Programación de comandos combinados (*shell scripts*)

Remo Suppi Boldrito

PID_00215378

UOC
Universitat Oberta de Catalunya

© 2014, FUOC. Se garantiza permiso para copiar, distribuir y modificar este documento según los términos de la GNU Free Documentation License, Version 1.2 o cualquiera posterior publicada por la Free Software Foundation, sin secciones invariantes ni textos de cubierta delantera o trasera. Se dispone de una copia de la licencia en el apartado "GNU Free Documentation License" de este documento.

Índice

Introducción

En este módulo veremos la importancia fundamental que tiene el intérprete de órdenes o comandos (*shell*) y sobre todo analizaremos con cierto detalle las posibilidades de estos para ejecutar ficheros de órdenes secuenciales y escritos en texto plano (ASCII) que serán interpretados por el *shell*. Estos ficheros, llamados *shell scripts*, son la herramienta fundamental de cualquier usuario avanzado e imprescindible para un administrador de sistemas *nix.

Un conocimiento práctico de *shell scripting* será necesario ya que el propio GNU/Linux se basa en este tipo de recurso para iniciar el sistema (por ejemplo, ejecutando los archivos de */etc/rcS.d*), por lo cual los administradores deben tener los conocimientos necesarios para trabajar con ellos, entender el funcionamiento del sistema, modificarlos y adecuarlos a las necesidades específicas del entorno en el cual trabajen.

Muchos autores consideran que hacer *scripting* es un arte, pero en mi opinión no es difícil de aprender, ya que se pueden aplicar técnicas de trabajar por etapas ("divide y vencerás"), que se ejecutará en forma secuencial y el conjunto de operadores/opciones no es tan extenso como para que genere dudas sobre qué recurso utilizar. Sí que la sintaxis será dependiente del *shell* utilizado, pero al ser un lenguaje interpretado con encadenamiento de secuencias de comandos, será muy fácil de depurar y poner en funcionamiento.

Hoy en día el *scripting* ha alcanzado muchos ámbitos con la potencialidad presentada por algunos lenguajes como PHP para desarrollar código del lado del servidor (originalmente diseñado para el desarrollo web de contenido dinámico); Perl, que es un lenguaje de programación (1987) que toma características de C, de *Bourne shell*, AWK, `sed`, Lisp entre otros; Phyton (1991), cuya filosofía hace hincapié en una sintaxis que favorezca un código legible siendo un lenguaje de programación (interpretado) que soporta orientación a objetos, programación imperativa y, en menor medida, programación funcional y con tipos dinámicos, o como Ruby (1993-5), que es un lenguaje de programación (también interpretado), reflexivo y orientado a objetos, que combina una sintaxis inspirada en Python y Perl con características de programación orientada a objetos similares a Smalltalk.

También un *shell script* es un método "*quick-and-dirty*" (que podría traducir como 'rápido y en borrador') de prototipado de una aplicación compleja para conseguir incluso un subconjunto limitado de la funcionalidad útil en una primera etapa del desarrollo de un proyecto. De esta manera, la estructura de

la aplicación y las grandes líneas puede ser probada y determinar cuáles serán las principales dificultades antes de proceder al desarrollo de código en C, C++, Java, u otro lenguaje estructurado y/o interpretado más complejo.

De acuerdo con M. Cooper, hay una serie de situaciones en las cuales él categóricamente aconseja "no utilizar shell scripts" si bien en mi opinión diría que se deberá ser prudente y analizar en profundidad el código desarrollado sobre todo desde el punto de vista de seguridad para evitar que se puedan realizar acciones más allá de las cuales fueron diseñadas (por ejemplo, en *scripting*, por el lado del servidor en aplicaciones web, es una de las principales causas de intrusión o ejecución no autorizadas). Entre las causas podemos enumerar: uso intensivo de recursos, operaciones matemáticas de alto rendimiento, aplicaciones complejas donde se necesite estructuración (por ejemplo, listas, árboles) y tipos de variables, situaciones en las que la seguridad sea un factor determinante, acceso a archivos extensos, matrices multidimensionales, trabajar con gráficos, acceso directo al hardware/comunicaciones. No obstante, por ejemplo, con la utilización de Perl, Phyton o Ruby, muchas de estas recomendaciones dejan de tener sentido y en algunos ámbitos científicos (por ejrmplo, en genética, bioinformática, química, etc.) el *scripting* es la forma habitual de trabajo.

