

# Programación con *bash*

# Índice de contenidos

1. Primer contacto	Pág. 3
2. Desvío de la salida y la entrada de los comandos	Pág. 3
3. Nombres de ficheros y metacaracteres	Pág. 5
4. Comillas y caracteres de escape	Pág. 6
5. Tuberías de comunicación: pipes	Pág. 6
6. Sustitución de comandos por su salida	Pág. 7
7. Secuencias de comandos	Pág. 8
8. Expresiones	Pág. 9
8.1. Variables	Pág. 9
8.2. Variables especiales	Pág. 13
8.3. Expresiones aritméticas	Pág. 14
8.4. El comando <code>test</code>	Pág. 16
9. Rutinas	Pág. 17
9.1. Algo más sobre los parámetros	Pág. 18
9.2. Valor de retorno	Pág. 20
9.3. Variables locales	Pág. 21
9.4. Bibliotecas de rutinas	Pág. 22
10. Sentencias de control	Pág. 22
10.1. <code>case ... in ... esac</code>	Pág. 23
10.2. <code>if ... then ... fi</code>	Pág. 23
10.3. <code>for ... do ... done</code>	Pág. 24
10.4. <code>while ... done</code>	Pág. 25
10.5. <code>until ... done</code>	Pág. 256

Un *shell* es un programa a medio camino entre el usuario y el sistema operativo. En UNIX hay multitud de *shells*, pero quizá uno de los más frecuentes es el *Bourne Shell* y su mejora *Bourne Again Shell (bash)*. Cada uno de los *shells* que existen tienen particularidades que lo hacen único, pero también muchas similitudes que nos permiten, una vez aprendido uno, trabajar con los demás sin demasiados problemas. En nuestro caso, trabajaremos con el *Bourne Again Shell (bash)*, ya que es el *shell* que trae *Guadalinex* por defecto.

El desarrollo de esta unidad didáctica o lección está pensado para personas que han tenido contacto con algún tipo de lenguaje de programación y que conoce los conceptos de variable, rutina, sentencia de control, ámbitos locales de declaración de identificadores, etc.

## 1. Primer contacto

En el momento en el que una persona obtiene una cuenta en una máquina UNIX, el administrador le asigna una *shell* de trabajo que será el que le dará la bienvenida cada vez que inicie una sesión en esa máquina.

Para averiguar cuál es el *shell* de trabajo que le ha asignado su administrador tan sólo tiene que ejecutar en el terminal el siguiente comando que muestra en pantalla el contenido de la variable `$SHELL`. Esta es la variable que el sistema utiliza de forma estándar para guardar el nombre del *shell* de trabajo.

```
$ echo $SHELL
```

Si no aparece la cadena `/bin/bash` que identifica al deberá cambiar su *shell* de trabajo para poder continuar esta lección. Para ello tan sólo debe ejecutar el comando `chsh`. Al hacerlo aparecerá una lista con los nombres completos de todos los *shell* disponibles en su sistema e instrucciones paso a paso que le permitirán cambiarlo de una forma muy sencilla.

A lo largo de toda la lección iremos mostrando numerosos ejemplos que nos

permitirán ilustrar cada uno de los conceptos estudiados. En algunos de ellos al final de algunas líneas aparece un texto en castellano precedido del símbolo #. Se trata de comentarios que *bash shell* ignora por completo.

## 2. Desvío de la salida y la entrada de los comandos

El *bash shell* permite desviar la entrada y la salida de todos los comandos de forma que estos puedan leer o escribir sus datos en un fichero, en la pantalla, en una línea de comunicaciones o en cualquier otro dispositivo sin que sea preciso cambiar una sola línea del código fuente.

Para desviar, por ejemplo, la salida del comando `ls` a un fichero llamado `lst` basta con teclear el siguiente comando:

```
$ ls -F > lst
```

Este comando crea primero el fichero `lst` y a continuación ejecuta `ls`, pero desviando toda la salida que produzca hacia este fichero. También es posible desviar la salida de un comando añadiéndola a un fichero ya existente. Por ejemplo, si quisiéramos añadir al fichero anterior la frase "estos eran mis ficheros", bastaría con teclear el siguiente comando:

```
$ echo "estos eran mis ficheros" >> lst
```

Si ahora mostramos en pantalla el contenido de `lst` obtendríamos el siguiente resultado:

```
$ cat lst
a.c b.c buf.c copia/ f1.dat f2.dat f3.dat
estos eran mis ficheros
```

La entrada estándar también se puede desviar. Por ejemplo, si deseamos enviar al usuario `juan` el fichero anterior, podemos indicar al comando `mail` que lea el texto a enviar desde este fichero de la siguiente forma:

```
$ mail juan < lst
```

También es posible desviar la entrada usando el símbolo `<<`, pero en este caso el resultado es diferente. Cuando tecleamos un comando de la forma

```
$ cmd << palabra
```

El *shell* crea un fichero temporal en el que introduce todas las líneas que lea de la

entrada estándar hasta encontrar una que contenga tan sólo la palabra indicada. A continuación ejecuta el comando desviando su entrada a través ese fichero temporal. En cierto modo, algo como

```
$ mail juan << fin
Hola Juan, te he enviado el listado de mis ficheros
fin
```

es equivalente a la secuencia de comandos

```
$ echo "Hola Juan, te he enviado el listado de mis ficheros" > tmp
$ mail juan < tmp
$ rm tmp
```

Este tipo de desvío se utiliza de forma casi exclusiva en los programas que construimos en el lenguaje del *shell*. En sesiones interactivas prácticamente no se utiliza.

