` devuelve `Lopez`.

Ejercicio 4.1

Imaginemos que tenemos un fichero con el saldo y nombre de un conjunto de clientes de la forma:

```
$ cat clientes
45340   Jose Carlos Martinez
24520   Mari Carmen Gutierrez
450     Luis Garcia Santos
44      Marcos Vallido Grandes
500     Carlos de la Fuente Lopez
```

Escribir un script que imprima los *N* clientes que más saldo tengan. El script recibirá como primer argumento el fichero de clientes y, opcionalmente como segundo argumento, el número *N* de clientes a imprimir. Si no se proporciona *N,* por defecto será 5. Luego la forma del comando podría ser:

```
mejoresclientes fichero [cuantos]
```

Para ello podemos usar el comando `sort` el cual ordena líneas, y después el comando `head` que saca las primeras líneas de la forma:

```
sort -nr $1 | head -${2:-5}
```

La opción `-n` dice a `sort` que ordene numéricamente (no alfabéticamente) y la opción `-r` dice a `sort` que saque los elementos ordenados de mayor a menor. `head` recibe como argumento el número de líneas a mostrar, por ejemplo `head -2` significa coger las primeras 2 líneas.

Este script aunque funciona es un poco críptico, y vamos a hacer unos cambios con vistas a mejorar su legibilidad. Por un lado vamos a poner comentarios (precedidos por `#`) al principio del fichero, y vamos a usar variables temporales para mejorar la legibilidad del programa. El resultado se muestra en el Listado 4.8.

```
# Script que saca los mejores clientes
# Tiene la forma:
#       mejoresclientes <fichero> [<cuantos>]

fichero=$1
cuantos=$2
defecto=5
sort -nr $fichero | head -${cuantos:-$defecto}
```

Listado 4.8: Script que lista los mejores clientes

Estos cambios que hemos hecho mejoran la legibilidad del script, pero no su tolerancia a errores, ya que si por ejemplo no pasamos el argumento `$1`, con el nombre del fichero, se ejecutará el script así:

```
sort -nr | head -$5
```

Y `sort` se quedará esperando a que entren datos por su entrada estándar hasta que el usuario pulse Ctrl+D o Ctrl+C. Esto se debe a que, aunque controlamos que el segundo argumento no sea nulo, no controlamos el primero.

En este caso podemos usar el operador de cadena `:?` para dar un mensaje de error cambiando:

```
fichero=$1
```

Por:

```
fichero=${1:?'no suministrado'}
```

Ahora, si no suministrados el argumento se produce el mensaje:

```
./mejoresclientes: 1: no suministrado
```

Podemos mejorar la legibilidad de este mensaje si hacemos:

```
fichero_clientes=$1
fichero_clientes=${fichero_clientes:?'no suministrado'}
```

Ahora si olvidamos proporcionar el argumento vemos el mensaje:

```
./mejoresclientes: fichero_clientes: no suministrado
```

Para ver un ejemplo del operador de cadena `:=` podríamos cambiar:

```
cuantos=$2
```

Por:

```
cuantos=${2:=5}
```

Pero esto no funciona porque estamos intentando asignar un valor a un parámetros posicional (que son de sólo lectura). Lo que sí podemos hacer es:

```
cuantos=$2
.........
sort -nr $fichero_clientes | head -${cuantos:=$defecto}
```

El Listado 4.9 muestra el resultado de hacer nuestro script más tolerante a errores:

```
# Script que saca los mejores clientes
# Tiene la forma
#    mejoresclientes <fichero> [<cuantos>]

fichero_clientes=$1
fichero_clientes=${fichero_clientes:?'no suministrado'}
cuantos=$2
defecto=5
sort -nr $fichero_clientes | head -${cuantos:=$defecto}
```

Listado 4.9: Script que lista los mejores clientes mejorado

3.2. Operadores de búsqueda de patrones

En la Tabla 4.2 se muestran los operadores de búsqueda de patrones existentes y una descripción de su comportamiento.

Un uso frecuente de los operadores de búsqueda de patrones es eliminar partes de la ruta de un fichero, como pueda ser el directorio o el nombre del fichero. Vamos a exponer algunos ejemplos de cómo funcionan estos operadores: Supongamos que tenemos la variable `ruta` cuyo valor es:

```
ruta=/usr/local/share/qemu/bios.core.bin
```

Y ejecutamos los siguientes operadores de búsqueda de patrones, entonces los resultados serían los siguientes:

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

En la búsqueda de patrones se pueden usar tanto los comodines tradicionales que vimos en el apartado 2 del Tema 1, como los comodines extendidos que vimos en el apartado 2.4 de ese mismo tema.

Operador	**Descripción**
`${`*`var`*`#`*`patron`*`}`	Si *`patron`* coincide con la primera parte del valor de *`var`*, borra la parte más pequeña que coincide y retorna el resto.
`${`*`var`*`##`*`patron`*`}`	Si *`patron`* coincide con la primera parte del valor de *`var`*, borra la parte más grande que coincide y retorna el resto.
`${`*`var`*`%`*`patron`*`}`	Si *`patron`* coincide con el final del valor de *`var`*, borra la parte más pequeña que coincide y retorna el resto.
`${`*`var`*`%%`*`patron`*`}`	Si *`patron`* coincide con el final del valor de *`var`*, borra la parte más grande que coincide y retorna el resto.
`${`*`var`*`/`*`patron`*`/`*`cadena`*`}` `${`*`var`*`//`*`patron`*`/`*`cadena`*`}`	La parte más grande de *`patron`* que coincide en *`var`* es reemplazada por *`cadena`*. La primera forma sólo reemplaza la primera ocurrencia, y la segunda forma reemplaza todas las ocurrencias. Si *`patron`* empieza por `#` debe producirse la coincidencia al principio de *`var`*, y si empieza por `%` debe producirse la coincidencia al final de *`var`*. Si *`cadena`* es nula se borran las ocurrencias. En ningún caso `var` se modifica, sólo se retorna su valor con modificaciones.

Tabla 4.2: Operadores de búsqueda de patrones

Ejercicio 4.2

En el mundo de GNU existen unas herramientas llamadas NetPBM[1] que permiten convertir entre muchos formatos gráficos. La herramienta suele convertir de formatos conocidos (gif, bmp, jpg) a un formato interno, o bien del formato interno a los formatos conocidos. Los formatos internos que utiliza son `.ppm` (Portable Pixel Map) para imágenes en color, `.pgm` (Portable Gray Map) para imágenes en escala de grises, y `.pbm` (Portable Bit Map) para imágenes formadas por bits de blanco y negro. A veces estos formatos aparecen bajo la extensión general `.pnm`, que abarca a todos ellos.

Nuestro objetivo es hacer un script llamado `bmptojpg` que reciba uno o dos nombres de fichero, el primero de tipo `.bmp` y el segundo de tipo `.jpg`. Si no se suministra el segundo argumento, el nombre del fichero será el mismo que el del primer argumento pero con la extensión `.jpg`. Para realizar las conversiones usaremos los comandos de NetPBM `bmptoppm` y `ppmtojpeg`. Los comandos reciben como argumento el fichero origen y emiten por la salida estándar el fichero en el formato destino.

Para obtener el primer argumento, o dar error si no se nos suministra, podemos usar el operador `:?` así:

```
fichero_entrada=${1:?'falta argumento'}
```

Para obtener los nombres de fichero intermedio y de salida usamos:

```
fichero_intermedio=${fichero_entrada%.bmp}.ppm
fichero_salida=${2:-${fichero_intermedio%.ppm}.jpg}
```

Obsérvese que para el nombre de salida usamos el operador `:-` para que si no se ha suministrado el segundo argumento usemos el nombre del fichero de entrada, pero con la extensión cambiada.

Luego el script que realiza esta operación se muestra en el Listado 4.10.

```
# Convierte un .bmp en un .jpg

fichero_entrada=${1:?'falta argumento'}
fichero_intermedio=${fichero_entrada%.bmp}.ppm
fichero_salida=${2:-${fichero_intermedio%.ppm}.jpg}

bmptoppm $fichero_entrada > $fichero_intermedio
ppmtojpeg $fichero_intermedio > $fichero_salida
```

Listado 4.10: Script que convierte un `.bmp` en un `.jpg`

[1] El paquete se encuentra en http://netpbm.sourceforge.net. En el caso de Mac OS X puede descargarlo directamente del proyecto Fink.

Este ejercicio, tal como está ahora, puede producir situaciones extrañas si el fichero de entrada no tiene la extensión `.bmp`, además el comando fallaría si el fichero de entrada no existe. En el siguiente tema usaremos los nuevos conocimientos que desarrollemos para arreglar estos problemas.

Ejercicio 4.3

Los directorios de búsqueda que almacena la variable de entorno `PATH` a veces resultan difíciles de ver debido a que cuesta encontrar el delimitador dos puntos (`:`). Escribir un script llamado `verpath` que muestre los directorios del `PATH` uno en cada línea.

Para hacer esto podemos sustituir el símbolo dos puntos (`:`) por un símbolo de nueva línea `'\n'` usando un operador de búsqueda de patrones de la forma:

```
$ echo -e ${PATH//:/'\n'}
/sw/bin
/sw/sbin
/bin
/sbin
/usr/bin
/usr/local/bin
/usr/sbin
/sw/bin
/usr/X11R6/bin
.
```

3.3. El operador longitud

El operador longitud nos permite obtener la longitud (en caracteres) del valor de una variable. Tiene la forma `${#var}` donde *var* es la variable cuyo valor queremos medir. Por ejemplo si la variable `nombre` vale `Fernando`, `${#nombre}` devolverá 8.

4.Sustitución de comandos

La **sustitución de comandos** nos permite usar la salida de un comando como si fuera el valor de una variable.

La sintaxis de la sustitución de comandos es:

`$(comando)`

En el Bourne Shell y el C Shell se utiliza la comilla hacía atrás, o comilla grave, es decir `` `comando` `` para realizar la sustitución de comandos. Aunque Bash mantiene esta sintaxis por compatibilidad hacía atrás, la forma recomendada es mediante el uso de paréntesis, que permiten anidar sustituciones de comandos.

Un ejemplo de uso de la sustitución de comandos es `$(pwd)`, que nos devuelve el directorio actual, y es equivalente a leer la variable de entorno `$PWD`.

Otro ejemplo es el uso de `$(ls $HOME)`, esta sustitución de comandos nos devuelve una variable con todos los ficheros del directorio home:

```
$ midir=$(ls $HOME)
$ echo $midir
Desktop Documents Library Movies Music Pictures Public
Sites autor jdevhome tmp xcode
```

También podemos cargar el contenido de un fichero en una variable de entorno usando `$(<fichero)`, donde *fichero* es el fichero a cargar. Por ejemplo, para cargar el contenido del fichero `copyright.txt` en la variable de entorno `copyright` hacemos:

```
$ copyright=$(<copyright.txt)
```

El comando `type -path fichero` nos da la ruta donde se encuentra el *fichero*. La opción `-path` hace que se ignoren keywords, comandos internos, etc. Si queremos obtener información detallada del fichero `bmptoppm` podemos usar el comando:

```
$ ls -l $(type -path bmptoppm)
-rwxr-xr-x 1 root  admin  8 12 Jan  2004 /sw/bin/bmptoppm
```

Si tenemos un conjunto de ficheros de la forma `tema*.txt` podemos usar la sustitución de comandos para abrir todos los que traten de Bash así:

```
$ emacs $(grep -l 'bash' tema*.txt)
```

La opción `-l` hace que `grep` devuelva sólo los nombres de fichero donde ha encontrado la palabra `'bash'`.

La sustitución de comandos se realiza dentro de las comillas dobles (o débiles) con lo que podemos extender las regla que vimos en el apartado 4.1 del Tema 3 para utilizar comillas fuertes siempre que dentro no tengamos una variable o sustitución de comando que queramos ejecutar.

Ejercicio 4.4

El comando `ls` no tiene ninguna forma de seleccionar ficheros por una determinada fecha de modificación. Hacer una función que nos devuelva los ficheros del directorio actual cuya fecha de modificación sea la dada en un argumento. Hacer luego un script llamado `lsfecha` al que le pasemos una fecha y nos devuelva los ficheros de ese directorios modificados en esa fecha.

La función que lista los ficheros modificados en una fecha podría buscar la fecha en la salida de `ls -lad` usando `grep`, es decir si ejecutamos:

```
$ ls -lad * | grep "3 Aug"
-rw-r--r--  1 fernando admin  6148  3 Aug 18:44 .DS_Store
-rwxr-xr-x  1 fernando admin   327  3 Aug 18:20 bmptojpg
-rw-r--r--  1 fernando admin 75688  3 Aug 18:18 casa.bmp
-rw-r--r--  1 fernando admin     0  3 Aug 18:19 casa.jpg
-rw-r--r--  1 fernando admin     0  3 Aug 18:20 casa.ppm
```

Sólo obtendremos los ficheros con fecha de modificación `"3 Aug"`. Si ahora con `cut` cortamos de la columna 54 al final obtenemos los nombres de los ficheros con esta fecha de modificación. Esto es lo que hace la función `ListaFecha()` del Listado 4.11.

```
# Script que encuentra los ficheros modificados en una
# determinada fecha con el formato '14 Jun' que usa
# ls -lad para dar las fechas de modificacion

# Funcion que lista ficheros del directorio
# actual modificados en la fecha $1
function ListaFecha
{
 ls -lad * | grep "$1" | cut -c54-
}

ls -lad $(ListaFecha "$1")
```

Listado 4.11: Script que lista los ficheros modificados en la fecha indicada

Para hacer el script tenemos que volver a ejecutar el comando `ls -la`, pero pasándole ahora como argumento los nombres de los ficheros que queremos

listar (los de la fecha que nos interese). Para ello podemos usar una sustitución de comando que ejecute la función `ListaFecha` con el argumento que recibe el script. Obsérvese que `"$1"` aparece entrecomillado, esto es necesario porque sino la fecha (p.e. `3 Aug`), al tener un espacio, hace que pasemos dos parámetros a `ListaFecha` (`3` en `$1`, y `Aug` en `$2`).

Ejercicio 4.5

Para acabar el tema vamos a realizar un ejercicio más completo que usa todo lo aprendido hasta ahora, y que vamos a continuar en el Tema 5 y Tema 6 con los nuevos conceptos de programación que veamos.

Bash dispone de los comandos internos `pushd` y `popd` que nos permiten movernos por los directorios de la siguiente forma: Al ejecutar `pushd` *`directorio`* nos movemos a ese directorio (como con `cd`) y el directorio anterior se guarda en una pila de forma que la hacer un `popd` regresamos al directorio que está en la cima de la pila. El comando `dirs` nos permite ver la pila de directorios que tenemos en cada momento. En parte, estos comandos son parecidos a el comando `cd -`, que nos lleva al anterior directorio en el que estábamos (al guardado en la variable de entorno `OLDPWD`), sólo que `pushd` y `popd` nos permiten almacenar cualquier número de directorios en la pila.

Nosotros en este ejercicio vamos a hacer los comandos `ira` *`directorio`* y `volver` que serían equivalentes a `pushd` y `popd`. Para almacenar la pila de directorios vamos a usar la variable de entorno `PILADIR` en la que almacenaremos los directorios separados por un espacio. Aunque en principio el espacio puede formar parte del nombre de un directorio, hasta que no avancemos más, y veamos los arrays en el Tema 6, vamos a utilizar esta forma de guardar la pila de directorios.

A continuación se muestra un ejemplo de ejecución de estos comandos:

Comando	**PILADIR**	**Directorio resultante**
`ira /usr`	`/usr /Users/fernando`	`/usr`
`ira /Volumen`	`/Volumen /usr /Users/fernando`	`/Volumen`
`ira /tmp`	`/tmp /Volumen /usr /Users/fernando`	`/tmp`
`volver`	`/Volumen /usr /Users/fernando`	`/Volumen`
`volver`	`/usr /Users/fernando`	`/usr`
`volver`	`/Users/fernando`	`~`

La forma de funcionar es la siguiente:

1. La primera vez que se ejecuta `ira()` (cuando `PILADIR` está vacía) se guarda el directorio actual en `PILADIR`, se hace un `cd` al directorio indicado, y se guarda el nuevo directorio en `PILADIR`.

2. Las demás veces que se llama a `ira()` se guarda el directorio destino en la pila.

3. Cuando se llama a `volver()` se saca un directorio de la pila, y se va al directorio que esté en la cima de la pila.

El Listado 4.12 muestra nuestra versión inicial de estas funciones. Podemos cargar el contenido de este fichero en el entorno con:

```
$ source piladir
```

Suponiendo que las funciones están guardadas en el fichero `piladir`.

```
# Funciones que implementan un movimiento en pila
# por los directorios

# Funcion que recibe en $1 el directorio al que cambiar,
# cambia y lo almacena en la pila PILADIR
function ira
{
 PILADIR="$1 ${PILADIR:-$PWD}"
 cd ${1:?"falta el directorio como argumento"}
 echo $PILADIR
}

function volver
{
 PILADIR=${PILADIR#* }
 cd ${PILADIR%% *}
 echo $PILADIR
}
```

Listado 4.12: Versión inicial de las funciones `ira` y `volver`

Vemos que el contenido de las funciones no es muy amplio. Vamos a comentarlo un poco: La primera línea de la función `ira()` lo que hace es guardar el directorio destino `$1` al principio de la pila, además el operador `:-` nos permite asignar el valor por defecto `$PWD` a `PILADIR` si ésta está vacía. La segunda línea cambia al directorio `$1`, y en caso de que el argumento `$1` no se haya suministrado el operador `:?` da un mensaje de error. Recuérdese que comentamos que el operador `:?` siempre sale del script en los shells no interactivos, pero en los shells interactivos (como desde donde nosotros lo ejecutamos) no abandona el shell, simplemente abandona la función.

Respecto a la función `volver()`, su primera línea saca de la pila el directorio de la cima, y la segunda línea coge en nuevo directorio de la cima como directorio destino del comando `cd`.

Tal como están ahora nuestras funciones tenemos algunos problemas que solucionaremos más adelante:

- ¿Qué pasa si el usuario da a `ira()` un directorio que no existe?
- ¿Qué pasa si el usuario intenta hacer un `volver()` cuando la pila está vacía?
- El otro problema es que, tal como están ahora las funciones, estas no funcionan si un directorio tiene un espacio dentro de su nombre.

Tema 5

Control de flujo

Sinopsis:

Las sentencias de control de flujo es un denominador común de la mayoría de los lenguajes de programación, incluido Bash.

En este tema veremos las sentencias de control de flujo (`if`, `else`, `for`, `while`, `until`, `case` *y* `select`*). Supondremos que el lector ya conoce lo que son, hipótesis bastante fácil de asumir teniendo en cuenta su gran difusión. Lo que vamos a hacer a lo largo de este tema es resaltar sus peculiaridades en el caso de Bash, y mostrar ejemplos de su funcionamiento con ejemplos que ayuden a recordar las muchas variantes que de ellas existen.*

1. Las sentencias condicionales

1.1. Las sentencias `if`, `elif` y `else`

La sentencia condicional tiene el siguiente formato:

```
if condicion
then
    sentencias
elif condicion
then
    sentencias
else
    sentencias
fi
```

Este lenguaje nos obliga a que las sentencias estén organizadas con estos retornos de carro, aunque algunas personas prefieren poner los `then` en la misma línea que los `if`, para lo cual debemos de usar el separador de comandos, que en Bash es el punto y coma (`;`) así:

```
if condicion ; then
    sentencias
elif condicion ; then
    sentencias
else
    sentencias
fi
```

1.2. Los códigos de terminación

En UNIX los comandos terminan devolviendo un código numérico al que se llama **código de terminación** (exit status) que indica si el comando tuvo éxito o no[1].

Aunque no es obligatorio que sea así, normalmente un código de terminación 0 significa que el comando terminó correctamente, y un código entre 1 y 255 corresponde a posibles códigos de error[2]. En cualquier caso siempre conviene consultar la documentación del comando para interpretar mejor sus códigos de terminación. Por ejemplo el comando `diff` devuelve 0 cuando no encuentra diferencias entre los ficheros comparados, 1 cuando hay

[1] Para los programadores C, este código coresponde con el retorno de la función `main()`

[2] El código de terminación 127 lo usa el shell para indicar que el comando no ha sido encontrado, con lo que no debe usase por los comandos para evitar confusiones

diferencias, y 2 cuando se produce algún error (p.e que uno de los ficheros pasados como argumento no se puede leer).

La sentencia `if` comprueba el código de terminación de un comando en la *`condicion`*, si éste es 0 la condición se evalúa como cierta. Luego la forma normal de escribir una sentencia condicional `if` es:

```
if comando ; then
    Procesamiento normal
else
    Procesamos el error
fi
```

En el ejercicio 4.5 del apartado 4 del Tema 4 vimos que la función:

```
function ira
{
 PILADIR="$1 ${PILADIR:-$PWD}"
 cd ${1:?"Falta el directorio como argumento"}
 echo $PILADIR
}
```

Siempre metía el directorio en la pila, incluso si el directorio no existía y el cambio de directorio fallaba. Vamos a aprovechar el código de terminación del comando `cd` para no meter este directorio en la pila si el cambio falla:

```
function ira
{
 if cd ${1:?"Falta el directorio como argumento"} ; then
    PILADIR="$1 ${PILADIR:-$OLDPWD}"
    echo $PILADIR
 else
    echo "Directorio $1 no valido"
 fi
}
```

Ahora se nos informa del error si `cd` falla, y no se actualiza la pila. Obsérvese que ahora el cambio de directorio con `cd` se produce antes de actualizar `PILADIR`, con lo que hemos cambiado `$PWD` por `$OLDPWD`.