Este módulo está orientado a recoger las principales características de Bash (*Bourne-Again Shell*) que es el *shell script* estándar en GNU/Linux, pero muchos de los principios explicados se pueden aplicar a Korn Shell o C Shell.

1. Introducción: el *shell*

El *shell* es una pieza de software que proporciona una interfaz para los usuarios en un sistema operativo y que provee acceso a los servicios del núcleo. Su nombre proviene de la envoltura externa de algunos moluscos, ya que es la "parte externa que protege al núcleo".

Los *shells* se dividen en dos categorías: línea de comandos y gráficos, ya sea si la interacción se realiza mediante una línea de comandos (CLI, *command line interface*) o en forma gráfica a través de una GUI (*graphical user interface*). En cualquier categoría el objetivo principal del *shell* es invocar o "lanzar" otro programa, sin embargo, suelen tener capacidades adicionales, tales como ver el contenido de los directorios, interpretar órdenes condicionales, trabajar con variables internas, gestionar interrupciones, redirigir entrada/salida, etc.

Si bien un *shell* gráfico es agradable para trabajar y permite a usuarios sin muchos conocimientos desempeñarse con cierta facilidad, los usuarios avanzados prefieren los de modo texto ya que permiten una forma más rápida y eficiente de trabajar. Todo es relativo, ya que en un servidor es probable que un usuario administrador no utilice ni siquiera interfaz gráfica, mientras que para un usuario de edición de vídeo nunca trabajará en modo texto. El principal atractivo de los sistema *nix es que también en modo gráfico se puede abrir un terminal y trabajar en modo texto como si estuviera en un *shell* texto, o cambiar interactivamente del modo gráfico y trabajar en modo texto (hasta con 6 terminales diferentes con *Crtl+Alt+F1*-6 generalmente) y luego regresar al modo gráfico con una simple combinación de teclas. En este apartado nos dedicaremos a *shell* en modo texto y las características avanzadas en la programación de *shell scripts*.

Entre los *shells* más populares (o históricos) en los sistemas *nix tenemos:

- Bourne shell (sh).
- Almquist shell (ash) o su versión en Debian (dash).
- Bourne-Again shell (bash).
- Korn shell (ksh).
- Z shell (zsh).
- C shell (csh) o la versión Tenex C shell (tcsh).

El *Bourne shell* ha sido el estándar *de facto* en los sistemas *nix, ya que fue distribuido por Unix Version 7 en 1977 y *Bourne-Again Shell* (Bash) es una versión mejorada del primero escrita por el proyecto GNU bajo licencia GPL, la cual se ha transformado en el estándar de los sistemas GNU/Linux. Como ya hemos

comentado, Bash será el intérprete que analizaremos, ya que, además de ser el estándar, es muy potente, posee características avanzadas, recoge las innovaciones planteadas en otros *shells* y permite ejecutar sin modificaciones (generalmente) *scripts* realizados para cualquier *shell* con sintaxis Bourne compatible. La orden `cat /etc/shells` nos proporciona los *shells* conocidos por el sistema Linux, independientemente de si están instalados o no. El *shell* por defecto para un usuario se obtendrá del último campo de la línea correspondiente al usuario del fichero */etc/passwd* y cambiar de *shell* simplemente significa ejecutar el nombre del *shell* sobre un terminal activo, por ejemplo:

```
RS@DebSyS:~$ echo $SHELL
/bin/bash
RS@DebSyS:~$ tcsh
DebSyS:~>
```

Una parte importante es lo que se refiere a los modos de ejecución de un *shell*. Se llama *login* interactivo cuando proviene de la ejecución de un *login*, lo cual implicará que se ejecutarán los archivos */etc/profile, ~/.bash_profile, ~/.bash_login o ~/.profile* (la primera de las dos que exista y se pueda leer) y *~/.bash_logout* cuando terminemos la sesión. Cuando es de *no-login* interactivo (se ejecuta un terminal nuevo) se ejecutará *~/.bashrc*, dado que un bash interactivo tiene un conjunto de opciones habilitadas a si no lo es (consultar el manual). Para saber si el shell es interactivo, ejecutar `echo $-` y nos deberá responder `himBH` donde la ***i*** indica que sí lo es.

