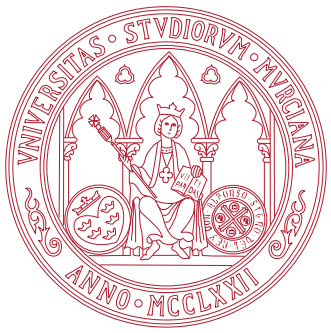

Universidad de Murcia

Facultad de Informática

Departamento de Ingeniería y Tecnología de Computadores

Área de Arquitectura y Tecnología de Computadores

Prácticas de

Introducción a los Sistemas Operativos

2º de Grado en Ingeniería Informática

Boletín de prácticas 3 – Introducción a la programación de *shell scripts* en Linux

Curso 2022/2023

Índice

1. Introducción

Partiendo de las destrezas adquiridas en los boletines anteriores en el manejo del shell como intérprete de órdenes que procesa todo lo que se escribe en el terminal, en este boletín veremos cómo podemos construir guiones shell en base, principalmente, a la agrupación adecuada de estas órdenes.

La programación de guiones shell es una de las herramientas más apreciadas por todos los administradores y muchos usuarios de UNIX/Linux, ya que permite automatizar tareas complejas y/o repetitivas, y ejecutarlas con una sola llamada al guion, incluso de manera automática a una hora preestablecida, sin intervención humana.

En este boletín veremos en primer lugar en qué consiste un guion shell y cuál es su funcionamiento. Tras ello, iremos describiendo sintáctica y semánticamente algunos de los elementos básicos que lo conforman: parámetros de entrada, variables internas, instrucciones aritméticas,... Finalmente, se describirán las estructuras de control más importantes que se pueden utilizar en un guion.

1.1. Objetivos

Al terminar el boletín el alumno debe ser capaz de:

- Comprender el funcionamiento básico de un guion shell.
- Utilizar apropiadamente los parámetros de entrada de un guion, sus variables internas y su código de salida.
- Comprender el funcionamiento de los distintos tipos de entrecomillado así como de la sustitución de órdenes.
- Utilizar órdenes de evaluación aritmética.
- Comprender el funcionamiento y poder usar correctamente la orden `test`.
- Utilizar apropiadamente las principales estructuras de control.

1.2. Órdenes utilizadas

Las órdenes que veremos en este boletín son:

- `clear`
- `echo`
- `let`
- `case`
- `break`
- `date`
- `exit`
- `test`
- `while`
- `shift`
- `whoami`
- `if`
- `for`
- `continue`

En las páginas de manual de cada una de estas órdenes encontrarás información detallada de cómo usarlas.

2. Guiones shell. Concepto y funcionamiento básico

En los boletines anteriores se introdujo el concepto de shell como intérprete de órdenes que procesa todo lo que se escribe en el terminal. Una de las principales características del shell es que puede programarse usando ficheros de texto a partir de órdenes internas y programas externos. Además, el shell ofrece construcciones y facilidades para hacer más sencilla su programación. Estos ficheros de texto se llaman *scripts*, *shell scripts* o *guiones shell*. El intérprete de órdenes seleccionado para realizar estas prácticas es el Bourne-Again Shell o `bash`, cuyo ejecutable es `/bin/bash` o `/usr/bin/bash`.

Supongamos que creamos el siguiente guion shell (un fichero de texto), llamado `limpiafecha.sh`, que contiene dos órdenes, la primera borra la pantalla y, a continuación, la segunda muestra la fecha:

```
clear
date
```

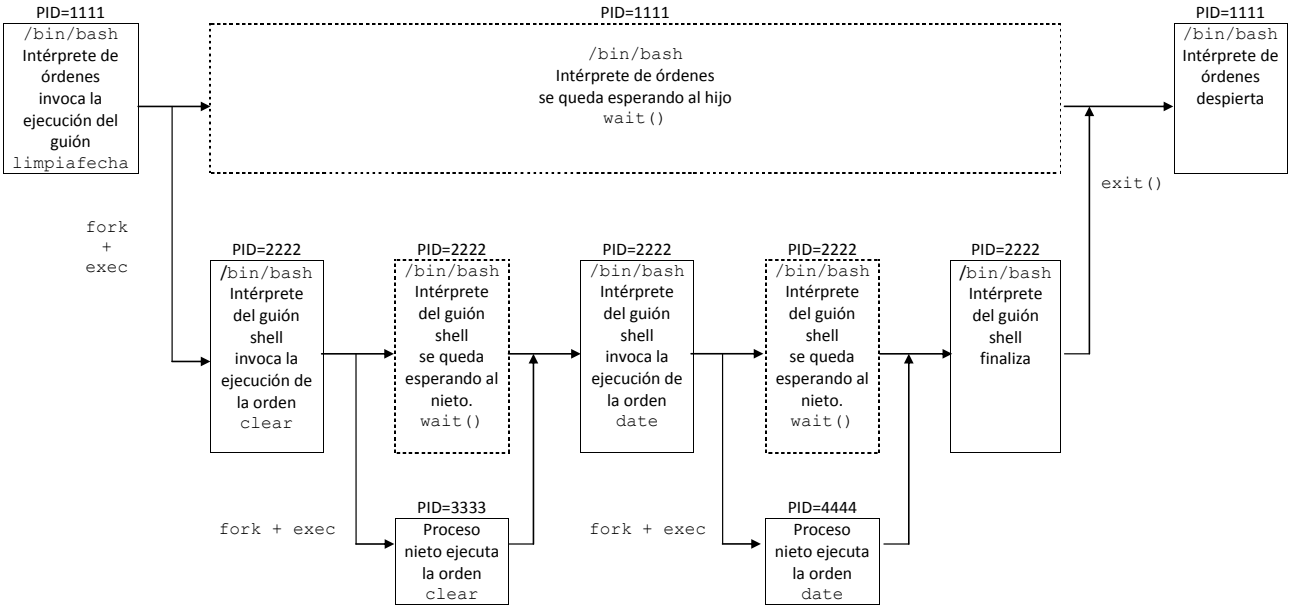

Figura 1: Esquema de funcionamiento de un guion shell para el ejemplo `limpiafecha.sh`.

Para que se ejecute el contenido del script, tenemos que invocar a un intérprete de órdenes (por ejemplo al `bash`), pasándole como parámetro el nombre de este guion:

```
$ bash limpiafecha.sh
```

Este nuevo intérprete de órdenes que estamos invocando no se ejecutará en modo interactivo, sino que irá interpretando el contenido del guion. El procedimiento que se sigue es el siguiente (figura 1):

1. El intérprete de órdenes `/bin/bash` (proceso padre) crea un proceso hijo mediante un `fork`. A continuación, este proceso hijo pone en funcionamiento un nuevo `/bin/bash`, mediante un `exec`, que se encargará de ir interpretando el contenido del guion shell.

2. El proceso padre se queda a la espera mientras no termine el nuevo proceso hijo de ejecutar el guion.

3. El proceso hijo hace un `fork` y, a continuación, el nuevo proceso creado ejecuta la orden `clear` mediante un `exec`. Por tanto, esta orden la ejecutará, un proceso nieto del shell incial.

4. El proceso hijo se queda a la espera de que termine la ejecución de `clear`.

5. Una vez que ha finalizado la ejecución de la orden `clear` (el proceso nieto ha terminado), el proceso hijo repite los mismos pasos para la orden `date` (creación de un nuevo proceso nieto que ejecutará esta orden).