1.3. Las sentencias `return` y `exit`

Los comandos UNIX retornan un código de terminación indicando el resultado de una operación, pero qué pasa si estamos haciendo una función: ¿Cómo retornamos el código de terminación?.

Para ello podemos usar la variable especial `?`, cuyo valor es `$?`, y que indica el código de terminación del último comando ejecutado por el shell. Por ejemplo:

```
$ cd direrroneo
-bash: cd: direrroneo: No such file or directory
$ echo $?
1
```

Al haber fallado el último comando `$?` vale 1. Sin embargo, si el comando se ejecuta bien `$?` valdrá 0:

```
$ cd direxistente
$ echo $?
0
```

La variable `?` debe de ser leída junto después de ejecutar el comando, luego es muy típico guardar su valor en una variable `ct=$?` para su posterior uso.

El otro elemento que necesitamos para devolver el código de terminación de una función es la sentencia `return [ct]`, la cual hace que abandonemos la función devolviendo el código de terminación que pasemos en *ct*. El parámetro *ct* es opcional y se puede omitir, en cuyo caso se devuelve el valor de la variable `?` en el momento de ejecutarse `return`, es decir, el código de terminación del último comando ejecutado. Si la función termina sin la sentencia `return` (como ha pasado hasta ahora), la función también devuelve el código de terminación de la última sentencia ejecutada.

`return` puede ser usada sólo dentro de funciones y scripts ejecutados con `source`. Por contra, la sentencia `exit [ct]` puede ser ejecutada en cualquier sitio, y lo que hace es abandonar el script (aunque se ejecute dentro de una función). La sentencia `exit` suele usarse para detectar situaciones erróneas que hacen que el programa deba detenerse, aunque a veces se utiliza para "cerrar" un programa.

Ejercicio 5.1

Hacer una función llamada `cd()` que sustituya al comando interno `cd` de forma que al ejecutar `cd`, además de cambiar de directorio, se de un mensaje que indique el antiguo y nuevo directorio.

En principio podríamos hacer la función:

```
cd()
{
 builtin cd "$@"
 echo "$OLDPWD -> $PWD"
```

```
}
```

Como la función `cd()` tiene preferencia sobre el comando interno `cd`, en principio al llamar a `cd` desde la función `cd()` se ejecuta la función y no el comando, con lo que entraríamos en un bucle infinito. Para ejecutar el comando interno y no la función podemos usar el comando interno `builtin`. Obsérvese que, tal como se explicó en el apartado 2.3 del Tema 4, entrecomillar `"$@"` es necesario por si el nombre de directorio tiene espacios.

Tal como está ahora la función, cambia de directorio e imprime el mensaje pero no devuelve el código de terminación del comando interno `cd`. Para ello tendríamos que modificar la función de la siguiente forma:

```
cd()
{
 builtin cd "$@"
 local ct=$?
 echo "$OLDPWD -> $PWD"
 return $ct
}
```

Obsérvese que tenemos que guardar el código de terminación en una variable, porque sino devolveríamos el código de terminación del comando `echo` y no del comando `cd`.

1.4. Operadores lógicos y códigos de terminación

Podemos combinar varios códigos de terminación de comandos mediante los **operadores lógicos** and (representada con `&&`) or (representada con `||`) y not (representada con `!`).

Estas operaciones, al igual que en otros lenguajes como C o Java, funcionan en shortcut, es decir el segundo operando sólo se evalúa si el primero no determina el resultado de la condición. Según esto la operación:

```
if cd /tmp && cp 001.tmp $HOME ; then
    ....
fi
```

Ejecuta el comando `cd /tmp`, y si éste tiene éxito (el código de terminación es 0), ejecuta el segundo comando `cp 001.tmp $HOME`, pero si el primer comando falla ya no se ejecuta el segundo porque para que se cumpla la condición ambos comandos deberían de tener éxito (si un operando falla ya no tiene sentido evaluar el otro).

Muchas veces se utiliza el operador `&&` para implementar un `if`. Para ello se aprovecha el hecho de que si el primer operando no se cumple, no se evalua el segundo. Por ejemplo si queremos imprimir un mensaje cuando `elementos` sea 0 podemos hacer:

```
[ elementos = 0 ] && echo "No quedan elementos"
```

El operador `||` por contra ejecuta el primer comando, y sólo si éste falla se ejecuta el segundo. Por ejemplo:

```
if cp /tmp/001.tmp ~/d.tmp || cp /tmp/002.tmp ~/d.tmp
then
   ....
fi
```

Aquí si el primer comando tiene éxito ya no se ejecuta el segundo comando (ya que tiene éxito el primer comando, con que se cumple uno de sus operandos).

Obsérvese que a diferencia de C, el código de terminación 0 es el que indica verdadero, y cualquier otro código indica falso.

Por último el operador ! niega un código de terminación. Por ejemplo:

```
if ! cp /tmp/001.tmp ~/d.tmp; then
    Procesa el error
fi
```

1.5. Test condicionales

La sentencia `if` lo único que sabe es evaluar códigos de terminación. Pero esto no significa que sólo podamos comprobar si un comando se ha ejecutado bien. El comando interno `test` nos permite comprobar otras muchas condiciones, que le pasamos como argumento, para acabar devolviéndonos un código de terminación. Una forma alternativa al comando `test` es el operador `[]` dentro del cual metemos la condición a evaluar, es decir, `test cadena1 = cadena2` es equivalente a `[ cadena1 = cadena2 ]`. Los espacios entre los corchetes y los operandos, o entre los operandos y el operador de comparación `=` son obligatorios. Con lo que es coherente con la sintaxis de `if` que hemos visto en el apartado anterior.

Usando los **test condicionales** podemos evaluar atributos de un fichero (si existe, que tipo de fichero es, que permisos tiene, etc.), comparar dos ficheros para ver cual de ellos es más reciente, comparar cadenas, e incluso comparar los números que almacenan las cadenas (comparación numérica).

1.5.1. Comparación de cadenas

Los operadores de comparación de cadenas se resumen en la Tabla 5.1. Las comparaciones que realizan < y > son lexicográficas, es decir comparaciones de diccionario, donde por ejemplo `q` es mayor que `perro`. Obsérvese que no existen operadores <= y >= ya que se pueden implementar mediante operaciones lógicas.

Operador	Verdadero si ...
str1 = *str2* [1]	Las cadenas son iguales
str1 != *str2*	Las cadenas son distintas
str1 < *str2*	*str1* es menor lexicográficamente a *str2*
str1 > *str2*	*str1* es mayor lexicográficamente a *str2*
-n *str1*	*str1* es no nula y tiene longitud mayor a cero
-z *str1*	*str1* es nula (tiene longitud cero)

Tabla 5.1: Operadores de comparación de cadenas

Usando operadores de comparación de cadenas podemos mejorar la función `volver()` del Ejercicio 4.5 del Tema 4 para que si la variable de entorno `PILADIR`, donde guardábamos la pila de directorios, está vacía no intente cambiar de directorio e informe del problema:

```
function volver
{
 if [ -n "$PILADIR" ]; then
    PILADIR=${PILADIR#* }
    cd ${PILADIR%% *}
    echo $PILADIR
 else
    echo "La pila esta vacia, no se cambio de directorio"
 fi
}
```

Hemos puesto `"$PILADIR"` entre comillas para que cuando se expanda se expanda por una sola cadena, y no tantas como directorios haya. Hay otra razón para ponerla entre comillas que se manifestará luego, y es que si `PILADIR` está vacía se expandirá por `[ -n ]` produciendo error. Con las comillas se expandirá por `[ -n "" ]` que es lo que queremos.

Otro ejemplo que podemos mejorar es el del Ejercicio 4.1 del Tema 4. Recuérdese que en el Listado 4.9 habíamos contemplado el caso de que no se suministrara el nombre de fichero de clientes como argumento:

[1] Obsérvese que sólo hay un símbolo =, lo cual confunde a veces a los programadores C.

```
fichero_clientes=$1
fichero_clientes=${fichero_clientes:?'no suministrado'}
```

Aunque si no se suministraba se daba un mensaje un poco críptico para el usuario:

```
./mejoresclientes: fichero_clientes: no suministrado
```

Ahora usando la sentencia `if` vamos a mejorar el programa para que de un mensaje más descriptivo:

```
fichero_clientes=$1
if [ -z "$fichero_clientes" ] ; then
  echo 'Use: mejoresclientes <fichero_clientes> [<N>]'
else
  cuantos=$2
  defecto=5
  sort -nr $fichero_clientes | head -${cuantos:=$defecto}
fi
```

Aunque normalmente se considera mejor técnica de programación el encerrar todo el código dentro de un bloque `if` y de otro bloque `else`, en este caso el bloque `if` corresponde a una situación de error que debería abortar todo el script, con lo que vamos a poner un `exit`, y dejamos el ejemplo como se muestra en el Listado 5.1.

```
# Script que saca los mejores clientes
# Tiene la forma
#    mejoresclientes <fichero> [<N>]

fichero_clientes=$1

if [ -z "$fichero_clientes" ] ; then
  echo 'Use: mejoresclientes <fichero_clientes> [<N>]'
  exit 1
fi

cuantos=$2
defecto=5
sort -nr $fichero_clientes | head -${cuantos:=$defecto}
```

Listado 5.1: Script que lista los mejores clientes mejorado

1.5.2. Comparación numérica de enteros

El shell también permite comparar variables que almacenan cadenas interpretando estas cadenas como números, para ello se deben de utilizar los operadores de la Tabla 5.2.

Operador	Descripción
-lt	Less Than
-le	Less than or Equal
-eq	EQual
-ge	Greater than or Equal
-gt	Greater Than
-ne	Not Equal

Tabla 5.2: Operadores de comparación numérica de cadenas

Los test de condición (las que van entre corchetes `[]`) también se pueden combinar usando los operadores lógicos `&&`, `||` y `!`.

```
if [ condicion ] && [ condicion ]; then
```

También es posible combinar comandos del shell con test de condición:

```
if comando && [ condicion ]; then
```

Además a nivel de test de condición (dentro de los `[]`) también se pueden usar operadores lógicos, pero en este caso debemos de usar los operadores `-a` (para and) y `-o` (para or).

Por ejemplo, la siguiente operación comprueba que `$reintegro` sea menor o igual a `$saldo`, y que `$reintegro` sea menor o igual a `$max_cajero`:

```
if [ $reintegro -le $saldo -a \
     $reintegro -le $max_cajero ]
then
   ....
fi
```

Aunque el operador lógico `-a` tiene menor precedencia que el operador de comparación `-le`, en expresiones complejas conviene usar paréntesis que ayuden a entender la expresión, pero si los usamos dentro de un test de condición conviene recordar dos reglas:

- Los paréntesis dentro de expresiones condicionales deben ir precedidos por el carácter de escape `\` (para evitar que se interpreten como una sustitución de comandos).

- Los paréntesis, al igual que los corchetes, deben de estar separados por un espacio.

Luego la operación anterior se puede escribir como:

```
if [ \( $reintegro -le $saldo \) -a \
     \( $reintegro -le $max_cajero \) ]
then
   ....
fi
```

1.5.3. Comprobar atributos de ficheros

El tercer tipo de operadores de comparación nos permiten comparar atributos de fichero. Existen 22 operadores de este tipo que se resumen en la Tabla 5.3.

Operador	Verdadero si ...
`-a` *fichero*	*fichero* existe
`-b` *fichero*	*fichero* existe y es un dispositivo de bloque
`-c` *fichero*	*fichero* existe y es un dispositivo de carácter
`-d` *fichero*	*fichero* existe y es un directorio
`-e` *fichero*	*fichero* existe (equivalente a `-a`)
`-f` *fichero*	*fichero* existe y es un fichero regular
`-g` *fichero*	*fichero* existe y tiene activo el bit de setgid
`-G` *fichero*	*fichero* existe y es poseído por el group ID efectivo
`-h` *fichero*	*fichero* existe y es un enlace simbólico
`-k` *fichero*	*fichero* existe y tiene el stricky bit activado
`-L` *fichero*	*fichero* existe y es un enlace simbólico
`-N` *fichero*	*fichero* existe y fue modificado desde la última lectura
`-O` *fichero*	*fichero* existe y es poseído por el user ID efectivo
`-p` *fichero*	*fichero* existe y es un pipe o named pipe
`-r` *fichero*	*fichero* existe y podemos leerlo
`-s` *fichero*	*fichero* existe y no está vacío
`-S` *fichero*	*fichero* existe y es un socket
`-u` *fichero*	*fichero* existe y tiene activo el bit de setuid
`-w` *fichero*	*fichero* existe y tenemos permiso de escritura
`-x` *fichero*	*fichero* existe y tenemos permiso de escritura, o de búsqueda si es un directorio
fich1 `-nt` *fich2*	La fecha de modificación de `fich1` más moderna que (Newer Than) la de `fich2`
fich1 `-ot` *fich2*	La fecha de modificación de `fich1` más antigua que (Older Than) la de `fich2`
fich1 `-ef` *fich2*	*fich1* y *fich2* son el mismo fichero (Equal File)

Tabla 5.3: Operadores para comprobar atributos de fichero

Usando estos operadores vamos a mejorar la función `ira()` del Ejercicio 4.5 del Tema 4 para que antes de entrar en un directorio compruebe que el directorio existe y que tenemos el permiso de listado (permiso `x`) del directorio. Además comprobaremos que el argumento que se nos ha pasado no sea el directorio donde ya estamos `$PWD`, para evitar meter dos veces seguidas el mismo valor en la pila. Según esto la función quedaría así:

```
function ira
{
 if [ -z "$1" ]; then
    echo "Use ira <directorio>"
    return 1
 fi
 if [ "$1" -ef "$PWD" ]; then
    echo "Ya esta en el directorio $PWD"
    return 2
 fi
 if [ \( -d "$1" \) -a \( -x "$1" \) ]; then
    cd $1
    PILADIR="$1 ${PILADIR:-$OLDPWD}"
    echo $PILADIR
 else
    echo "Directorio $1 no valido"
 fi
}
```

Obsérvese que la comparación del directorio a donde cambia `$1` con el `$PWD` la hemos hecho con `-ef`, y no con `=`, ya que de esta forma al ejecutar:

```
$ ira .
Ya esta en el directorio /Users/fernando
```

Estamos comparando `.` con `$PWD`, los cuales no son iguales desde el punto de vista de `=` pero sí lo son desde el punto de vista de `-ef`.

Para comprobar que `$1` sea un directorio válido hemos usado los operadores `-d` y `-x` que comprueban respectivamente que sea un directorio, y que tengamos permiso de cambiar a él.

2. El bucle `for`

El bucle `for` en Bash es un poco distinto a los bucles `for` tradicionales de otros lenguajes como C o Java, sino que se parece más al bucle `for each` de otros lenguajes, ya que aquí no se repite un número fijo de veces, sino que se procesan las palabras de una frase una a una.

Su sintaxis es la siguiente:

```
for var [in lista]
do
   .....
   Sentencias que usan $var
   .....
done
```

Si se omite `in lista`, se recorre el contenido de `$@`, pero aunque vayamos a recorrer esta variable, en este tutorial la indicaremos explícitamente por claridad.

Por ejemplo si queremos recorrer una lista de planetas podemos hacer:

```
for planeta in Mercury Venus Terra Marte Jupiter Saturno
do
  echo $planeta  # Imprime cada planeta en una línea
done
```

La *lista* del bucle `for` puede contener comodines. Por ejemplo, el siguiente bucle muestra información detallada de todos los ficheros en el directorio actual:

```
for fichero in *
do
  ls -l "$fichero"
done
```

Para recorrer los argumentos recibidos por el script, lo correcto es utilizar `"$@"` entrecomillado, ya que como explicamos en el apartado 2.3 del Tema 4 tanto `$*` como `$@` sin entrecomillar interpretan mal los argumentos con espacios, y `"$*"` entrecomillado considera un sólo elemento a todos los argumentos.

Por ejemplo, en el Listado 5.2 hemos modificamos el script `recibe`, que hicimos en el apartado 2.3 del Tema 4, para que recorra con un bucle tanto `"$*"` como `"$@"`:

```
# Bucles que recorren los argumentos

for arg in "$*"
do
   echo "Elemento:$arg"
done

for arg in "$@"
do
   echo "Elemento:$arg"
done
```

Listado 5.2: Bucle for sobre "$*" y "$@"

Si ahora lo ejecutamos obtenemos la siguiente salida:

```
$ recibe "El perro" "La casa"
Elemento:El perro La casa
Elemento:El perro
Elemento:La casa
```

Vemos que "$*" ha interpretado como un sólo elemento a los dos argumentos.

El delimitador que usa el bucle `for` para la variable *`lista`* es el que indiquemos en `IFS`, y por defecto este delimitador es el espacio, aunque en ocasiones (com en el próximo ejercicio) conviene cambiarlo.

Ejercicio 5.2

Hacer un script llamado `listapath` que nos liste todos los ficheros ejecutables que haya en los directorios de la variable de entorno `PATH`. Además deberá de avisar si encuentra en `PATH` un directorio que no exista.

El Listado 5.3 muestra la solución que proponemos al ejercicio. Es script empieza cambiando `IFS` para que se use el símbolo `:` como separador de los elementos de la lista. El bucle `for` de esta forma puede recorrer todos los elementos de la variable `PATH`. En cada iteración del bucle principal se comprueba que el directorio exista, si es así se llama a la función `ListaEjecutables()`, la cual recibe el nombre de un directorio y lista sus ficheros con permiso de ejecución.

Para obtener los ficheros de un directorio en una lista, la función `ListaEjecutables()` usa la opción `-1` del comando `ls`, que hace que cada fichero se escriba en una línea distinta. En este caso además tenemos que cambiar el delimitador de elementos (la variable `IFS`) a un retorno de carro.

```
# Script que muestra todos los ficheros ejecutables
# que hay en el PATH

# Funcion que lista los ejecutables de un directorio
function ListaEjecutables
{
IFS='
'
 ficheros=$(ls -1 $1)
 for fichero in $ficheros
 do
   path_fichero="$1/$fichero"
   if [ -x $path_fichero ]; then
      echo  $path_fichero
   fi
 done
 IFS=':'
}

IFS=':'
for dir in $PATH
do
  if [ -z "$dir" ]; then
     echo "ENCONTRADO UN DIRECTORIO VACIO"
     exit 1
  elif ! [ -d "$dir" ]; then
     echo "$dir NO ES UN DIRECTORIO VALIDO"
     exit 1
  else
     ListaEjecutables $dir
  fi
done
```

Listado 5.3: Script que lista los ejecutables del `PATH`

3.Los bucles `while` y `until`

Los bucles `while` y `until` serán más útiles cuando los combinemos con características que veremos en el próximo tema, como la aritmética con enteros, la entrada y salida de variables, y el procesamiento de opciones de la línea de comandos. Aun así veremos en este apartado algunos ejemplos útiles de su utilización. Su sintaxis es:

```
while comando
do
    .....
done
```

```
until comando
do
    .....
done
```

En este caso el comando también puede ser una condición encerrada entre `[]`. La única diferencia entre `while` y `until` es que `while` se ejecuta mientras que el código de terminación del comando sea exitoso, es decir 0, mientras que `until` se ejecuta hasta que el código de terminación sea exitoso, según esto `until` se puede interpretar como ejecutar varias veces un comando hasta que tenga exito.

Ejercicio 5.3

Volver a hacer un script que muestre los directorios de `PATH`, tal como hicimos en el Ejercicio 4.3, pero usando ahora un bucle `while` y aprovechando el hecho de que una cadena en un test de condición evalúa por verdadero cuando no está vacía, y por falso cuando está vacía.

```
# Script que muestra los directorios de PATH

path=$PATH
while [ $path ]; do
     echo ${path%%:*}
     if [ ${path#*:} = $path ]; then
        path=
     else
        path=${path#*:}
     fi
done
```

Listado 5.4: Script que lista los directorios de `PATH`

El Listado 5.4 muestra la solución propuesta. La comprobación `[ ${path#*:} = $path ]` se hace para que cuando quede un sólo directorio sin dos puntos pongamos la variable `path` a vacía, ya que el patrón del operador de búsqueda de patrones no se cumple si la cadena no tiene el símbolo dos puntos (`:`).

4. La sentencia `case`

Mientras que esta sentencia en lenguajes como C o Java se usa para comprobar el valor de una variable simple, como un entero o un carácter, en Bash esta sentencia permite realizar una comparación de patrones con la cadena a examinar.

Su sintaxis es la siguiente:

```
case cadena in
   patron1)
     Sentencias ;;
   patron2)
     Sentencias ;;
   .....
esac
```

Cada patrón puede estar formado por varios patrones separados por el carácter `|`. Si `cadena` cumple alguno de los patrones, se ejecutan sus correspondientes sentencias (las cuales se separan por `;`) hasta `;;`.

Ejercicio 5.4

Utilizando los comandos NetPBM que vimos en el Ejercicio 4.2 hacer un script llamado `tojpg` que reciba uno o más nombres de ficheros con las extensiones `.tga`, `.pcx`, `.xpm`, `.tif` o `.gif`, y genere ficheros con el mismo nombre pero con la extensión `.jpg`.

Puede usar los comandos `tgatoppm`, `xpmtoppm`, `pcxtoppm`, `tifftopnm` y `giftopnm` para generar un fichero `.ppm` intermedio, y luego usando `ppmtojpeg` obtener el fichero `.jpg` final.

