Introducción a BASH

Francisco Alonso Sarría

Índice

1 Introducción

Podemos considerar a la SHELL como un entorno de trabajo que permite al usuario lanzar programas mediante órdenes que suelen formarse con el nombre de un programa y un conjunto de opciones o parámetros.

Por ejemplo la orden:

```
ls -l *.tif
```

Producirá un listado (orden `ls`) de todos los ficheros con extensión tiff (parámetro `*.tif`). La opción `-l` establece que será un listado largo, incluyendo diversas características de los ficheros.

Una SHELL incluye, además de la posibilidad de ejecutar programas, una serie de comandos básicos para el manejo de los archivos y directorios del ordenador, así como herramientas diversas para procesar la información presente en los archivos.

La ventaja de trabajar directamente con la SHELL, en un terminal de texto, es la flexibilidad y la posibilidad de automatizar tareas mediante *scripts*, pequeños programas que incluyen varias órdenes y herramientas que permiten a aquellas interactuar unas con otras. El precio que hay que pagar es la necesidad de aprender un lenguaje para sacarle partido a la SHELL.

Por ejemplo, ante el problema de convertir 200 ficheros de imagen de formato tiff a formato jpg. ¿Como se haría con un programa basado en iconos y botones? Repitiendo 200 veces una secuencia de movimientos y pulsaciones de ratón.

¿Cómo se haría en linea de comandos? Repitiendo 200 veces una orden similar a `convert fichero.tif fichero.jpg` donde fichero se sustituye cada vez por el nombre de un fichero.

Una posibilidad sería hacer una lista de tareas, es decir escribir en un fichero de texto 200 veces la orden y editar cada linea para poner los nombres correctos de los ficheros.

```
convert fichero_1.tif fichero_1.jpg
convert fichero_2.tif fichero_2.jpg
convert fichero_3.tif fichero_3.jpg
..       ..         ..
convert fichero_N.tif fichero_N.jpg
```

A continuación bastaría con copiar y pegar estas lineas en la terminal.

Una posibilidad más inteligente es convertir la lista de tareas en un script. Asumiendo que queremos transformar todos los ficheros tiff del directorio de trabajo, el siguiente script realizará el trabajo por nosotros.

```
for i in $(ls *.tif);do
    o=$(echo $i|sed 's/tif/jpg/');
    convert $i $o;
done
```

Además de ahorrarnos tecleo, este script nos ahorrará tener que estar pendientes de que el programa haya terminado una transformación para iniciar la siguiente.

Aunque no entiendas del todo el anterior script quedate con la idea de que la primera linea inicia un bucle y define todos los ficheros de entrada `i` a los que se va a aplicar el contenido del bucle; la segunda linea genera el nombre del fichero de salida (`o`) que corresponde a cada fichero de entrada sustituyendo `tif` por `jpg`; la tercera linea ejecuta la transformación mediante una llamada al programa `convert` y la cuarta cierra el bucle.

2 Operaciones con archivos

Un sistema operativo debe proporcionar una serie de facilidades para manejar archivos, al menos para los archivos en formato ASCII. A continuación se exponen las operaciones más habituales con ficheros ASCII y los comandos para ejecutarlas en los sistemas operativos tipo Unix:

- Listado de ficheros `ls`
- Creación, lectura y actualización, mediante un editor como `emacs`, `vi`
- Copiar, mover, renombrar y borrar archivos, `cp`, `mv`, `rm`, `mkdir`
- Visualización `cat`, `more`, `less`, `head`, `tail`
- Partición del fichero en trozos `split` (por filas), `cut` (por columnas)
- Concatenación `cat` (por filas), `join` (por columnas)
- Consulta y sustitución `sed`, `grep`
- Ordenación `sort`
- Búsqueda de ayuda `man`

2.1 Obtener un listado de los archivos

El comando **ls** lista archivos del directorio actual (**ls**) o de cualquier otro (**ls /bin**). Si se añade la opción -l hace el listado en formato largo, dando detalles. La salida obtenida consta de renglones parecidos a

-rw-rw-rw- 1 pp users 138 Apr 5 19:34 leame

y se interpretan así:

- El primer carácter indica el tipo de archivo de que se trata, con esta convención:
  - `-` archivo común,
  - `d` directorio,
  - `l` enlace o referencia a otro archivo.