Existen un conjunto de comandos internos[1] a *shell*, es decir, integrados con el código de *shell*, que para Bourne son:

(1)Consultar el manual para una descripción completa.

:, ., `break`, `cd`, `continue`, `eval`, `exec`, `exit`, `export`, `getopts`, `hash`, `pwd`, `readonly`, `return`, `set`, `shift`, `test`, `[`, `times`, `trap`, `umask`, `unset`.

Y además Bash incluye:

`alias`, `bind`, `builtin`, `command`, `declare`, `echo`, `enable`, `help`, `let`, `local`, `logout`, `printf`, `read`, `shopt`, `type`, `typeset`, `ulimit`, `unalias`.

Cuando Bash ejecuta un *script shell*, crea un proceso hijo que ejecuta otro Bash, el cual lee las líneas del archivo (una línea por vez), las interpreta y ejecuta como si vinieran de teclado. El proceso Bash padre espera mientras el Bash hijo ejecuta el *script* hasta el final que el control vuelve al proceso padre, el cual vuelve a poner el *prompt* nuevamente. Un comando de *shell* será algo tan simple como `touch archivo1 archivo2 archivo3` que consiste en el propio comando seguido de argumentos, separados por espacios considerando *archivo1* el primer argumento y así sucesivamente.

1.1. Redirecciones y pipes

Existen 3 descriptores de ficheros: *stdin*, *stdout* y *stderr* (la abreviatura std significa estándar) y en la mayoría de los *shells* (incluso en Bash) se puede redirigir *stdout* y *stderr* (juntas o separadas) a un fichero, *stdout* a *stderr* y viceversa. Todas ellas se representan por un número, donde el número 0 representa a *stdin*, 1 a *stdout*, y 2 a *stderr*.

Por ejemplo, enviar *stdout* del comando *ls* a un fichero será: `ls -l > dir.txt`, donde se creará un fichero llamado 'dir.txt', que contendrá lo que se vería en la pantalla si se ejecutara `ls -l`.

Para enviar la salida *stderr* de un programa a un fichero, haremos `grep xx yy 2> error.txt`. Para enviar la *stdout* a la *stderr*, haremos `grep xx yy 1>&2` y a la inversa simplemente intercambiando el 1 por 2, es decir, `grep xx yy 2>&1`.

Si queremos que la ejecución de un comando no genere actividad por pantalla, lo que se denomina ejecución silenciosa, solamente debemos redirigir todas sus salidas a */dev/null*, por ejemplo pensando en un comando del `cron` que queremos que borre todos los archivos acabados en .mov del sistema: `rm -f $(find / -name "*.mov") &> /dev/null` pero se debe ir con cuidado y estar muy seguro, ya que no tendremos ninguna salida por pantalla.

Los *pipes* permiten utilizar en forma simple tanto la salida de una orden como la entrada de otra, por ejemplo, `ls -l | sed -e "s/[aeio]/u/g"`, donde se ejecuta el comando `ls` y su salida, en vez de imprimirse en la pantalla, se envía (por un tubo o *pipe*) al programa `sed`, que imprime su salida correspondiente. Por ejemplo, para buscar en el fichero */etc/passwd* todas las líneas que acaben con false podríamos hacer `cat /etc/passwd | grep false$`, donde se ejecuta el `cat`, y su salida se pasa al `grep` donde el $ al final de la palabra le está indicando al `grep` que es final de línea.

1.2. Aspectos generales

Si la entrada no es un comentario, es decir (dejando de lado los espacios en blanco y tabulador) la cadena **no** comienza por #, el *shell* lee y lo divide en palabras y operadores, que se convierten en comandos, operadores y otras construcciones. A partir de este momento, se realizan las expansiones y sustituciones, las redirecciones, y finalmente, la ejecución de los comandos.

En Bash se podrán tener funciones, que son una agrupación de comandos que se pueden invocar posteriormente. Y cuando se invoca el nombre de la función de *shell* (se usa como un nombre de comando simple), la lista de comandos

relacionados con el nombre de la función se ejecutará teniendo en cuenta que las funciones se ejecutan en el contexto del *shell* en curso, es decir, no se crea ningún proceso nuevo para ello.