El Listado 5.5 muestra la solución propuesta. Obsérvese que como podemos recibir un número variable de ficheros como argumento hemos hecho un bucle `for` sobre el parámetro `"$@"`.

```
# Script que convierte ficheros de imagen al formato .jpg

for fichero in "$@"
do
   fichero_ppm=${fichero%.*}.ppm
   case $fichero in
        *.jpg) exit 0;;
        *.tga) tgatoppm $fichero > $fichero_ppm;;
        *.xpm) xpmtoppm $fichero > $fichero_ppm;;
        *.pcx) pcxtoppm $fichero > $fichero_ppm;;
```

```
        *.tif) tifftopnm $fichero > $fichero_ppm;;
        *.gif) figtoppm $fichero > $fichero_ppm;;
        *.pnm|*.ppm) ;;
        *) echo "Formato .${fichero##*.} no soportado"
           exit 1;;
   esac
   fichero_salida=${fichero_ppm%.ppm}.jpg
   pnmtojpeg $fichero_ppm > $fichero_salida
   if ! [ $fichero = $fichero_ppm ]; then
      rm $fichero_ppm
   fidone
```

Listado 5.5: Script que convierte diversos formatos de imagen a `.jpg`

Ejercicio 5.5

Escribir una función que implemente el comando del Korn Shell `cd` *`old new`*, donde cuando este comando recibe dos argumentos, coge el path del directorio actual y busca el patrón *`old`* , si lo encuentra lo sustituye por *`new`* y intenta cambiar a ese directorio. Cuando recibe cero o un argumentos actua como normalmente.

Por ejemplo si en el directorio `/Users/fernando` escribimos:

```
$ cd fernando carolina
/Users/carolina
```

Cambia `fernando` por `carolina` en el path, y cambia a ese directorio.

El Listado 5.6 muestra la solución propuesta. Recuérdese que el número de argumentos recibidos venía en `$#`, y cuando vale 2 es cuando aplicamos la sustitución.

```
# Funcion que se comporta como en el Korn Shell

function cd
{
 case $# in
      0|1) builtin cd $@ ;;
      2)   destino=${PWD//$1/$2}
           if [ $destino = $PWD ]; then
              echo " sustitucion a $destino no valida"
           elif ! cd $destino ; then
              echo "Directorio $destino no valido"
           fi;;
      *)   echo "Numero erroneo de argumentos";;
 esac
}
```

Listado 5.6: Implementación del comando `cd` del Korn Shell

5. La sentencia `select`

La sentencia `select` nos permite generar fácilmente un menú simple. Su sintaxis es la siguiente:

```
select variable [in lista]
do
   Sentencias que usan $variable
done
```

Vemos que tiene la misma sintaxis que el bucle `for`, excepto por la keyword `select` en vez de `for`. De hecho si omitimos `in lista` también se usa por defecto `$@`.

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

Aunque el bucle de `select` es infinito (lo cual nos permite volver a pedir una opción cuantas veces haga falta), el bucle se puede abandonar usando la sentencia `break`. La sentencia `break`, al igual que en C y Java, se usa para abandonar un bucle, y se puede usar en el caso, tanto de `select`, como de los bucles `for`, `while` y `until`. Pero a diferencia de C y Java no sirve para abandonar la sentencia `case`, sino que ésta se abandona usando los dos puntos comas `;;`.

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

Ejercicio 5.6

Escribir una función llamada `elegirdir()` que permita al usuario elegir un directorio de los disponibles en la variable de entorno `PILADIR` (del Ejercicio 4.5). El directorio elegido se mueve al principio de la pila `PILADIR` y se convierte en el directorio actual.

El Listado 5.7 muestra la solución propuesta. La función cambia el prompt `PS3` antes de ejecutar `select`. El cuerpo de `select` se ejecutará tantas veces haga falta, hasta que se elija una opción válida (`$dir` no sea nula). El resto de funcionalidad ya debería ser conocida por el lector.

```
function elegirdir
{
 if [ -z "$PILADIR" ]; then # Si no hay directorios error
    echo "No hay directorios en la pila"
 fi
 PS3='Eliga directorio:'
 select dir in $PILADIR
 do
 if [ $dir ]; then # Se ha elegido directorio valido
    if [ -z "${PILADIR%%* *}" ]
    then # Mas de un dir en la pila
       piladir=$PILADIR
       PILADIR="$dir ${piladir%%$dir*}"
       PILADIR="$PILADIR ${piladir##*$dir}"
    fi
    cd $dir
    echo $PILADIR
    break
 else
    echo "Opcion no valida"
 fi
 done
}
```

Listado 5.7: Implementación de la función `elegirdir()`

En ejemplo de ejecución es el siguiente:

```
$ echo $PILADIR
/var/ /usr /tmp/ /Users/fernando
$ elegirdir
1) /var/
2) /usr
3) /tmp/
4) /Users/fernando
Eliga directorio:3
/tmp/ /var/ /usr /Users/fernando
```

Tema 6

Opciones de la línea de comandos, expresiones aritméticas y arrays

Sinopsis:

En este momento ya somos capaces de realizar nuestros propios scripts, aunque todavía faltan por conocer como se realizan otras tareas comunes de programación.

A lo largo de este tema aprenderemos a realizar dos operaciones frecuentes en la programación de scripts: La primera es recibir opciones (precedidas por un guión) en la línea de comandos. La segunda es aprender a realizar operaciones aritméticas con las variables, de esta forma superaremos la limitación que estábamos teniendo, de que todas nuestras variables sólo contenían cadenas de caracteres.

1. Opciones de la línea de comandos

Es muy típico que los comandos UNIX tengan el formato:

```
comando [-opciones] argumentos
```

Es decir, las opciones suelen preceder a los argumentos y tienen un guión delante.

Las opciones, al igual que los argumentos se reciben en las variables posiciónales, con lo que si por ejemplo ejecutamos `hacer -o esto.txt aquello.txt`, en `$1` recibimos `-o`, en `$2` recibimos `esto.txt` y en `$3` recibimos `aquello.txt`. Luego en principio para tratar las opciones no necesitaríamos aprender nada más. El problema está en que normalmente las opciones son "opcionales", es decir, pueden darse o no, con lo que el script que procesa la opción anterior debería tener la forma:

```
if [ $1 = -o ]; then
   Ejecuta la operación con $2 y $3
else
   Ejecuta la operación con $1 y $2
fi
```

En consecuencia, cuando el número de opciones crece, la programación de scripts se vuelve engorrosa.

1.1. La sentencia `shift`

Afortunadamente la sentencia `shift` nos permite solucionar este engorro elegantemente. Esta sentencia tiene el formato:

```
shift [n]
```

Donde *n* es el número de desplazamientos a la izquierda que queremos hacer con los argumentos. Si se omite *n* por defecto vale 1. Luego, en el ejemplo anterior, si ejecutamos el comando `shift 1`, `$1` pasará a ser `esto.txt`, `$2` pasa a ser `aquello.txt`, y la opción se pierde.

Esto nos permite hacer es script anterior más sencillo:

```
if [ $1 = -o ]; then
   Procesa -o
   shift
fi
Ejecuta la operación con $1 y $2
```

Podemos extender el ejemplo anterior a un comando `hacer` que recibe tres posibles opciones: `-a`, `-b` y `-c`. La forma de implementarlo sería ahora:

```
while [ -n "$(echo $1 | grep '^-')" ]
do
   case $1 in
        -a) Procesa opción -a;;
        -b) Procesa opción -b;;
        -c) Procesa opción -c;;
        * ) echo 'Use hacer [-a] [-b] [-c] args...'
            exit 1;;
   esac
   shift
done
Procesa los argumentos
```

La condición del bucle busca la expresión regular `'^-'` que significa "cualquier cosa que empieza por guión". Obsérvese que cada vez que procesamos una opción ejecutamos `shift` para desplazar una vez los argumentos.

Algunas opciones tienen sus propios argumentos, en cuyo caso el argumento suele preceder a la opción. Por ejemplo, imaginemos que la opción `-b` recibe a continuación un argumento. En este caso deberíamos de modificar el bucle anterior para leer este argumento de la siguiente forma:

```
while [ -n "$(echo $1 | grep '^-')" ]
do
   case $1 in
        -a) Procesa opción -a;;
        -b) Procesa opción -b
            $2 es el argumento de la opción
            shift;;
        -c) Procesa opción -c;;
        * ) echo 'Use hacer [-a] [-b arg] [-c] args...'
            exit 1;;
   esac
   shift
done
Procesa los argumentos
```

Vemos que ahora también hacemos un `shift` en caso de encontrar la opción `-b`, ya que el argumento de la opción es un argumento más.

1.2. El comando interno `getopts`

Cuando la complejidad de las posibles opciones y argumentos de un comando crece, el comando interno `getopts` nos permite procesar opciones de línea de comandos más cómodamente[1].

Además `getopts` nos permite procesar opciones cuando están agrupadas (p.e. `-abc` en vez de `-a -b -c`) y argumentos de opciones cuando estos no están separados de la opción por un espacio (`-barg` en vez de `-b arg`)[2].

El comando interno `getopts` es una evolución del antiguo comando `getopt` del Bourne Shell, el cual se integra mejor en la sintaxis del lenguaje y es más flexible.

Generalmente el comando `getopts` se usa junto con un bucle `while` como vamos a ver. El comando `getopts` recibe dos argumentos: El primero es una cadena con las letras de las opciones que vamos a reconocer. Si la opción tiene un argumento se precede por dos puntos (`:`). El argumento de la opción lo podemos leer consultando la variable de entorno `OPTARG`. El comando `getopts` coge una de las opciones de la línea de comandos y la almacena en una variable cuyo nombre se da como segundo argumento. Mientras que el comando encuentra opciones devuelve el código de terminación 0, cuando no encuentra más opciones devuelve el código de terminación 1.

En ejemplo del apartado anterior lo podemos procesar con `getopts` así:

```
while getopts ":ab:c" opt
do
   case $opt in
        a ) Procesa opción -a;;
        b ) Procesa opción -b
            $OPTARG es el argumento de la opción
        c ) Procesa opción -c;;
        \?) echo 'Use hacer [-a] [-b arg] [-c] args...'
            exit 1;;
   esac
done
shift $(($OPTIND -1))
Procesa los argumentos
```

Obsérvese que en la variable `opt` obtenemos la opción sin guión delante. Por desgracia, por defecto `getopts` produce un error de la forma

[1] Un inconvenente que tiene este comando es que las opciones deben de ser de una sóla letra, es decir `-a` sería una opción válida, pero `-activa` no.

[2] Las Command Syntax Standard Rules del UNIX User's Manual desaconsejan esta forma de pasar argumentos a las opciones, aunque en la práctica muchos comandos UNIX la utilizan.

`cmd:getopts:illegal option` si pasamos una opción no válida. El símbolo `:` delante de la cadena de opciones se usa para evitar que `getopts` produzca un mensaje de error si recibe una opción inválida. Lo que sí que ocurre es que al recibir una opción inválida la variable `opt` se queda valiendo `?`, y se ejecuta el caso que informa al usuario del error. Obsérvese que `?` está precedido por el carácter de escape debido a que es un carácter especial que no queremos que se interprete como parte del patrón.

Obsérvese también que no hacemos `shift` dentro del bucle ya que `getopts` es lo suficientemente inteligente como para devolvernos cada vez una opción distinta. Lo que sí hemos hecho al final del bucle es desplazar tantas veces como argumentos con opciones hayamos encontrado. Para ello `getopts` almacena en la variable de entorno `OPTIND` el número del primer argumento a procesar. Por ejemplo en `hacer -ab esto.txt` la variable `$OPTIND` valdrá 3, y en `hacer -a -b esto.txt` la variable `$OPTIND` valdrá 4. La expresión `$(($OPTIND -1))` es una expresión aritmética (que veremos en breve en este tema).

Ejercicio 6.1

Mejorar el script `tojpg` del Ejercicio 5.4 para que tenga la posibilidad de redimensionar imágenes y de ponerlas borde. Para ello el comando pasará a tener el siguiente formato:

`tojpg [-s` *`escala`*`] [-b grosor`*`borde`*`]` *`ficheros`*`...`

Para realizar este ejercicio puede utilizar dos nuevos comandos de NetPBM: El primero es el comando `ppmscale`, el cual tiene la sintaxis:

`pnmscale` *`scale files`*`...`

Que escala las imágenes en el factor que diga *`scale`* (p.e. si *`scale`* es 2 aumenta su tamaño al doble).

El otro comando es:

`pnmmargin` *`borderwidth files`*`...`

El cual pone un borde del grosor *`borderwidth`* a la imagen.

El Listado 6.1 muestra la solución propuesta. Usando `getopts` almacenamos en las variables `escala` y `grosor` los valores de estas opciones. Si estos valores no se proporcionan las variables quedan vacías y durante la ejecución del bucle `for` principal no se tendrá en cuenta esta opción.

```
# Script que convierte ficheros de imagen al formato .jpg

# Lee las opciones
while getopts ":s:b:" opt
do
   case $opt in
        s ) escala=$OPTARG;;
        b ) grosor=$OPTARG;;
        \?) echo "Use: tojpg [-s escala] " \
                 "[-b grosorborde] ficheros..."
            exit 1;;
   esac
done
shift $(($OPTIND -1))

for fichero in $@
do
   # Convierte a .ppm
   fichero_ppm=${fichero%.*}.ppm
   case $fichero in
        *.jpg) exit 0;;
        *.tga) tgatoppm $fichero > $fichero_ppm;;
        *.xpm) xpmtoppm $fichero > $fichero_ppm;;
        *.pcx) pcxtoppm $fichero > $fichero_ppm;;
        *.tif) tifftopnm $fichero > $fichero_ppm;;
        *.gif) figtoppm $fichero > $fichero_ppm;;
        *.pnm|*.ppm) ;;
        *) echo "Formato .${fichero##*.} no soportado"
           exit 1;;
   esac
   # Aplica las opciones
   if [ $escala ]; then
      cp $fichero_ppm aux.$fichero_ppm
      pnmscale $escala aux.$fichero_ppm > $fichero_ppm
      rm aux.$fichero_ppm
   fi
   if [ $grosorborde ]; then
      cp $fichero_ppm aux.$fichero_ppm
      pnmmargin $grosor aux.$fichero_ppm > $fichero_ppm
      rm aux.$fichero_ppm
   fi
   # Genera el fichero final
   fichero_salida=${fichero_ppm%.ppm}.jpg
   pnmtojpeg $fichero_ppm > $fichero_salida
   if ! [ $fichero = $fichero_ppm ]; then
      rm $fichero_ppm
   fi
done
```

Listado 6.1: Script que usa opciones para procesar imágenes

2. Variables con tipo

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

```
declare [-afFirx] [-p] name[=value] ...
```

La Tabla 6.1 describe las opciones que puede recibir este comando. Una peculiaridad de este comando es que para activar un atributo se precede la opción por un guión `-`, con lo que para desactivar un atributo decidieron preceder la opción por un `+`.

Opción	Descripción
`-a`	La variable es de tipo array
`-f`	Mostrar el nombre e implementación de las funciones
`-F`	Mostrar sólo el nombre de las funciones
`-i`	La variable es de tipo entero
`-r`	La variable es de sólo lectura
`-x`	Exporta la variable (equivalente a `export`)

Tabla 6.1: Opciones del comando interno `declare`

Si escribimos `declare` sin argumentos nos muestra todas las variables de entorno. Si usamos la opción `-f` nos muestra sólo los nombres de funciones y su implementación, y si usamos la opción `-F` nos muestra sólo los nombres de las funciones existentes.

Las variables que se declaran con `declare` dentro de una función son variables locales a la función, de la misma forma que si hubiésemos usado el modificador `local`.

La opción `-i` nos permite declarar una variable de tipo entero, lo cual permite que podamos realizar operaciones aritméticas con ella. Por ejemplo, si usamos variables de entorno normales para realizar operaciones aritméticas:

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

Sin embargo si ahora usamos variables de tipo entero:

```
$ declare -i var1=5
$ declare -i var2=4
$ declare -i resultado
$ resultado=$var1*$var2
$ echo $resultado
20
```

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

```
$ resultado=4*6
$ echo $resultado
24
```

E incluso podemos recoger resultados de operaciones con variables inexistentes:

```
$ resultado=4*var_inexistente
$ echo $resultado
0
```

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

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

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

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

Existe otro comando interno llamado `readonly` que nos permite declarar variables de sólo lectura, pero que tiene más opciones que `declare -r`. En concreto `readonly -p` nos muestra todas las variables de sólo lectura:

```
$ readonly -p
declare -ar BASH_VERSINFO='([0]="2" [1]="05b" [2]="0"
[3]="1" [4]="release" [5]="powerpc-apple-darwin7.0")'
declare -ir EUID="503"
declare -ir PPID="686"
declare -ir UID="503"
```

Además se nos indica si la variable es de tipo array (`-a`) o entero (`-i`).

Usando la opción `-f` podemos hacer a una función de sólo lectura (que el usuario no pueda modificar la función). Por ejemplo:

```
$ readonly -f ira
```

Hace que la función `ira()` no se pueda modificar.

Muchas veces se ataca un sistema modificando una función que se sabe que va a ejecutar un script en modo súperusuario (haciendo que la función haga algo distinto, o algo más de lo que hacía originalmente). Para evitar este ataque las funciones que van a ejecutarse en modo súperusuario se deben de definir sólo dentro del script que las usa, aunque a veces se necesitan llamar desde fuera y es recomendable protegerlas con la opción `-r`, ya que una vez que una función se marca como de sólo lectura ya no se puede quitar este permiso. Es decir si usamos la opción `-n` para quitar este atributo, en principio el comando parece funcionar:

```
$ readonly -n ira
```

Pero si luego intentamos redefinir la función se producirá un error indicando que la función sigue siendo de sólo lectura.

3. Expresiones aritméticas

Como hemos visto ya en el apartado 1.2, las expresiones aritméticas van encerradas entre `$((` y `))`[1].

Las expresiones aritméticas, al igual que las variables y la sustitución de comandos, se evalúan dentro de las comillas blandas, con lo que finalmente vamos a formular la siguiente regla: *Dentro de las comillas blandas se evalúan sólo los elementos que van precedidos por un símbolo de* `$`. Estrictamente hablando esta regla no es del todo cierta, ya que las expresiones aritméticas pueden no tener un símbolo de `$` delante (podemos escribir `((OPTIND -1))` en vez de `$((OPTIND -1))`), pero esto no se recomienda hacerlo por uniformidad del lenguaje.

Ejercicio 6.2

Basándonos en el comando `date +%j`, que nos devuelve el número de día Juliano, hacer un script que nos diga cuantos días quedan hasta el próximo 31 de Diciembre.

El comando se puede implementar restando a 365 días el número de días transcurridos así:

```
$ echo "$(( 365 - $(date +%j) )) dias para el 31 de Dic"
```

3.1. Similitud con las expresiones aritméticas C

Las expresiones aritméticas de Bash se han diseñado de forma equivalente a las expresiones C[2], luego si conoce C ya sabe escribir expresiones aritméticas complejas en Bash. Por ejemplo `$((x+=2))` añade 2 a `x`. Si no conoce C o Java, le recomendamos consultar estos operadores en un manual, ya que en el resto de este tutorial vamos a suponer que el lector conoce estos operadores.

Aunque algunos operadores (p.e. `*` o los paréntesis) son caracteres especiales para Bash, no hace falta precederlos por el carácter de escape siempre que estén dentro de `$((...))`. Igualmente, a las variables que están dentro de la expresión aritmética no hace falta precederlas por `$` para obtener su valor.

[1] También podemos encerrarlas entre `$[` y `]`, pero esta forma está desestimada por Bash, con lo que no se recomienda usarla.
[2] Además bash añade el operador `**` para exponenciar que no forma parte de C.

Bash además nos permite usar los operadores relacionales (<, >, <= ,>= ,== ,!=) y lógicos de C (!, &&, ||) interpretando, al igual que C, el 1 como cierto y el 0 como falso. Conviene no confundir los operadores relacionales de las expresiones aritméticas con los operadores de comparación numérica de cadenas que vimos en la Tabla 5.2. Los primeros van dentro de `$((...))`, mientras que los segundos forman parte de una expresión que pasamos al comando interno `test` o encerramos entre corchetes `[...]`. Lo que sí se cumple es que `$((...))` devuelve el código de terminación 0 cuando la expresión se evalúa como cierta, y viceversa.

3.2. El comando interno `let`

El comando interno `let` nos permite asignar el resultado de una expresión aritmética a una variable. Tiene la siguiente sintaxis:

`let` *`var=expresion`*

`expresion` es cualquier expresión aritmética y no necesita estar encerrada entre `$((...))`.

El comando `let`, a diferencia del comando `declare -i`, no crea una variable de tipo entero, sino una variable de tipo cadena de caracteres normal. Por ejemplo:

```
$ let a=4*3
$ declare -p a
declare -- a="12"
```

Vemos que `a` es del tipo normal. Mientras que si usamos `declare -i` nos la crea de tipo entero:

```
$ declare -i a=3*4
$ declare -p a
declare -i a="12"
```

Al igual que pasa con las variables normales, las declaradas con `let` no pueden tener espacios entre la variable y el signo `=`, ni entre el signo `=` y el valor. Aunque sí pueden tener espacios si encerramos la expresión entre comillas:

```
$ let x=" (9*5) / 7 "
$ echo $x
6
```

Vemos que la aritmética entera pierde los redondeos.