6. Si quedasen órdenes por ejecutar se seguiría el mismo procedimiento (el proceso hijo crea un proceso nieto por cada orden a ejecutar).

7. Cuando finaliza el proceso hijo (se han ejecutado todas las órdenes del guion shell), el proceso padre reanuda su ejecución.

En este ejemplo introductorio, para la ejecución de las dos líneas del interior del guion se han creados sendos procesos nietos. Como veremos más adelante, esto no siempre ocurre de esta manera, pues cuando en una línea del script se utilizan únicamente órdenes internas del bash, dicha línea puede ser ejecutada por el proceso hijo directamente, sin necesidad de crear un proceso nieto.

Un guion shell puede incluir comentarios. Para ello, se debe anteponer el carácter # al texto que constituye dicho comentario. Por ejemplo:

```
clear      # borrar la pantalla
date       # mostrar la fecha
```

Por último, para acabar esta sección introductoria, cabe indicar que si se le dan permisos de ejecución al guion, como por ejemplo así:

```
$ chmod 744 limpiafecha.sh
```

entonces podremos llamarlo directamente, como a cualquier otro ejecutable:

```
$ ./limpiafecha.sh
```

En este caso, podemos elegir el shell que queremos que interprete nuestro guion, indicándolo en la primera línea de éste. Así, por ejemplo, si queremos asegurarnos de que nuestro guion sea interpretado por `/bin/bash`, independientemente del shell desde el que sea llamado, le debemos añadir la línea `#!/bin/bash` al principio. De esta manera, el guion de ejemplo, `limpiafecha.sh`, quedaría así:

```
#!/bin/bash
clear      # borrar la pantalla
date       # mostrar la fecha
```

Cabe destacar que esta primera línea del guion simplemente será ignorada si el guion es ejecutado de la primera manera que se indicó al inicio de esta sección:

```
$ bash limpiafecha.sh
```

3. Variables, parámetros de entrada y código de salida

3.1. Variables

Cada intérprete tiene unas variables ligadas a él, a las que el usuario puede añadir tantas como desee. Para dar un valor a una variable `variable` se usa la sintaxis:

```
variable=valor
```

En bash no existen tipos de datos. Por tanto, no es necesario declarar ninguna variable. Cada variable se crea con tan solo asignarle un valor a su referencia. En un principio, este valor es simplemente una cadena de caracteres, que se interpretará según el contexto como un número, un carácter o una cadena de caracteres.

SINTAXIS. Asignación de valor a una variable: uso de espacios en blanco.
A la hora de realizar una asignación de valor a una variable no puede haber espacios entre el nombre de la variable, el signo = y el valor. Por otra parte, si se desea que el valor contenga espacios, es necesario utilizar comillas para delimitar su inicio y su final.

Para obtener el valor de una variable hay que anteponerle a su nombre el carácter $. Por ejemplo, para visualizar el valor de una variable, usando la orden `echo`:

```
echo $variable
```

De esta forma, podemos, a partir de `limpiafecha.sh`, escribir `limpiafechasaludo.sh`:

```
#!/bin/bash
clear
date
saludoprimero=Hola
numeroboletin=3
```

```
saludosegundo="Bienvenido al boletín"
saludotercero="Prácticas   de   ISO"
echo $saludoprimero
echo $saludosegundo $numeroboletin
echo $saludotercero
```

Con lo que si lo ejecutamos ahora, veremos en pantalla algo como:

```
jue mar 21 11:17:10 CET 2019
Hola
Bienvenido al boletín 3
Prácticas de ISO
```

Como podemos apreciar, la frase que formaba el contenido de la variable `saludotercero` se ha mostrado en pantalla correctamente, pero se han reducido los múltiples espacios en blanco que separaban cada palabra. Esto es debido a que la labor de la orden `echo` consiste en mostrar en pantalla, usando un espacio en blanco como separador, cada uno de los parámetros que se le pasen. Pues bien, en este caso han sido tres parámetros los que se le han pasado (`Prácticas`, `de` e `ISO`), ya que, como en cualquier orden del shell, un parámetro se toma como un conjunto de caracteres separado por uno o varios espacios en blanco de otro o, incluso, separado por un tabulador o por un salto de línea. En la siguiente sección veremos cómo podemos mostrar un frase como la de este ejemplo preservando todos esos espacios en blanco existentes.

Además de en el interior de un guion, también se pueden utilizar las variables directamente en el intérprete de órdenes. Por ejemplo:

```
$ nombre=Pepito
$ echo Hola $nombre
Hola Pepito
```

También se pueden usar las variables para contener el texto correspondiente a la invocación de una orden. Por ejemplo:

```
$  ls
   fichero01
   fichero02
   fichero03

$  mils=ls

$  mils                          # No hace nada porque trata
   mils: command not found       # de ejecutar la orden 'mils'
                                 # que no existe tal cual

$  $mils                         # sustituye '$mils' por su contenido,
   fichero01                     # es decir, el texto 'ls',
   fichero02                     # y entonces ejecuta 'ls'
   fichero03

$  echo $mils                    # Muestra el contenido de la
   ls                            # variable 'mils', es decir,
                                 # el texto 'ls'
```

En este ejemplo, en primer lugar se crea una variable llamada `mils` cuyo contenido es la cadena de caracteres `ls`.

En la siguiente línea, al teclear `mils`, el intérprete de órdenes intenta ejecutar una orden llamada tal cual, pero, al no encontrarla, simplemente sacará un mensaje de error.

A continuación, al anteponer el carácter $ a la cadena `mils`, se está indicando que lo primero que se quiere hacer es obtener el valor de una variable. El intérprete de órdenes realiza esa labor previa, sustituyendo el nombre de esta variable por su valor, en este caso por la cadena `"ls"`. Tras ello, el intérprete de órdenes ejecuta la orden resultante: `ls`.

Por último, con la orden `echo $mils`, en primer lugar el intérprete de órdenes realiza la labor previa de sustituir `$mils` por su valor: `ls`. A continuación, el intérprete de órdenes ejecuta la orden resultante, `echo ls`, que simplemente mostrará la cadena `ls` en pantalla.

3.2. Parámetros

Como cualquier programa, un guion shell puede recibir parámetros en la línea de órdenes para procesarlos durante su ejecución. Los parámetros recibidos se guardan en una serie de variables predefinidas que el script puede consultar cuando lo necesite. Los nombres de estas variables son:

```
$1 $2 $3 ... ${10} ${11} ${12} ...
```

- La variable `$0` contiene el nombre con el que se ha invocado el script, `$1` contiene el primer parámetro, `$2` contiene el segundo parámetro,....
- La variable `$@` contiene todos los parámetros recibidos.
- La variable `$#` contiene el número de parámetros recibidos.