### 3. Nombres de ficheros y metacaracteres

Todos los *shell* de UNIX permiten el uso de metacaracteres para representar de forma simplificada conjuntos más o menos amplios de nombres de ficheros cuyos nombres encajan en un cierto patrón. A continuación se muestran los cuatro metacaracteres existentes:

- \*, que concuerda con cualquier cadena de caracteres, incluida la cadena vacía.
- ?, que concuerda tan sólo con un carácter.
- [ $a_1a_2 \dots a_n$ ], que concuerda con cualquiera de los caracteres entre los corchetes. Es posible especificar un rango separando el carácter inicial y final del mismo mediante un guión. Por ejemplo [ $a-z$ ] encaja en cualquier letra minúscula (exceptuando la ñ).
- [ $!a_1a_2 \dots a_n$ ], que concuerda con cualquier carácter que no aparezca entre los corchetes. Como en el caso anterior, es posible especificar rangos de caracteres usando el guión.

A continuación se muestran varios ejemplos de uso de los metacaracteres:

```
$ ls -FR
a.c b.c buf.c copia/ f1.dat f2.dat f3.dat

./copia:
a.c b.c buf.c f1.dat f2.dat f3.dat
$ # ficheros cuyo nombre terminan en .c
$ls *.c
a.c b.c buf.c
```

```

$ # ficheros .c cuyo nombre base tiene un carácter
$ ls ?.c
a.c b.c
$ # ficheros cuyo nombre incluye al menos un dígito
$ ls *[0-9]*
f1.dat f2.dat f3.dat
$ # ficheros .c con nombre base de un carácter bajo el directorio actual
$ ls */?.c
copia/a.c copia/b.c
$ # ficheros cuyo nombre tiene tres caracteres y el central es un punto
$ # en el directorio actual y en los inmediatamente inferiores
$ ls ?? */??.?
a.c b.c copia/a.c copia/b.c

```

En el ejemplo hemos mostrado diversos usos de los patrones con el comando `ls`, pero se pueden utilizar casi en cualquier lugar y siempre que el *shell* los encuentra los sustituye de forma automática por la lista completa de nombres de ficheros que encajan en el mismo. Por ejemplo:

```

$ echo "Estos son los ficheros de mi directorio: " *
Estos son los ficheros de mi directorio: a.c b.c c.c

```

## 4. Comillas y caracteres de escape

Tal y como hemos podido ver hasta ahora, el *shell* reserva para uso propio ciertos caracteres. Algunos de ellos son poco comunes en la escritura, pero otros son de uso frecuente y se necesita del algún mecanismo que nos permita impedir al *shell* los interprete con su significado especial.

El primer mecanismo consiste en escribir esos caracteres entre comillas simples o dobles. En el primer caso, todo lo que escribamos entre comillas se interpretará literalmente, esto es, si aparece un `*` se interpretará como el carácter asterisco y no como la lista de todos los nombres ficheros en el directorio actual. En el segundo caso tan sólo se interpretarán literalmente los metacaracteres, mientras que las variables serán sustituidas por sus valores. De momento tan sólo conocemos la variable `$SHELL` que nos permite conocer cuál es el nombre de nuestro *shell* de trabajo, pero será suficiente para ilustrar el comportamiento de las comillas.

```

$ ls
f.dat main.c
$ echo '$SHELL *'
$SHELL *
$ echo "$SHELL *"
/bin/bash *
$ echo $SHELL *
/bin/bash a.c b.c buf.c copia f1.dat f2.dat f3.dat

```

Las comillas permiten delimitar una cadena de texto. Si tan sólo queremos interpretar literalmente un carácter podemos precederlo de una barra invertida.

```
$ ls
f.dat main.c
$ echo \$$SHELL \*
$$SHELL *
$ echo "\$$SHELL \*"
$$SHELL *
$ echo $$SHELL *
/bin/bash a.c b.c buf.c copia f1.dat f2.dat f3.dat
```

## 5. Tuberías y comunicaciones: pipes

Situaciones en las que la salida de un comando se utiliza como la entrada de otro son muy frecuentes en la práctica. Por ejemplo, supongamos que deseamos obtener un listado ordenado alfabéticamente con información acerca de todos los usuarios conectados actualmente en la máquina. Sabemos que el comando `who` permite obtener la información acerca de los usuarios conectados y que `sort` puede ordenarla alfabéticamente. Una posible solución es:

```
$ who > tmp
$ sort < tmp
usuario1 :0 Sep 09 19:53
usuario2 :1 Sep 09 15:45
$ rm tmp
```

En este caso hemos usado el fichero `tmp` para guardar de forma temporal la información proporcionada por `who` antes de usar el comando `sort`. Una forma simplificada de hacer esto es usar tuberías, tal y como se muestra a continuación:

```
$ who | sort
usuario1 :0 Sep 09 19:53
usuario2 :1 Sep 09 15:45
```

El efecto es similar a la creación de un archivo temporal, con la única diferencia de que el uso de las tuberías es mucho más eficiente.

