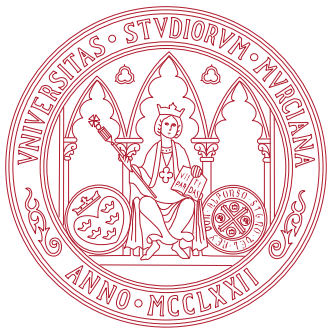

Universidad de Murcia

Facultad de Informática

Departamento de Ingeniería y Tecnología de Computadores

Área de Arquitectura y Tecnología de Computadores

PRÁCTICAS DE I.S.O.

2º de Grado en Ingeniería Informática

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

Curso 2017/2018

Índice

1. Concepto de guión shell

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*.

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 script, incluso de manera automática a una hora preestablecida, sin intervención humana.

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`. El resto del contenido de este documento está centrado en este intérprete de órdenes.

2. Funcionamiento de un guión shell

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

```
clear
date
```

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 guión de esta manera:

```
$ bash limpiafecha
```

Este nuevo intérprete de órdenes que estamos invocando no se ejecutará en modo interactivo, sino que irá interpretando el contenido del guión. 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 guión shell.
2. El proceso padre se queda a la espera mientras no termine el nuevo proceso hijo de ejecutar el guión.
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 guión shell), el proceso padre reanuda su ejecución.

En este ejemplo introductorio, para la ejecución de las dos líneas del interior del guión se han creados sendos procesos nietos. Como veremos más adelante, esto no ocurre siempre de esta manera, pues encontraremos muchos ejemplos de líneas de un guión que son interpretadas directamente por el proceso hijo, bien porque son líneas de código que no llaman a ninguna orden del bash, o bien porque invocan únicamente a órdenes internas del bash que, por tanto, pueden ser ejecutadas por el proceso hijo directamente, sin necesidad de crear un proceso nieto.

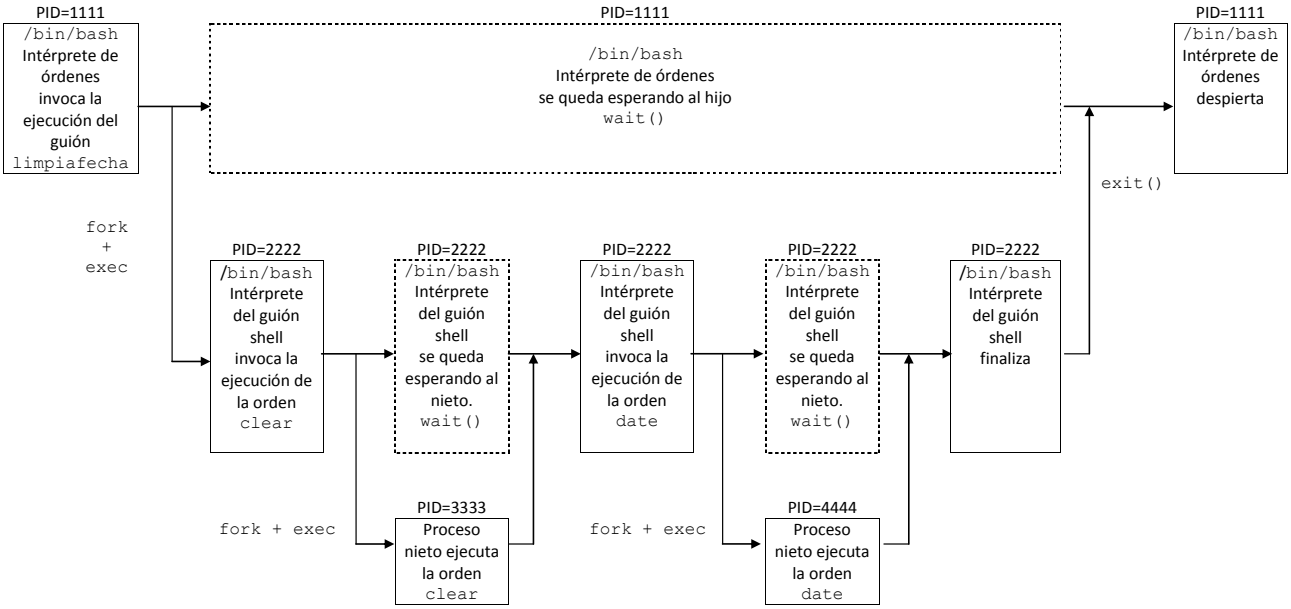

Figura 1: Esquema de funcionamiento de un guión shell para el ejemplo `limpiafecha`.