Ejercicio 6.3

El comando `du` nos dice el tamaño ocupado por cada subdirectorio de un directorio más o menos de la siguiente forma:

```
$ du
3       ./X11-unix
512     ./a503
2       ./Items
536     .
```

En principio nos da el tamaño de disco en bloques que pueden se de 1024, 2048 o 4096 bytes (en concreto en Mac OS X son de 2048 bytes). Si preferimos el tamaño ocupado en KBytes podemos usar la opción `-k`, y si sólo queremos un resumen (la última línea del ejemplo anterior) podemos usar la opción `-s`.

Hacer un script llamado `ocupa` que nos diga el tamaño ocupado por el directorio que le pasemos como argumento en bytes, KB y MB.

```
# Nos dice cuanto ocupa cada subdirectorio
# de un directorio

# Comprueba argumento
if [ -z $1 ]; then
   echo "Use: ocupa <directorio>"
   exit 1
fi

# Obtiene los directorios y su tamano en una lista
lista=$(du -k)
IFS='
'
for fila in $lista
do
    dir=$(echo $fila|cut -f 2)
    let kb=$(echo $fila|cut -f 1)
    let b=1024*kb
    let mb=kb/1024
    echo "$mb MB,       $kb KB, $b B     $dir"
done
```

Listado 6.2: Script que muestra el espacio ocupado por cada subdirectorio de un directorio

El Listado 6.2 muestra la solución propuesta. Obsérvese que la variable `IFS` está declarada de la forma:

```
IFS='
'
```

Ya que por defecto este delimitador es el espacio, y no queremos que el espacio se consideren cambio de palabra, sino que queremos que lo sea el cambio de línea.

3.3. Sentencias de control de flujo aritméticas

Las expresiones aritméticas pueden usarse en las distintas sentencias de control de flujo, en cuyo caso la expresión va entre dobles paréntesis, pero sin el `$` delante, por ejemplo, el **`if` aritmético** tendría la forma:

```
if ((expresión aritmética)); then
   cuerpo
fi
```

O el **`while` aritmético** tendría la forma:

```
while ((expresion aritmética))
do
  cuerpo
done
```

Ejercicio 6.4

Implementar un juego que genere un número aleatorio entre 0 y 99, y que nos lo pregunte hasta que lo acertemos, o fallemos 10 veces. Para ello usar un `if` y un `while` aritmético. Puede generar un número aleatorio leyendo la variable `$RANDOM` la cual cada vez que se lee devuelve un número distinto.

```
let intentos=10
let solucion=$RANDOM%100
while ((intentos-->0))
do
  read -p "Indique un numero:" numero
  if ((numero==solucion)); then
     echo "ACERTASTE ERA $solucion, ENHORABUENA!"
     exit
  elif ((numero<solucion)); then
     echo "Es mayor"
  else
     echo "Es menor"
  fi
done
echo "PERDISTE POR SUPERAR LOS 10 INTENTOS"
```

Listado 6.3: Juego de adivinar un número

El Listado 6.3 muestra la solución propuesta. Obsérvese que al ser una expresión aritmética en la condición del `if` se usa `==` y no `=`. La expresión `intentos-->0` primero comprueba que `intentos` sea mayor a 0, y luego le aplica el operador `--` que decrementa en uno `intentos`.

En el apartado 2 del Tema 5 vimos que el bucle `for` permitía recorrer los elementos de una lista. Existe otro tipo de bucle que se asemeja más a los bucles de C que es el **bucle `for` aritmético**. Este tiene la sintaxis:

```
for (( inicialización ; condición ; actualizacion ))
do
    cuerpo
done
```

En este caso los espacios en `((inicialización;condición; actualizacion))` no son necesarios. La Figura 6.1 muestra el flujo de control de este bucle, que es similar al de C.

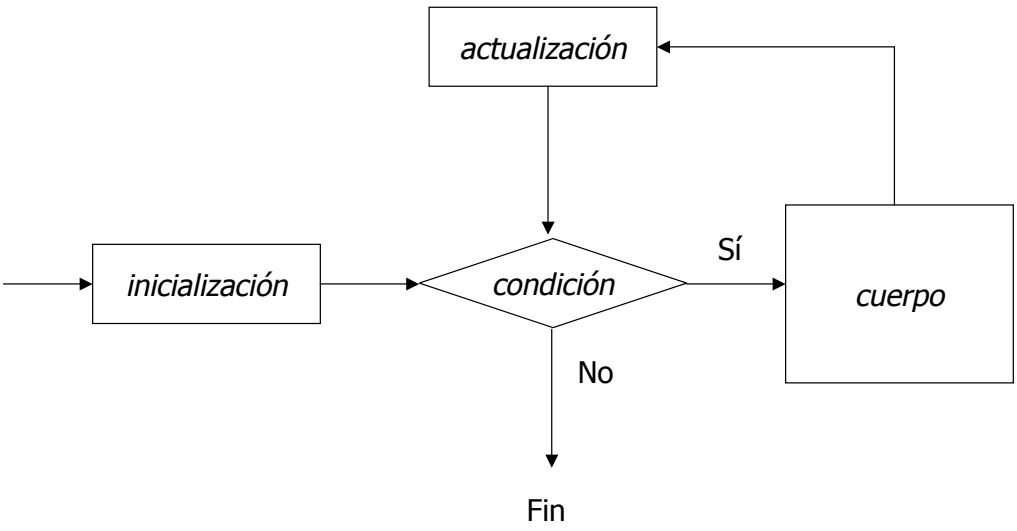

Figura 6.1: Diagrama de flujo del bucle `for`

Ejercicio 6.5

Escribir un script llamado `tablamultiplicar` que pida un número e imprima su tabla de multiplicar. Para pedir el número puede usar la sentencia `read variable` que pide un valor por teclado y lo mete en *`variable`*. El Listado 6.4 muestra la solución propuesta.

En los bucles `for` aritméticos no es necesario utilizar expresiones aritméticas en la inicialización, condición y actualización, pero sí en el cuerpo.

```
echo -n "Indique numero:"
read n
for((i=0;i<=10;i++))
do
  echo "${n}x$i=$(($n*$i))"
done
```

Listado 6.4: Script que imprime la tabla de multiplicar de un número

3.4. Arrays

Bash introdujo los arrays en la versión 2.0. Actualmente Bash sólo soporta trabajar con arrays unidimensionales.

Para declarar un array podemos usar el comando `declare -a` de la siguiente forma:

```
$ declare -a A
```

Si ahora preguntamos por la variable `A`, nos dice que es un array vacío:

```
$ declare -p A
declare -a A='()'
```

Realmente no hace falta declarar un array con `declare -a`, podemos crearlo asignándole directamente valores así:

```
$ B=(Jamon 4 Melon)
$ declare -p B
declare -a B='([0]="Jamon" [1]="4" [2]="Melon")'
```

Los elementos en los arrays Bash empiezan a contar en 0, aunque podemos cambiar los índices de los elementos indicándolos explicitamente:

```
$ C=([5]=Melon [0]=Jamon [3]=400)
$ declare -p C
declare -a C='([0]="Jamon" [3]="400" [5]="Melon")'
```

Obsérvese que no hace falta suministrar los elementos en orden, ni suministrarlos todos, los índices donde no colocamos un valor, simplemente valdrán "cadena nula".

Si sólo indicamos algunos índices, los demás los asigna continuando las cuenta desde el último índice asignado:

```
$ C=([5]=Cosa Casa Perro)
```

```
$ declare -p C
declare -a C='([5]="Cosa" [6]="Casa" [7]="Perro")'
```

Una forma útil de rellenar un array es a partir de las líneas devueltas por una sustitución de comandos usando la forma de inicialización `B=(...)`, y dentro de los paréntesis ponemos la sustitución de comandos. Por ejemplo, para obtener un array con los nombres de ficheros de un directorio podemos usar:

```
$ B=($(ls -1))
$ declare -p B
declare -a B='([0]="Desktop" [1]="Documents"
[2]="Library" [3]="Movies" [4]="Music" [5]=Pictures"
```

Para acceder a los elementos usamos el operador corchete `[]` para indicar el índice del elemento a acceder, y en este caso es obligatorio encerrar entre llaves `{}` la variable:

```
$ echo ${B[2]}
Library
```

Si no indicamos índice de elemento, por defecto nos coge el elemento de índice 0:

```
$ echo $B
Desktop
```

Y ésta es la razón por la que hay que encerrar los elementos entre llaves, porque sino nos coge el primer elemento:

```
$ echo $B[3]
Movies[3]
```

También podemos inicializar un array introduciendo valores directamente con el operador corchete:

```
$ D[2]=Casa
$ D[0]=Avion
$ declare -p D
declare -a D='([0]="Avion" [2]="Casa")'
```

Podemos usar los índices especiales `*` y `@`, los cuales retornan todos los elementos del array de la misma forma que lo hacen los parámetros posiciónales: Cuando no están encerrados entre comillas débiles ambos devuelven una cadena con los elementos separados por espacio, pero cuando se encierran entre comillas débiles `@` devuelve una cadena con los elementos separados por espacio, y `*` devuelve una cadena con los elementos separados por el valor de `IFS`. En cualquiera de los casos, el array se pierde, y lo que recibimos es una cadena de caracteres:

```
$ IFS=,
$ echo ${C[*]}
Cosa Casa Perro
$ echo ${C[@]}
Cosa Casa Perro
$ echo "${C[@]}"
Cosa Casa Perro
$ echo "${C[*]}"
Cosa,Casa,Perro
```

Al igual que en los parámetros posiciónales podemos iterar sobre un array con un bucle `for`:

```
for e in ${C[*]}
do
  echo $e
done
```

Donde al ser `${C[*]}` una cadena de caracteres hubiéramos también podido usar `${C[@]}` o `"${C[@]}"` pero si hacemos `"${C[*]}"` lo que obtenemos es una cadena con los elementos de ésta separados por el valor de `IFS` (coma en este ejemplo), y no por espacio. Sin embargo el bucle hubiera funcionado si los elementos están separados por el valor de `IFS`, y de hecho en ocasiones (p.e. cuando los elementos tienen espacios) conviene utilizar esta última solución.

Los elementos de un array a los que no se asigna valor tienen una cadena nula, con lo que el bucle anterior sólo imprimiría los elementos que existan en el array `C`.

Podemos consultar la longitud de un array usando la forma `${#array[@]}`, donde array es el nombre del *`array`* a consultar su longitud. Por ejemplo, en el ejercicio anterior:

```
$ echo ${#C[@]}
3
```

Nos dice que hay tres elementos aunque los índices lleguen hasta el índice 7, ya que existen posiciones sin asignar en el array, `#` sólo nos devuelve el número de posiciones ocupadas.

También debemos tener en cuenta que dentro de los corchetes va una `@`, y no un índice, ya que sino nos devuelve la longitud (en caracteres) de ese elemento. Por ejemplo, si en el índice 5 teníamos la cadena `"Casa"`:

```
$ echo ${#C[5]}
4
```

Si queremos saber que elementos son no nulos en un array podemos usar la forma `${!array[@]}`, donde *array* es el nombre del array a consultar[1]:

```
$ echo ${!C[@]}
5 6 7
```

Si asignamos un array compuesto a un array se pierden los valores existentes, y se asignan los nuevos. Por ejemplo si a `C` del ejemplo anterior le asignamos:

```
$ C=(Hola "Que tal" Adios)
$ declare -p C
declare -a C='([0]="Hola" [1]="Que tal" [2]="Adios")'
```

Se pierden los elementos 5, 6 y 7 que teníamos asignados.

Podemos borrar una entrada de un array usando `unset` sobre ese índice:

```
$ unset C[2]
$ declare -p C
declare -a C='([0]="Hola" [1]="Que tal")'
```

O bien borrar todo el array usando `unset` sobre el nombre del array:

```
$ unset C
$ declare -p C
bash: declare: C: not found
```

Obsérvese que este comportamiento es distinto al de asignar un valor al nombre del array, que lo que hace es asignárselo al elemento de índice 0.

Por último vamos a comentar que algunas de las variables de entorno de Bash son arrays. Por ejemplo `BASH_VERSINFO` es un array de sólo lectura con información sobre la instancia actual de Bash:

```
$ declare -p BASH_VERSINFO
declare -ar BASH_VERSINFO='([0]="3" [1]="00" [2]="0"
[3]="1" [4]="release" [5]="powerpc-apple-darwin7.9.0")'
```

O por ejemplo `PIPESTATUS` es un array con los códigos de terminación de los comandos del último pipe ejecutado. Por ejemplo si ejecutamos `ls|more`, el comando acaba correctamente, y después consultamos este array:

[1] Esta funcionalidad sólo está disponible a partir de Bash 3.0

```
$ declare -p PIPESTATUS
declare -a PIPESTATUS='([0]="0" [1]="0")'
```

Ejercicio 6.6

Implementar las funciones `ira()`, `volver()` y `elegirdir()` del Ejercicio 5.6 para que los directorios se almacenen en un array, y permitir así que los nombres de directorios tengan espacios.

La solución definitiva a este ejercicio la mostramos y en el Listado 6.5. Esta solución debería ahora de se fácil de seguir por el lector. Reacuérdese que dentro de un bucle `for` aritmético (como el de `elegirdir()`) no era necesario usar expresiones aritméticas en la inicialización, condición y actualización, pero sí en el cuerpo.

```
# Funciones que implementan un movimiento en pila por los
directorios

# Funcion que recibe en $1 el directorio al que cambiar,
# cambia y lo almacena en la pila PILADIR
function ira
{
 if [ -z "$1" ]; then
    echo "Use ira <directorio>"
    return 1
 fi
 if [ "$1" -ef "$PWD" ]; then
    echo "Ya esta en el directorio $PWD"
    return 2
 fi
 if [ \( -d "$1" \) -a \( -x "$1" \) ]; then
    PILADIR[${#PILADIR[@]}]=$PWD
    cd $1
    echo ${PILADIR[@]}
 else
    echo "Directorio $1 no valido"
 fi
}

function volver
{
 if [ ${#PILADIR[@]} -gt 0 ]; then
    cd ${PILADIR[$((${#PILADIR[@]}-1))]}
    unset PILADIR[$((${#PILADIR[@]}-1))]
    echo ${PILADIR[@]}
 else
    echo "La pila esta vacia, no se cambio de directorio"
 fi
}
```

```
function elegirdir
{
 # Si no hay directorios da error
 if [ ${#PILADIR[@]} -eq 0 ]; then
    echo "No hay directorios en la pila"
 fi
 PS3='Eliga directorio:'
 select dir in ${PILADIR[@]}
 do
 if [ $dir ]; then # Se ha elegido directorio valido
    # Mas de un dir en la pila
    if [ ${#PILADIR[@]} -gt 1 ]; then
       aux=${PILADIR[$(($REPLY-1))]}
       echo "Elegida la opcion $REPLY con $aux"
       n_elementos=${#PILADIR[@]}
       for ((i=$REPLY-1;i<n_elementos-1;i++))
           do
           PILADIR[$i]=${PILADIR[$((i+1))]}
           done
       PILADIR[$((n_elementos-1))]=$aux
    fi
    cd $dir
    echo ${PILADIR[@]}
    break
 else
    echo "Opcion no valida"
 fi
 done
}
```

Listado 6.5: Solución a `ira()` `volver()` y `elegirdir()` usando arrays

Tema 7

Redirecciones

Sinopsis:

Este tema trata diversos aspectos relacionados con las redirecciones. Empezaremos estudiando sus tipos en profundidad, para luego estudiar las operaciones de entrada/salida más comunes. En este tema también veremos los llamados bloques de comandos, que nos permiten hacer redirección a un grupo de comandos.

El tema abraca otro grupo de comandos que muchas veces aparecen relacionados con problemas complejos de redirección. Acabaremos viendo formas de hacer que el shell ejecute los comandos que tenemos almacenados en una variable, lo cual nos permitirá crear programas en tiempo de ejecución.

1. Redirecciones

En el apartado 4 del Tema 1 vimos los principales operadores de redirección: `>`, `<`, `>>`, `2>` y `|`. Realmente existen otros muchos operadores de redirección que hemos postergado su explicación hasta ahora. Todos estos operadores están resumidos en la Tabla 7.1.

Operador	Descripción
cmd1`|`*cmd2*	Pipe; Coge la salida estándar de *cmd1* y la envía a la entrada estándar de *cmd2*.
`>`*fichero*	Redirige la salida estándar del programa al *fichero*.
`<`*fichero*	Redirige el contenido de *fichero* a la entrada estándar del programa.
`>>`*fichero*	Redirige la salida estándar del programa al *fichero*. Añade esta a *fichero* si éste ya existe.
`>|`*fichero*	Redirige la salida estándar del programa al *fichero*. Sobrescribe a éste incluso si la opción `noclobber` está activada.
`<>`*fichero*	Usa a *fichero* tanto para la entrada estándar como para la salida estándar.
n`<>`*fichero*	Usa a *fichero* tanto para la entrada como para la salida del descriptor de fichero *n*.
`<<`*etiqueta*	Here document. Véase el apartado 1.3.
`<<<`*texto*	Here string. Véase el apartado 1.3.
n`>`*fichero*	Envía el valor del descriptor de fichero *n* al *fichero*.
n`<`*fichero*	Obtiene el valor del descriptor de fichero *n* del *fichero*.
n`>>`*fichero*	Envía el valor del descriptor de fichero *n* al *fichero*. Añade al final del fichero si éste existe.
n`>|`*fichero*	Envía el valor del descriptor de fichero *n* al *fichero*. Sobrescribe a éste incluso si la opción `noclobber` está activada.
n`>&`	Enlaza la salida estándar en el descriptor de fichero *n*
n`<&`	Enalza la entrada estándar en el descriptor de fichero *n*
n`>&`*m*	Enlaza el descriptor de salida *m* en el descriptor de fichero *n*
n`<&`*m*	Enlaza el descriptor de entrada *m* en el descriptor de fichero *n*
`&>`*fichero*	Redirige la salida estándar y la salida de error estándar al *fichero*
`<&-`	Cierra la entrada estándar
`>&-`	Cierra la salida estándar
n`>&-`	Cierra el descriptor de salida *n*
n`<&-`	Cierra el descriptor de entrada *n*

Tabla 7.1: Operadores de redirección

El operador de redirección < se pueden usar de forma aislada para hacer algo parecido al comando `touch`. Para ello usamos:

```
$ > nuevo.txt
```

Que crea el fichero `nuevo.txt` si no existe, o lo deja con 0 bytes si éste sí que existe.

En Bash existe la opción `noclobber` que por defecto no está activa (y que se puede activar con el comando `set -o noclobber`) que hace que si intentamos sobrescribir un fichero con el operador > (redirigir a un fichero existente) el shell produzca un error y no nos deje ejecutar esta operación. Podemos forzar que se sobrescriban los ficheros, incluso con esta opción activada, con el operador `>|`.

1.1. Los descriptores de fichero

Como ya hemos comentado en el apartado 4 del Tema 1, toda aplicación tiene siempre tres descriptores de fichero abiertos, el 0 para la entrada estándar (stderr), el 1 para la salida estándar (stdout), y el 2 para la salida de errores estándar (stderr).

Nos quedan los números del 3 al 9 para abrir descriptores adicionales[1]. Esto se hace así porque a veces es útil asignar un número de descriptor adicional a los descriptores estándar como si fueran una copia adicional de este enlace. Otras veces resulta útil asignar un número de descriptor adicional a un fichero al que luego nos vamos a referir por su descriptor.

El operador *n*`>&`*m* enlaza el descriptor *n* en el descriptor de salida *m*. Por ejemplo `ls -yz >> resultado.log 2>&1` captura el resultado de la opción ilegal `-yz` y la envía por stdout, en vez de a stderr. Es decir si hacemos:

```
$ ls -yz >> resultado.log
ls: illegal option -- y
usage: ls [-ABCFGHLPRSTWZabcdfghiklnoqrstuvx1] [file ...]
$ ls -yz >> resultado.log 2>&1
```

En el primer comando stderr va a la pantalla, y en el segundo va a el fichero `resultado.log`.

Es importante tener en cuenta que `ls -yz 2>&1 >> command.log` no daría el mismo resultado porque stdout se debe cambiar antes de redirigir

[1] El descriptor 5 puede dar problemas ya que cuando el shell ejecuta un subshell el subproceso hereda este descriptor.

stderr a stdout. Y si lo que hubiéramos ejecutado hubiera sido `ls 1>&2` lo que habríamos hecho es enviar el listado de directorios (stdout) a la salida de errores estándar (stderr).

El operador *n*`<&`*m* enlaza el descriptor *n* en el descriptor de entrada *m*. Por ejemplo el comando `cat 3< clave.h 0<&3` enlaza el descriptor de entrada 3 con el fichero `clave.h`, y luego cambia stdin para que en vez de leer de teclado, lea del descriptor de fichero 3, con lo que el contenido de `clave.h` actúa como entrada al comando `cat`.

Cuando a *n*`>&`*m* se le omite *m* (se usa *n*`&>`), se usa stdout como *m*, y cuando a *n*`<&`*m* se le omite *m* (se usa *n*`<&`), se usa stdin como *m*.

Cuando un descriptor de fichero *n* no está asignado podemos usar *n*`>`*fichero* para asignar el descriptor de fichero de salida *n* al *fichero*, y *n*`<`*fichero* para asignar el descriptor de fichero de entrada *n* al *fichero*. Por ejemplo `ls 3>mis.ficheros 1>&3` asigna el descriptor de fichero de salida 3 al fichero `mis.ficheros`, y después enlaza la salida estándar con el descriptor 3, con lo que la salida de `ls` va al fichero `mis.ficheros`.