Uno de los problemas de las tuberías es que el usuario no puede observar cuál es la información que corre a través de ellas. Para conseguirlo se puede usar el comando `tee` `f` que toma su entrada estándar y la vuelca en un fichero de nombre `f` al mismo tiempo que en su salida estándar. De esta forma, el comando:

```
$ who | tee who.txt | sort
```

```
usuario1 :0 Sep 09 19:53
usuario2 :1 Sep 09 15:45
```

muestra efectivamente la información ordenada alfabéticamente acerca de los usuarios conectados, pero también la vuelca el listado (sin ordenar) en el fichero `who.txt`.

```
$ cat who.txt
usuario2 :1 Sep 09 15:45
usuario1 :0 Sep 09 19:53
```

## 6. Sustitución de comandos por su salida

Las tuberías nos proporcionan un mecanismo sencillo para utilizar la salida de un comando como entrada de otro. Pero existen multitud de ocasiones en las que lo que realmente necesitamos es usar la salida de uno o varios comandos como parámetros para ejecutar otro comando, no como su entrada estándar.

En estos casos es posible utilizar la técnica conocida como sustitución de un comando por su salida. Esto se consigue ejecutando el comando entre tildes graves (```) y toda la información que muestre en su salida estándar sustituirá a la cadena entre las tildes graves.

Por ejemplo:

```
$ echo La fecha de hoy es `date`
La fecha de hoy es vie sep 09 20:08:54 CEST 2005
```

El comando `date` se ha ejecutado entre tildes graves, por lo que todo aquello que produzca en su salida estándar es capturado y colocado en su lugar.

Es posible combinar tantas sustituciones de comando como sean precisas. Por ejemplo, el siguiente comando muestra la fecha actual y el número de usuarios conectados en la máquina.

```
$ echo Hoy es `date` y hay `who | wc -l` usuarios
Hoy es vie sep 09 20:08:54 CEST 2005 y hay 2 usuarios
```

Este tipo de sustituciones resultan muy potentes y útiles en multitud de ocasiones. Por ejemplo, si quisiéramos editar todos aquellos ficheros `.txt` que contengan la palabra `fecha` bastaría con ejecutar el siguiente comando:

```
$ vi `grep -l fecha *.txt`
```

`grep -l fecha *.txt` muestra en su salida estándar el nombre de todos aquellos ficheros en el directorio actual terminados en `.txt` que contienen la cadena de caracteres

fecha. Por lo tanto, escribir este comando es equivalente a escribir detrás de vi los nombres de todos esos ficheros.

## 7. Secuencias de comandos

Para ejecutar varios comandos uno detrás de otro, la forma más sencilla es escribirlos en varias líneas, introduciendo un retorno de carro al final de cada uno. Pero existen situaciones en las que tenemos que escribir necesariamente dos comandos en una misma línea.

Por ejemplo supongamos que el comando `morosos` produce un listado con las direcciones de correo electrónico de todos los morosos de nuestra empresa en el fichero `morosos.txt`. Si deseamos enviar una carta a todos ellos podemos usar la siguiente sustitución de comando:

```
$ morosos  
$ mail 'cat morosos.txt' < carta
```

`morosos` no produce la lista de morosos en su salida estándar, sino en el fichero `morosos.txt`, por lo que una vez ejecutado el comando es necesario usar `cat` para obtener las direcciones de la lista. Esto exige ejecutar en secuencia dos comandos y para ello podemos usar el punto y coma como separador en vez de teclear dos líneas.

```
$ morosos; mail 'cat morosos.txt' < carta
```

Pero pensemos por un momento en que el comando `morosos` falla por cualquier razón y no produce el fichero `morosos.txt`. En este caso el comando `cat` fallaría también al no encontrarlo o bien leería un ficheros de morosos resultado de una ejecución previa del comando. Es evidente que tan sólo debe enviarse la carta si el programa de `morosos` se ha ejecutado correctamente.

En estos casos podemos usar la composición secuencial de comandos con el símbolo `&&`. `cmd1 && cmd2 && ... && cmdn` ejecuta en secuencia los comandos indicados hasta que falle alguno de ellos. En ese caso los comandos que vienen a continuación no se ejecutan. De esta forma, el comando que buscábamos para enviar nuestras cartas a los morosos sería el siguiente:

```
$ morosos && mail 'cat morosos.txt' < carta
```

Otra posibilidad para ejecutar comandos en secuencia es separándolos con el símbolo `||`. `cmd1 || cmd2 || ... || cmdn` ejecuta los comandos especificados hasta que uno de ellos no falle. De esta forma si queremos mostrar un mensaje de error en caso de que no se haya podido enviar la carta a todos los morosos, podemos usar el siguiente

comando:

```
$ ( morosos && mail 'cat morosos.txt' < carta ) || echo Error
```

Fíjese en que hemos utilizado paréntesis para agrupar los comandos delante del símbolo `||`. Estos hacen que los comandos en su interior se agrupen y se ejecuten como si de un único comando se tratase. De esta forma si cualquiera de los dos falla aparecerá en pantalla el mensaje `Error`.

## 8. Expresiones

Bash shell proporciona un rudimentario lenguaje de expresiones con el que podemos llevar a cabo las operaciones aritméticas, lógicas o de cadena más comunes. En esta sección las estudiaremos todas empezando por las más sencillas de todas, las variables.

### 8.1 Variables

Al igual que cualquier otro lenguaje de programación, los programas escritos en bash shell pueden usar variables para almacenar información de forma temporal. La única diferencia entre estas variables y las de cualquier otro lenguaje de programación de uso habitual es que todas se tratan como si de cadenas de caracteres se tratase y además pueden contener datos de longitud arbitraria. Esto significa que aunque una variable contenga el valor 123, el *shell* interpretará ese valor como una cadena de tres caracteres, nunca como un número entero.