Un **parámetro** es una entidad que almacena valores, que puede ser un nombre, un número o un valor especial. Una **variable** es un parámetro que almacena un nombre y tiene atributos (1 o más o ninguno) y se crean con la sentencia `declare` y se remueven con `unset` y si no se les da valor, se les asigna la cadena nula.

Por último, existen un conjunto de expansiones que realiza el *shell* que se lleva a cabo en cada línea y que pueden ser enumeradas como: de tilde/comillas, parámetros y variables, substitución de comandos, de aritmética, de separación de palabras y de nombres de archivos.

2. Elementos básicos de un *shell script*

2.1. ¿Qué es un *shell script*?

Un *shell script* es simplemente un archivo (mysys.sh para nosotros) que tiene el siguiente contenido (hemos numerado las líneas para referirnos a ellas con el comando el `cat -n` pero no forman parte del archivo):

```
RS@debian:~$ cat -n mysys.sh
1 #!/bin/bash
2 clear; echo "Información dada por el shell script mysys.sh. "
3 echo "Hola, $USER"
4 echo
5 echo "La fecha es `date`, y esta semana `date +"%V"`."
6 echo
7 echo "Usuarios conectados:"
8 w | cut -d " " -f 1 - | grep -v USER | sort -u
9 echo
10 echo "El sistema es `uname -s` y el procesador es `uname -m`."
11 echo
12 echo "El sistema está encendido desde hace:"
13 uptime
14 echo
15 echo "¡Esto es todo amigos!"
```

Para ejecutarlo podemos hacer `bash mysys.sh` o bien cambiarle los atributos para hacerlo ejecutable y ejecutarlo como una orden (al final se muestra la salida después de la ejecución):

```
RS@debian:~$ chmod 744 mysys.sh

RS@debian:~$ ./mysys.sh

Información dada por el shell script mysys.sh.
Hola, RS

La fecha es Sun Jun  8 10:47:33 CEST 2014, y es la semana 23.

Usuarios conectados: RS

El sistema es Linux y el procesador es x86_64.

El sistema está encendido desde hace:
```

```
10:53:07 up 1 day, 11:55,  1 user,  load average: 0.12, 0.25, 0.33


¡Esto es todo amigos!
```

#!/bin/bash

Así evitamos que si el usuario tiene otro *shell*, su ejecución dé errores de sintaxis, es decir, garantizamos que este script siempre se ejecutará con bash.

El script comienza con "#!", que es una línea especial para indicarle con qué *shell* se debe interpretar este script. La línea 2 es un ejemplo de dos comandos (uno borra la pantalla y otro imprime lo que está a la derecha) en la misma línea, por lo que deben estar separados por ";". El mensaje se imprime con la sentencia `echo` (comando interno) pero también se podría haber utilizado la sentencia `printf` (también comando interno) para una salida con formato. En cuanto a los aspectos más interesantes del resto, la línea 3 muestra el valor de una variable, la 4 forma una cadena de caracteres (*string*) con la salida de un comando (reemplazo de valores), la 6 ejecuta la secuencia de 4 comandos con parámetros encadenados por pipes "|" y la 7 (similar a la 4) también reemplaza valores de comandos para formar una cadena de caracteres antes de imprimirse por pantalla.

Para depurar un *shell script* podemos ejecutar el *script* con `bash -x mysys.sh` o incluir en la primera línea el -x: `#!/bin/bash -x`.

También se puede depurar una sección de código incluyendo:

```
set -x # activa debugging desde aquí
código a depurar
set +x # para el debugging
```

2.2. Variables y *arrays*

Se pueden usar variables pero no existen tipos de datos. Una variable de bash puede contener un número, un carácter o una cadena de caracteres; no se necesita declarar una variable, ya que se creará con solo asignarle un valor. También se puede declarar con `declare` y después asignarle un valor:

```
#!/bin/bash
a="Hola UOC"
echo $a
declare b
echo $b
b="Cómo están Uds.?"
echo $B
```