También podemos usar el operador *n*`<>`*fichero* para que *n* sea a la vez un descriptor de entrada y de salida para *fichero*. Por ejemplo el siguiente comando lee el fichero `clientes.txt` y luego escribe en él los clientes ordenados:

```
$ cat clientes.txt
Juan
Pedro
Ana
$ sort 3<>clientes.txt 0<&3 1>&3
$ cat clientes.txt
Juan
Pedro
Ana
Ana
Juan
Pedro
```

Obsérvese que los clientes ordenados no sobrescriben a los existentes, sino que se escriben a continuación. Esto se debe a que al leerlos, el puntero a fichero se ha colocado al final, y cuando después escribimos estamos escribiendo al final de éste.

El operador *n*`<>`*fichero* se puede abreviar como `<>`*fichero*, en cuyo caso stdin se convierte en un descriptor de fichero de entrada/salida con fichero como fuente y destino. Es decir, el comando anterior se podría haber escrito como `sort <>clientes.txt 1>&0`. El final del comando `1>&0` se usa

para que stdout se envíe a stdin (el fichero `clientes.txt`). Téngase en cuenta que:

```
$ sort 0<clientes.txt 1>&0
sort: -: write error: Bad file descriptor
```

Falla porque por defecto stdin es un descriptor de sólo lectura.

A veces queremos enviar tanto stdout como stderr a un mismo fichero, siempre podemos redirigir ambos stream de salida así: `gcc *.c 1> errores.log 2>errores.log`, pero una forma alternativa más corta es usar el operador `&>`*`fichero`* de la forma `gcc *.c &>errores.log`.

Si lo que queremos es que el resultado de un comando vaya a la salida de errores estándar, en vez de a la salida estándar, podemos usar `&>2`. Por ejemplo, para dar un mensaje en la stderr podemos hacer:

```
echo "Operación incorrecta" &>2
```

1.2. El comando `exec`

El comando interno `exec` nos permite cambiar las entradas y salidas de un conjunto de comandos (de todos los que se ejecuten a partir de él).

Por ejemplo el Listado 7.1 hemos usado `exec` para que la entrada estándar la lea del fichero `data-file`. El operador *`n`*`<&-` se usa para cerrar el fichero con descriptor *`n`*.

```
exec 6<&0          # Enlaza el descriptor  6 a stdin.
                   # Esto se hace para salvar stdin
exec < data-file   # Reemplaza stdin por el
                   # fichero data-file

read a1            # Lee la primera línea de data-file
read a2            # Lee la segunda línea de data-file

echo
echo "Líneas leidas del fichero"
echo "-------------------------"
echo $a1
echo $a2

exec 0<&6 6<&-     # Restaura stdin
```

Listado 7.1: Script que lee de un fichero

El Listado 7.2 muestra la operación contraría, ahora pretendemos enviar todas las salidas estándar de un conjunto de comandos a un fichero. Obsérvese que el comando sólo captura las salidas estándar de los comandos, para capturar también las salidas de error estándar deberíamos de haber redirigido stderr (p.e. usando `&>fichero`).

```
LOGFILE=logfile.txt

exec 6>&1           # Salva stdout

exec > $LOGFILE     # Reemplaza stdout por
                    # el fichero logfile.log
# Todas las salidas van a logfile.log
.............................
.............................
.............................
exec 1>&6 6>&-      # Retaura stdout y cierra el fichero
```

Listado 7.2: Script que captura las salidas estándar

1.3. Here documents

El operador `<<etiqueta` fuerza a que la entrada a un comando sea la stdin hasta encontrar `etiqueta`. Este texto que va hasta encontrar `etiqueta` es lo que se llama **here document**.

Por ejemplo podemos hacer que `cat` recoja texto hasta que encuentre una línea con un sólo punto, y guarde este texto en el fichero `mensaje.txt` así:

```
$ cat  >> mensaje.txt << .
> Hola.
> Esto es un mensaje de correo
> Adios.
> .
```

El recoger texto de stdin y mandarlo a un comando es lo que ya se hace por defecto, con lo que en principio este operador no sería muy útil, excepto por detectar el final del documento. Pero los here document muchas veces se utilizan para dar la entrada a un comando que no queremos que escriba el usuario, sino que escribimos nosotros mismos dentro de un script. Por ejemplo, el script del Listado 7.3 sube un fichero a un servidor FTP usando el comando `ftp`. El script recibe cinco argumentos, el fichero a subir, el servidor FTP a donde subirlo, el usuario, el password, y la ruta del servidor FTP donde depositarlo. Obsérvese que el here document puede tener variables que se expanden por su valor antes de ser pasadas a la entrada estándar del comando.

```
# Script que sube un fichero a un servidor de FTP
#  Recibe los siguientes argumentos
# $1 Fichero a subir
# $2 Servidor FTP
# $3 Nombre de usuario
# $4 Password
# $5 Directorio destino

ftp << FIN
open $1
$2
$3
cd $4
binary
put $5
quit
FIN
```

Listado 7.3: Script que sube un fichero a un servidor FTP usando el comando `ftp`

Tras ejecutar este comando en mi máquina obtengo la siguiente salida:

```
$ subirfichero ftp.macprogramadores.org fernando *******[1]
/www/comun esto.txt
Connected to ftp.macprogramadores.org.
220---------- Welcome to Pure-FTPd [TLS] ----------
220-You are user number 4 of 50 allowed.
220-Local time is now 17:23. Server port: 21.
Name (ftp.macprogramadores.org:fernando):
331 User fernando OK. Password required
Password:
230-User fernando has group access to:  macprog
230 OK. Current restricted directory is /
Remote system type is UNIX.
Using binary mode to transfer files.
250 OK. Current directory is /www/comun
200 TYPE is now 8-bit binary
local: esto.txt remote: esto.txt
229 Extended Passive mode OK (|||49743|)
150 Accepted data connection
226-File successfully transferred
226 0.191 seconds (measured here), 41.96 bytes per second
8 bytes sent in 00:00 (0.02 KB/s)
221-Goodbye. You uploaded 1 and downloaded 0 kbytes.
221 Logout.
```

[1] Aquí debería de ir el password del servidor no los asteriscos que aparecen

Dejamos como ejercicio al lector el que el comando no muestre en consola la evolución del proceso, sino que sólamente diga si ha tenido éxito o no.

Una variante de los here document son los **here string**, los cuales se llevan a cabo usando el operador `<<<texto`, y que simplemente envían texto a la entrada estándar del comando.

Por ejemplo, si queremos añadir una línea al principio del fichero `esto.txt` y guardarlo en `estonuevo.txt` podemos usar el comando:

```
$ titulo="Mi titulo de ejemplo"
$ cat - esto.txt <<<$titulo > estonuevo.txt
```

El guión como argumento de `cat` (al igual que en otros comandos como `grep` o `sort`) indica que ahí va lo que reciba por la entrada estándar (el here string en nuestro ejemplo). Obsérvese que el here string (al igual que el here document) puede contener variables, en cuyo caso las expanden por su valor antes de pasar el here document al comando.

2. Entrada y salida de texto

En este apartado comentaremos detalladamente los comandos `echo`, `printf` y `read`, los cuales nos permiten realizar las operaciones de entrada/salida que requiere un script.

2.1. El comando interno `echo`

Como ya sabemos, `echo` simplemente escribe en la salida estándar los argumentos que recibe.

La Tabla 7.2 muestra las opciones que podemos pasar a `echo`. La única opción que resulta nueva es `-e`. Esta opción activa la interpretación de las secuencias de escape, que por defecto `echo` las ignora, es decir:

```
$ echo "Hola\nAdios"
Hola\nAdios
$ echo -e "Hola\nAdios"
Hola
Adios
```

Opción	Descripción
`-e`	Activa la interpretación de caracteres precedidos por el carácter de escape.
`-E`	Desactiva la interpretación de caracteres precedidos por el carácter de escape. Es la opción por defecto.
`-n`	Omite el carácter `\n` al final de la línea (es equivalente a la secuencia de escape `\c`).

Tabla 7.2: Opciones del comando `echo`

La Tabla 7.3 muestra las secuencias de escape que acepta `echo` (cuando se usa la opción `-e`). Para que estas tengan éxito deben de encerrarse entre comillas simples o dobles, ya que sino el carácter de escape `\` es interpretado por el shell y no por `echo`.

Secuencia de escape	Descripción
`\a`	Produce un sonido "poom" (alert).
`\b`	Borrar hacía atrás (backspace).
`\c`	Omite el `\n` al final (es equivalente a la opción `-n`). Debe colocarse al final de la línea
`\f`	Cambio de página de impresora (formfeed)
`\n`	Cambio de línea

\r	Retorno de carro
\t	Tabulador

Tabla 7.3: Secuencias de escape del comando echo

El carácter de escape \b se puede usar para borrar texto escrito. Por ejemplo, el Listado 7.4 muestra un script que permite ir escribiendo el porcentaje realizado de un proceso.

```
for ((i=0;i<10;i++))
do
  echo -n "Procesado $i"
  sleep 1
  echo -ne "\b\b\b\b\b\b\b\b\b\b"
done
```

Listado 7.4: Script que muestra un proceso usando \b

El bucle se repite de 0 a 9, sin embargo esto se puede complicar si queremos hacerlo de 0 a 100 ya que estos números tienen distinto número de dígitos. Sin embargo podemos usar \r que retorna el cursor en el terminal sin introducir un cambio de línea (como hace \n). El Listado 7.5 muestra como podría quedar a ahora el script.

```
for ((i=0;i<10;i++))
do
  echo -n "Procesado $i"
  sleep 1
  echo -ne "\r"
done
```

Listado 7.5: Script que muestra un proceso usando \r

2.2. El comando interno `printf`

Aunque el comando echo es suficiente en la mayoría de los casos, en ocasiones, especialmente a la hora de formatear texto en pantalla, es necesaria más flexibilidad, y en este caso se utiliza el comando printf.

Este comando, en principio, puede escribir cadenas igual que echo:

```
$ printf "Hola mundo"
Hola mundo
```

Aunque a diferencia de echo no imprime el cambio de línea del final, sino que debemos indicarlo explícitamente. Además, printf por defecto interpreta las secuencias de escape.

```
$ printf "Hola mundo\n"
Hola mundo
```

El formato de este comando es muy parecido al de la función `printf()` de C:

```
printf cadenaformato [argumentos]
```

Dentro de la **cadena de formato** que recibe como primer argumento pueden aparecer **especificadores de formato**, los cuales empiezan por `%` y se describen en la Tabla 7.4.

Especificador	Descripción
`%c`	Imprime el primer carácter de una variable cadena
`%d`	Imprime un número decimal
`%i`	Igual que `%d` (integer)
`%e`	Formato exponencial `b.precisione[+-]e`
`%E`	Formato exponencial `b.precisionE[+-]e`
`%f`	Número en coma flotante `b.precision`
`%g`	El que menos ocupa entre `%e` y `%f`
`%G`	El que menos ocupa entre `%E` y `%f`
`%o`	Octal sin signo
`%s`	Cadena (string)
`%b`	Interpreta las secuencias de escape del argumento cadena
`%q`	Escribe el argumento cadena de forma que pueda ser usado como entrada a otro comando
`%u`	Número sin signo (unsigned)
`%x`	Hexadecimal con letras en minúscula
`%X`	Hexadecimal con letras en mayúscula
`%%`	`%` literal

Tabla 7.4: Especificadores de formato de `printf`

Un ejemplo de uso de los especificadores de formato es el siguiente:

```
$ n=30
$ nombre=Fernando
$ printf "El cliente %d es %s\n" $n $nombre
El cliente 30 es Fernando
```

El especificador de formato tiene el siguiente formato general:

```
%[Flag][Ancho][.Precision]Especificador
```

El significado de estos campos depende de si el especificador de formato es para una cadena de caracteres o para un número.

En el caso de que sea una cadena *Ancho* se refiere al ancho mínimo que ocupa la cadena. Por ejemplo:

```
$ printf "|%10s|\n" Hola
|      Hola|
```

Si queremos indicar un máximo para la cadena debemos de usar el campo *Precicion*:

```
$ printf "|%-10.10s|\n" "Fernando Lopez"
|Fernando L|
```

El `Flag` - se usa para indicar que queremos alienar a la izquierda:

```
$ printf "|%-10s|\n" Hola
|Hola      |
```

Dos variantes de `%s` son `%b` y `%q`. El especificador de formato `%b` hace que se interpreten las secuencias de escape de los argumentos. Es decir:

```
$ printf "%s\n" "Hola\nAdios"
Hola\nAdios
$ printf "%b\n" "Hola\nAdios"
Hola
Adios
```

El especificador de formato `%q` escribe el argumento de forma que pueda ser usado como entrada de otro comando. Por ejemplo:

```
$ printf "%s\n" "Hola a todos"
Hola a todos
$ printf "%q\n" "Hola a todos"
Hola\ a\ todos
```

Bash no puede manejar expresiones con números en punto flotante, sólo enteros, pero sí puede recibir la salida de un comando que devuelva una cadena con un número en punto flotante. En este caso podemos formatear este número usando `%e`, `%E`, `%f`, `%g`, `%G`. Entonces *Ancho* es el número total de dígitos a utilizar y *Precision* indica el número de dígitos a la derecha del punto. Por ejemplo:

```
$ n=123.4567
$ printf "|%9.3f|\n" $n
|  123.457|
$ printf "|%11.3E|\n" $n
|  1.235E+02|
```

Respecto a los *Flag* que puede tener un número, estos se resumen en la Tabla 7.5.

Flag	Descripción
+	Preceder por + a los números positivos
espacio	Preceder por un espacio a los números positivos
0	Rellenar con 0's a la izquierda
#	Preceder por 0 a los números en octal %o y por 0x a los números en hexadecimal %x y %X

Tabla 7.5: Flags de los especificadores de formato para números

Por ejemplo:

```
$ printf "|%+10d|\n" 56
|       +56|
$ printf "|%010d|\n" 56
|0000000056|
$ printf "|%x|\n" 56
|38|
$ printf "|%#x|\n" 56
|0x38|
```

2.3. El comando interno `read`

La sintaxis de este comandos es:

```
read var1 var2...
```

Esta sentencia lee una línea de la entrada estándar y la parte en palabras separadas por el símbolo que indique la variable IFS (por defecto espacio o tabulador). Las palabras se asignan a *var1*, *var2*, etc. Si hay más palabras que variables las últimas palabras se asignan a la última variable. Por ejemplo:

```
$ read var1 var2 var3
Esto es una prueba
$ echo $var1
Esto
$ echo $var2
es
$ echo $var3
una prueba
```

Si omitimos las variables, la línea entera se asigna a la variable REPLY.

La convención de la programación de shells dice que las preguntas que hagamos al usuario deben hacerse emitiendo un mensaje por la stderr. La razón de usar esta salida es que cuando los usuarios ejecutan una tarea en background suelen redirigir (a `/dev/null`) la stdout para evitar que en su terminal aparezcan mensajes de estado que no desea ver. Luego la forma de hacer una pregunta al usuario debería de ser:

```
echo -n "Responda a esta pregunta: " &>2
read respuesta
```

Resulta que el comando interno `select`, que vimos en el Tema 5, también pide una opción usando stderr.

Aunque el comando `read` es una primitiva básica de la programación convencional. En este tutorial no ha aparecido hasta ahora porque es un "patito feo" de la programación de scripts, ya que los scripts deberían de ser realizados uniendo comandos independientes `cut`, `grep`, `soft`, que van pasándose un texto el cual van procesando. El uso de `read` rompe esta forma de programar, con lo que se recomienda usarlo con moderación. A lo mejor resulta más conveniente pedir un dato como argumento que pedir al usuario que lo introduzca con `read`. O al menos ofrecer las dos opciones.

Antes de acabar con el comando `read` vamos a comentar las principales opciones del comando `read`, que se resumen en la Tabla 7.6.

Opción	Descripción
`-a`	Permite leer las palabras como elementos de un array
`-d`	Permite indicar un delimitador de fin de línea
`-e`	Activa las teclas de edición de readline
`-n` *max*	Permite leer como máximo *max* caracteres.
`-p`	Permite indicar un texto de prompt
`-t`	Permite indicar un timeout para la operación de lectura

Tabla 7.6: Opciones del comando `read`

La primera opción que vamos a comentar es `-a`, que permite leer las palabras como elementos de un array.

```
$ read -a frase
Hola que tal
$ declare -p frase
declare -a frase='([0]="Hola" [1]="que" [2]="tal")'
```

La opción `-d` nos permite indicar un delimitador de fin de línea de forma que la línea se lee hasta encontrar este delimitador.

La opción `-e` es recomendable usarla en general siempre, ya que permite que se puedan usar todas las combinaciones de teclas de readline en el prompt de `read`.

La opción `-n` nos permite especificar un número máximo de caracteres a leer. Si se intentan escribir más caracteres que los indicados en esta opción simplemente se acaba la operación de `read`.

La opción `-p` nos permite aportar un texto de prompt al comando que se imprime antes de pedir el dato:

```
$ read -p "Cual es tu nombre:"
Cual es tu nombre:Fernando
```

Por último la opción `-t` nos permite dar un tiempo máximo para el prompt, momento a partir del cual se continua con el script. Esto es especialmente útil para los scripts de instalación, donde a veces el administrador no está presente esperando a que la instalación acabe.

3. Los bloques de comandos

Podemos usar los operadores de redirección para cambiar la entrada o salida estándar de una función. El Listado 7.6 muestra una función que usamos para que, consultando el fichero `/etc/group`, nos devuelva el ID de un grupo de usuarios. Por ejemplo, si este script está almacenado en un fichero llamado `iddegrupo`:

```
$ iddegrupo kmem
2
```

Lo peculiar de la función es que cuando la ejecutamos estamos cambiando su entrada estándar para que `read` lea de esta entrada.

```
function IDdeGrupo
{
 IFS=:
 while read grupo asterisco ID resto
 do
   if [ $1 = $grupo ]; then
      echo $ID
      return
   fi
 done
}

IDdeGrupo $1 < /etc/group
```

Listado 7.6: Función a la que redirigimos la entrada

Para que esta función use como entrada estándar el fichero `/etc/group` necesitamos redireccionar su entrada cuando la ejecutamos. Si queremos que la entrada estándar de una función siempre esté redirigida al mismo fichero podemos indicarlo en su implementación como muestra el Listado 7.7.

```
function IDdeGrupo
{
 IFS=:
 while read grupo asterisco ID resto
 do
   if [ $1 = $grupo ]; then
      echo $ID
      return
   fi
 done
} < /etc/group
```

Listado 7.7: Función con la entrada redirigida

Ahora podemos ejecutar la función (además del script):

```
$ source iddegrupo
$ IDdeGrupo www
70
```

Incluso, si el único que va a utilizar esa entrada es el bucle `while`, podemos redirigir la entrada estándar durante la ejecución del bucle `while` como muestra el Listado 7.8.

```
function IDdeGrupo
{
 IFS=:
 while read grupo asterisco ID resto
 do
   if [ $1 = $grupo ]; then
      echo $ID
      return
   fi
 done < /etc/group
}
```

Listado 7.8: Redirigir la entrada de un bucle

Por último, también podemos redirigir la entrada de un conjunto de comandos creando un llamado **bloque de comandos**, los cuales encierran un conjunto de comandos entre llaves, tal como muestra el Listado 7.9.

```
{
 IFS=:
 while read grupo asterisco ID resto
 do
   if [ $1 = $grupo ]; then
      echo $ID
      exit
   fi
 done
} < /etc/group
```

Listado 7.9: Bloque de comandos con la entrada redirigida

Lógicamente, también podemos modificar la salida estándar de un bloque de comandos. El Listado 7.10 muestra un ejemplo de como redirigir esta salida a un fichero.

```
{
 IFS=:
 while read grupo asterisco ID resto
 do
   if [ $1 = $grupo ]; then
      echo $ID
      exit
   fi
 done
} < /etc/group > idbuscado
```

Listado 7.10: Ejemplo de redirección de la entrada y salida de un bloque de comandos

También podemos redirigir la salida de un bloque de comandos a otro comando. El Listado 7.11 muestra un ejemplo de un bloque de comandos que pasa su salida a `cut` usando un pipe. En este caso, el bloque de comandos tiene redirigida su entrada estándar para leer de `/etc/group`, y la salida del bloque de comandos se envía a través de un pipe a `cut`.

```
{
 IFS=:
 while read grupo asterisco ID resto
 do
   if [ $1 = $grupo ]; then
      echo "$grupo:$asterisco:$ID:resto"
      break
   fi
 done < /etc/group
} | cut -d ':' -f 3
```

Listado 7.11: Ejemplo de redirección de la salida de un bloque de comandos con un pipe

En el apartado 1.2 de este tema vimos que podíamos usar el comando `exec` para modificar la entrada o salida estándar de un conjunto de comandos, sin embargo el uso de bloques de comandos es una solución más elegante, y se recomienda siempre que se pueda usar.

4. Los comandos `comand`, `builtin` y `enable`

Como vimos en el Tema 4, cuando introducimos un comando en Bash el orden de preferencia en la búsqueda del símbolo por parte de Bash es: Primero las funciones, luego los comandos internos y por último los ficheros de scripts y ejecutables del `PATH`. Los comandos internos `command`, `builtin` y `enable` nos permiten alterar este orden de preferencia.