Existen dos tipos principales de variables, las de entorno y las locales. No existe diferencia alguna entre ellas desde el punto de vista de su uso. La única diferencia es que cuando un *shell* invoca a otro, las primeras son heredadas de forma automática por el *shell* hijo, mientras que las del segundo grupo no. Por ejemplo, suponga que en su *shell* de trabajo actual las variables `PATH` y `PWD` son variables de entorno y `L1` y `L2` son variables locales.

El *bash shell* define automáticamente algunas variables de entorno que son de gran interés. A continuación se muestran las más importantes:

<b><i>Nombre</i></b>	<b><i>Significado</i></b>
HOME	Nombre del directorio con la cuenta del usuario

PATH	Un conjunto de nombres de directorio separados por el símbolo : en los que buscar los comandos
LOGNAME	Nombre del usuario
SHELL	Nombre completo del <i>shell</i> que se está utilizando
TERM	Tipo de terminal que se está utilizando

También define de forma automática diversas variables locales que toman valores por defecto cada vez que se ejecuta una copia del *shell*. La siguiente tabla recoge las más importantes:

<b>Nombre</b>	<b>Significado</b>
PPID	Número identificador del proceso padre
PWD	Nombre del directorio de trabajo actual
OLDPWD	Nombre del anterior directorio de trabajo antes de ejecutar por última vez el comando <code>cd</code>
RANDOM	Un número entero generado al azar
PS1	Cadena que presenta el <i>shell</i> cada vez que solicita un comando
PS2	Cadena que presenta el <i>shell</i> cada vez que solicita la continuación de un comando que ocupa varias líneas de pantalla

### 8.1.1 Definición y consulta

En *bash shell* no es necesario declarar las variables. Para introducir una nueva tan sólo es necesario asignarle un valor utilizando la siguiente sintaxis (fíjese en que no existe ningún espacio en blanco al rededor del signo =):

```
$ mi_nombre=juan
$ mi_maquina=bicho.es
```

Para la consulta de variables es necesario preceder el nombre de las variables del signo \$. Por ejemplo:

```
$ mi_direccion=$mi_nombre@$mi_maquina
$ mi_cuenta=file://$mi_maquina/$LOGNAME
```

```

$ echo $mi_direccion
juan@bicho.es
$ echo mi_direccion
mi_direccion
$ mi_otra_direccion=mi_nombre@mi_maquina
$echo $mi_otra_direccion
mi_nombre@mi_maquina

```

Todas las variables que definamos usando este método son variables locales, y por lo tanto no son heredadas por los *shells* que se ejecuten a partir del actual. Para definir una variable de entorno se debe usar el comando `export`. Por ejemplo:

```

$ export mi_direccion=$mi_nombre@$mi_maquina

```

crea una variable de entorno llamada `mi_direccion` que será heredada por cualquier *subshell*. Cualquier variable local puede convertirse en variable de entorno usando el comando `export variable`. Para ver la lista de todas las variables de entorno se puede usar el comando `export` sin ningún parámetro.

### 8.1.2 Consulta avanzada de variables

Anteponer un `$` al nombre de una variable es el modo de consulta más sencillo para las variables, pero existen otras más complejos que se recogen en la siguiente tabla:

<i>Patrón</i>	<i>Significado</i>
<code>\${v}</code>	Idéntico a <code>\$v</code> . Se utiliza en los casos de ambigüedad
<code>\${#v}</code>	Número de caracteres de la cadena que contiene <code>v</code>
<code>\${#v[*]}</code>	Número de elementos del vector <code>v</code>
<code>\${v-c}</code>	Si <code>v</code> ha sido definida su valor, en otro caso <code>c</code>
<code>\${v=c}</code>	Si <code>v</code> ha sido definida su valor, en otro caso le asigna ahora el valor <code>c</code>
<code>\${v+c}</code>	Si <code>v</code> ha sido definida <code>c</code> , en otro caso la cadena vacía

<b>Patrón</b>	<b>Significado</b>
<code>\${v?c}</code>	Si <i>v</i> ha sido definida su valor, en otro caso se muestra en la salida de errores la cadena <i>c</i> y termina la ejecución del <i>shell script</i>
<code>\${v:+c}</code> <code>\${v:=c}</code> <code>\${v:-c}</code> <code>\${v:?c}</code>	Similar a los modos de consulta en que no aparece el símbolo <code>:</code> , pero a las variables además de estar definidas se les exige que el valor que contienen no sea la cadena vacía.
<code>\${v#p}</code> <code>\${v##p}</code>	El valor de la variable, pero eliminando del mismo todos los caracteres iniciales que encajen en el patrón <i>p</i> . Si usamos <code>#</code> se elimina la subcadena más corta y si usamos <code>##</code> se elimina la más larga.
<code>\${v%p}</code> <code>\${v%%p}</code>	Similar a <code>\${v#p}</code> y <code>\${v##p}</code> pero eliminando la subcadena por el final.

A continuación se muestra un ejemplo de uso de todos estos modos avanzados de acceso a las variables.

```

$ mi_nombre=juan
$ mi_maquina=bicho.es
$ mi_direccion=$mi_nombre@$mi_maquina
$ echo ${#mi_nombre}
4
$ cd trash
$ echo $PWD ${#PWD}
/home/juan/trash 16
$ echo $HOME
/home/juan
$ echo ${PWD#$HOME/}
trash
$ ls
f.dat
$ mi_fichero=f.dat
$ cp $mi_fichero ${mi_fichero%.dat}.bak
$ ls
f.dat f.bak

```

### 8.1.3 Vectores

*Bash shell* ofrece la posibilidad de usar vectores de valores utilizando una sintaxis con corchetes para los subíndices. La única limitación en este caso es que el índice de los mismos debe estar entre 0 y 511, por lo que tan sólo son posibles vectores de hasta 512 componentes.