A continuación se muestra un sencillo ejemplo de un guion shell que muestra los cuatro primeros parámetros recibidos:

```
#!/bin/bash
echo "El nombre del programa es: $0"
echo "---------------------------------"
echo "El primer parámetro recibido es: $1"
echo "El segundo parámetro recibido es: $2"
echo "El tercer parámetro recibido es: $3"
echo "El cuarto parámetro recibido es: $4"
echo "---------------------------------"
echo "El conjunto de parámetros recibidos es: $@"
echo "El número total de parámetros recibidos es: $#"
```

La orden `shift` mueve todos los parámetros una posición a la izquierda. Esto hace que el contenido del parámetro `$1` desaparezca y sea reemplazado por el contenido de `$2`, que `$2` sea reemplazado por `$3`, etc. Por tanto, con cada `shift` el contenido de `$@` se va actualizando, desapareciendo el parámetro que se encontraba a la izquierda y, de igual manera, se decrementa el valor de `$#`. Un ejemplo sencillo de un guion shell que muestra el nombre del ejecutable, el número total de parámetros, todos los parámetros y los cuatro primeros parámetros es el script `parametros2.sh`:

```
#!/bin/bash
echo El nombre del programa es: $0
echo El número total de parámetros es: $#
echo Todos los parámetros recibidos son: $@
echo El primer parámetro recibido es: $1
shift
echo El segundo parámetro recibido es $1
shift
echo El tercer parámetro recibido es $1
echo El cuarto parámetro recibido es $2
echo Los parámetros, tras ejecutar dos shifts, son: $@
echo El número de parámetros, tras ejecutar dos shifts, es: $#
```

Un ejemplo de uso de este guion sería:

```
$ bash parametros2.sh a b c d e f g h i
El nombre del programa es: parametros2.sh
El número total de parámetros es: 9
Todos los parámetros recibidos son: a b c d e f g h i
El primer parámetro recibido es: a
El segundo parámetro recibido es: b
El tercer parámetro recibido es: c
El cuarto parámetro recibido es: d
Los parámetros que tenemos tras dos shifts, son: c d e f g h i
El número total de parámetros, tras dos shifts, es: 7
```

En general, si deseamos acceder al parámetro i-ésimo, podemos usar la expresión `${!i}`, siendo `i` una variable previamente definida y con valor numérico. Con lo que un programa similar al ejemplo anterior podría ser este guion, llamado `parametros.sh`:

```
#!/bin/bash
echo El nombre del programa es $0
echo El número total de parámetros es $#
echo Todos los parámetros recibidos son $@

i=1
echo El primer parámetro recibido es ${!i}
i=2
echo El segundo parámetro recibido es ${!i}
i=3
echo El tercer parámetro recibido es ${!i}
i=4
echo El cuarto parámetro recibido es ${!i}
```

De esta manera, un par de ejemplos de uso de este último guion podrían ser:

```
$ bash parametros.sh El Rey León fue espectacular
  El nombre del programa es: parametros.sh
  El número total de parámetros es: 5
  Todos los parámetros recibidos son: El Rey León fue espectacular
  El primer parámetro recibido es: El
  El segundo parámetro recibido es: Rey
  El tercer parámetro recibido es: León
  El cuarto parámetro recibido es: fue

$ bash parametros.sh "El Rey León" fue espectacular
  El nombre del programa es: parametros3.sh
  El número total de parámetros es: 3
  Todos los parámetros recibidos son: El Rey León fue espectacular
  El primer parámetro recibido es: El Rey León
  El segundo parámetro recibido es: fue
  El tercer parámetro recibido es: espectacular
  El cuarto parámetro recibido es:
```

En el primero ejemplo, el guion ha recibido 5 parámetros: `El`, `Rey`, `León`, `fue` y `espectacular`; mientras que en el segundo ejemplo de uso, se han utilizado las comillas dobles para agrupar las 3 primeras palabras como una única cadena de caracteres que se pasa como primer parámetro, por lo que los parámetros han sido solamente 3: `"El Rey León"`, `fue` y `espectacular`.

3.3. Código de salida de un guion shell

En Linux, todos los procesos cuando finalizan devuelven un valor a su proceso padre. A este valor se le llama código de salida (*exit code*) o estado de salida (*exit status*). En el estándar POSIX la convención es que se devuelva un valor 0 cuando la ejecución haya sido correcta y un valor entre 1 y 255 cuando la ejecución haya fallado de alguna manera.

Una forma para saber si una orden ha finalizado con éxito o ha tenido problemas es consultando el valor de la variable `$?`. Esta variable contiene en cada momento el código de salida devuelto por la última orden ejecutada.

Veamos un ejemplo de su uso:

```
$ mkdir d
$ cd d
$ touch f1
$ touch f2
$ touch f3

$ ls f*
f1  f2  f3


$ echo $?
0

$ ls g*
ls: no se puede acceder a 'g*': No such file or directory

$ echo $?
2

$ echo $?
0
```

En este ejemplo, la primera vez que se consulta el valor de la variable `$?` (orden `echo $?`) vemos que éste es igual a 0, pues la ejecución de la orden que precede a esta consulta (`ls f*`) resultó correcta. Tras ello, se ejecuta la orden `ls g*`. Observamos que dicha ejecución hace que se muestre un mensaje de error en pantalla y, además, deja almacenado en `$?` el código de error, en este caso igual a 2, tal como observamos al ejecutar `echo $?` justo a continuación. Finalmente, como esta ejecución de la orden `echo $?` se ha llevado a cabo sin problemas, el valor de dicha variable vuelve a ser igual a 0, como podemos observar.

De igual forma, podemos anteponer el caracter «!» a la orden a evaluar si lo que nos interesa es considerar lo opuesto del código retornado por dicha orden. Así, continuando con el ejemplo anterior tendríamos:

```
$ ! ls f*
f1  f2  f3

$ echo $?
1

$ ! ls g*
ls: no se puede acceder a 'g*': No such file or directory

$ echo $?
0
```

Como veremos más adelante, el código de salida de una orden se utilizará como condición en las instrucciones `if` y `while`. En la instrucción `if` de cara a determinar, condicionalmente, qué instrucciones ejecutar, y en la instrucción `while` para saber si se debe continuar con la siguiente iteración del bucle o no.

En un guion shell, la ejecución de la orden `exit n` provoca que el script finalice, devolviendo el valor `n` como código de salida. En caso de que el script acabe sin haber ejecutado la orden `exit`, el código de salida devuelto corresponderá al de la última orden ejecutada dentro de dicho script.

A continuación, se muestra un ejemplo donde el guion `llamar.sh` llama a otro guion de nombre `num.sh`. Este último muestra un par de mensajes y finaliza. Cuando `num.sh` termina, devuelve, usando `exit`, el valor 1 al proceso que lo ha llamado. La ejecución continúa por `llamar.sh` que muestra el valor devuelto por `num.sh`.

`llamar.sh`:

```
#!/bin/bash
echo "Inicio del guion $0"
echo "Llamando al guion num.sh"
bash num.sh
res=$?
echo "El guion num.sh ha devuelto el valor $res"
echo "Fin del guion $0"
```

`num.sh`:

```
#!/bin/bash
echo "Inicio del guion $0"
echo "Fin del guion  $0"
exit 1
```

Con lo que al ejecutarlos tendríamos:

```
$ bash llamar.sh
Inicio del guion llamar.sh
Llamando al guion num.sh
Inicio del guion num.sh
Fin del guion num.sh ha devuelto el valor 1
El guion num.sh
Fin del guion llamar.sh
```

4. Caracteres especiales y de entrecomillado

Los mecanismos de protección se emplean para que los caracteres que son especiales para `bash` no se traten de forma especial, para que palabras reservadas no sean reconocidas como tales y para evitar la evaluación de variables.

Entre los caracteres que son especiales para el bash tenemos los comodines para la expansión de rutas, vistos en el boletín 1, los caracteres de control para la separación de órdenes, que veremos en el boletín 4, y los metacaracteres, que se utilizan para separar palabras y que son: `* $ | & ; ( ) { } < >` `espacio` y `tabulador`.