Como se puede ver, se crea una variable *a* y se le asigna un valor (que para delimitarlo se deben poner entre " ") y se recupera el VALOR de esta variable poniéndole un '$' al principio, y si no se antepone el $, solo imprimirá el nombre de la variable no su valor. Por ejemplo, para hacer un script que haga una

copia (*backup*) de un directorio, incluyendo la fecha y hora en el momento de hacerlo en el nombre del archivo (intentad hacer esto tan fácil con comandos en un sistema W y terminaréis frustrados):

```
#!/bin/bash
OF=/home/$USER-$(date +%d%m%Y).tgz
tar -cZf $OF /home/$USER
```

Este *script* introduce algo nuevo, ya que estamos creando una variable y asignando un valor (resultado de la ejecución de un comando en el momento de la ejecución). Fijaos en la expresión $(date +%d%m%Y) que se pone entre () para capturar su valor. USER es una variable de entorno y solamente se reemplazará por su valor. Si quisiéramos agregar (concatenar) a la variable USER, por ejemplo, un *string* más, deberíamos delimitar la variable con {}. Por ejemplo:

```
RS@debian:~$OF=/home/${USER}uppi-$(date +%d%m%Y).tgz
RS@debian:~$ echo $OF
/home/RSuppi-17042010.tgz
```

Así, el nombre del fichero será distinto cada día. Es interesante ver el reemplazo de comandos que hace el bash con los (), por ejemplo, `echo $(ls)`.

Las variable locales pueden declararse anteponiendo *local* y eso delimita su ámbito.

```
#!/bin/bash
HOLA=Hola
function uni {
local HOLA=UOC
echo -n $HOLA
}
echo -n $HOLA
uni
echo $HOLA
```

La salida será *HolaUOCHola*, donde hemos definido una función con una variable local que recupera su valor cuando se termina de ejecutar la función.

Las variables las podemos declarar como **ARRAY[INDEXNR]=valor**, donde *INDEXNR* es un número positivo (comenzando desde 0) y también podemos declarar un *array* como `declare -a ARRAYNAME` y se pueden hacer asignaciones múltiples con: **ARRAY=(value1 value2 ... valueN)**

```
RS@debian:~$ array=( 1 2 3)
RS@debian:~$ echo $array
1
RS@debian:~$ echo ${array[*]}
```

```
1 2 3
RS@debian:~$ echo ${array[2]}
3
RS@debian:~$ array[3]=4
RS@debian:~$ echo ${array[*]}
1 2 3 4
RS@debian:~$ echo ${array[3]}
4
RS@debian:~$ unset array[1]
RS@debian:~$ echo ${array[*]}
1 3 4
```

2.3. Estructuras condicionales

Las estructuras condicionales le permiten decidir si se realiza una acción o no; esta decisión se toma evaluando una expresión.

La estructura condicional más básica es: **if** expresión; **then** sentencia; **fi** donde 'sentencia' solo se ejecuta si 'expresión' se evalúa como verdadera. '2<1' es una expresión que se evalúa falsa, mientras que '2>1' se evalúa verdadera.

Los condicionales tienen otras formas, como: **if** expresión; **then** sentencia1; **else** sentencia2 **fi**. Aquí 'sentencia1' se ejecuta si 'expresión' es verdadera. De otra manera se ejecuta 'sentencia2'.

Otra forma más de condicional es: **if** expresión1; **then** sentencia1; **elif** expresión2; **then** sentencia2; **else** sentencia3 **fi**. En esta forma solo se añade "else if 'expresión2' then 'sentencia2'", que hace que sentencia2 se ejecute si expresión2 se evalúa verdadera. La sintaxis general es:

```
if [expresión];
then
código si 'expresión' es verdadera.
fi
```

Un ejemplo en el que el código que se ejecutará si la expresión entre corchetes es verdadera se encuentra entre la palabra *then* y la palabra *fi*, que indica el final del código ejecutado condicionalmente:

```
#!/bin/bash
if [ "pirulo" = "pirulo" ]; then
echo expresión evaluada como verdadera
fi
```

Otro ejemplo con variables y comparación por igualdad y por diferencia (= o !=):

```
#!/bin/bash
T1="pirulo"
T2="pirulon"
if [ "$T1" = "$T2" ]; then
echo expresión evaluada como verdadera
else
echo expresión evaluada como falsa
fi
if [ "$T1" != "$T2" ]; then
echo expresión evaluada como verdadera
else
echo expresión evaluada como falsa
fi
```