Para crear un vector basta con asignar un valor a uno de sus componentes. Los restantes se irán creando de forma dinámica conforme se vayan necesitando. Para acceder a una componente particular se usa la sintaxis `${v[i]}`. Por ejemplo:

```
$ amigos[0]=Juan
$ amigos[1]=Luís
$ amigos[2]=María
$ amigos[3]=Ana
$ echo ${amigos[0]} está casado con ${amigos[2]}
Juan está casado con María
$ echo Tengo ${#amigos[*]} en la agenda
Tengo 4 amigos en la agenda
$ Mis amigos son ${amigos[*]}
Mis amigos son Juan Luís María Ana
```

La notación `${v[*]}` se utiliza para acceder a todos los elementos del vector de forma conjunta.

## 8.2 Variables especiales

El *bash shell* permite el uso de algunas variables especiales que se recogen en la siguiente tabla. Su valor no se puede cambiar de la forma que hemos visto hasta ahora y generalmente sólo se utilizan en los programas escritos en *shell script*, nunca cuando el *shell* se utiliza de forma interactiva.

<b>Nombre</b>	<b>Significado</b>
<code>\$\$</code>	Número de proceso del <i>shell script</i> en el que se está utilizando
<code>\$0</code>	Nombre completo del <i>shell script</i> en que se está utilizando
<code>\$1..\$9</code>	<code>\$n</code> hace referencia al n-ésimo argumento de la línea de comandos
<code>\$?</code>	El código de retorno del último comando ejecutado
<code>\$#</code>	Número de argumentos
<code>\$*</code>	La lista de todos los argumentos

Todas las variables que hemos mostrado son especiales en el sentido de que no es posible darles un valor utilizando los métodos que hemos estudiado hasta el momento. De todas formas las variables de la forma `$i` son una excepción, ya que es posible cambiar sus valores a voluntad, aunque utilizando el comando `set` y no los mecanismos habituales.

Cuando `set` se ejecuta con parámetros, lo que hace es asignarlos a las variables `$0`, `$1`, `$2`, etcétera. Por ejemplo, a veces suele ser preciso determinar la hora actual a partir del comando `date`. El problema es que este comando muestra en su salida estándar una cadena que informa no sólo de la hora actual sino también del día de la semana, el mes, el año etcétera.

```
$ date
lun sep 12 13:08:54 CEST 2005
```

Para obtener sólo la hora podríamos ejecutar el siguiente comando:

```
$ set 'date'
$ echo $4
```

Al ejecutar `date` entre tildes graves, su salida se convierte en los parámetros del comando `set`, por lo que el comando anterior es equivalente a teclear:

```
$ set lun sep 12 13:11:06 CEST 2005
```

Una vez se ha ejecutado, las variables `$0`, `$1`, ... reciben los valores `lun`, `sep`, ... Por lo tanto `$4` contendrá una cadena que representa la hora actual.

## 8.3 Expresiones aritméticas

*Bash shell* ofrece a los usuarios la posibilidad de realizar operaciones aritméticas sencillas mediante el uso del comando `let`. Este comando aplicado sobre una expresión aritmética devuelve como código de retorno el resultado de evaluarla.

En la expresión se pueden usar los operadores que se recogen en la siguiente tabla, agrupados en orden descendente de prioridad:

<i>Operador</i>	<i>Significado</i>
-	Menos unario
!	Negación lógica
* / %	Multiplicación, división y resto
+ -	Suma y resta

<= >= < >	Operadores relacionales
== !=	Igualdad y desigualdad
=	Asignación a variable
&&	Y lógico
	O lógico

Todos estos operadores, excepto el de asignación, son asociativos por la izquierda. Las expresiones pueden usar paréntesis para modificar el orden de evaluación, pero en ningún momento se comprueba si alguna de ellas produce desbordamiento.

A continuación se muestran varios ejemplos (Fíjese en que no hay ningún espacio en blanco alrededor del signo =, ni tampoco alrededor de los operadores):

```
$ let x=2+3
$ echo $x
5
$ let y=x*5
$ echo $y
25
```

El único punto de interés es que dentro de las expresiones del comando `let` las variables, pese a que se está consultando su valor, no es necesario preceder su nombre del signo `$`. En cualquier caso, podemos hacerlo si así lo deseamos, como se muestra a continuación:

```
$ let x=2+3
$ echo $x
5
$ let y=$x*5
$ echo $y
25
```

Generalmente no existe ningún problema con la mayoría de las expresiones, salvo en aquellas en las que aparecen los operadores `*`, `<` o `>`, puesto que estos caracteres, como ya sabemos, tienen un significado especial para el *shell*. Para evitar que el *shell* los interprete de una forma especial, lo mejor es escribir la expresión entre comillas o bien usar la notación

```
(( expresión ))
```

que es casi equivalente a escribir

```
let " expresión "
```

La única diferencia es que al usar los dobles paréntesis se permite usar espacios en blanco alrededor de todos los operandos y operadores.

```

$ (( x = 2 + 3 ))
$ echo $x
5
$ (( y = $x * 5 ))
$ echo $y
25