Hay 3 mecanismos de protección: el carácter de escape, las comillas simples y las comillas dobles[1].

- Una barra inclinada invertida no entrecomillada, \, es el carácter de escape que se utiliza para preservar el valor literal del siguiente carácter que lo acompaña. Por ejemplo:

[1]Las comillas simples y dobles son las que aparecen en la tecla que hay a la derecha del 0 y en la tecla del 2, respectivamente.

```
$ echo "Yesterday I won \$5,000 in Las Vegas."
Yesterday I won $5,000 in Las Vegas.
```

Existe una excepción del uso indicado para la barra inclinada invertida, que es cuando va seguida de <nueva-línea>. En este caso, la combinación \<nueva-línea> se trata como una continuación de línea (esto es, se quita del flujo de entrada y no se tiene en cuenta). Por ejemplo:

```
$ ls \
> -l
```

sería equivalente a ejecutar:

```
$ ls -l
```

- Encerrar caracteres entre comillas simples (`' '`) preserva el valor literal de todos ellos. Una comilla simple no puede estar entre comillas simples, ni siquiera precedida de una barra invertida.
- Encerrar caracteres entre comillas dobles (`" "`) preserva el valor literal de todos ellos, con la excepción de `$`, `` ` `` (comilla simple invertida) y `\`. Los caracteres `$` y `` ` `` mantienen su significado especial dentro de comillas dobles. La barra invertida mantiene su significado especial solamente cuando está seguida por uno de los siguientes caracteres: `$`, `` ` ``, `"`, `\` o <nueva-línea>. Una comilla doble puede aparecer entre otras comillas dobles precedida de una barra invertida.

Así, por ejemplo, el programa `comillas.sh`:

```
#!/bin/bash
usuario="Pepe"
sueldo=2500
echo 'CON COMILLAS SIMPLES: El usuario $usuario gana $sueldo $ al mes'
echo "CON COMILLAS DOBLES: El usuario $usuario gana $sueldo \$ al mes"
```

mostraría la siguiente salida al ejecutarse:

```
$ bash comillas.sh
CON COMILLAS SIMPLES: El usuario $usuario gana $sueldo $ al mes
CON COMILLAS DOBLES: El usuario Pepe gana 2500 $ al mes
```

Como podemos apreciar, con el uso de las comillas dobles podemos expandir el valor de las variables antes de ejecutar `echo`, mientras que con las comillas simples el texto se ha mostrado literalmente, sin ninguna sustitución.

5. Sustitución de órdenes

Si se encierra una cadena entre paréntesis y se precede de un signo `$`, `$(cadena)`, se fuerza al shell a ejecutar `cadena` como una orden y sustituir todo el texto `$(cadena)` por la salida estándar que produce[2].

Veamos un ejemplo: el programa `saludo.sh`:

[2]El mismo efecto de sustitución de órdenes mostrado cuando encerramos una cadena entre paréntesis precedida de un signo $ se puede conseguir al encerrar una cadena entre comillas simples invertidas (es decir `` ` ` ``). En cualquier caso, puede ser aconsejable utilizar siempre la primera manera mostrada (paréntesis precedido de $) de cara a evitar que, cuando leamos o escribamos código, confundamos las comillas simples inversas con las comillas simples normales. Estas últimas tienen un uso bien distinto, como se ha visto anteriormente.

```
#!/bin/bash

#almacena en la variable 'fecha' la salida estándar
#de la orden 'date'
fecha=$(date)

#almacena en la variable 'usuario' la salida estandar
#de la orden 'whoami', es decir, nombre del usuario actual
usuario=$(whoami)

#muestra en pantalla el mensaje de saludo
echo "Hola $usuario. La fecha de hoy es: $fecha"
```

mostraría la siguiente salida al ejecutarse:

```
$ bash saludo.sh
Hola alumno. La fecha de hoy es: mar sep 22 19:20:50 CEST 2018
```

Un programa equivalente al anterior sería:

```
#!/bin/bash
echo "Hola $(whoami). La fecha de hoy es: $(date)"
```

donde la sustitución de órdenes se realiza directamente dentro del entrecomillado doble.

6. Ejemplos de uso combinado de variables, entrecomillado y sustitución de órdenes

En esta sección vamos a ver y comentar un conjunto de ejemplos que combinan el uso de la orden `echo` con algunos de los conceptos descritos en las secciones anteriores. Partimos de un directorio en el que tenemos 3 ficheros: `f1`, `f2` y `f3`. Es decir:

```
$ ls -l *
-rw-rw-r--. 1 alumno alumno 0 sep 27 11:27 f1
-rw-rw-r--. 1 alumno alumno 0 sep 27 11:27 f2
-rw-rw-r--. 1 alumno alumno 0 sep 27 11:27 f3
```

1. En un primer ejemplo, vemos cómo la ejecución de `echo` muestra la cadena de caracteres que se le pasa como parámetro tal cual, al llevar comillas simples:

   ```
   $ echo 'ls -l *'
   ls -l *
   ```

2. En un segundo ejemplo, podemos ver cómo la orden `echo` también muestra la cadena de caracteres tal cual, porque, aunque la cadena esté con comillas dobles, no hay en su interior ninguna variable que sustituir, ni orden que invocar previamente:

   ```
   $ echo "ls -l *"
   ls -l *
   ```

3. Veamos ahora un ejemplo donde no se utiliza ningún tipo de entrecomillado. En este caso, como aparece un comodín que permite especificar múltiples ficheros al mismo tiempo, tal como vimos en el primer boletín, lo primero que se hace es la expansión de dicho comodín y es el resultado de esta expansión (el `*` se expande a la cadena de todos los nombres de entradas (ficheros y subdirectorios) del directorio actual: `f1 f2 f3`) lo que conforma realmente el resto de parámetros de `echo`. Por tanto, la orden `echo` actuará mostrando en pantalla las 5 cadenas de caracteres que le llegan como parámetros: `ls`, `-l`, `f1`, `f2` y `f3`:

```
$ echo ls -l *
ls -l f1 f2 f3
```

4. En un nuevo ejemplo, primero se invoca la orden de listado, `ls -l *`, cuya salida estándar queda recogida dentro de las comillas dobles y, por tanto, conforma una cadena de caracteres que se le pasa como un único parámetro a la orden `echo`:

```
$ echo "$(ls -l *)"
-rw-rw-r--. 1 alumno alumno 0 sep 27 11:27 f1
-rw-rw-r--. 1 alumno alumno 0 sep 27 11:27 f2
-rw-rw-r--. 1 alumno alumno 0 sep 27 11:27 f3
```

5. Ahora, como ocurrió en el caso anterior, primero se invoca la orden de listado, `ls -l *`. Sin embargo, en esta ocasión, la salida producida por esta orden no queda recogida entre comillas dobles, por lo que el intérprete entiende que los saltos de línea que se encuentran en este texto son separadores de parámetros para la orden `echo`. La ejecución de esta orden simplemente sustituye estos separadores por espacios en blanco cuando los muestra en pantalla:

```
$ echo $(ls -l *)
-rw-rw-r--. 1 alumno alumno 0 sep 27 11:27 f1 -rw-rw-r--. 1 alumno
alumno 0 sep 27 11:27 f2 -rw-rw-r--. 1 alumno alumno 0 sep 27 11:27
f3
```