`command` hace que no se busquen alias ni nombres de funciones, sólo comandos internos y comandos de fichero. En el Ejercicio 5.1 redefinimos en comando `cd` creando la función `cd()`. Aunque usamos `builtin` para implementarlo, éste también es un buen ejemplo de lugar donde podemos usar `command` para evitar que la función `cd()` se llame a sí misma.

```
cd()
{
 command cd "$@"
 local ct=$?
 echo "$OLDPWD -> $PWD"
 return $ct
}
```

`builtin` es similar a `command`, pero es más restrictivo, sólo busca comandos internos.

`enable` nos permite desactivar el nombre de un comando interno, lo cual permite que un comando de fichero pueda ejecutarse sin necesidad de dar toda la ruta del fichero. Un ejemplo de comando interno que muchas veces da lugar a errores durante las primeras lecciones de programación en Bash es el comando interno `test`. Muchas veces queremos probar algo y creamos un script con el nombre `test`, pero cuando vamos a ejecutarlo resulta que no funciona como esperamos porque lo que estamos haciendo es ejecutar el comando interno `test`, y no nuestro script de prueba.

Podemos usar el comando `enable -a` para ver todos los comandos internos, y si están habilitados o no.

5. El comando interno `eval`

El comando `eval` nos permite pasar el valor de una variable al interprete de comandos para que lo ejecute. Por ejemplo:

```
$ eval "ls"
```

Hace que el interprete de comandos ejecute `ls`.

Aunque en principio puede parecer un poco extraño, el comando `eval` resulta muy potente ya que el programa puede construir programas en tiempo de ejecución y luego ejecutarlos. Estrictamente hablando no necesitaríamos disponer del comando `eval` para hacer esto con Bash, ya que siempre podemos crear un fichero con los comandos a ejecutar y luego ejecutarlo usando `source`. Pero `eval` nos evita el tedio de tener que crear un fichero.

La cadena que pasemos a `eval` puede contener variables, lo cual aumenta su flexibilidad. Por ejemplo:

```
$ fich=resultado.txt
$ eval "ls -la > $fich"
```

El comando que pasa `eval` a Bash sería `ls -la > resultado.txt`.

El argumento de `eval` también se podría haber pasado entre comillas fuertes, pero en este caso Bash recibiría el comando `ls -la $fich`, el cual produce el mismo resultado, ya que Bash expande `$fich` antes de ejecutar el comando.

El formato general de `eval` es:

```
eval arg1 arg2 ...
```

Con lo que también podríamos hacer omitido las comillas, y el comando hubiera funcionado igual de bien. Aunque ahora cada palabra del comando hubiera sido considerada un argumento, y `eval` hubiera tenido que reconstruir los argumentos. En general se recomienda entrecomillar el argumento.

Ejercicio 7.1

Hacer un script llamado `fondo` que ejecute un comando (que recibe como argumento) en background y que redirija tanto su salida estándar como su salida de errores estándar a un fichero llamado `resultado.log`

Por ejemplo podemos hacer:

```
$ fondo du -d 1 /
```

La implementación del script usando `eval` es tan sencilla como esta:

```
eval "$@" > resultado.log 2>&1 &
```

Ejercicio 7.2

Implementar la utilidad `make` usando un script. La utilidad `make` lee un fichero con una serie de reglas de la forma:

```
target : source1 source2
     Comandos a ejecutar
```

De forma que si alguno de los ficheros source es más reciente que el target ejecuta los comandos de la regla. Un ejemplo de regla típico se muestra en el Listado 7.12:

```
programa.o : programa.c programa.h
     gcc -c programa.c
programa : programa.o
     gcc programa.o -o programa
```

Listado 7.12: Ejemplo de fichero de `make`

Donde si `programa.c` o `programa.h` son más recientes que `programa.o`, compila el fichero `programa.c` para generar el fichero objeto `programa.o`, y si el fichero objeto `programa.o` es más reciente que el ejecutable `programa`, vuelve a enlazar el fichero objeto.

Podemos implementar este programa con un script como el que se muestra en el Listado 7.13. El script ejecuta la función `ejecutareglas()` con la entrada estándar redirigida al fichero que pasemos como argumento del script, en nuestro caso el fichero tendrá la forma del Listado 7.12.

La función se ejecuta mientras que haya texto que procesar, es decir mientras que `read` devuelva el código de terminación 0. Para ello hemos hecho un bucle infinito que sólo se acaba al cumplirse este fin de fichero. El comando `true` es un comando que siempre devuelve el código de terminación 0.

Ahora al ejecutarlo obtenemos:

```
$ touch programa.c
$ mimake Makefile
```

```
Regla:  programa.o : programa.c programa.h
Cmd:    gcc -c programa.c
Regla:  programa : programa.o
Cmd:    gcc programa.o -o programa
```

```
function ejecutareglas
{
 while true
 do
    # Carga la regla
    if ! read target dospuntos sources ; then
       return 0
    fi
    # Ignora lineas en blanco, comentarios y comandos
    while [ "$target" = "" ] || \
          ! [ "${target#'#'}" = "$target" ] || \
          ! [ "$dospuntos" = ":" ]
    do
      if ! read target dospuntos sources; then
         return 0
      fi
    done
    echo -e "Regla:\t$target $dospuntos $sources"
    # Comprueba si algun source es mas reciente
    # que el target
    for src in $sources
    do
      if [ $src -nt $target ]; then
         read comando
         echo -e "Cmd:\t${comando#\t}"
         eval "${comando#\t}"
         break
      fi
    done
 done
}

ejecutareglas < $1
```

Listado 7.13: Script que implementa el comportamiento de `make`

Tema 8

Control de procesos

Sinopsis:

Una característica de UNIX es que el usuario ejerce un control sobre los procesos que en el sistema se están ejecutando. Aunque este control también sería posible tenerlo desde otros sistemas operativos más orientados al usuario doméstico, estos sistemas tradicionalmente han tratado de ocultar la gestión de procesos al usuario, en pro de la facilidad de uso.

Empezaremos este tema viendo las primitivas de gestión de procesos que ofrece el shell, para centrarnos luego en estudiar las técnicas de comunicación entre procesos que podemos controlar desde Bash.

1.IDs de procesos y números de jobs

Los sistemas UNIX asignan un **ID de proceso** a cada proceso que ejecutamos. Podemos ver este ID cuando ejecutamos un proceso en background usando `&`. Por ejemplo:

```
$ esto &
[1] 766
```

766 es el ID de proceso, y el 1 sería el **número de job**, el cual es asignado por el shell (no por el sistema operativo).

Si ejecutamos más procesos en background el shell les va asignando números de job consecutivos. Por ejemplo:

```
$ eso &
[2] 772
$ aquello &
[3] 774
```

El shell nos indica los números de job que van acabando:

```
[1]+  Done                    esto
```

En principio, estos mensajes se dan sólo después de haber ejecutado otro comando, pero podemos hacer que se dé el mensaje nada más acabe de ejecutarse el proceso fijando esta opción en el shell con el comando `set -b`.

En breve explicaremos que significa el símbolo `+` que precede al número de job. Si un proceso acaba con un código de terminación distinto de 0, el shell nos lo indicaría:

```
[1]+  Exit 1                  esto
```

El shell da otros tipos de mensajes cuando ocurre algo anormal a un proceso en background. Veremos estas circunstancias a lo largo de este tema.

2. Control de jobs

2.1. Foreground y background

Los procesos lanzados en background pierden la entrada estándar del terminal, con lo que no pueden leer del terminal, pero mantienen la salida estándar y salida de errores estándar asociadas al terminal, con lo que si escriben un mensaje, lo veremos. Por ello, como ya hemos comentado, muchas veces los procesos se lanzan redirigiendo la salida estándar a `/dev/null`, y dejando que sólo la salida de errores estándar aparezca en el terminal (para detectar si el proceso tiene problemas).

Una vez que lanzamos un job en background éste se está ejecutando hasta que acaba o necesita leer un valor de la entrada estándar. Podemos conocer el estado de los procesos que tengamos en background usando el comando `jobs`:

```
$ jobs
[1]   Stopped                 esto
[2]-  Running                 eso
[3]+  Running                 aquello
```

En este caso el comando `eso` está ejecutando correctamente (`Running`), pero el proceso `esto` está parado (`Stopped`), lo cual indica que posiblemente esté esperando una entrada por teclado, y como la entrada estándar esta liberada del terminal, no puede leer. En este caso debemos de pasar el proceso a foreground, usando el comando `fg`, e introducir el dato que está pidiendo:

```
$ fg %esto
Indique el fichero: prueba.txt
```

El comando `fg` sin argumentos pone en foreground el proceso más reciente (el 3 en este caso). Si queremos indicar un proceso a poner en foreground debemos indicar su número de job o su nombre precedidos por `%`[1]. Realmente no hace falta indicar todo su nombre, basta con indicar el principio de éste. Por ejemplo `fg %aq` pondría en foreground el job número 3.

El `+` y `-` que parece al ejecutar `jobs` indican respectivamente el proceso más reciente y el anterior al más reciente. Estos símbolos también pueden usarse para referirse a los procesos. Por ejemplo `fg +` pondría en foreground el job 3.

[1] Realmente en las versiones recientes de Bash no es necesario preceder por `%`, pero lo vamos a hacer por uniformidad con lo que vamos a ver más adelante

El comando `jobs` también tiene otras opciones interesantes que vamos a comentar aquí. La opción `-l` hace que `jobs` muestre también el ID de proceso:

```
$ jobs -l
[1]   766 Stopped                 esto
[2]-  772 Running                 eso
[3]+  774 Running                 aquello
```

La opción `-p` hace que jobs muestre sólo el ID de los procesos de background (esto nos resultará útil en el Ejercicio 8.1):

```
$ jobs -p
766
772
774
```

La opción `-r` muestra sólo los jobs que están ejecutándose (running), la opción `-s` muestra sólo los que están parados (stopped), y la opción `-n` los que han cambiando de estado desde la última vez que jobs nos los mostró. La opción `-x` nos permite ejecutar un proceso (en foreground). Si a esta última opción la damos un número de proceso, nos lo sustituye por su ID de proceso. Por ejemplo:

```
$ jobs -x echo %2
772
```

2.2. Suspender y reanudar un job

Una vez que tenemos un proceso en foreground, bien sea por haberlo puesto con `fg`, o por no haber usado `&` para que quede en background, éste tiene el control del teclado, y si el proceso es largo, conviene pasarlo a background de nuevo.

Para ello podemos usar la combinación de teclas Ctrl+Z que pasa el proceso que tenga el control del teclado a background y lo deja parado. Después podemos usar el comando `bg` para volver a pasar el proceso a ejecución.

Por ejemplo, si usamos el comando `du` para medir la ocupación de cada directorio del disco y nos damos cuenta de que la operación está tardando, podemos pararlo con Ctrl+Z, y reanudar su ejecución con `bg`. Además, como sólo hay un job en background, podríamos haber usado `bg` sin argumentos.

```
$ du -d 1 / > ocupaciondisco.txt
^Z
[1]+  Stopped                 du -d 1 / >ocupaciondisco.txt
```

```
$ jobs
[1]+  Stopped                 du -d 1 / >ocupaciondisco.txt
$ bg %1
[1]+ du -d 1 / >ocupaciondisco.txt &
$ jobs
[1]+  Running                 du -d 1 / >ocupaciondisco.txt &
```

2.3. El comando `ps`

El comando `ps` nos da información sobre los procesos que se están ejecutando en una máquina. Si ejecutamos `ps` sin argumentos nos da información sobre el proceso del shell y los procesos en background que se están ejecutando en este shell.

Por desgracia la forma de funcionar de este comando depende de si estamos en un UNIX de la familia BSD (p.e. Mac OS X), o en UNIX de la familia System V (p.e. Linux). En la primera familia nos da cinco columnas de información: El ID del proceso, el terminal donde se está ejecutando, el estado del proceso, el tiempo de CPU consumido y el comando. Por ejemplo en Mac OS X con un proceso en background obtenemos:

```
$ ps
PID  TT  STAT      TIME COMMAND
709 std  Ss     0:00.22 -bash
759 std  R      0:07.29 du -d 1 /
```

Obsérvese que el proceso `ps` no informa de su propia existencia, cosa que el `ps` de la familia System V sí hace.

En el System V se producen sólo cuatro de los cinco campos anteriores, el estado del proceso no se da. Por ejemplo en Linux obtendríamos esta salida:

```
$ ps
PID TTY          TIME CMD
152 tty2     00:00:01 bash
214 tty2     00:00:03 du
217 tty2     00:00:00 ps
```

En ambos podemos obtener información extendida sobre todos los procesos asociados a nuestro usuario (nos sólo los del terminal actual) poniendo la opción `-u`. Por ejemplo en Mac OS X obtenemos esta salida:

```
$ ps -u
USER     PID %CPU %MEM  VSZ RSS TT STAT STAR TIME  CMD
fernando 831 7.6  0.1 18060 304 p2 R+  10:58 00.14  du
fernando 709 0.0  0.2 18644 860 std Ss 10:41 00.23 -bash
fernando 799 0.0  0.2 18644 848 p2 Ss  10:58 00.11 -bash
```

%CPU indica la ocupación actual de CPU que está haciendo el proceso, %MEM su ocupación de memoria, VSZ la memoria virtual ocupada en kilobytes (no en porcentaje), RSS la memoria bloqueada por el proceso (en kilobytes), STAR la hora a la que se lanzó el proceso, y TIME el consumo total en tiempo de CPU que ha hecho el proceso. El comando nos dice que en el terminal actual (std) sólo se está ejecutando Bash (y el comando ps), mientras que en otro terminal (p2) se está ejecutando Bash y el comando du.

En Linux obtenemos esta otra salida:

```
$ ps -u
USER     PID %CPU %MEM VSZ  RSS  TTY STAT STAR TIME CMD
fernando 151  0.0 2.1 2620 1300 tty1 Ss+  0:38 0:01 -bash
fernando 152  0.1 1.2 2620  788 tty2 Ss   0:38 0:01 -bash
fernando 274 85.0 1.6 2072 1008 tty1 R    0:59 0:01 du
fernando 275  0.0 1.2 2276  788 tty2 R+   0:59 0:00 ps -u
```

Ahora la salida es idéntica, excepto que ps no se oculta a sí mismo.

Dentro del estado encontramos un conjunto de letras, cada una con un significado de acuerdo a la Tabla 8.1. Auque hay pequeñas variaciones entre los dos sistemas, los estados descritos en la Tabla 8.1 son los coincidentes, que son la mayoría.

Estado	Descripción
D	(Disk) Proceso realizando una operación de E/S a disco.
R	(Running) Ejecutando.
S	(Sleeping) Proceso dormido.
T	(Traced o sToped) Proceso parado o detenido por el depurador.
Z	(Zombie) Proceso zombie.
+	El proceso es el foreground process group leader del terminal.
<	Al proceso se le ha bajado la prioridad.
>	Al proceso se le ha subido la prioridad.
s	El proceso es un session leader

Tabla 8.1: Estados del comandos ps

En el ps -u ejecutado antes en Mac OS X se nos indica que los **session leader** (programa con los que nos logamos) son Bash, pero el proceso que tiene el control de la entrada estándar del terminal es du. Intente interpretar los estados obtenidos por ps -u en la máquina Linux.

La opción -a muestra los procesos de todos los usuarios (estén en el terminal que estén). Aun usando esta opción no obtenemos información sobre todos los procesos, ya que esta opción no muestra información de los procesos que no están asociados a un terminal, los llamados demonios. Si queremos obtener información sobre estos procesos sin terminal debemos de usar la

opción `-x`. En general, en ambas familias un comando que muestra descripción detallada de todos los procesos es `ps -aux`. Le recomendamos acordarse de esta forma de ejecutar `ps`, y normalmente no necesitara usar ninguna más.

2.4. El comando `top`

El comando `top` muestra información actualizada en tiempo real de los procesos que están consumiendo más CPU. Además proporciona otros tipos de información, como por ejemplo, la memoria ocupada o la ocupación de CPU. Use la tecla `q` para salir de `top`.

La Tabla 8.2 muestra algunas opciones interesantes de `top`.

Opción	Descripción
`a`	Muestra información acumulativa
`U`	Muestra sólo los procesos del usuario indicado
`p`	Muestra información del proceso indicado

Tabla 8.2: Principales opciones de `top`

La opción `-a` muestra información acumulativa de los procesos en vez de información puntual, esta información normalmente es más significativa para encontrar los procesos que más CPU están consumiendo. Por desgracia esta opción sólo está disponible en la familia BSD.

La opción `-U` nos permite obtener información de un determinado usuario. Por ejemplo `top -U fernando` da información de los procesos de este usuario.

La opción `-p` nos da información sobre los procesos indicados. Por ejemplo `top -p 151 -p 152` da información sobre los procesos indicados. Esta opción sólo está disponible en la familia System V.

3.Señales

Una **señal** es un mensaje que un proceso envía a otro. Normalmente es el proceso padre el que envía mensajes a los procesos hijos que crea. Ya hemos visto que un proceso se puede comunicar con otro usando un pipeline, las señales son otra técnica de comunicación entre procesos. De hecho ambos forman parte de lo que en los libros de sistemas operativos se llama técnicas de IPC (Inter Process Communication).

Las señales tienen números (de 1 al número de señales que soporta el sistema) y nombres. Podemos obtener una lista de las señales que soporta nuestro sistema con el comando `kill -l`. También puede obtener información sobre éstas usando `man 7 signal`. La Tabla 8.3 muestra un resumen de las principales señales que existen. Cuando escribimos scripts tenemos que tener en cuenta que los nombres de señales son más portables que sus números, con lo que nosotros nos vamos a referir a ellas por nombre.

Señal	Descripción	Acción
SIGHUP	El proceso padre ha terminado	Term
SIGINT	Interrumpido desde el teclado (con Ctrl-C)	Term
SIGQUIT	Cerrado desde el teclado (con Ctrl+\)	Term
SIGILL	Se ha intentado ejecutar una instrucción ilegal	Core
SIGABRT	Se ha abortado el proceso con la función `abort()`	Core
SIGFPE	Excepción en punto flotante	Core
SIGKILL	Mata el proceso	Term
SIGSEGV	Acceso a una dirección de memoria inválida	Core
SIGPIPE	Se ha intentado escribir en un pipe roto	Term
SIGALRM	Temporizador de una alarma puesta con la función `alarm()`	Term
SIGTERM	Señal de terminación	Term
SIGUSR1	Señal de usuario	Term
SIGUSR2	Señal de usuario	Term
SIGCHLD	Un hijo ha parado o terminado	Ign
SIGSTOP	Parar el proceso	Stop
SIGCONT	Continua si está parado	-
SIGTSTP	Proceso parado desde el terminal (con Ctrl+Z)	Stop
SIGTTIN	El proceso ejecutando en background (sin control del terminal) ha intentado leer de la entrada estándar	Stop
SIGTTOU	El proceso ejecutando en background (sin control del terminal) ha intentado escribir en la salida estándar	Stop
SIGTRAP	Trap de breakpoint	Core
SIGURG	Datos urgentes en el socket	Ign
SIGXCPU	Excedido el tiempo de CPU	Core
SIGXFSZ	Excedido el espacio en disco	Core

Tabla 8.3: Resumen de las principales señales

El campo Acción de la Tabla 8.3 indica cual es el comportamiento por defecto de un proceso ante una señal, y puede tomar uno de estos cuatro valores:

- Term. Se termina el proceso.
- Ign. Se ignora la señal.
- Core. Se termina el proceso y se hace un dump del core.
- Stop. Se para el proceso.

3.1. Combinaciones de teclas que envían señales

Existen varias combinaciones de teclas que actúan sobre el proceso que se esté ejecutando en foreground en el terminal.

La combinación de teclas Ctrl+C envía la señal `SIGINT` al proceso, con lo que éste debería de terminar. Ctrl+Z envía la señal `SIGTSTP` al proceso, con lo que éste se detiene. Ctrl+\ manda la señal `SIGQUIT` al proceso, y se debe usar sólo cuando el proceso no responde a `SIGINT`. Esto se debe a que a veces los procesos capturan la `SIGINT`, y en ella hacen desinicializaciones y cierres de fichero. Si esta operación de cierre tarda en realizarse, o se queda bloqueada por algún error, el usuario puede enviar la señal `SIGQUIT` que normalmente no está capturada por los procesos, y el proceso termina. Esta terminación brusca puede dejar ficheros sin cerrar, y no se recomienda usarla más que si el cierre con `SIGINT` no ha funcionado.

Podemos usar el comando `stty` para crear nuevas combinaciones de teclas que envíen señales al proceso el foreground usando `stty` *`señal ^letra`*. donde *`señal`* es el nombre de la señal en minúsculas y sin el prefijo `SIG`. Por ejemplo para que Ctrl+Q produzca la señal `SIGQUIT` podemos usar:

```
$ stty quit ^Q
```

3.2. El comando interno `kill`

Podemos usar el comando interno `kill` para enviar un comando a cualquier proceso que hayamos creado, no sólo al que esté ejecutándose en foreground. Además, si tenemos permiso de administración, podemos enviar mensajes con `kill` a procesos de otros usuarios.

`kill` recibe como argumento el ID de proceso, el número de job (precedido por `%`) o el nombre del comando (también precedido por `%`). En estos dos últimos casos necesitamos que el proceso sea un job de nuestro terminal.

Por defecto `kill` envía la señal `SIGTERM` al proceso, la cual causa que esté termine limpiamente, al igual que pasa con la señal `SIGINT` producida por Ctrl+C. Podemos indicar que `kill` envíe una señal distinta al proceso precediendo el nombre o el número de la señal por un guión.