```

## 8.4 El comando test

El comando test nos permite llevar a cabo operaciones lógicas sencillas con ficheros, cadenas y números enteros. El formato de este comando es

```
test expresion
```

O de forma equivalente

```
[ expresion ]
```

La expresión puede tener cualquiera de los formatos que se muestran en la siguiente tabla:

<i>Patrón</i>	<i>Significado</i>
-d f	Cierto si f existe y es un directorio
-f f	Cierto si f existe y no es un directorio
-h f	Cierto si f existe y es un enlace simbólico
-n c	Cierto si la cadena c no está vacía
-r f	Cierto si se tiene permiso de lectura en el fichero f
-w f	Cierto si se tiene permiso de escritura en el fichero f
-x f	Cierto si se tiene permiso de ejecución en el fichero f
-z c	Cierto si la cadena c está vacía
c1 = c2	Cierto si las dos cadenas son iguales
c1 != c2	Cierto si las dos cadenas son diferentes
e1 -eq e2	Cierto si los dos enteros son iguales

<code>e1 -ne e2</code>	Cierto si los dos enteros son diferentes
<code>e1 -gt e2</code>	Cierto si e1 es mayor estricto que e2
<code>e1 -ge e2</code>	Cierto si e1 es mayor o igual que e2
<code>e1 -lt e2</code>	Cierto si e1 es menor estricto que e2
<code>e1 -le e2</code>	Cierto si e1 es menor o igual que e2
<code>! e</code>	Cierto si la expresión e es falsa
<code>e1 -a e2</code>	Cierto si las dos expresiones son ciertas
<code>e1 -o e2</code>	Cierto si la primera o la segunda expresión o ambas son ciertas
<code>\( e \)</code>	Los paréntesis se usan para agrupar expresiones y cambiar el orden de evaluación

## 9. Rutinas

El *bash shell* permite definir bibliotecas de rutinas de utilidad que pueden ser compartidas por multitud de *shell scripts*. Para escribir una rutina se utiliza la siguiente sintaxis:

```
nombre_rutina ()
{
  lista de comandos
}
```

Para llamar a una rutina basta con escribir su nombre seguida de los parámetros adecuados, exactamente igual que si estuviésemos ejecutando cualquier otro comando. En el siguiente ejemplo se muestra una sencilla rutina que muestra un mensaje de saludo. Por conveniencia, conviene escribir todas las rutinas en ficheros a los que es preciso dar permiso de ejecución utilizando el comando `chmod`.

```
$ cat > saludo.bsh
#!/bin/bash
saludo ()
{
  echo ¡Hola $1!
}
# Principal
saludo $LOGNAME
saludo amigo $LOGNAME
saludo
```

```
^D
$ chmod ugo+x saludo.bsh
```

La rutina `saludo` se limita a mostrar en pantalla un mensaje dando la bienvenida a la persona cuyo nombre se le pasa como primer parámetro. No es necesario declarar la rutina de una forma especial para indicar que admite parámetros y tampoco se realiza ningún tipo de comprobación en cuanto al número de parámetros reales que se utilizan en la llamada. Para acceder al parámetro  $i$ -ésimo de una rutina se usa la notación  $\$i$ , pero si en la llamada ese parámetro no ha sido suministrado entonces se sustituye por una cadena vacía.

De esta forma, si ejecutamos el anterior *shell script* obtendremos la salida que se muestra a continuación:

```
$ saludo.bsh
¡Hola juan!
¡Hola amigo!
¡Hola !
```

Fíjese en que en la segunda llamada hemos usado dos parámetros y en la tercera ninguno. En el primer caso el parámetro extra es ignorado y en el segundo es sustituido por una cadena vacía.

## 9.1 Algo más sobre parámetros

Desgraciadamente, el mecanismo de variables  $\$i$  tan sólo nos permite acceder a los nueve primeros parámetros que se le pasen a la rutina. Si es preciso usar más parámetros dentro de una rutina tenemos que recurrir al comando `shift`. Cada vez que se ejecuta desplaza los parámetros que se han pasado a una rutina una posición a la izquierda, de forma que el primero se pierde, el segundo se convierte en el primero y así sucesivamente.

El siguiente ejemplo muestra la forma en que funciona este comando:

```
$ cat > shift.bsh
#!/bin/bash
desplaza ()
{
echo $1 $2 $3
shift
echo $1 $2 $3
shift
echo $1 $2 $3
shift
}

# Programa principal

desplaza a b c
^D
```

```
$ chmod ugo+x shift.bsh
$ shift.bsh
a b c
b c
c
```

Para acceder a la lista completa de parámetros podemos usar la variable especial `$*`, que se equivale a una cadena con todos los parámetros que se han pasado a la rutina separados por espacios en blanco.