- `rwxrw-rw` son los permisos del archivo. Los tres grupos de 3 caracteres indican permisos para el dueño del archivo (pepe), su grupo (users) y el resto del mundo.
  - `r` (read) permiso para leer el archivo
  - `w` (write) permiso para modificar o eliminar el archivo
  - `x` (execute) si se trata de un archivo, permiso para ejecutarlo como programa; si se trata de un directorio, permiso para ingresar en él y recorrerlo.
- `1` cantidad de enlaces, referencias a este archivo desde otros archivos ubicados en diferentes lugares.
- `pepe` nombre del usuario dueño del archivo.
- `users` nombre del grupo al que pertenece el archivo
- `138` tamaño en bytes del archivo.
- `Apr 5 19:34` fecha y hora de última modificación. Si no aparece el año, se asume el año corriente.
- `leame` nombre del archivo. Notar que el nombre del archivo está siempre al final.

`ls -a` muestra también archivos ocultos, normalmente no visibles en el listado. Los archivos cuyo nombre empieza con un punto son ocultos. Las entradas `.` y `..` representan el directorio actual y el directorio padre, respectivamente.

2.2 Leer el contenido de un archivo

El comando `cat fichero` muestra el contenido de *fichero*.

El comando `more fichero` presenta el fichero página a página (hay que pulsar una tecla para pasar de página)

El comando `less fichero` similar a `more` pero con la posibilidad de ir arriba y abajo dentro del fichero

El comando `head -n fichero` presenta las n primeras lineas del fichero en pantalla

El comando `tail -n fichero` presenta las n últimas lineas del fichero en pantalla

2.3 Gestión de archivos

El comando `cp fichero1 fichero2` copia fichero1 con el nombre fichero2.

El comando `mv fichero1 fichero2` mueve o renombra fichero1 afichero2.

El comando `rm fichero` borra el archivo.

El comando `mkdir directorio` crea un nuevo directorio.

2.4 Partición de archivos

El programa `split` divide un fichero en varios ficheros. La opción `-l n` determina cuantas lineas iran a cada fichero, el parámetro `prefijo` determina cual será el prefijo con el que se formarán los nombres de los ficheros de salida (se les añadirán combinaciónes de 2 letras: aa, ab, ac,...). Suponiendo que `fichero1` tiene 4500 lineas, la orden:

```
split -l 1000 fichero1 fich
```

generará 5 ficheros: `fichaa`, `fichab`, `fichac`,`fichad`, `fichae`. Los cuatro primeros contendrán 1000 lineas y el último 500.

El programa `cut` selecciona determinadas columnas de un fichero. La opción `-d` permite determinar el carácter que se utiliza como separador de columnas, la opción `-f` permite elegir que columnas queremos extraer. La salida de `cut` se dirige a la pantalla pero la podemos redirigir a un fichero.

Si el contenido de `fic1` es:

```
1 alpha 2 azul
2 alpha 3 rojo
3 beta 3 rojo
4 gamma 2 rojo
```

La orden:

```
cut -f 2,4 -d " " fic1
```

producirá como salida:

```
alpha azul
alpha rojo
beta rojo
gamma rojo
```

2.5 Concatenación de archivos

El programa `cat` permite también concatenar archivos. La orden `cat fichero1>>fichero2` copia el contenido de fichero1 al final de fichero2

Para concatenar archivos por columnas se utiliza `join`. A partir de dos ficheros `fic1` y `fic2` genera una serie de lineas concatenando aquellas que tengan el mismo valor en un determinado campo (columna) que actúa como campo clave.

Las opciones más importantes que se pueden pasar a `join` son:

- `-1 n` donde n es la columna que actuará como campo clave en el primer fichero (por defecto es la primera).
- `-2 n` donde n es la columna que actuará como campo clave en el segundo fichero (por defecto es la primera).
- `-t c` donde c es el carácter que se utiliza como separador de campos (por defecto es el espacio).