6. Obtenemos el mismo resultado que en el ejemplo 4 si usamos una variable donde quede recogida la salida estándar de la orden `ls -l *` y, a continuación, le pasamos como parámetro a la orden `echo` el contenido de esa variable, conformado como una única cadena de caracteres mediante el uso de comillas dobles:

```
$ listado=$(ls -l *)
$ echo "$listado"
-rw-rw-r--. 1 alumno alumno 0 sep 27 11:27 f1
-rw-rw-r--. 1 alumno alumno 0 sep 27 11:27 f2
-rw-rw-r--. 1 alumno alumno 0 sep 27 11:27 f3
```

7. Obtenemos el mismo resultado que en el ejemplo 5 si guardamos en una variable la salida de la orden `ls -l *` y, a continuación, le pasamos el contenido de esa variable a la orden `echo` sin usar comillas:

```
$ listado=$(ls -l *)
$ echo $listado
-rw-rw-r--. 1 alumno alumno 0 sep 27 11:27 f1 -rw-rw-r--. 1 alumno
alumno 0 sep 27 11:27 f2 -rw-rw-r--. 1 alumno alumno 0 sep 27 11:27
f3
```

8. Por último, en el siguiente ejemplo, al estar la cadena con comillas simples, no se invoca ninguna orden antes de que actúe la orden `echo`, la cuál, por tanto, muestra en pantalla la cadena de caracteres que recibe como parámetro, `$(ls -l *)`, sin más:

```
$ echo '$(ls -l *)'
   $(ls -l *)
```

-, +	Menos y más unarios
**	Exponenciación
*, /, %	Multiplicación, división, resto
+, -	Adición, sustracción
=, +=, -=, *=, /=, %=,	Asignación: simple, después de la suma, de la resta,...

Tabla 1: Algunos operadores aritméticos permitidos por bash, en orden de precedencia decreciente.

7. Evaluación aritmética

El shell permite que se evalúen expresiones aritméticas. La tabla 1 muestra algunos operadores que se pueden utilizar en estas expresiones, en orden de precedencia decreciente y agrupando a aquellos de igual precedencia.

La evaluación de las expresiones aritméticas se realiza teniendo en cuenta una serie de reglas:

- La evaluación se hace con enteros largos sin comprobación de desbordamiento, aunque la división por 0 se atrapa y se señala como un error.
- Las subexpresiones entre paréntesis se evalúan primero y pueden sustituir a las reglas de precedencia establecidas en la tabla 1.
- Se permite que las variables del shell actúen como operandos: se realiza la expansión de variables antes de la evaluación de la expresión.
- El valor de un operando se fuerza a un entero largo dentro de una expresión.
- Las constantes con un `0` inicial se interpretan como números octales, mientras que un `0x` ó `0X` inicial denota un número en hexadecimal.

Con la orden `let` se pueden evaluar las expresiones aritméticas que se le pasen como argumentos. Algunos ejemplos de su uso serían:

```
$ a=5
$ b=8
$ let c=$a+$b
$ echo "El resultado de sumar $a y $b es: $c"

El resultado de sumar 5 y 8 es: 13

$ let b=7%5
$ echo "El resto de la división es: $b"

El resto de la división es: 2
```

SINTAXIS. Orden `let`: uso de espacios en blanco.
En la orden `let` no se debe dejar espacios en la expresión a evaluar: ni alrededor del =, ni entre los diferentes operandos y operadores.

8. La orden `test`

La orden `test` evalúa si la expresión que recibe como parámetro es verdadera o falsa, devolviendo como código de salida un 0 o un 1 respectivamente. Con esta orden podemos comparar valores de variables, así como conocer las propiedades de un fichero, como veremos a continuación. La sintaxis de esta orden es:

```
test expresión
```

SINTAXIS. Orden `test`: uso de espacios en blanco.
Al utilizar la orden `test`, es necesario incluir espacios en blanco entre los diferentes componentes de la expresión (operadores, operandos, paréntesis,...).

La expresión puede incluir operadores de comparación como los siguientes:

- Para **números**: `arg1 OP arg2`, donde `OP` puede ser uno de los siguientes:

`-eq`	Igual a
`-ne`	Distinto de
`-lt`	Menor que
`-le`	Menor o igual que
`-gt`	Mayor que
`-ge`	Mayor o igual que

Por ejemplo, directamente desde el intérprete de órdenes:

```
$ let a=50-40
$ let b=20-10
$ test $a -eq $b    # ¿contenido de $a es igual al de $b?
$ echo $?           #
0                   # respuesta: sí

$ test $a -ge $b    # ¿contenido de $a es mayor o igual que el de $b?
$ echo $?           #
0                   # respuesta: sí

$ test $a -gt $b    # ¿contenido de $a es mayor que el de $b?
$ echo $?           #
1                   # respuesta: no

$ let a=50-40
$ let b=22-10
$ test $a -eq $b    # ¿contenido de $a es igual al de $b?
$ echo $?           #
1                   # respuesta: no

$ test $a -gt $b    # ¿contenido de $a es mayor que el de $b?
$ echo $?           #
1                   # respuesta: no

$ test $a -lt $b    # ¿contenido de $a es menor que el de $b?
$ echo $?           #
0                   # respuesta: sí

$ test $a -le $b    # ¿contenido de $a es menor o igual que el de $b?
$ echo $?           #
0                   # respuesta: sí
```

Es importante destacar que en las comparaciones con números, si utilizamos una variable que no está definida, se producirá un error de sintaxis en la ejecución de esta orden. Por ejemplo:

```
$ let a=10
$ let b=50
$ test $a -eq $c
-bash: test: 10: unary operator expected
$ echo $?
2
```

En este ejemplo, la variable `c` no está inicializada, por lo que a la orden `test` le llega como parámetros una expresión formada por el valor de la variable `a`, 10, la cadena `-eq` y una cadena de texto vacía en representación de la variable no definida `c`. Por esta razón, la ejecución de esta orden responde con un mensaje de error donde indica que si solamente le pasamos un operando, 10, necesitaría un operador unario y no un operador binario, como es el caso de `-eq`.

- Para **caracteres alfabéticos** o **cadenas** teniendo en cuenta el orden lexicográfico[3]:

`-z cadena`	Verdad si la longitud de `cadena` es cero.
`-n cadena`	Verdad si la longitud de `cadena` no es cero.
`cadena1 = cadena2`	Verdad si las cadenas son iguales. También se puede emplear == en vez de =.
`cadena1 != cadena2`	Verdad si las cadenas no son iguales.
`cadena1 \<cadena2`	Verdad si `cadena1` se ordena lexicográficamente antes de `cadena2`.
`cadena1 \>cadena2`	Verdad si `cadena1` se ordena lexicográficamente después de `cadena2`.

Observa que para poder utilizar los operadores < y > es necesario protegerlos, por ejemplo anteponiéndoles una barra inclinada invertida, de cara a preservar su valor literal. De esta manera, el shell no los va a intentar tratar previamente creyendo que son un direccionamiento de la entrada y de la salida de la orden, respectivamente, y, por tanto, le llegarán intactos a la orden `test`.