Para terminar este estudio que estamos realizando de los parámetros, pruebe el siguiente *shell script*:

```
$ cat > curioso.bsh
#!/bin/bash
mas_curioso ()
{
    echo curioso ha recibido $# parámetros
}

curioso ()
{
    echo curioso ha recibido $# parámetros
    mas_curioso $*
}

# Programa principal
curioso "hola $LOGNAME"
^D
$ chmod ugo+x curioso.bsh
```

Si lo ejecuta verá que la salida es la siguiente:

```
$ curioso.bsh
curioso ha recibido 1 parámetros
mas_curioso ha recibido 2 parámetros
```

¿Cómo es posible que si a la rutina `curioso` se le pasa un parámetro y lo utiliza para llamar a `mas_curioso`, esta rutina reciba dos? La razón es que el parámetro que se le ha pasado a `curioso` es una cadena con un espacio en blanco y el comando

```
mas_curioso $*
```

se expande como

```
mas_curioso hola juan
```

Al no haber comillas alrededor del parámetro en esta llamada, el *shell* interpreta

que `mas_curioso` debe recibir dos parámetros: uno con la cadena `hola` y otro con la cadena `juan`. Encerrar la variable `$*` entre comillas tampoco sirve de mucho, ya que en este caso si pasamos más de un parámetro a la rutina `curioso`, entonces `mas_curioso` sólo recibirá uno.

El problema de los espacios es quizá uno de los más importantes cuando trabajamos con *shell scripts*, puesto que suelen ser una fuente importante de errores. Desgraciadamente no existe ninguna solución simple.

## 9.2 Valor de retorno

Las rutinas escritas en *bash shell* j pueden devolver valores de retorno enteros mediante el uso del comando `return`. Pese a que este comando se puede utilizar en cualquier punto de la rutina, provocando su terminación inmediata, lo más adecuado es usarlo siempre al final. Para determinar cuál es el valor de retorno de cualquier rutina se puede usar la variable especial `$?`.

A continuación se muestra un ejemplo de una rutina que calcula y devuelve la suma de los dos números que se le pasan.

```
$ cat > suma.bsh
#!/bin/bash
suma ()
{
  (( result = $1 + $2 ))
  return $result
}

# Programa principal

suma 1 2
echo "1 + 2 = $?"
^D
$ chmod ugo+x suma.bsh
$ suma.bsh
1 + 2 = 3
```

La variable `$?` devuelve realmente el código de salida del último comando ejecutado, y una llamada a rutina no es más que un caso particular de ejecución de un comando. Esto significa que si deseamos utilizar en varias ocasiones el valor de retorno de una rutina, necesariamente deberemos guardarlo en una variable auxiliar. Por ejemplo, el siguiente código

```
$ cat > suma.bsh
...
suma 1 2
echo "1 + 2 = $?"
echo "2 + 1 = $?"
```

produce al ejecutarse la salida siguiente:

```
$ suma.bsh
1 + 2 = 3
2 + 1 = 0
```

La razón es que cuando se usa `$?` por primera vez devuelve el resultado de haber ejecutado la llamada `suma 1 2`. En cambio cuando se usa por segunda vez devuelve el código de salida del comando anterior, esto es, de `echo "1 + 2 = $?"`. Como este comando se ha ejecutado con éxito, la segunda vez que se consulta la variable `$?` devuelve el código de error `0` indicativo de que el último comando se ejecutó adecuadamente.

Mediante el comando `return` tan sólo es posible devolver valores enteros. Si deseamos devolver texto entonces tenemos que usar el comando `echo` para escribirlo en la salida estándar de la función y realizar una sustitución de comando para capturar su salida. A continuación mostramos un ejemplo en el que la rutina `saludo` devuelve un mensaje saludando al usuario que la ejecuta.

```
saludo ()
{
echo "Hola $LOGNAME."
echo 'date'
}

msg='saludo'
echo $msg
```

### 9.3 Variables locales

Dentro de una rutina todas las variables que se definan son globales por defecto, esto es, son visibles no sólo dentro del ámbito de la rutina sino de todo el *shell script* en el que aparecen. Para introducir una variable de ámbito local se utiliza el comando `typeset` seguido del nombre de la variable. Cuando la definamos posteriormente tan sólo será conocida dentro de la rutina en que ha sido declarada.

En el siguiente ejemplo se ilustra el uso de este comando.

```
$ cat > typeset.bsh
#!/bin/bash
locales ()
{
typeset x
(( x = $1 + $2 ))
echo local: x = $x
return $x
}
```

```
# programa principal
x=$LOGNAME
echo global: x = $x
locales 1 2
echo global: x = $x
^D
$ chmod ugo+x typeset.bsh
$ typeset.bsh
global: x = juan
local: x = 3
global: x = juan
```

## 9.4 Biblioteca de rutinas

En muchas ocasiones las rutinas que escribimos para un programa podrían usarse en otros muchos. El *bash shell* nos proporciona un sencillo mecanismo para agrupar rutinas en forma de bibliotecas.

Para construir una biblioteca tan sólo tiene que teclear las rutinas que la componen en un fichero, generalmente con la extensión *.bsh*, que debe encontrarse en alguno de los directorios que indica la variable *\$PATH*. Para cargar la biblioteca y poder hacer uso de sus rutinas debemos usar un comando especial llamado *."*. No, no crea que es una errata. El nombre del comando es un punto.

Por ejemplo:

```
$ . saludo.bash
¡Hola pepe!
¡Hola amigo!
¡Hola !
$ saludo Juan
¡Hola Juan!
$ _
```

Todos los comandos que no aparezcan dentro de una rutina en una biblioteca se ejecutan al cargarla. De ahí que en el ejemplo anterior apareciesen los tres saludos iniciales.