2.6 sed y grep

El comando `grep` permite buscar las líneas que contienen una cadena de caracteres especificada mediante una expresión regular. Lee la entrada estándar o una lista de archivos y muestra en la salida sólo aquellas líneas que contienen la expresión indicada. La sintaxis es:

```
grep patrón archivos
```

donde el patrón a buscar es una expresión regular.

Crea un archivo con los días de la semana, uno por línea; llamarle dias y prueba las siguientes órdenes:

```
grep martes dias
grep tes dias
```

Entre las opciones de grep se cuentan `-i` para evitar distinguir entre mayúsculas de minúsculas, `-n` para mostrar el número de línea en que se produce la coincidencia y `-v` para buscar líneas que no contengan la expresión indicada.

2.7 sort

El comando `sort` permite la ordenación del contenido de un fichero por caracteres ASCII o por valor numérico. La ordenación ASCII es la más parecida a la alfabética; sigue el orden del juego de caracteres ASCII. En la ordenación numérica se respeta la ordenación por valor numérico de la cadena de caracteres: 101 va después de 21; en ordenamiento ASCII sería al revés.

```
sort arch1 ordena según el código ASCII.
sort -n arch2.num ordena numéricamente.
```

Si no se indican campos de ordenación, la comparación se hace sobre toda la línea. Si se indican campos, la comparación se hace considerando la cadena de caracteres iniciada en el primer caracter del primer campo hasta el último caracter del último campo.

```
sort -t: -k1,3 arch1.txt
```

ordena por campos separados por ":", tomando en cuenta para la comparación los caracteres desde el primero del campo 1 hasta el último del campo 3.

```
sort -t: -k1.3,3.5 arch1.txt
```

ordena por campos tomando en cuenta desde el 3er. caracter del campo 1 hasta el 5to. caracter del campo 3.

```
sort -nr arch2.num
```

ordena en orden numérico descendente.

```
sort -k3 arch3.txt
```

ordena alfabéticamente, usando como cadena de comparación la comprendida desde el primer caracter del 3er. campo hasta el fin de lína. Como no se indica separador, los campos se definen por blancos (espacio o tabulador).

Otras opciones interesantes son `-f` que ordena sin distinguir entre mayúsculas y minúsculas; y `-r` que ordena en orden inverso.

2.8 Búsqueda de ayuda

BASH dispone de un programa para generar ayuda relativa a cualquier programa o comando del sistema (`man`). Por ejemplo:

```
man sort
```

mostrará en pantalla la ayuda de dicho programa.

3 Variables

Un script puede ser una simple lista de ordenes de sistema. Sin embargo para que sean realmente útiles los scripts necesitan tener cierta capacidad de generalización. Para ello es necesario el uso de variables:

```
x=10
echo $x
```

Como ves, cuando se define una variable no hay que precederla de un `$` pero si cuando se utiliza.

El comando `echo Mensaje_en_pantalla` muestra en la pantalla el mensaje indicado.

`echo Mensaje_en_pantalla>fichero` Escribe el mensaje en el archivo fichero.

`echo Otro_mensaje_en_pantalla»fichero` Concatena el mensaje en el archivo fichero.

Otra posibilidad a la hora de definir una variable es asignar a esta el resultado de la ejecución de una orden, para ello basta con poner la orden entre paréntesis precedida de un `$`. Puede verlo en el siguiente ejemplo:

```
x=$(seq 1 10)
echo $x
```

La orden `seq` simplemente devuelve la secuencia de números solicitada, en este caso se ha almacenado en la variable `x` cuyo valor pasa a ser:

```
1 2 3 4 5 6 7 8 9 10
```

3.1 Expresiones aritméticas

En BASH podemos introducir expresiones aritméticas sencillas que sólo admiten números enteros:

```
a=3;b=5;c=4;d=7
y=$(( ($a*$b + $c*$d)/6 ))
echo $y
```

El resultado será 7 debido al redondeo.