2.1. Ejercicios

1. Vamos a ver otra manera de ejecutar un guión shell directamente. Utiliza un editor de texto para escribir el primer ejemplo de guión shell mostrado anteriormente y ponerle `limpiafecha` como nombre:

```
#!/bin/bash -u
clear
date
```

Ahora, para poder probarlo, abre una consola de texto y desde la línea de órdenes:

- Dale permisos de ejecución:

```
$ chmod 744 limpiafecha
```

- Ejecútalo:

```
$ ./limpiafecha
```

Un guión shell puede incluir comentarios. Para ello se utiliza el carácter # al inicio del texto que constituye el comentario. Además, en un guión shell se puede especificar el shell concreto con el que se debe interpretar o ejecutar, indicándolo en la primera línea de la siguiente forma (el carácter # no es un comentario en este caso):

```
#!/bin/bash -u
```

Se ha incluido la opción `-u` que, como veremos más adelante, resulta de gran utilidad a la hora de depurar la ejecución del guión.

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:

```
echo $variable
```

Un ejemplo del uso de las variables en la línea de órdenes directamente, sin llegar a usar un script, sería:

```
$  mils="ls -l"                              # Se crea una nueva variable

$  mils                                      # No hace nada porque busca
   mils: command not found                   # el ejecutable mils que no
                                             # existe

$  $mils                                     # Ejecuta la orden "ls -l"
   drwxrwxr-x 2 javiercm javiercm 4096 Nov  8 10:33 dir2
   -rw-rw-r-- 1 javiercm javiercm   40 Oct 19 20:05 f1

$  echo $mils                                # Muestra el contenido de la
   ls -l                                     # variable mils,
                                             # es decir, "ls -l"
```

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

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 -l"`. Tras ello, el intérprete de órdenes ejecuta la orden resultante: `ls -l`.

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 -l"`. A continuación, el intérprete de órdenes ejecuta la orden resultante, `echo "ls -l"`, que simplemente mostrará la cadena `"ls -l"` en pantalla.

Las siguientes variables predefinidas son muy útiles al programar los guiones shell:

- `$!`: identificador de proceso (PID) de la última orden ejecutada en segundo plano.
- `$$`: el PID del shell actual (comúnmente utilizado para crear nombres de ficheros únicos).

3.2. Parámetros

Como cualquier programa, un guión 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 guión shell que muestra los cuatro primeros parámetros recibidos:

```
#!/bin/bash -u
echo "El nombre del programa es $0"
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 "El número total de prámetros 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 `$3`, etc. De igual forma, el valor de `$#` se decrementa en uno con cada ejecución de un `shift`.

Un ejemplo sencillo de un guión shell que muestra el nombre del ejecutable, el número total de parámetros, todos los parámetros y los cuatro primeros parámetros es el siguiente:

```
#!/bin/bash -u
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 "El número total de parámetros, tras dos shifts, es $#"
```

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:

```
#!/bin/bash -u
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}"
echo "El número total de parámetros, sin hacer shifts, es $#"
```

3.3. Código de salida de un guión 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.

En un guión 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.

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.

3.4. Ejercicios

1. Escribe y prueba los siguientes guiones:

   `llamar.sh`:

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

   `num.sh`:

   ```
   #!/bin/bash -u
   echo "Inicio del guion $0"
   echo "El PID de $0 es $$"
   echo "Fin del guion  $0"
   exit 1
   ```

   El proceso correspondiente al programa `llamar.sh` muestra su número PID y después llama a un programa llamado `num.sh`. A su vez, `num.sh` también muestra su PID. Finalmente, cuando el proceso `num.sh` termina, devuelve, usando `exit`, el valor 1 al proceso que lo ha llamado. La ejecución continúa por el proceso `llamar.sh` que muestra el valor devuelto por `num.sh` (este valor devuelto por `exit` ha quedado recogido en la variable `$?` al ser el valor devuelto por la última orden ejecutada).

2. Haz un guión shell llamado `priult` que devuelva los argumentos primero y último que se le han pasado. Si se llama con:

   ```
   priult hola qué tal estás
   ```

debe responder:

```
El primer argumento es hola
El último argumento es estás
```

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.

Los metacaracteres `* $ | & ; ( ) { } < >` `espacio` y `tab` tienen un significado especial para el shell y deben ser protegidos o entrecomillados si quieren representarse a sí mismos. Hay 3 mecanismos de protección: el carácter de escape, las comillas simples y las comillas dobles[1].

- Una barra inclinada inversa (o invertida) no entrecomillada (\) es el carácter de escape (no confundir con el código ASCII cuyo valor es 27 en decimal), el cual preserva el valor literal del siguiente carácter que lo acompaña, con la excepción de <nueva-línea>. Si la barra invertida no está entre comillas, 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 -u
usuario="Pepe"
sueldo=2500
echo 'PRUEBA PRIMERA: El usuario $usuario gana $sueldo $ al mes'
echo "PRUEBA SEGUNDA: El usuario $usuario gana $sueldo \$ al mes"
```

mostraría la siguiente salida:

```
$ ./comillas.sh
PRUEBA PRIMERA: El usuario $usuario gana $sueldo $ al mes
PRUEBA SEGUNDA: 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`. Con estas comillas, para poder utilizar el carácter `$` literalmente, sin significado especial, hemos tenido que anteponerle la barra de escape.

[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.

Por otro lado, encerrar una cadena entre paréntesis precedida de un signo $, `$(cadena)`, supone forzar 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`:

```
#!/bin/bash -u

#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"
usuario=$(whoami)

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

mostraría la siguiente salida si se ejecuta:

```
$./saludo.sh
Hola javier. La fecha de hoy es: mar sep 22 19:20:50 CEST 2015
```

Un programa equivalente al anterior sería:

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

Veamos un conjunto de ejemplos que combinan la orden `echo` con conceptos descritos en esta sección:

- 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 *
  ```

- 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 *
  ```

- 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, a la orden `echo` le llegan cinco cadenas de caracteres como parámetros: `ls`, `-l`, `f1`, `f2` y `f3`:

[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. 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.

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

- 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 es lo que muestra la orden `echo` en pantalla:

```
$ echo "$(ls -l *)"
total 8
-rw-rw-r-- 1 javiercm javiercm  5 Sep 10  2013 f1
-rw-rw-r-- 1 javiercm javiercm 17 Sep 10  2013 f2
-rw-rw-r-- 1 javiercm javiercm  0 Sep 10  2013 f3
```

- 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 separadadores por espacios en blanco cuando los muestra en pantalla:

```
$ echo $(ls -l *)
total 8  -rw-rw-r-- 1 javiercm javiercm 5 Sep 10  2013 f1   -rw-
rw-r--   1 javiercm javiercm 17 Sep 10  2013 f2    -rw-rw-r-- 1
javiercm  javiercm  0 Sep 10  2013 f3
```

- Por último, en este ejemplo, al estar la cadena con comillas simples no se invoca ninguna orden antes de que actúe `echo` que, por tanto, muestra en pantalla la cadena de caracteres que recibe como parámetro, sin más:

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

4.1. Ejercicios

1. Escribe un guión shell que, tomando el primer argumento que se le pasa como el nombre de un usuario, nos devuelva cuántas veces está conectada esa persona.