A partir de ahora, en todos los ejercicios usaremos una biblioteca muy útil que nos permitirá mostrar mensajes de aviso o de error. Mostramos su código a continuación:

```
#!/bin/bsh
# error.bash
error ()
{
    echo "$*" 1>&2
    exit 1
}

warning ()
```

```
{
  echo "$*" 1>&2
}
```

## 10. Sentencias de control

El *bash shell* soporta un conjunto amplio de sentencias de control estructurado que nos permiten tomar decisiones o repetir ciertos grupos de comandos con facilidad. En esta sección supondremos que el lector está familiarizado con las sentencias de control de algún otro lenguaje de programación, por lo que la descripción que vamos a realizar de las mismas será bastante somera.

### 10.1 case ... in ... esac

La sintaxis de esta sentencia de control es la siguiente:

```
case e in
p1,1 | p1,2 | ... | p1,n1 ) lista de comandos ;;
p1,1 | p1,2 | ... | p1,n2 ) lista de comandos ;;
...
pm,1 | pm,2 | ... | pm,n1 ) lista de comandos ;;
esac
```

Esta sentencia intenta encajar *e* en alguno de los patrones  $p_{i,j}$  y ejecuta la lista de comandos asociada al primer patrón en que encaje.

A continuación se muestra un ejemplo en el que se desarrolla un sencillo programa que muestra al usuario el calendario del mes que él mismo decida. El programa acepta un parámetro en el que el usuario indica el mes que quiere ver ya sea en formato numérico o usando las tres primeras letras del nombre del mes en cualquier combinación de mayúsculas o minúsculas. Si el mes seleccionado no es correcto se muestra un mensaje de error en la pantalla.

```
$ cat > calendario.bsh
#!/bin/bash
mes=$1
case $mes in
  [1-9]|10|11|12) ;;
  [Ee][Nn][Ee]) mes=1 ;;
  [Ff][Ee][Bb]) mes=2 ;;
  [Mm][Aa][Rr]) mes=3 ;;
  [Aa][Bb][Rr]) mes=4 ;;
  ...
  [Dd][Ii][Cc]) mes=12 ;;
  *) echo Error
  exit
  ;;
```

```
esac
set `date`
cal $mes $7
^D
```

En este programa hemos usado el comando `set `date`` para obtener el año actual.

## 10.2 if ... then ... fi

Este comando permite bifurcar la ejecución de un *shell script* en función a un conjunto de expresiones condicionales. Su sintaxis es la siguiente:

```
if l1
then
  l2
elif l3
then
  l4
elif l5
then
  l6
...
else
  ln
fi
```

El comando `if` ejecuta inicialmente la lista de comandos  $l_1$ . Si el último comando de esta lista tiene éxito (devuelve el código 0), entonces se ejecuta la lista de comandos  $l_2$ . Si el último comando falla se repite el mismo esquema con cada una de las ramas `elif`. Si no se ejecuta con éxito la sentencia asociada a ninguna de ellas entonces se ejecuta la lista de sentencias asociada a la rama `else`. Tanto las ramas `elif` como la rama `else` son opcionales.

A continuación se muestra un sencillo ejemplo que ilustra el comando `if`.

```
$ cat > if.bsh
#!/bin/bash
echo "Teclee un número: "
read num
if (( num > 0 ))
then
  echo $num es positivo
elif (( num == 0 ))
  echo $num es nulo
else
  echo $num es negativo
fi
^D
$ chmod ugo+x if.bsh
$ if.bsh
```

```
Teclee un número:
12
12 es positivo
$ if.bsh
Teclee un número
-1
-1 es negativo
```

En este ejemplo hemos hecho uso por primera vez del comando `read` que juega un papel complementario a `echo`, pues sirve para leer datos desde el terminal.

## 10.3 for ... do ... done

Los bucles `for` permiten ejecutar una lista de comandos en varias ocasiones bajo el control de una variable que toma en cada iteración un valor diferente. A continuación mostramos la sintaxis:

```
for v in c1 c2 ... cn
do
    lista de sentencias
done
```

El comando `for` ejecuta la lista de sentencias que se le indica asignando en cada vuelta uno de los valores `c1 c2 ... cn` a la variable de control `v`. Para especificar los valores que toma la variable de control se pueden usar patrones arbitrarios que se sustituirán por los nombres de los ficheros que encajen en los mismos.

En el ejemplo que se muestra a continuación se desarrolla un pequeño programa que muestra los nombre de todos los ficheros de código C que residen en el directorio actual y en el `/tmp`.

```
$ cat > for.bsh
#!/bin/bash
for f in "Los ficheros de código C son: " *.c /tmp/*.c
do
    echo $f
done
^D
$ chmod ugo+x for.bsh
```

## 10.4 while ... done

La sintaxis de este comando es la siguiente:

```
while l1
do
    l2
```

```
done
```

`while` ejecuta inicialmente la lista de comandos  $l_1$  y finaliza si el último comando en ella falla. En otro caso ejecuta la lista de comandos  $l_2$  y a continuación repite el mismo esquema una y otra vez.

En el siguiente ejemplo se desarrolla un programa que solicita al usuario un número hasta que este está entre 1 y 10.

```
num=0
while (( num < 1 || num > 10 ))
do
  read num
done
```

## 10.5 until ... done

La sintaxis de este comando es la siguiente:

```
until l1
do
  l2
done
```

`until` ejecuta inicialmente la lista de comandos  $l_1$  y finaliza si el último comando en ella tiene éxito. En otro caso ejecuta la lista de comandos  $l_2$  y a continuación repite el mismo esquema una y otra vez.

En el siguiente ejemplo se desarrolla un programa que solicita al usuario un número hasta que este está entre 1 y 10.

```
num=0
until (( num >= 1 && num <= 10 ))
do
  read num
done
```