La orden `let` permite ejecutar cálculos sencillos evitando los pares de paréntesis iniciales y finales:

```
a=3
let b=$a+3
let c=$a*3
echo $a $b $c
```

El resultado será: `3 6 9`.

Admite también divisiones:

```
a=30
let a=$a/3
echo $a
```

pero sólo enteras. En caso de necesitar calculos más complejos es preferible utilizar awk:

```
a=4;b=7
c=$(echo $a $b|awk '{print sqrt($1*$1+$2*$2)}')
echo $a $b $c
```

En este último ejemplo, además de utilizar awk para hacer una raiz cuadrada, se ha utilizado la sintáxis `c=$(orden)` para asignar como valor de una variable el resultado de una orden al sistema.

3.2 Expresiones lógicas

El número de expresiones lógicas que pueden verificarse es muy grande, incluyendo operadores para cadenas de carácteres y números enteros, pero no para reales.

3.2.1 Operadores de comparación numéricos

- Igual `-eq`
- No igual `-ne`
- Menor que `-lt`
- Menor o igual que `-le`
- Mayor que `-gt`
- Mayor o igual que `-ge`

El comando `test` nos sirve para realizar comparaciones, el valor que devuelve es 0 si la comparación es cierta y 1 si no lo es. Por ejemplo el script:

```
num=5
test $num -eq 10
$?
```

devolverá `1`. En este script se ha utilizado la expresión `$?` que devuelve el valor devuelto por la última orden ejecutada.

3.2.2 Operadores de comparación de texto

Aunque resulte algo antiintuitivo, BASH utiliza los comparadores habituales en matemáticas para comparar textos (mientras que para números utiliza los que se han visto anteriormente). Así la lista de comparadores es:

- Igual =
- No igual `!=`
- Menor que <

- Mayor que >

Para utilizar el comando `test` con textos es necesario entrecomillar las variables:

```
a=Elefante;b=Cocodrilo
test "$a" = "$b"
echo $?
test "$a" != "$b"
echo $?
```

Los resultados serán `1` en el primer caso y `0` en el segundo.

Podemos encadenar condiciones con los operadores `Y` lógico (`&&`), `O` lógico (`||` y `NO` (`!`). Por ejemplo:

```
test "$a" != "$b" && test 2 -eq 2
```

Recuerda que en operaciones lógicas `||` tiene la misma precedencia que la suma y `&&` la misma que el producto, así que cuando sea necesario habrá que poner paréntesis, es decir que las siguientes expresiones no son iguales y no producirán el mismo resultado:

```
test "$a" == "$b" && test 2 -eq 3 || test 2 -eq 2
test "$a" == "$b" && ( test 2 -eq 3 || test 2 -eq 2 )
```

3.2.3 Operadores lógicos con ficheros

Existen diversos operadores para consultar características sobre los ficheros presentes en el sistema. Por ejemplo:

```
test -e mifichero.txt
echo $?
```

devolverá 0 si el fichero existe. Puedes consultar la lista de pruebas que puedes ejecutar sobre los archivos en Gallardo & Wolf (2002).

3.3 Arrays

También podemos definir arrays en BASH:

```
declare -a identificador
identificador=(1 22 33  40 51)
echo ${identificador[3]}
```

Hay que tener en cuenta que:

- Son necesarias las llaves
- El primer elemento del array es el 0
- Si se sustituye el índice entre corchetes por un asterisco, devuelve todos los valores

Así el resultado del anterior script será `40` ya que es el tercer elemento del array.

Si tras el script anterior escribimos:

```
identificador[3]=50
echo ${identificador[*]}
```

El resultado será:

```
1 22 33 50 51
```

como ves podemos modificar directamente los elementos de un array.

3.4 Concatenación de variables

Para concatenar dos variables de texto en BASH basta con escribirlas juntas tal como se puede ver en los siguientes ejemplos:

```
extension=txt;fichero=datos
echo $fichero.$extension
```

```
extension=txt;fichero=datos
fichero=${fichero}001.$extension
```

Si no resulta evidente donde termina el nombre de la variable es necesario delimitarlo explicitamente con llaves tal como se ve en el segundo ejemplo.