Por ejemplo, directamente desde el intérprete de órdenes:

```
$ a="Pepe Lopez"
$ b="Pepe Perez"
$ test -n "$a"       # ¿contiene $a una cadena de longitud > 0?
$ echo $?
0                    # respuesta: sí
$ test -n "$c"       # ¿contiene $c una cadena de longitud > 0?
$ echo $?
1                    # respuesta: no
$ test -z "$c"       # ¿contiene $c una cadena de longitud = 0?
$ echo $?
0                    # respuesta: sí
$ test "$a" \< "$b"  # ¿la cadena de $a es anterior a la de $b?
$ echo $?
0                    # respuesta: sí
$ test "$a" \> "$b"  # ¿la cadena de $a es posterior a la de $b?
```

[3]El orden lexicográfico es el que se utiliza para ordenar caracteres. Normalmente se diferencia entre letras mayúsculas y minúsculas, y además se consideran los números y los signos de puntuación. En los diccionarios se utiliza orden lexicográfico, pero en ellos no se hace diferencia entre mayúsculas y minúsculas.

```
$ echo $?
1                    # respuesta: no
```

- En la expresión se pueden incluir **comprobaciones sobre ficheros**, entre otras:

`-e fichero`	El fichero existe.
`-r fichero`	El fichero existe y tengo permiso de lectura.
`-w fichero`	El fichero existe y tengo permiso de escritura.
`-x fichero`	El fichero existe y tengo permiso de ejecución.
`-f fichero`	El fichero existe y es regular.
`-s fichero`	El fichero existe y es de tamaño mayor a cero.
`-d fichero`	El fichero existe y es un directorio.

Veamos algunos ejemplos de uso desde el intérprete de órdenes:

```
$ ls -l
total 8
drwxrwxr-x. 2 alumno alumno 4096 mar 18 11:24 d1
-rw-rw-r--. 1 alumno alumno    5 mar 18 11:24 f1
-rw-rw-r--. 1 alumno alumno    0 mar 16 00:33 f2
-rw-rw-r--. 1 alumno alumno    0 mar 16 00:33 f3

$ test -e f1          # ¿existe el fichero f1?
$ echo $?             #
0                     # respuesta: sí

$ test -e d1          # ¿existe el fichero d1?
$ echo $?             #
0                     # respuesta: sí

$ test -e f5          # ¿existe el fichero f5?
$ echo $?             #
1                     # respuesta: no

$ test -r f2          # ¿f2 es un fichero con permiso de lectura?
$ echo $?             #
0                     # respuesta: sí

$ chmod a-r f2        # le quitamos permiso lectura a f2
$ ls -l f2
--w--w----. 1 alumno alumno    0 mar 16 00:35 f2

$ test -r f2          # ¿f2 es un fichero con permiso de lectura?
$ echo $?             #
1                     # respuesta: no

$ test -r f2          # ¿f2 es un fichero con permiso de escritura?
$ echo $?             #
0                     # respuesta: sí

$ chmod a-w f2        # le quitamos permiso escritura a f2
$ ls -l f2
```

```
----------. 1 alumno alumno    0 mar 16 00:35 f2

$ test -w f2          # ¿f2 es un fichero con permiso de escritura?
$ echo $?             #
1                     # respuesta: no


$ test -x f2          # ¿f2 es un fichero con permiso de ejecución?
$ echo $?             #
1                     # respuesta: no

$ test -s f1          # ¿f1 es un fichero con tamaño mayor que cero?
$ echo $?
0                     # respuesta: sí

$ test -s f2          # ¿f2 es un fichero con tamaño mayor que cero?
$ echo $?
1                     # respuesta: no

$ test -f f2          # ¿f2 es un fichero regular?
$ echo $?
0                     # respuesta: sí

$ test -f d1          # ¿d1 es un fichero regular ?
$ echo $?
1                     # respuesta: no

$ test -d d1          # ¿d1 es un directorio ?
$ echo $?
0                     # respuesta: sí
```

- Además, se pueden incluir **operadores lógicos** y **paréntesis**:

`-o`	OR
`-a`	AND
`!`	NOT
`\(`	Paréntesis izquierdo
`\)`	Paréntesis derecho

Por ejemplo:

```
$ a=10
$ b=20
$ c=30
$ d=40

$ test \( $a -lt $b \) -a \( $c -lt $d \)    # ¿se cumple que
$ echo $?                                    # (a<b)y(c<d)?
0                                            # respuesta: sí

$ test \( $a -lt $b \) -a \( $c -gt $d \)    # ¿se cumple que
$ echo $?                                    # (a<b)y(c>d)?
```

```
1                                                   # respuesta: no

$ test \( $a -lt $b \) -o \( $c -gt $d \)          # ¿se cumple que
$ echo $?                                           # (a<b)o(c>d)?
0                                                   # respuesta: sí
```

Observamos que, de igual forma que antes habíamos hecho para los operadores < y >, protegemos ahora los paréntesis, por ejemplo anteponiéndoles una barra inclinada invertida.

En la siguiente sección veremos varios ejemplos de uso de esta orden en combinación con diferentes estructuras de control.

9. Estructuras de control

9.1. Condiciones: `if` y `case`

En un guion shell se pueden introducir condiciones, de forma que determinadas órdenes sólo se ejecuten cuando éstas se cumplen. Para ello se utilizan las órdenes `if` y `case`.

9.1.1. Orden `if`

La sintaxis de la orden `if` es la siguiente:

```
if  <orden_a_evaluar_if>
then
      <lista_ordenes_if>

elif  <orden_a_evaluar_elif_1>
then
      <lista_ordenes_elif_1>
      # (el bloque elif y sus órdenes son opcionales)

elif  <orden_a_evaluar_elif_2>
then
      <lista_ordenes_elif_2>
      # (el bloque elif y sus órdenes son opcionales)
...
else
      <lista_ordenes_else>
      # (el bloque else y sus órdenes son opcionales)
fi
```

El funcionamiento será el siguiente: se ejecuta la orden que acompaña al `if`, `orden_a_evaluar_if`. En caso de que el código de retorno devuelto por esa orden sea 0 (verdadero), se ejecutará la lista de órdenes `lista_ordenes_if`. En caso contrario, se aplica el mismo método al primer `elif`, luego, si es necesario, al segundo,... Si finalmente ninguna orden evaluada ha devuelto 0, se ejecutará la lista de órdenes `lista_ordenes_else`. Obviamente, los bloques correspondientes a los `elif` y al `else` son opcionales.

Un ejemplo del funcionamiento de la orden `if` sería:

```
#!/bin/bash
if grep -q main prac.c
then
        echo "Encontrada la palabra clave main en prac.c"
else
        echo "No encontrada la palabra clave main en prac.c"
fi
```

En este ejemplo se usa la orden `grep` para buscar la palabra `main` en el fichero `prac.c`. La instrucción `if` utiliza el código retornado por la orden `grep` para saber si la búsqueda ha tenido éxito o no. La opción `-q` de la orden `grep` se utiliza para que su ejecución no envíe nada a su salida estándar, es decir, que no muestre nada en pantalla.

De igual forma, podemos anteponer el caracter «!» a la orden a evaluar si lo que nos interesa es considerar lo opuesto del código retornado por dicha orden. Es decir, un ejemplo equivalente al anterior podría ser:

```
#!/bin/bash
if ! grep -q main prac.c
then
        echo "No Encontrada la palabra clave main en prac.c"
else
        echo "Encontrada la palabra clave main en prac.c"
fi
```

En el siguiente ejemplo, si el script no recibe exactamente dos argumentos al ejecutarlo, muestra un mensaje de error (redireccionando la salida de la orden `echo` al descriptor 2, conocido como *salida estándar de error*, tal como se vió en el boletín anterior) y finaliza su ejecución (orden `exit`) devolviendo el código de salida 1:

```
#!/bin/bash
if test $# -ne 2
then
     echo "se necesitan dos argumentos" >&2
     exit 1
fi

# resto de código del script
```

El siguiente script comprueba el valor del primer parámetro. Si es un fichero regular (`-f`) visualiza su contenido; si no lo es, entonces comprueba si es un directorio y, si es así, lista su contenido. En otro caso, muestra un mensaje de error.

```
#!/bin/bash
if test -f "$1"
then
     cat $1
elif test -d "$1"
then
     ls -l "$1"
else
     echo "$1 no es fichero ni directorio" >&2
     exit 1
fi
```

Nota: En el anterior ejemplo se usan las comillas dobles para manejar el primer parámetro, `"$1"`, por si en su contenido hay algún espacio en blanco.