Por ejemplo si el proceso `esto` tiene el número de job 1, podemos enviarle la señal `SIGTERM` usando `kill %1`[1]. Si tiene éxito veremos un mensaje de la forma:

```
[1]   766 Terminated              esto
```

Sino podemos enviarle la señal `SIGQUIT` usando `kill -QUIT %1`[2]. Si tenemos éxito recibimos el mensaje:

```
[1]   766 Exited  131             esto
```

Don de 131 es el código de terminación del proceso.

Si aun así no responde, el último recurso es enviar la señal `SIGKILL`, con `kill -KILL %1`[3], que es una señal que los procesos no pueden capturar, con lo que no le queda más remedio que ser matado por el sistema operativo.

Ejercicio 8.1

Hacer un script llamado `killalljobs` que envíe la señal pasada como argumento a todos los procesos en background.

Usando el comando `jobs -p`, que vimos antes, podemos hacerlo en una sola línea así:

```
kill "$@" $(jobs -p)
```

Existe un comando llamado `killall` *`patron`* que nos permite enviar la señal `SIGTERM` (o otra si se la indicamos como opción) a todos los procesos que tengan a *`patron`* en su nombre. Realmente este comando es equivalente a `kill`, excepto porque no hay que preceder el nombre de proceso por `%`.

[1] Tenga cuidado de no ejecutar el comando `kill 1` ya que si ejecuta este comando como root terminará el proceso `init`, un proceso muy importante que posiblemente hará que todo su sistema se vuelva inestable.
[2] Siempre se puede omitir el `SIG` del nombre de la señal, es decir, usar `kill -QUIT %1` en vez de `kill -SIGQUIT %1`
[3] Esta opción muchas veces se documentoa como `kill -9`, nosotros preferimos usar el nombre de la señal y no su número.

4. Capturar señales desde un script

4.1. El comando interno `trap`

Hemos comentado que los programas C pueden capturar señales y actuar en consecuencia. Los scripts Bash no son menos y también tienen su mecanismo de captura de señales usando el comando interno `trap`, el cual tiene el siguiente formato:

`trap` *`cmd sig1 sig2 ...`*

`cmd` es el comando que queremos ejecutar al capturar alguna de las señales *`sig1 sig2 ...`* Lógicamente *`cmd`* puede ser una función o un script, y las señales se pueden dar por número o por nombre. El comando `trap` también se puede ejecutar sin argumentos, en cuyo caso nos da la lista de traps que están fijados.

El Listado 8.1 muestra un script sencillo que captura la señal `SIGINT` (producida por Ctrl+C), e imprime un mensaje indicando que ha capturado la señal.

```
trap "echo 'Pulsaste Ctrl+C!'" INT

while true
do
  sleep 60;
  echo "Cambio de minuto"
done
```

Listado 8.1: Script que captura la `SIGINT`

Ahora al ejecutarlo y pulsar Ctrl+C obtenemos:

```
$ capturasenal
^CPulsaste Ctrl+C!
Cambio de minuto
^CPulsaste Ctrl+C!
Cambio de minuto
```

Obsérvese que al recibir la señal `SIGINT` Bash se la pasa al comando `sleep`, con lo que éste acaba, pero luego se ejecuta el trap del script, y por esta razón el script no acaba.

Para pararlo ahora con Ctrl+C tenemos un problema ya que hemos cambiado la opción por defecto, que es Term, por imprimir un mensaje. Para terminar

el script puede pararlo con Ctrl+Z y luego hacerle un `kill` (que envía la `SIGTERM` no la `SIGINT`):

```
^Z
[1]+  Stopped                 capturasenal
$ kill %1
[1]+  Terminated              capturasenal
```

Podemos ahora añadir la captura de la señal `SIGTERM` como muestra el Listado 8.2.

```
trap "echo 'Pulsaste Ctrl+C!'" INT
trap "echo 'Intentaste terminarme!'" TERM

while true
do
  sleep 60;
  echo "Cambio de minuto"
done
```

Listado 8.2: Script que captura la `SIGINT` y la `SIGTERM`

Si ahora ejecutamos el script en background:

```
$ capturasenal &
[1] 1504
```

E intentamos terminarlo:

```
$ jobs
[1]+  Running                 capturasenal &
$ kill %1
Intentaste terminarme!
Cambio de minuto
```

El script se defiende. Siempre podemos terminarlo enviándole la `SIGKILL` que sabemos que no la puede capturar:

```
$ kill -KILL %1
[1]+  Killed                  capturasenal
```

4.2. Traps y funciones

Como sabemos, las funciones se ejecutan en el mismo proceso que el script que las llama, en consecuencia dentro de una función se puede detectar un trap fijado por el script, y viceversa, un trap fijado por una función sigue activo cuando ésta termina. Por ejemplo, en el Listado 8.3 se muestra una

función que fija un trap. En trap seguirá activo cuando nos metamos en el bucle.

```
function fijatrap
{
 trap "echo 'Pulsaste Ctrl+C!'" INT
}

fijatrap
while true
do
  sleep 60;
  echo "Cambio de minuto"
done
```

Listado 8.3: Función que fija un trap

4.3. IDs de proceso

Vamos a ver aquí otras dos variables especiales: `$` y `!` (a cuyo valor accedemos con `$$` y `$!`). La primera almacena el ID de nuestro proceso, la segunda almacena el ID del último proceso en background que ejecutamos.

Por ejemplo, si ejecutamos un proceso en background, y a continuación preguntamos por el valor de `$!`:

```
$ ls > /dev/null &
[1] 795
$ echo $!
795
```

Si desde el terminal preguntamos por `$$`:

```
$ echo $$
744
```

Nos devuelve el ID de proceso del shell Bash en el que estamos.

El directorio `/tmp` (muchos sistemas tienen también el directorio `/var/tmp`) es un directorio destinado a almacenar ficheros temporales que se borrar al apagar la máquina. Esto evita que la máquina se llene de ficheros temporales que ocupan disco de forma innecesaria. Muchas veces se usa el ID de proceso del script para asignar nombre a los ficheros temporales. Por ejemplo, en el Ejemplo 6.1 hicimos un script llamado `tojpg` que convertía un fichero en cualquier formato a formato `.jpg`. Además el script nos permitía escalar y poner borde al fichero, pero para hacer esto último teníamos que

hacer una copia temporal del fichero, y luego pasárselo a los comandos `pnmscale` y `pnmmargin`:

```
# Aplica las opciones
if [ $escala ]; then
   cp $fichero_ppm aux.$fichero_ppm
   pnmscale $escala aux.$fichero_ppm > $fichero_ppm
   rm aux.$fichero_ppm
fi
if [ $grosorborde ]; then
   cp $fichero_ppm aux.$fichero_ppm
   pnmmargin $grosorborde aux.$fichero_ppm > $fichero_ppm
   rm aux.$fichero_ppm
fi
```

Esta forma de hacer el script implicaría que si en mitad de la ejecución fallase el script (o fuera interrumpido con Ctrl+C), los ficheros temporales quedarían sin borrar.

Vamos a modificar esto para que los ficheros se creen en el directorio temporal, y para que les asignemos un nombre único, que incluya el ID de proceso de nuestro script.

El script quedaría ahora de la siguiente forma:

```
# Aplica las opciones
if [ $escala ]; then
   cp $fichero_ppm /tmp/${$}$fichero_ppm
   pnmscale $escala /tmp/${$}$fichero_ppm > $fichero_ppm
   rm /tmp/${$}$fichero_ppm
fi
if [ $grosorborde ]; then
   cp $fichero_ppm /tmp/${$}$fichero_ppm
   pnmmargin $grosorborde /tmp/${$}$fichero_ppm > \
             $fichero_ppm
   rm /tmp/${$}$fichero_ppm
fi
```

4.4. Ignorar señales

Si lo que queremos es ignorar una señal, simplemente tenemos que pasar una cadena vacía (`""` ó `''`) en el argumento *cmd* de `trap`.

El ejemplo clásico de señal que muchas veces se quiere ignorar es la señal `SIGHUP` (hangup), la cual recibe un proceso cuando su padre termina (p.e. el shell) y produce que el proceso hijo también termine. Por ejemplo, podemos hacer la siguiente función que lanza un comando de forma que éste no termina al terminar el shell:

```
function ignorarhup
{
 trap "" HUP
 eval "$@"
 trap - HUP
}
```

La opción – pasada a trap restaura la señal `SIGHUP` para que los siguientes comandos que ejecutemos capturen esta señal, y sí que terminen al recibirla.

Ahora podemos lanzar un comando así:

```
$ ignorarhup du -d 1/ > ocupacion.txt
```

Actualmente existe un comando UNIX que hace esto mismo, que es el script `nohup` cuya implementación se muestra en el Listado 8.4.

```
trap "" HUP
eval "$@" > nohup.out 2>&1
trap - HUP
```

Listado 8.4: Implementación de `nohup`

Es decir, básicamente el comando evalúa con `eval` los argumentos recibidos y redirige tanto la salida estándar como la salida de errores estándar al fichero `nohup.out`.

Por último vamos a comentar que existe un comando interno, llamado `disown`, que recibe como argumento un job y elimina el proceso de la lista de jobs controlados por el shell (con lo que no recibiría la señal `SIGHUP` cuando el shell que lo lanzó termine). La opción `-h` (hook) de este comando realiza la misma función de `nohup`, manteniendo al proceso en la lista de jobs, pero no enviándole la señal `SIGHUP` cuando el shell termina. También existe la opción `-a` (all) que libera a todos los procesos en background de la lista de jobs del shell.

5.Corutinas

Llamamos **corutinas** a un conjunto de dos o más procesos ejecutados concurrentemente por el shell, y opcionalmente con la posibilidad de comunicarse ente ellos.

Un pipe es un ejemplo de corutinas. Cuando invocamos un pipe, p.e. `ls|more`, el shell llama a un conjunto de primitivas, o llamadas al sistema. En concreto, el shell dice al SO que realice las siguientes operaciones (si es usted programador C, entre paréntesis le indicamos la primitiva del SO usada):

1. Crear dos procesos que llamaremos P1 y P2 (usa la primitiva `folk()`, la cual crea otro proceso, y devuelve el ID del nuevo proceso al hilo del padre, y el ID 0 al proceso hijo).
2. Conecta la salida estándar de P1 a la entrada estándar de P2 (usando la función `pipe()`).
3. Ejecuta `/bin/ls` en P1 (usando `exec()` que reemplaza la imagen del proceso actual por una nueva imagen).
4. Ejecuta `/bin/more` en el proceso P2 (usando `exec()`).
5. Espera a que ambos procesos acaben (usando la primitiva `wait()`).

Si no se necesita que dos procesos se comuniquen entre ellos, la forma de ejecutarlos es más sencilla. Por ejemplo, si queremos lanzar los procesos `comer` y `beber` como corutinas, podemos hacer el siguiente script:

```
comer &
beber
```

Si `beber` es el último proceso en acabar, esta solución funciona, pero si `comer` sigue ejecutando después de que acabe de ejecutarse el script, `comer` se convertiría en un proceso huérfano (también llamado zombie).

En general esto es algo indeseable, y para solucionarlo existe el comando interno `wait`, el cual para al proceso del script hasta que todos los procesos de background han acabado. Luego la forma correcta de lanzar las corutinas anteriores sería:

```
comer &
beber
wait
```

El comando interno `wait` también puede recibir como argumento el ID o el número de job del proceso al que queremos esperar.

Ejercicio 8.2

Implementar un script llamado `mcp` que copie el fichero que pasamos como primer argumento al resto de argumentos. Es decir, el comando tendrá el formato:

`mcp` *`fuente destino1 destino2 ...`*

La solución consiste en lanzar varios procesos como corutinas, y esperar a que todos acaben con `wait`, tal como muestra el Listado 8.5.

```
fuente=$1
shift
for destino in "$@"
do
   cp $fuente $destino &
done
wait
```

Listado 8.5: Implementación de `mcp`

6.Subshells

Vamos a ver otra técnica de comunicación entre procesos, que es la comunicación entre un subshell y el shell padre. En el Tema 3 vimos que cuando ejecutábamos un script, estábamos creando un proceso distinto en el que se ejecutaba el script. Ahora vamos a ver que dentro de un script, un conjunto de comandos pueden ejecutarse también como un proceso aparte.

En el apartado 3 del Tema 7 vimos los bloques de comandos, donde podíamos encerrar un conjunto de comandos entre llaves, y redirigir su entrada o salida estándar. Los **subshells** son parecidos a los bloques de comandos, donde también podemos redirigir su entrada y salida estándar, sólo que ahora se encierran los comandos entre paréntesis y el subshell, a diferencia del bloque de comandos, se ejecuta en un proceso aparte. Por ejemplo, el Listado 8.6 muestra un subshell que genera los números del 0 al 9, y después se los pasa por un pipe a `sort` para que los ordene de mayor a menor.

```
(
 for ((i=0;i<=9;i++))
 do
   echo $i
 done
) | sort -r
```

Listado 8.6: Ejemplo de subshell

La principal diferencia entre un subshell y un bloque de comandos es que el primero se ejecuta en un proceso aparte, con lo que es menos eficiente, pero a cambio no modifica variables del shell actual, con lo que existe mayor encapsulación[1]. Por ejemplo, si vamos a fijar un trap, lo podemos fijar en un subshell para no afectar al resto del script.

Conviene aclarar que cuando ejecutamos un subshell se hereda del proceso padre: El directorio actual, las variables de entorno exportadas y la entrada y salida estándar, así como la de errores. Y no se hereda: Las variables no exportadas y los traps de señales.

[1] Recuerdese que siempre que ejecuamos un comando externo se ejecuta en un proceso aparte, con lo que el uso de subshell no enlentece mucho más.

7. La sustitución de procesos

La **sustitución de procesos** es un complemento a la sustitución de comandos que vimos en el apartado 4 del Tema 4.

Reacuérdese que la sustitución de comandos nos permitía asignar la salida estándar de un comando a una variable. La sustitución de procesos lo que nos permite el asignar la salida estándar de un proceso a un **named pipe**, los cuales se suelen almacenar como ficheros en el directorio `/dev/fd/`, y este fichero se pasa a otro comando para que procese su contenido.

Existen dos operadores de sustitución de procesos: `<(`*comando*`)` que asigna la salida estándar del comando a un fichero (named pipe) de sólo lectura, y `>(`*comando*`)` que asigna la salida del comando a un named pipe de sólo escritura.

Por ejemplo `grep "prueba.txt" <(ls -la)` es equivalente a `ls -la|grep "prueba.txt"`.

Realmente la sustitución de comandos lo que nos devuelve es el nombre del fichero donde se ha depositado la salida estándar del comando. Por ejemplo:

```
$ echo <(ls -la)
/dev/fd/63
```

podemos usar la sustitución de procesos para encontrar las diferencias entre los ficheros de los directorios `dir1` y `dir2` así:

```
$ diff <(ls -la dir1) <(ls -la dir2)
```

O bien, si tenemos dos comandos `comando1` y `comando2` que producen una determinada salida, podemos comparar sus salidas usando:

```
$ diff <(comando1) <(comando2)
```

Tema 9

Depurar scripts

Sinopsis:

En este tema veremos técnicas útiles para depurar sus scripts. La técnica más básica de depuración, que seguramente ya conozca, es llenar el scripts de `echo`*s que muestran como evoluciona el programa. El objetivo de este tema es que pueda usar más y mejores técnicas a la hora de depurar sus scripts.*

En la primera parte del tema veremos como se usan estás técnicas. En la segunda parte veremos paso a paso como construir un depurador de scripts con Bash.

1. Opciones de Bash para depuración

El shell tiene una serie de opciones para la depuración las cuales, o bien se pasan como argumentos al lanzar `bash`, o bien se activan con el comando `set -o` *`opcion`*. Si usamos `set -o` *`opcion`* estamos activando la opción, si usamos `set +o` *`opcion`* la estamos desactivando. Es decir, al revés de lo que parece. Las opciones que vamos a empezar comentando se describen en la Tabla 9.1.

Opción de `set`	Opción de `bash`	Descripción
`noexec`	`-n`	No ejecuta los comandos, sólo comprueba su sintaxis.
`verbose`	`-v`	Imprime los comandos antes de ejecutarlos
`xtrace`	`-x`	Imprime los comandos a interpretar y las distintas expansiones que se realizan antes de ejecutarlo

Tabla 9.1: Opciones de Bash para depuración

La opción `xtrace` nos muestra tanto el comando a ejecutar, como la expansión de las sustituciones de parámetros, de las sustituciones de comandos, y todas las demás sustituciones que se realizen. Por ejemplo, si ejecutamos el comando `listafecha` del Ejercicio 4.4 con esta opción obtenemos:

```
$ bash -x lsfecha '9 Aug'
++ ListaFecha '9 Aug'
++ ls -lad Makefile aquello lsfecha
++ grep '9 Aug'
++ cut -c54-
+ ls -lad aquello
-rwxr-xr-x  1 fernando  admin   80 19 Aug 22:16 aquello
```

El mencionado script se repite en el Listado 9.1 por claridad.

```
function ListaFecha
{
 ls -lad * | grep "$1" | cut -c54-
}

ls -lad $(ListaFecha "$1")
```

Listado 9.1: Implementación de `lsfecha`

Cada símbolo + al principio de una línea indica un nivel de expansión.

En el ejemplo, la sustitución de comandos `$(ListaFecha "$1")` produce una expansión, y llama a la función `ListaFecha`, la cual ejecuta cada uno de los comandos del pipe expandidos, es decir, observe que en vez de `ListaFecha "$1"` aparece `ListaFecha '9 Aug'`. Al acabar la sustitución de comandos se reduce en uno el nivel de expansiones y se ejecuta el comando `ls -lad aquello` donde `Makefile aquello lsfecha` es el resultado de la expansión de la sustitución de comandos.

El símbolo `+` es personalizable usando el cuarto prompt `PS4`. Por ejemplo si hacemos:

```
$ export PS4='xtrace->'
$ bash -x lsfecha '9 Aug'
xxtrace->ListaFecha '9 Aug'
xxtrace->ls -lad Makefile aquello lsfecha
xxtrace->grep '9 Aug'
xxtrace->cut -c54-
xtrace->ls -lad aquello
-rwxr-xr-x  1 fernando  admin   80 19 Aug 22:16 aquello
```

Obsérvese que para múltiples niveles de expansión el prompt imprime sólo el primer carácter de `PS4` con el fin de hacer más legible la traza.

Podemos personalizar aun más el prompt poniendo variables en éste. Por ejemplo, la variable especial `$LINENO` nos permite saber la línea del script que estamos ejecutando. Tal como hicimos con `PS1` en el Tema 3, podemos posponer la evaluación de esta variable hasta que se vaya a mostrar el prompt encerrándola entre comillas fuertes:

```
$ export PS4='$0:$LINENO:'
$ bash -x lsfecha '9 Aug'
llsfecha:12:ListaFecha '9 Aug'
llsfecha:9:ls -lad Makefile aquello lsfecha
llsfecha:9:grep '9 Aug'
llsfecha:9:cut -c54-
lsfecha:12:ls -lad aquello
-rwxr-xr-x  1 fernando  admin   80 19 Aug 22:16 aquello
```

Lo importante de usar la opción `xtrace` es que nos permite encontrar errores en nuestro programa al poder ver como se expanden los valores. Por ejemplo, si tenemos un script de la forma:

```
fntcb=mifich.txt
mejorcli=$(cut -f3 $futcb)
```

el cual se queda colgado, y lo ejecutamos con la opción `xtrace` vemos que nos da la salida:

```
+ fntcb=mifich.txt
++ cut -f3
```

y esto nos ayuda a identificar que el problema está en que la variable `fntcb` no se ha expandido correctamente y `cut` está leyendo de la entrada estándar. ¿Ve por qué?.

La opción `noexec` sirve para que Bash no ejecute los comandos, sólo lea los comandos y compruebe su sintaxis. Sin embargo una vez que activa esta opción con `set -o noexec` ya no podrá volver a desactivarla, ya que el comando `set +o noexec` será parseado pero no ejecutado por Bash. En consecuencia use está opción sólo para pasársela a otro subshell de la forma:

```
$ bash -n lsfecha '9 Aug'
```

2. Fake signals

Las **fake signals** (falsas señales) son un mecanismo muy potente de ayuda a la depuración. Se trata de señales producidas por Bash, y no por un programa o suceso externo al shell. Estas señales se resumen en la Tabla 9.2 y las vamos a comentar a continuación.

Señal	Descripción
`SIGEXIT`	El script acabó de ejecutarse
`SIGERR`	Un comando a retornado un código de terminación distinto de 0
`SIGDEBUG`	El shell va a ejecutar una sentencia
`SIGRETURN`	Una función o script ejecutado con `source` ha acabado.

Tabla 9.2: Fake signals de Bash

En los siguientes subapartados vamos a comentar estas señales con más detalle.

2.1. La señal `SIGEXIT`

Esta señal se activa justo antes de terminar de ejecutarse un proceso. La señal se debe de solicitar por el proceso (no por el proceso padre que lanza el proceso), es decir, si desde el shell ejecutamos el script `miscript` así:

```
$ trap "echo 'Acabo el script'" EXIT
$ miscript
```

La señal no se produce, sino que el `trap` debe de estar dentro del script, por ejemplo, si hacemos el siguiente script:

```
trap "echo 'Acabo el script'" EXIT
echo "Empieza el script"
```

Al ejecutarlo obtenemos[1]:

```
$ miscript
Empieza el script
Acabo el script
```

La señal se lanza independientemente de como acabe el script: Por ejecutar la última línea, o por encontrar un `exit`.