El entrecomillado simple convierte toda la concatenación en un literal.

```
extension=txt;fichero=datos
fichero='${fichero}001.$extension'
echo $fichero
```

la salida de este último script será

```
${fichero}001.$extension
```

4 Interacción con el usuario

4.1 Hacer ejecutable un script

Hasta ahora, los ejemplos que se han visto podían copiarse y pegarse directamente. En muchos casos es una buena idea abrir un editor de textos y escribir las órdenes en él para luego copiarlas y pegarlas en el terminal de texto ya que es mucho más fácil editar sobre un editor que sobre el terminal.

Sin embargo para que los scripts sean realmente útiles es necesario convertirlos en programas que puedan ser ejecutados por el usuario. Para ello debes decirle al sistema que tu fichero de texto que contiene las órdenes puede ser ejecutado. Para ello debes modificar el *modo* del fichero:

```
chmod 755 miscript
```

De este modo le concedes permiso de lectura, escritura y ejecución al dueño del fichero (o sea a ti mismo) y permiso de lectura y ejecución al resto de los usuarios.

Por otra parte el sistema debe saber, al ejecutar tu programa, a que intérprete de órdenes se dirigen estas; puesto que estamos programando para BASH escribiremos como primera linea del programa:

```
#!/bin/bash
```

Este sistema puede utilizarse con programas desarrollados para cualquier lenguaje interpretado:

```
#!/usr/bin/perl
#!/usr/bin/awk
#!/usr/bin/tclsh
```

Siempre que, por supuesto, el intérprete este disponible en el sistema y las órdenes presentes en el fichero correspondan a ese lenguaje.

4.2 Parámetros que se pasan al programa

.

A un script, como a cualquier tipo de programa se le puede pasar cualquier número de parámetros. BASH utiliza parámetros posicionales y dentro del script se hace referencia a ellos mediante las variables $1 para el primer parámetro, $2 para el segundo, etc.

Suponiendo que el script `parametros` contiene:

```
#!/bin/sh
echo $3 $2 $1
```

La siguiente llamada:

```
~$ parametros uno dos tres
```

producirá la siguiente salida:

```
tres dos uno
```

4.3 Escribiendo texto en pantalla

Ya has visto como `echo` es el comando adecuado para producir salidas de texto en pantalla. Se trata de un comando bastante primitivo, una opción más interesante sería utilizar `printf`. Este programa utiliza como primer parámetro una cadena de texto que especifica el formato con el que se van a escribir las variables, y a continuación estas variables.

```
quien=mundo
printf "Hola %s\n" $quien
```

%d	Número entero
%nd	Número entero formateado a n caracteres
%f	Número real
%$m.n$f	Número real con n decimales formateado a m caracteres
%s	Cadena de carácteres

El comando `printf` es equivalente a funciones que, con el mismo nombre, están disponibles en C o AWK; permite formatear la salida de texto según un patrón entrecomillado. Este patrón puede estar formado por caracteres, caractéres de control precedidos por \(\t es el tabulador y \n el retorno de carro) o códigos que reservan posiciones para las variables que se van a escribir (ver la siguiente tabla).

4.4 Pidiendo información al usuario

El comando `read` espera a que el usuario introduzca un dato mediante el teclado (hay que pulsar retorno de carro para que `read` entienda que el usuario ha terminado). Puede servir simplemente para dar al usuario control sobre el tiempo de ejecución del script, pero resulta más útil para permitir que el usuario de valor sobre la marcha a las variables:

```
read algo
echo $algo
```

Puede utilizarse de forma más sofisticada añadiendo un prompt para que el usuario sepa que hacer:

```
read -p "Dime algo: " -a algo
echo Has dicho $algo
```

Más interesante puede ser utilizar arrays en combinación con el comando `select` para generar menús para el usuario:

```
declare -a acciones
acciones=(copiar renombrar borrar)
select accion in ${acciones[*]};do
   echo Has elegido $accion
done
```