   NOTA: Utiliza la orden `who` para conocer todos los usuarios conectados, `grep` para buscar el usuario que nos interesa, y `wc` para contar el número de líneas.

   Solución:

```
#!/bin/bash -u
veces=$(who | grep -w ^$1 | wc -l)
echo "$1 está conectado $veces veces"
```

5. Evaluación aritmética

El shell permite que se evalúen expresiones aritméticas, según ciertas reglas (La tabla 1 muestra algunos operadores, en orden de precedencia decreciente y agrupando a aquellos de igual precedencia):

- 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.

-, +	Menos y más unarios
**	Exponenciación
*, /, %	Multiplicación, división, resto
+, -	Adición, sustracción

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

- 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 parámetro 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.

Veamos dos maneras de realizar operaciones aritméticas en `bash`:

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

```
$ 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
```

2. Mediante `$((expresión))`. Varios ejemplos de su uso serían:

```
$ echo "El resultado de la suma es: $((6+7))"
El resultado de la suma es: 13

$ echo "El resto de la división es: $((7%5))"
El resto de la división es: 2
```

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

5.1. Ejercicios

1. Escribe un guión shell que solicite al usuario que introduzca dos números (no serán argumentos del guión sino variables) e imprima su suma.

   NOTA[3]: utiliza la orden `read` para leer desde teclado estos números.

   Solución:

[3]Cuando en la resolución de un ejercicio se precise usar una orden no vista hasta el momento, se indicará mediante una nota en el enunciado de dicho ejercicio.

```
#!/bin/bash -u
echo "Introduzca un número:"
read numero1
echo "Introduzca otro número:"
read numero2
let respuesta=$numero1+$numero2
echo "$numero1 + $numero2 = $respuesta"
```

2. Crea un shell script llamado `doble` que pida un número por teclado y calcule su doble. Ejemplo:

```
Introduzca un número para calcular el doble: 89
El doble de 89 es 178
```

6. 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 puede ser una de las dos que se muestran a continuación:

```
test expresión
```

o bien:

```
[ expresión ]
```

SINTAXIS. Orden `test`: uso de espacios en blanco.
Al utilizar la orden `test`, es necesario incluir espacios en blanco entre la expresión y los corchetes, así como 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
$ echo $?
0

$ let a=50-40
$ let b=22-10
$ test $a -eq $b
$ echo $?
1
```

Es importante destacar que en las comparaciones con números, si utilizamos una variable que no está definida, saldrá un mensaje de error. El siguiente ejemplo, al no estar la variable `e` definida, mostrará un mensaje de error indicando que se ha encontrado un operador inesperado:

```
$ let a=50-40
$ let b=20-10
$ test $a -eq $c
-bash: test: 10: unary operator expected
$ echo $?
2
```

- Para **caracteres alfabéticos** o **cadenas** teniendo en cuenta el orden lexicográfico[4]:

`-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 clasifica lexicográficamente después de `cadena2`.

- En la expresión se pueden incluir **operaciones con 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.

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

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

A continuación veremos distintos ejemplos de utilización de la orden `test` junto a la orden `if` (la sintaxis de la orden `if` se verá con detalle en la siguiente sección), con el fin de aclarar su funcionamiento. Uno de los usos más comunes de la variable `$#` es para validar el número de argumentos necesarios en un programa shell. 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:

[4]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.

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

# resto de código del script
```

El siguiente ejemplo comprueba el valor del primer parámetro. Si es un fichero regular (`-f`) se visualiza su contenido; sino, entonces se comprueba si es un directorio y, si es así, se lista su contenido. En otro caso, se muestra un mensaje de error.

```
if test -f "$1"       # ¿ es un fichero regular ?
then
     cat $1
elif test -d "$1"     # ¿ es un directorio ?
then
     ls -l "$1"
else                  # no es ni fichero ni directorio
     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.

Comparando dos cadenas:

```
#! /bin/bash -u
cad1=hola
cad2=Hola

if [ "$cad1" != "$cad2" ]
then
       echo "cad1 ($cad1) no es igual a cad2 ($cad2)"
       echo "'$cad1'"
fi

if [ "$cad1" = "$cad2" ]
then
       echo "cad1 ($cad1) es igual a cad2 ($cad2)"
fi
```

Nota: En el ejemplo anterior, se muestra cómo, de nuevo, es conveniente utilizar las comillas dobles para manejar variables que contengan cadenas de caracteres. Si no se usan comillas y ocurre que `$cad1` y/o `$cad2` son cadenas vacías, entonces aparecería un error sintáctico en la instrucción `if`.

El siguiente ejemplo 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 -u
if [ \( $1 -ge 100 \) -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"
```

6.1. Ejercicios

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

```
$ ./num_arg hola mi amigo
El guión shell num_arg ha recibido 3 argumentos
$ echo $?
0

$ ./num_arg
El guión shell num_arg no ha recibido ningún argumento
$ echo $?
1
```

2. Mejora el shell script llamado doble (visto anteriormente: pide un número por teclado y calcula su doble), para que únicamente calcule el doble si el valor del número introducido está entre 100 y 200. En caso contrario, deberá mostrar el oportuno mensaje de aviso al usuario por la salida estándar de error y acabar, devolviendo un 1 como código de salida.

7. Estructuras de control

7.1. Condiciones: `if y case`

En un guión 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`. 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:

```
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.

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

```
case $variable in
    valor_1)                <lista_ordenes_1>
        ;;
    valor_2|valor_3)        <lista_ordenes_2_3>
        ;;
    valor_4)                <lista_ordenes_4>
        ;;
    ...

    valor_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: `valor_1`, `valor_2`, `valor_3`, `valor_4`, ..., `valor_n`; pudiendo utilizar comodines (ver boletín 1) al especificar dichos valores. 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 valores dentro de un mismo caso (cuando se indica `valor_2 | valor_3`).

Finalmente, para el caso de que la variable no coincida con ningún valor concreto 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:

```
case $resultado in
     1)   echo "El resultado es 1"
          ;;
     2)   echo "El resultado es 2"
          ;;
     3|4)   echo "El resultado es 3 o 4"
```

```
        ;;
    *)   echo "El resultado no es un entero entre 1 y 4"
        ;;
esac
```

7.2. Bucles condicionales: `while`

También es posible ejecutar bloques de órdenes de forma iterativa dependiendo de una condición evaluada como en la instrucción `if`, utilizando un bucle `while` con la siguiente sintaxis:

```
while    «orden_a_evaluar»   # Mientras el código de retorno sea 0
do
    ...
done
```

Un ejemplo del funcionamiento sería:

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

En este ejemplo, la orden a evaluar como condición de continuación del bucle sería un `test` (escrita en el formato de corchetes), 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.

7.3. Bucles incondicionales: `for`

Con la orden `for` se ejecutan bloques de órdenes, permitiendo que en cada iteración una determinada variable tome un valor distinto. La sintaxis es la siguiente:

```
for var in lista
do
    uso de $var
done
```

Por ejemplo:

```
for i in 10 30 70
do
     echo Mi número favorito es $i # toma los valores 10, 30  y 70
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 4) puede resultarnos de utilidad. Por ejemplo:

```
for i in $(seq 0 5 25)
do
     # uso de $i que toma los valores 0, 5, 10, 15, 20 y 25
done
```

7.4. 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 sigue a `done`, haciendo que el bucle termine antes de tiempo. La orden `continue`, por el contrario, transfiere el control a `done`, haciendo que se evalúe de nuevo la condición, es decir, 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 uso de `break` sería:

```
# Muestra todos los parámetros. Si uno es una "f" finaliza

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

Un ejemplo de uso de `continue` sería:

```
# Muestra todos los parámetros, si uno de ellos es una "f"
# se lo salta y continúa el bucle

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

7.5. Ejercicios

1. Mejora el guión llamado `num_arg` (visto anteriormente: devuelve el número de argumentos con el que ha sido llamado) de forma que muestre una lista de todos los argumentos pasados o bien que indique que no tiene argumentos. Por ejemplo:

   ```
   $ ./num_arg hola mi amigo
   El guión shell num_arg ha recibido 3 argumentos
   argumento 1: hola
   argumento 2: mi
   argumento 3: amigo
   ```

2. Mejora el guión llamado `doble` (visto anteriormente: pide un número por teclado y calcula su doble) para que antes de terminar pregunte si deseamos calcular otro doble, en cuyo caso no terminará. Por ejemplo:

```
$ .\doble
Introduzca un número para calcular el doble:
89
El doble de 89 es 178

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

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

3. Escribe un guión que evalúa la extensión de un fichero cuyo nombre recibe como primer argumento. Si ésta se corresponde con "txt", copia el fichero al directorio `~/copias`. Si es otra la extensión, no hace nada y presenta un mensaje.

   Solución:

```
case $1 in
*.txt)
       cp $1 ~/copias
       ;;
*.doc | *.bak)
       # Tan sólo como ejemplo de otras extensiones
       ;;
*)
       echo "$1 tiene extensión desconocida"
       ;;
esac
```

4. Escribe un guión que cambie el sufijo de todos los archivos `*.tex` a `*.latex` del directorio actual.

   NOTA: utiliza la orden `basename` para quitar el sufijo del nombre de un fichero. Esta orden recibe como primer parámetro el nombre de un fichero (con ruta absoluta o relativa) y como segundo parámetro (opcional) un cadena de caracteres. Al ejecutarla, muestra en su salida estándar el nombre del fichero sin la ruta y sin el sufijo especificado como segundo parámetro. Por ejemplo:

```
$ basename datos.txt
datos.txt

$ basename /home/alumno/datos.txt
datos.txt

$ basename datos.txt .txt
datos
```

   Solución:

```
#!/bin/bash -u
for nombre in *.tex
do
    #quita sufijo "tex" de "nombre"
```

```
    nuevo_nombre=$(basename $nombre tex)

    #añade sufijo "latex" a "nuevo_nombre"
     nuevo_nombre="$nuevo_nombre""latex"

     #renombra fichero
     mv $f $nuevo_nombre
done
```

5. Escribe un guión que muestre un sencillo reloj que va actualizándose cada segundo, hasta ser matado con Ctrl-C.

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

   Solución:

```
#!/bin/bash -u
while true
do
        clear
        echo "==========="
        date +"%r"
        echo "==========="
        sleep 1
done
```

6. Crea un shell script llamado `tabla` que, a partir de un número que se le pasará como argumento, obtenga la tabla de multiplicar de ese número. Si se llama con:

```
tabla 5
```

   debe responder:

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

   Mejora el shell script para que se verifique que sólo se le ha pasado un argumento y que éste es un número entre 0 y 10.

7. Escribe un guión shell para gestionar un menú de opciones que permita al usuario elegir entre ver el directorio actual, copiar ficheros, editar ficheros o imprimir ficheros.

   NOTA: utiliza la orden `read` para leer desde teclado la opción elegida por el usuario.

   NOTA: utiliza la orden `lpr` para imprimir un fichero.

   Solución:

```
while true
do
    clear
```

```
        echo "
              Ver directorio actual..........[1]
              Copiar ficheros................[2]
              Editar ficheros................[3]
              Imprimir fichero...............[4]
              Salir del menú.................[5]"
        read i
        case $i in
        1)    ls
              ;;
        2)    echo "Introduzca [desde]"
              read x
              echo "Introduzca [hasta]"
              read y
              cp $x $y
              ;;
        3)    echo "¿Nombre de fichero a editar?"
              read x
              vi $x
              ;;
        4)    echo "¿Nombre de fichero a imprimir?"
              read x
              lpr $x
              ;;
        5)    clear
              break
              ;;
        esac
        echo "Pulse INTRO para continuar"
        read x

done
```

8. 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.*