[1] La señal sólo se produce cuando ejecutamos el script como un comando, no cuando lo ejecutamos con `source`.

2.2. La señal SIGERR

La señal SIGERR se lanza siempre que un comando de un script acaba con un código de terminación distinto de 0. La función que la captura puede hacer uso de la variable ? para obtener su valor. Por ejemplo:

```
function CapturadoERR
{
 ct=$?
 echo "El comando devolvio el codigo de terminación $ct"
}
trap CapturadoERR ERR
```

Sería una buena mejora el incluir el número de línea donde se ha producido el código de terminación, pero si hacemos:

```
function CapturadoERR
{
 ct=$?
 echo "Codigo de terminación $ct en linea $LINENO"
}
trap CapturadoERR ERR
```

Lo que acabamos obteniendo es el número de línea de la sentencia de la función CapturadoERR. Podemos solucionar este problema así:

```
function CapturadoERR
{
 ct=$?
 echo "Codigo de terminación $ct en linea $LINENO"
}
trap 'CapturadoERR $LINENO' ERR
```

Donde debemos de encerrar entre comillas fuertes el comando a ejecutar por trap, ya que sino la variable se sustituiría por el número de línea del comando trap. Sin embargo, al encerrarlo entre comillas fuertes, la variable no se sustituye por su valor hasta el momento de ejecutar la función, y se sustituye por el valor de la línea donde se detecta el código de terminación erróneo.

También existe una forma alternativa de pedir al shell que nos informe si un comando acaba con un código de terminación distinto de 0, que es fijando la opción del shell set -o errtrace (o set -E)[1].

[1] La opción errtrace está disponible sólo a partir de Bash 3.0

2.3. La señal SIGDEBUG

La señal SIGDEBUG, cuando se activa con trap, se lanza justo antes de ejecutar un comando. El principal uso de está función lo veremos en el apartado 3 donde construiremos un depurador.

Un problema que tiene esta señal es que no se hereda en las funciones que ejecutemos desde el script, con lo que si queremos heredarla tenemos tres opciones: Activarla con trap dentro de cada función, declarar la función con declare -t que hace que la función si herede el trap, o fijar la opción del shell set -o functrace (o set -F), que hace que todas las funciones hereden el trap de SIGDEBUG.[1]

2.4. La señal SIGRETURN

La señal SIGRETURN se lanza cada vez que retornamos de una función, o retornamos de ejecutar un script con source. La señal no se lanza cuando acabamos de ejecutar un comando script (a no ser que lo ejecutemos con source). Si queremos hacer esto último debemos usar SIGEXIT.

Al igual que SIGDEBUG, la señal SIGRETURN no es heredada por las funciones. De nuevo podemos hacer que una función herede esta señal, declarando a la función con declare -t, o bien activando la opción set -o functrace.

[1] La herencia de traps con declare -t, set -o functrace o set -F sólo están disponibles a partir de Bash 3.0

3. Un depurador Bash

En este apartado construiremos un pequeño depurador de scripts Bash. Para ello usaremos los conceptos que hemos aprendido en los apartados anteriores.

La mayoría de los depuradores tienen numerosas características que ayudan al programador a seguir paso a paso la ejecución de un programa. Entre ellas está el poder ejecutar y parar el programa en determinados puntos, llamados breakpoints, así como el poder examinar y cambiar el valor de las variables.

Nuestro pequeño depurador va a disponer de las siguientes características:

- Posibilidad de especificar breakpoints, tanto por número de línea como por condición booleana a cumplirse.
- Posibilidad de ejecutar el programa paso a paso, que es lo que se llama el stepping.
- Posibilidad de leer y cambiar el valor de las variables del script depurado
- Posibilidad de imprimir el programa con indicadores de donde se encuentran los breakpoints, y donde se encuentra detenido el programa.

3.1. Estructura del depurador

El depurador que vamos a hacer, que llamaremos `bashdb`, es un depurador que recibe como argumento un script a depurar, que llamaremos **script original**, y lo almacena en otro script, al que llamaremos **script modificado**, al cual le hemos añadido cierta funcionalidad que nos va a ayudar a ejecutar el script original paso a paso. Este proceso será transparente al usuario, de forma que él sólo será consciente de la existencia del script original.

El depurador constará de tres módulos: El driver (fichero `bashdb`), el preámbulo (fichero `bashdb.pre`) y las funciones del depurador (fichero `bashdb.fn`). Vamos a comentar cada uno de estos módulos con más detalle en los próximos subapartados.

3.2. El driver

El driver es el encargado de configurar el entorno y coordinar el comportamiento de los demás módulos para depurar el script.

```
# Driver del depurador

# Comprueba argumentos
if (( $#<1 )); then
  echo "Use: bashdb <script>" >&2
  exit 1
fi
_original=$1
if [ ! -r $_original ]; then
   echo "No se puede leer el fichero $original" &>2
   exit 1
fi
# Convierte a $1 en $0 y pone en su sitio los
# argumentos del script original
shift

# Crea el fichero modificado
_tmpdir=/tmp
_libdir=.
_modificado=$_tmpdir/bashdb.$$
cat $_libdir/bashdb.pre $_original > $_modificado

# Y lo ejecuta
exec bash $_modificado $_libdir $_original "$@"
```

Listado 9.2: Implementación del driver del depurador `bashdb`

El Listado 9.2 muestra su implementación. `bashdb` recibe como primer argumento el nombre del script a ejecutar, y el resto de argumentos son los argumentos del script.

Si `bashdb` pasa los test iniciales, construye un fichero temporal en el que guarda el script modificado, el cual consta del preámbulo y el script original. La variable `_libdir` indica el directorio donde están situados los ficheros del depurador. En principio está fijada al directorio actual, pero, una vez acabado el programa que estamos haciendo, podemos cambiar estos ficheros a otro sitio (p.e. `/usr/local/lib`).

En el apartado 1.2 del Tema 7 vimos como se usaba el comando `exec` para modificar la entrada/salida de todos los comandos posteriores. El comando `exec` también se puede usar para reemplazar el script actual que está ejecutando Bash por otro que le pasamos como argumento (`$_modificado` en nuestro caso). Esto nos evita crear un subproceso aparte y ejecutar el script modificado en nuestro propio proceso. El script recibe dos argumentos: El directorio de librerías (`$_libdir`), y el nombre del fichero original (`$_original`). Obsérvese que todas las variables del depurador las hemos precedido por guión bajo para reducir conflictos con variables del script original.

3.3. El preámbulo

El preámbulo se ejecuta antes que el script original, y configura a este último. Su implementación se muestra en el Listado 9.3.

```
# Implementacion del preambulo de bashdb

# Recoge los argumentos
_modificado=$0
_libdir=$1
_original=$2
shift 2

# Declara variables necesarias
declare -a _lineas
declare -a _lineasbp
let _trace=0

# Activa el que SIGDEBUG se produzca dentro de
# las funciones del script original
set -o functrace

# Carga las funciones
source $_libdir/bashdb.fn

# Carga en _lineas las lineas del script original
let _i=0
while read
do
  _lineas[$_i]=$REPLY
  let _i=$_i+1
done < $_original

# Indica que nada mas empezar ejecute la
# primera sentencia del script original
let _steps=1

# Fija traps
trap '_finscript' EXIT
trap '_steptrap $(($LINENO-35))' DEBUG
```

Listado 9.3: Implementación del preámbulo en el fichero `bashdb.pre`

El preámbulo empieza recogiendo los argumentos que recibe del driver y aplica un `shift` de forma que `$0` acaba siendo el nombre del script original y el resto de los argumentos del script se colocan a partir de `$1`. Después declara los arrays `_lineas` y `_lineasbp` donde se guarda respectivamente

las líneas del script original y los breakpoints que vayamos fijando. También desactiva la traza (poniendo `_trace` a 0). Hecho esto el bucle `while` carga el script original en el array `_lineas`. Estas líneas será necesario tenerlas cargadas en memoria por dos razones: Para poderlas imprimir junto con los breakpoints, y poderlas mostrar cuando el modo de trace esté activado. Obsérvese que `read` no recibe como argumento una posición del array, sino que leemos de la variable `$REPLY`, esto está hecho así porque `$REPLY` preserva los espacios que indentan las líneas del script original.

Por último fijamos dos traps, uno para que cuando acabe el script original `_finscript()` libere el fichero temporal, y otro para que se ejecute `_steptrap()` cada vez que avance un paso el fichero original. Como veremos a continuación, `_steptrap()` para el depurador cuando `_steps` valga 0, o cuando se esté sobre un breakpoint. `_steps` puede tomar un valor positivo indicando el número de pasos a avanzar (p.e `_step=3` indica que avancemos 3 pasos y paremos), puede ser 0 en cuyo caso para el depurador, o puede ser un número negativo en cuyo caso no para el depurador.

3.4. Funciones del depurador

3.4.1. Avanzar paso a paso

Estas funciones estarán definidas en el fichero `bashdb.fn`. La primera de ellas es `_steptrap()`, la cual, cuando se activa el trap `SIGDEBUG`, es llamada por el shell, después de leer, y antes de ejecutar cada línea del script. Su implementación se muestra en el Listado 9.4.

```
# Cada vez que se va a ejecutar una linea
function _steptrap
{
 _lineaactual=$1
 # Si estamos trazando imprime la linea ejecutada
 (( $_trace )) &&\
    _msg "$PS4:$_lineaactual:${_lineas[_lineaactual-1]}"
 # Si hemos llegado al final sale
 if (( _lineaactual==${#_lineas[@]} )) ;then
    exit 0
 fi
 # Decrementa _steps (solo si es mayor o igual a 0)
 if (( $_steps >= 0 )); then
    let _steps=$_steps-1
 fi
 if _tienebp $_lineaactual; then
    _msg "Detenido en breakpoint en linea $_lineaactual"
    _cmdprompt
 elif [ -n "$_condbc" ] && eval $condbc; then
    _msg "Se cumplio la condicion \'$_condbc\' en la"\
```

```
          " linea $_lineaactual"
     _cmdprompt
 elif (( $_steps==0 )); then
     _msg "Parado en linea $_lineaactual"
     _cmdprompt
 fi
}

# Imprime los argumentos
function _msg
{
 echo -e "$@" >&2
}
```

Listado 9.4: Implementación de `_steptrap()`

Cada vez que se ejecuta está función se decrementa `_steps` (el número de pasos a dar antes de parar) siempre que `_steps` sea mayor o igual a 0. Al llegar a 0 es cuando debemos parar.

La función comprueba si se cumple un breakpoint (por línea, o por condición), en cuyo caso para mostrando el prompt, y también para si se cumple que `_steps` vale 0. Si no se cumplen estas condiciones la función retorna y se ejecuta la siguiente línea.

3.4.2. El menú de comandos

Cada vez que se ejecuta la función `_cmdprompt()` se imprime un prompt, en la salida de errores estándar, y se ejecutan los comandos introducidos por el usuario hasta que esté abandona (opción `q` (quit)), pida ejecutar sin traza (opción `g` (go)) la cual ejecuta hasta encontrar un breakpoint o acabar el programa, o hace stepping (opción `s` (step)). La Tabla 9.3 muestra un resumen de los comandos del menú (que imprime `_menu()` cuando introducimos en el prompt el comando `h` u `?`).

Comando	Acción
`bp` *N*	Pone un breakpoint en la línea *N*
`bp`	Lista los breakpoints
`bc` *condición*	Para cuando se cumple la condición *condición*
`bc`	Borra el breakpoint condicional
`cb` *N*	Borra el breakpoint en la línea *N*
`cb`	Borra todos los breakpoints
`p`	Muestra el texto del script original junto con los breakpoint y la posición actual
`g`	Empieza/continua con la ejecución (Go)
`s` [*N*]	Ejecuta *N* pasos (por defecto 1 paso)
`x`	Activa/desactiva la traza

h,?	Imprime un menú de ayuda
!*cmd*	Pasa el comando *cmd* al shell
q	Salir (Quit)

Tabla 9.3: Comandos del depurador

Dentro de las funciones no necesitamos preceder las líneas por guión bajo ya que podemos usar variables locales. El Listado 9.5 muestra la función que recoge los comandos del prompt.

```
function _cmdprompt
{
 local cmd args
 while read -e -p "bashbd>" cmd args
 do
   case $cmd in
        \?|h ) _menu;;
        bc ) _ponbc $args;;
        bp ) _ponbp $args;;
        cb ) _borrabp $args;;
        p ) _print;;
        g ) return;;
        q ) exit 0;;
        s ) let _steps=${args:-1}
            return;;
        x ) _xtrace;;
        !* ) eval ${cmd#!} $args;;
        * )_msg "Comando incorrecto: '$cmd'";;
   esac
 done
}

function _menu
{
 _msg 'Comandos de bashdb:
 bp N           Pone un breakpoint en la linea N
 bp             Lista los breakpoints actuales
 bc cadena      Pone un breakpoint con condicion cadena
 bc             Borra el breakpoint condicional
 cb N           Borra el breakpoint en la linea N
 cb             Borra todos los breakpoints
 p              Imprime el script depurado
 g              Empieza/continua con la ejecucion
 s [N]          Ejecuta N sentencias (por defecto N=1)
 x              Activa/desactiva la traza
 h,?            Imprime este menu
 ! cadena       Pasa cadena al shell
 q              Salir'
 return 1
}
```

Listado 9.5: Implementación de `_cmdprompt()`

3.4.3. Los breakpoints por número de línea

Vamos a estudiar ahora los comandos asociados a los breakpoint de número de línea. El comando `bp` llama a la función `_ponbp()`, la cual puede hacer dos cosas: Si no recibe argumentos lista los breakpoints llamando a `_print()`, sino fija un breakpoint en el argumento dado. El Listado 9.6 muestra la implementación de `_ponbp()`.

```
# Pode un breakpoint o los lista si no recibe parametros
function _ponbp
{
 if [ -z "$1" ]; then
    _print
 elif [ -n "$(echo $1|grep '^[0-9]*')" ]; then
    if [ -n "${_lineas[$1]}" ]; then
       local i
       _lineasbp=($(echo $( (for i in ${_lineasbp[*]} $1
                             do
                                echo $i
                             done) | sort -n) ))
    else
       _msg "La linea $1 esta vacia"
    fi
 else
 _msg "Por favor de como argumento un valor numerico"
 fi
}

# Imprime las lineas del script con breakpoint
# y pos actual
function _print
{
 (
  local i       # Itera el array _lineas
  local j=0     # Itera el array _lineasbp
  local bp=' '  # Simbolo de breakpoint
  local pos=' ' # Simbolos de pos actual
  for ((i=0 ; i < ${#_lineas[@]} ; i++ ))
  do
    if [ ${_lineasbp[$j]} ] &&\
      (( ${_lineasbp[$j]} == $i )); then
       bp='*'
       let j=$j+1
    else
       bp=' '
    fi
    if (( $_lineaactual == $i )); then
       pos='>'
```

```
    else
       pos=' '
    fi
    echo "$i:$bp$pos${_lineas[$i]}"
  done
 ) | more
}
```

Listado 9.6: Implementación de `_ponbp()`

Hay dos problemas que se pueden producir a la hora de que el usuario ponga un breakpoint: El primero es que ponga el breakpoint más allá de la longitud del script original, en este caso simplemente el breakpoint nunca se alcanzará. El segundo es que ponga un breakpoint en una línea en banco, la cual no produce la señal `SIGDEBUG`, y al usar el comando `g` el programa no se detendrá. Para detectar este segundo caso hemos puesto la condición `[ -n "${_lineas[$1]}" ]`.

Después de realizar estos test podemos añadir el breakpoint al array `_lineasbp`, el cual tiene los números de líneas donde hay breakpoints. Para hacer esto necesitamos un código un poco más extraño de lo normal, tal como puede apreciar. La idea es generar un array con los elementos del array más el nuevo elemento `${_lineasbp[*]} $1`, después, este texto se pasa por el comando `sort -n` para ordenar los números y por último generamos un array encerrando la sustitución de comandos entre paréntesis, tal como se explicó en el apartado 3.4 del Tema 6.

```
# Borra el breakpoint indicado, o todos si no
# se da argumento
function _borrabp
{
 if [ -z "$1" ]; then
    unset _lineasbp[*]
    _msg "Todos los breakpoints fueron eliminados"
 elif [ $(echo $1|grep '^[0-9]*') ]; then
    local i
    _lineasbp=( $(echo $(for i in ${_lineasbp[*]}
                         do
                         if (($1!=$i)); then
                            echo $i
                         fi
                         done) ) )
    _msg "Breakpoint en linea $1 eliminado"
 else
    _msg "Especifique un numero de linea valido"
 fi
}
```

Listado 9.7: Implementación de `_borrabp()`

Para poder borrar breakpoint hemos hecho la función `_borrabp()` que se muestra en el Listado 9.7. Su funcionamiento es parecido al de `_ponbp()`.

La otra función relacionada con breakpoint de número de línea es `_tienebp()`, la cual nos dice su hay un breakpoint en la línea pasada como argumento. Esta función es llamada por `_steptrap()` cada vez que se ejecuta con el fin de comprobar si en esa línea hay un breakpoint. Su implementación se muestra en el Listado 9.8.

```
# Comprueba si la linea tiene breakpoint
function _tienebp
{
 local i
 if [ "$_lineasbp" ]; then
    for (( i=0 ; i<${#_lineasbp[@]} ; i++ ))
    do
       if (( ${_lineasbp[$i]} == $1 )); then
          return 0
       fi
    done
 fi
 return 1
}
```

Listado 9.8: Implementación de `_tienebp()`

3.4.4. Los breakpoints condicionales

Nuestro depurador proporciona otro método de detener el script original: Usar un **breakpoint condicional**, que es una cadena especificada por el usuario que se evalúa como un comando (usando `eval`). Si la condición se cumple (el código de terminación es 0), el depurador se para (se ejecuta `_cmdprompt()`).

Esto nos permite, por ejemplo ver cuando una variable alcanza un valor (p.e. `(($x<0))`), o cuando se ha escrito un determinado texto a fichero (`grep` *`texto fichero`*).

Para fijar esta condición usamos el comando `bc` *`cadena`*. Para eliminarla usamos `bc` sin argumentos, esto instala la cadena vacía que es ignorada ya que `_steptrap()` evalúa esta cadena sólo si no es nula. Esto es lo que significa en el Listado 9.4:

```
elif [ -n "$_condbc" ] && eval $_condbc; then
    _msg "Se cumplio $_condbc en la linea $_lineaactual"
    _cmdprompt
```

La función `_ponbc()`, que aparece en el Listado 9.9, es la encargada de quitar o poner el breakpoint condicional.

```
# Pone o quieta el breakpoint condicional
function _ponbc
{
 if [ -n "$*" ]; then
    _condbc="$*"
    _msg "Breakpoint condicional para:\'$*\'"
 else
    _condbc=
    _msg "Breakpoint condicioonal desactivado"
 fi
}
```

Listado 9.9: Implementación de `_ponbc()`

3.4.5. Trazar la ejecución

Como vimos en el apartado 1, bash tiene la opción `xtrace` que se podía fijar con el comando `set -o xtrace`, o desactivar con `set +o xtrace`. Por desgracia, activar esta opción implicaría que se trazase el comportamiento del depurador, con lo que lo que vamos a hacer es que dentro de la función `_steptrap()` comprobaremos el valor de una variable flag llamada `_trace`, si esta está activada mostraremos un mensaje de traza. Esto es lo que significa la siguiente sentencia que aparece en la función `_steptrap()` del Listado 9.4:

```
(( $trace )) && _msg \
"$PS4:$_lineaactual:${_lineas[_lineaactual]}"
```

La función `_xtrace()`, que se muestra en el Listado 9.10, lo que hace simplemente es modificar el valor de este flag.

```
# Cambia el valor del flag _trace
function _xtrace
{
 let _trace="! $_trace"
 if (( $_trace )); then
    _msg "Activada la traza"
 else
    _msg "Desactivada la traza"
 fi
}
```

Listado 9.10: Implementación de `_xtrace()`

3.5. Ejemplo de ejecución

Para ver como funciona el depurador, imaginemos que tenemos un script llamado `prueba` como el que muestra el Listado 9.11.

```
a=0
echo "La variable a vale $a"
a=1
echo "La variable a vale $a"
```

Listado 9.11: Ejemplo de script de prueba

A continuación se muestra un ejemplo de la ejecución del depurador para depurar este script:

```
$ bashdb prueba
Parado en linea 0
bashbd>x
Activada la traza
bashbd>p
0: >a=0
1:  echo "La variable a vale $a"
2:  a=1
3:  echo "La variable a vale $a"
bashbd>bp 3
bashbd>p
0: >a=0
1:  echo "La variable a vale $a"
2:  a=1
3:* echo "La variable a vale $a"
bashbd>s
+ :1:a=0
Parado en linea 1
bashbd>g
La variable a vale 0
+ :2:echo "La variable a vale $a"
+ :3:a=1
Detenido en breakpoint en linea 3
bashbd>s
La variable a vale 1
+ :4:echo "La variable a vale $a"
```

Referencias

Para la elaboración de este texto nos hemos basado principalmente en los textos que se enumeran a continuación.

[1] Cameron Newham, *"Learning the bash Shell, Third Edition"*, O'Reilly March 2005

[2] John Paul Wallington, *"Bash Reference Manual"*, August 2005.

[3] Mendel Cooper, *"Advanced Bash-Scripting Guide"*, Free Software Foundation, July 2002.