5 Tuberias y redirecciones

A veces es útil enviar la salida de un programa directamente a la pantalla (el comportamiento por defecto), pero en otros casos será más interesante redirigir esta salida a otro lugar. Cualquier programa informático puede concebirse como un sistema que transforma un archivo de entrada en otro de salida. En Unix este hecho es especialmente evidente ya que el sistema proporciona una gran flexibilidad para construir sistemas de proceso de datos mediante la integración de comandos. Las *tuberías* | y las *redirecciones* > o >> que son los elementos clave para conseguir esta integración.

```
ls -l>listado.txt
```

Crea un fichero de texto llamado `listado.txt` que contendrá el listado de ficheros producido con `ls`. El inconveniente es que si `listado.txt` existía previamente lo eliminará. Para evitarlo se puede utilizar:

```
ls -l>>listado.txt
```

que, en caso de que `listado.txt` existiera previamente, lo mantendrá y escribirá la salida de `ls -l` a continuación del contenido preexistente.

El programa `cat` proporciona como salida el contenido del archivo de entrada pero este se puede redirigir a otro archivo:

```
cat archivo1 > archivo2
```

De esta forma, `archivo2` será una copia de `archivo1`. Si `archivo2` existía previamente habrá sido eliminado. Si en lugar de eleminarlo hubiesemos querido añadir a `archivo2` el contenido de `archivo1`, la orden hubiese sido:

```
cat archivo1 >> archivo2
```

Un tercer tipo de redirección es < que permite que una orden tome sus datos de entrada de un fichero.

Las tuberías | permiten pasar a un programa la salida de otro; por ejemplo:

```
ls -l|more
```

Pemitirá ver página por página el listado producido por `-l`.

```
ls|awk 'print $8'|grep 4
```

redirige el listado que se obtiene como salida de `ls` a `grep` con el parámetro `4` que seleccionará sólo aquellos ficheros que incluyan el número 4 en su nombre.

La filosofía de Unix se basa en muchas herramientas pequeñas (como `cat`, `more`, `grep` o `ls`) que hacen una tarea sencilla y su potencia reside en la capacidad

de integrar (mediante tuberías redirecciones y otros elementos de programación) varias de estas herramientas para hacer una tarea compleja.

6 Control del flujo

BASH, además de un entorno de trabajo, es un lenguaje de programación, y como cualquiera de ellos necesita decidir que acciones ejecutar según los resultados de operaciones anteriores. Además es necesario automatizar la repetición de determinadas acciones un número fijo de veces o en función de que se cumpla o no una condición.

6.1 Condicionales

Ya se ha visto como existen diversos operadores que permiten determinar si se cumple una determinada condición y devuelven un valor cierto (0) o falso (1). Ahora se verá como se puede hacer que los valores de estas comparaciones determinen cual será el rumbo que cogerá un programa.

6.1.1 if

Permite seleccionar entre unas pocas opciones:

```
if [ ``$x'' = ``$k'' ]; then
   echo Son iguales
else
   echo No son iguales
fi
```

La identación de lineas que has visto en este ejemplo no es obligatoria pero ayuda a que el programa sea más legible, en los ejemplos que siguen aparece a menudo.

```
if [ $edad -le 18 ]; then
   echo Joven
else
   echo Mayor
fi
```

6.1.2 Estructuras case

Una alternativa a `if` cuando las opciones posibles son varias es la herramienta `case`:

```
case $opcion in
-f)
   echo Opcion -f;;
-k)
   echo Opción -k;;
fichero)
   echo fichero;;
*)
   echo Opción inválida;;
esac
```

```
case $edad in
   8|9|10|11|12|13) echo niño ;;
   14|15|16|17|18) echo joven ;;
   *) echo mayor;;
esac
```

6.2 Bucles

6.2.1 Bucles con for

El comando `for` ejecuta el bucle de instrucciones situado entre `do` y `done` para el conjunto de valores de la variable especificada en la orden (el conjunto de valores también se especifica en la orden).