En el siguiente ejemplo, el script comprueba si las dos cadenas de caracteres que recibe como parámetros son iguales. En caso contrario, indica qué orden lexicográfico existe entre ellas:

```
#!/bin/bash
cadena1="$1"
cadena2="$2"
```

```
if test "$cadena1" == "$cadena2"
then
        echo "Las cadenas $cadena1 y $cadena2 son iguales"
else
        echo "Las cadenas $cadena1 y $cadena2 son diferentes"

        if test "$cadena1" \< "$cadena2"
        then
                echo "La cadena $cadena1 es anterior a $cadena2"
        else
                echo "La cadena $cadena1 es posterior a $cadena2"
        fi
fi
```

Nota: En el ejemplo anterior, se muestra cómo, de nuevo, es conveniente utilizar las comillas dobles para manejar variables que contienen cadenas de caracteres, sobretodo si éstas pueden tener espacios en blanco. Además, si no se usaran las comillas y una de las variables contuviese una cadena vacía, se produciría un error de sintaxis a la hora de ejecutar la orden `test`.

En el siguiente ejemplo el script recibe dos números enteros como parámetros, comprueba que ambos sean valores menores que 100 y, en tal caso, calcula su suma y muestra el resultado. En caso contrario, muestra un mensaje de error y, mediante la orden `exit`, finaliza con código de salida errónea igual a 1.

```
#! /bin/bash
if test \( $1 -ge 100 \) -o \( $2 -ge 100 \)
then
    echo "Los parametros deben ser números menores de 100" >&2
    echo "USO: $0 numero1 numero2" >&2
    exit 1
fi
let suma=$1+$2
echo "La suma de $1 y $2 es: $suma"
```

9.1.2. Orden `case`

La sintaxis de la orden `case` es la siguiente:

```
case $variable in
    patrón_1)
             <lista_ordenes_1>
             ;;
    patrón_2|patrón_3)
             <lista_ordenes_2_3>
             ;;
    patrón_4)
             <lista_ordenes_4>
             ;;
    ...

    patrón_n)
             <lista_ordenes_n>
             ;;

    *)
```

```
                <lista_ordenes_por_defecto>
                ;;
    esac
```

Se ejecutará aquella lista de órdenes del primer caso en el que el valor de la variable coincida con alguno de los valores considerados en los patrones: `patrón_1`, `patrón_2`, `patrón_3`, `patrón_4`, ..., `patrón_n`; pudiendo utilizar comodines (ver boletín 1) al especificar dichos patrones. Cada una de estas listas de órdenes debe finalizar con el delimitador doble punto y coma, `;;`.

Como se puede observar, también se pueden indicar varios posibles patrones dentro de un mismo caso (cuando se indica `patrón_2 | patrón_3`).

Finalmente, para el caso de que la variable no coincida con ningún patrón de los considerados, podemos indicar un caso por defecto. Este caso por defecto se colocará como último caso de ejecución, indicando que es válido para cualquier valor de la variable, mediante el uso de `*` como comodín.

Un ejemplo de su funcionamiento podría ser este guion shell, llamado `diasanto.sh`, que recibe como argumento el nombre de una persona y le indica qué día del año es su santo en un par de casos (San José para Pepe, Pepa, Jose y Josefa; y San Francisco para Paco, Paca, Francisco y Francisca):

```
#!/bin/bash
echo "nombre: $1"
case "$1" in
        Pac[oa]|Francisc[ao])
           echo "Tu santo es el 4 de octubre: San Francisco"
           ;;
        Pep[ea]|Jose|Josefa)
           echo "Tu santo es el 19 de marzo: San José"
           ;;
        *)
         echo "Lo siento, no sé qué día es tu santo"
         ;;
esac
```

9.2. Bucles condicionales: `while`

También es posible ejecutar bloques de órdenes de forma iterativa dependiendo del estado de salida de una orden evaluada, como se hacía en la instrucción `if`, utilizando un bucle `while` con la siguiente sintaxis:

```
while <orden_a_evaluar>   # Mientras el código de retorno sea 0
do
    <conjunto_instrucciones>
done
```

Un ejemplo del funcionamiento sería:

```
# Muestra todos los parámetros
while test ! -z "$1"
do
      echo "Parámetro: $1"
      shift
done
```

En este ejemplo, la orden a evaluar como condición de continuación del bucle sería un `test`, donde se comprueba si el primer parámetro del script, `$1`, es distinto de la cadena vacía, `!-z`. En tal caso, la orden `test` devuelve un 0, con lo que la instrucción `while` considerará que se cumple la condición para continuar la ejecución del bucle.

9.3. Bucles incondicionales: `for`

Con la orden `for` se ejecutan bloques de órdenes, permitiendo que en las sucesivas iteraciones una determinada variable vaya tomando como valor cada uno de los especificados en una lista. La sintaxis es la siguiente:

```
for variable in <lista_de_valores>
do
    <conjunto_instrucciones>
done
```

Por ejemplo:

```
echo "Los 3 números premiados en el sorteo de Navidad han sido:"
for i in 23454 12332 09884
do
     echo $i
done
```

Aunque la lista de valores del `for` puede ser arbitraria (incluyendo no sólo números, sino cualquier otro tipo de cadena o expresión), a menudo lo que queremos es generar secuencias de valores numéricos al estilo de la instrucción *for* de los lenguajes de programación convencionales. En este caso, la orden `seq`, combinada con el mecanismo de sustitución de órdenes (véase el apartado 5) puede resultarnos de utilidad. La orden `seq` genera una secuencia de números, comenzando en el valor que se le pasa como primer parámetro, llegando, sin superarlo al valor que se le pasa como tercer parámetro, con un incremento entre los números igual al segundo parámetro que recibe. Así, por ejemplo, si ejecutamos el guion `multiplos.sh`:

```
#!bin/bash
echo "Los múltiplos de $1 menores o igual que $2 son:"
for i in $(seq 0 $1 $2)
do
    echo $i
done
```

podríamos obtener esto:

```
$ bash multiplos.sh 3 20
Los múltiplos de 3 menores o igual que 20 son:
0
3
6
9
12
15
18
```

9.4. Ruptura de bucles: `break y continue`

Las órdenes `break` y `continue` sirven para interrumpir la ejecución secuencial del cuerpo de un bucle.

- La orden `break` transfiere el control a la orden que hay tras la instrucción `done`, haciendo que el bucle termine antes de tiempo.
- La orden `continue` transfiere el control a la instrucción `done`, con lo que la ejecución del bucle continúa en la siguiente iteración.

En ambos casos, las órdenes del cuerpo del bucle siguientes a estas sentencias no se ejecutan. Lo normal es que formen parte de una sentencia condicional.