En el siguiente ejemplo se muestra la tabla del 2, se ha utilizado el comando `printf` para conseguir un adecuado formateado de la salida:

```
for v in $(seq 1 10);do
         let v2=$v*2;
         printf "%d*%d=%d\n" $v 2 $v2
done
```

El siguiente script muestra un bucle integrado dentro de otro para mostrar las tablas de multiplicar.

```
for v in $(seq 1 10);do
    for v2 in $(seq 1 10); do
         let v3=$v*$v2;
         printf "%d*%d=%d\t" $v $v2 $v3
    done
    printf "\n"
done
```

6.2.2 Bucles con while

El comando `while` ejecuta el bucle de instrucciones situado entre `do` y `done` mientras se cumpla la condición especificada como parámetro en la llamada. Por ejemplo el siguiente script escribe los números del 1 al 10:

```
a=1
while test $a -le 10;do
   echo $a
   let a=$a+1
done
```

En el siguiente ejemplo se verán diversas opciones nuevas:

```
while who|grep pepe>/dev/null;do
  sleep 30
done
echo ... y ahora lanzo el proceso gordo
```

Este script comprueba cada 30 segundos si el usuario pepe está conectado al ordenador y sólo cuando se haya desconectado finaliza el bucle y pasa a ejecutar el resto del script que, en este caso, podría ser el típico proceso que consume muchos recursos y es preferible ejecutar cuando no haya otros usuarios trabajando.

La orden `who` presenta un listado de los usuarios conectados al sistema; este listado se pasa mediante una tubería a la orden `grep pepe` que dará una respuesta positiva si pepe está incluido en el listado. En ese caso esperará 30 segundos (`sleep 30` y volverá a consultar.

La salida de `grep pepe` se dirige a `/dev/null` que es un dispositivo virtual que se utiliza para evitar que la información llegue constantemente a la terminal de salida.

6.2.3 Bucles con until

El comando `until` permite realizar el proceso contrario, es decir ejecutar el bucle hasta que se cumpla la condición especificada. El siguiente script es equivalente al anterior pero ahora esperamos al usuario pepe para lanzar un proceso.

```
usuario=pepe
until who|grep \$usuario>/dev/null;do
  sleep 30
done
echo ... y ahora lanzo el proceso
```

BASH, al igual que casi todos los lenguajes de programación, dispone de dos órdenes para modificar el comportamiento de los bucles: `break` y `continue`. El primero rompe la ejecución del bucle y pasa a la siguiente linea tras `done`; el segundo rompe la ejecución del bucle pero pasa a la siguiente iteración.

```
for i in $(seq 1 10);do
   if test $i -eq 6;then
       break
   fi
   echo $i
done
```

```
for i in $(seq 1 10);do
   if test $i -eq 6;then
       continue
   fi
   echo $i
done
```

7 Funciones

Un script de shell también puede utilizar funciones. Estas encapsulan un conjunto de acciones que se van a ejecutar de la misma manera varias veces en un programa. Para definir una función basta con escribir el nombre de la misma seguido de `()` y,

entre corchetes, el conjunto de ordenes de que consta; sin embargo, por claridad, es preferible preceder el nombre de la función de la alabra clave `function`.

Si la función se define en una sola linea, hay que tener la precaución de terminar la última orden incluida dentro de ella con `;`

```
listado () {ls -la;}
listado
```

Una función sin parámetros no resulta muy útil. En realidad para estos casos es más habitual utilizar el comando `alias`:

```
alias listado=''ls -la''
listado
```

A una función se le pueden pasar los parámetros que sean necesarios, en el siguiente ejemplo se presenta una función para calcular el factorial de un número que es pasado como parámetro. Como ves dentro de la función se utiliza el mismo sistema para hacer referencia a los diferentes parámetros que se vio para los parámetros de un script.

```
function factorial(){
   f=1
   for i in $(seq 2 $1);do
     f=$(($f*$i))
   done
   echo $f
}
factorial 12
```

El siguiente ejemplo introduce una función para esperar a la conexión de un usuario, en este caso el nombre del usuario se pasa como parámetro a la función:

```
esperar_a(){
  usuario=$1
  until who|grep $usuario>/dev/null;do
    sleep 5
  done
}
```