Un ejemplo de `break` sería este script:

```
#!/bin/bash
# Muestra todos los parámetros.
# Si uno es la cadena "fin" entonces acaba

while test $# -gt 0
do
      if test "$1" = "fin"
      then
            break
      fi
      echo "Parámetro: $1"
      shift
done
```

Un ejemplo de uso de `continue` sería el siguiente script:

```
#!/bin/bash
# Muestra todos los parámetros
# excepto los que son la cadena "salta"

while test $# -gt 0
do
      if test "$1" = "salta"
      then
          shift
          continue
      fi
      echo "Parámetro: $1"
      shift
done
```

10. Ejercicios

1. Crea un shell script llamado `num_arg.sh`, que muestre los argumentos con los que ha sido llamado. Además, este guion debe devolver como código de salida un 0 (`exit 0`) si se ha pasado algún argumento y 1 (`exit 1`) en caso contrario. Ejemplo de uso:

```
$ bash num_arg.sh hola amigo Paco López
El guion shell num_arg.sh ha recibido 4 argumentos:
 hola
 amigo
 Paco
 Lopez
$ echo $?
0
```

```
$ bash num_arg.sh hola amigo "José Antonio González"
El guion shell num_arg.sh ha recibido 3 argumentos
 hola
 amigo
 José Antonio González
$ echo $?
0

$ bash num_arg.sh
El guion shell num_arg no ha recibido ningún argumento
$ echo $?
1
```

2. Escribe un guion shell llamado `opera.sh` que muestre en pantalla el resultado de sumar, restar, multiplicar y dividir los dos números que se le pasan como argumentos

   Ejemplo de uso:

```
$ bash opera.sh 34 20
La suma de 34 más 20 es 54
La resta de 34 menos 20 es 14
El producto de 34 por 20 es 680
La división entera de 34 entre 20 es 1
```

3. Crea un script llamado `doble.sh` que muestre el valor doble del número que se le pase como primer parámetro. Ejemplo de uso:

```
$ bash doble.sh 35
El doble de 35 es 70
```

4. Modifica el shell script llamado `doble.sh` visto anteriormente, para que únicamente calcule el doble si el valor del número introducido como argumento está entre 100 y 200, ambos inclusive. En caso contrario, deberá mostrar el oportuno mensaje de aviso al usuario y acabar, devolviendo un 1 como código de salida. El nuevo script se llamará `doble_02.sh`. Ejemplo de uso:

```
$ bash doble_02.sh 111
El doble de 111 es 222
$ echo $?
0

$ bash doble_02.sh 35
El argumento debe ser un número entre 100 y 200
$ echo $?
1

$ bash doble_02.sh 235
El argumento debe ser un número entre 100 y 200
$ echo $?
1
```

5. A partir del guion llamado `doble.sh`, visto anteriormente, crea `doble_interactivo.sh`, que no recoja ningún parámetro, sino que pida interactivamente al usuario un número, calcule

su doble y lo muestre. A continuación, debe preguntar al usuario si desea calcular otro doble, solicitándole que responda `S` o `N`. En caso de que el usuario responda `S`, se repetirá todo el proceso. Por ejemplo:

```
$ bash doble_interactivo.sh
Introduce un número para calcular el doble:
89
El doble de 89 es 178

¿Deseas calcular otro doble (S/N)?
S
Introduce un número para calcular el doble:
9
El doble de 9 es 18

¿Deseas calcular otro doble (S/N)?
N
```

NOTA: Utiliza la orden `read` para almacenar en una variable el valor que se introduce desde la entrada estándar, tal como lo hace el siguiente guion llamado `saludo_interactivo.sh`:

```
#!/bin/bash
echo "¿Cómo te llamas?"
read nombre
echo "Buenos días, $nombre"
```

6. Escribe un guion llamado `tratafichero.sh` que reciba como parámetro el nombre de un fichero. Si dicho nombre termina en `.txt`, se debe mostrar el contenido del fichero, si el nombre termina en `.sh` , se debe ejecutar (se supone que será otro guion shell). En otro caso, simplemente se muestra un mensaje indicando que no sabe tratar el fichero indicado como argumento. Ejemplo de uso:

```
$ bash tratafichero.sh saludo_interactivo.sh
Ejecutando el script saludo_interactivo.sh:
¿Cómo te llamas?
Pepe
Buenos dias, Pepe

$ bash tratafichero.sh saludo.txt
Contenido del fichero saludo.txt:
Buenos dias a todos !
Bienvenidos !
```

7. Escribe un guion llamado `reloj.sh` que muestre en pantalla un sencillo reloj digital que va actualizándose cada segundo, hasta ser finalizado con `Ctrl+C`.

   NOTA: Puedes usar la orden `date` para obtener la fecha y hora del sistema (incluyendo los segundos).

8. Crea un shell script llamado `tabla.sh` que muestra la tabla de multiplicar del número que se le pasa como argumento. Un ejemplo de uso sería:

```
$ bash tabla.sh 5
```

```
TABLA DE MULTIPLICAR DEL 5
==========================
5 * 1 = 5
5 * 2 = 10
...
5 * 9 = 45
5 * 10 = 50
```

9. Mejora el shell script `tabla.sh` para que verifique que el argumento es un número mayor que 0 y menor que 10. En caso contrario, debe mostrar un mensaje de error y finalizar sin hacer nada más. El nuevo script se llamará `tabla_02.sh`.

10. Amplía la funcionalidad del guion llamado `diasanto.sh` visto anteriormente (recibe como argumento el nombre de una persona y le indica qué día del año es su santo) para que también contemple apropiadamente a estos nombres: Pepita, Pepito, Antonio y Antonia. El nuevo script se llamará `diasanto_02.sh`.

11. Crea un guion shell llamado `cuentalineas.sh` que muestra el número de líneas de todos los ficheros regulares cuyos nombres se le pasan como parámetros. Si alguno de los parámetros no corresponde con un fichero regular simplemente lo debe ignorar. Ejemplo de uso:

```
$bash cuentalineas.sh saludo.txt dir4 sumanumeros.sh fff 34 llamar.sh
  Fichero: saludo.txt.  Número de líneas: 3
  Fichero: sumanumeros.sh.  Número de líneas: 7
  Fichero: llamar.sh.  Número de líneas: 6
```

11. Bibliografía

- Página de manual del intérprete de órdenes Bash (`man bash`).
- *El libro de UNIX*,
  S. M. Sarwar *et alt*, ISBN: 8478290605. Addison-Wesley, 2005.
- *Linux: Domine la administración del sistema*, 2ª edición.
  Sébastien Rohaut. ISBN 9782746073425. Eni, 2012.
- *Shell & Utilities: Detailed Toc*. The Open Group Base Specifications.
  http://pubs.opengroup.org/onlinepubs/9699919799/utilities/contents.html.
- *Unix shell patterns*
  (http://wiki.c2.com/?UnixShellPatterns), J. Coplien *et alt*.
- *Programación en BASH - COMO de introducción*
  http://es.tldp.org/COMO-INSFLUG/COMOs/Bash-Prog-Intro-COMO/,
  Mike G. (traducido por Gabriel Rodríguez).
- *Shell & Utilities: Detailed Toc*
  http://pubs.opengroup.org/onlinepubs/9699919799/utilities/contents.html,
  The Open Group Base Specifications.
- *Espacio Linux. Portal y comunidad GNU*
  http://www.espaciolinux.com/.
- *Linux Shell Scripting Tutorial (LSST) v2.0*
  https://bash.cyberciti.biz/guide/Main_Page,
  Vivek Gite *et al*.