Un ejemplo de expresión de comprobación si existe un fichero (vigilar con los espacios después de [y antes de]):

```
FILE=~/.basrc
if [ -f $FILE ]; then
echo el fichero $FILE existe
else
echo fichero no encontrado
fi
if [ 'test -f $FILE' ]
echo Otra forma de expresión para saber si existe o no
fi
```

Existen versiones cortas del **if**, como por ejemplo:

```
[ -z "${COLUMNS:-}" ] && COLUMNS=80
```

Es la versión corta de:

```
if [ -z "${COLUMNS:-}" ]; then
COLUMNS=80
fi
```

Otro ejemplo de uso de la sentencia **if** y variables:

```
#!/bin/bash
FILE=/var/log/syslog
MYMAIL="adminp@master.hpc.local"
SUBJECT="Error - $(hostname)"
BODY="Existen ERRORES en $(hostname) @ $(date). Ver syslog."
OK="OK: No se detectan errores en Syslog."
WARN="Posibles Errores. Analizar"
# Verificar que $FILE existe
```

```
if test ! -f "$FILE"
then
  echo "Error: $FILE no existe. Analizar cuál es el problema."
  exit 1
fi
# Buscamos errores
errorres=$(grep -c -i "rror" $FILE)
# Si?
if [ $errores -gt 0 ]
then    # Si ...
  echo "$BODY" | mail -s "$SUBJECT" $MYMAIL
  echo "Mail enviado ..."
else    # No
  echo "$OK"
fi
```

Resumen

La sentencia `if then else` se puede resumir como:

```
if list; then list; [ elif list; then list; ] ... [ else list; ] fi
```

Se debe prestar atención a los ";" y a las palabras clave. No es necesario poner toda la sentencia en una sola línea.

2.4. Los bucles

El bucle **for** tiene dos formas, una de las cuales es diferente a las de otros lenguajes de programación, ya que permite iterar sobre una serie de 'palabras' contenidas dentro de una cadena. La otra forma del **for** sigue los patrones de los lenguajes de programación habituales. El bucle **while** ejecuta una sección de código si la expresión de control es verdadera, y solo se detiene cuando es falsa (o se encuentra una interrupción explícita dentro del código en ejecución, por ejemplo a través de **break**). El bucle **until** es casi idéntico al bucle **while**, excepto en que el código se ejecuta mientras la expresión de control se evalúe como falsa. Veamos unos ejemplos y observemos la sintaxis:

```
#!/bin/bash
for i in $( ls ); do
echo elemento: $i
done
```

En la segunda línea declaramos *i* como la variable que recibirá los diferentes valores contenidos en `$( ls )`, que serán los elementos del directorio obtenido en el momento de la ejecución. El bucle se repetirá (entre do y done) tantas veces como elementos tenga la variable y en cada iteración la variable *i* adquirirá un valor diferente de la lista. Otra sintaxis aceptada podrá ser:

```
#!/bin/bash
for i in `seq 1 10`;
```

```
do
echo $i
done
```

En la segunda forma del **for** la sintaxis es **for ((expr1 ; expr2 ; expr3)) ; do list ; done**, donde primero se evalúa la expresión **expr1**, luego se evalúa la expresión repetidamente hasta que es 0. Cada vez que la expresión **expr2** es diferente de cero, se ejecuta **list** y la expresión aritmética **expr3** es evaluada. Si alguna de las expresiones se omite, esta siempre será evaluada como 1. Por ejemplo:

```
#!/bin/bash
for (( c=1; c<=5; c++ ))
do
   echo "Repetición Nro. $c "
done
```

Un bucle infinito podría ser:

```
#!/bin/bash
for (( ; ; ))
do
echo "Bucle infinito [ introducir CTRL+C para finalizar]"
done
```

La sintaxis del **while** es:

```
#!/bin/bash
contador=0
while [ $contador -lt 10 ]; do
echo El contador es $contador
let contador=contador+1
done
```

Como podemos ver, además de **while** hemos introducido una comparación en la expresión por menor "-lt" y una operación numérica con let contador=contador+1. La sintaxis del **until** es equivalente:

```
#!/bin/bash
contador=20
until [ $contador -lt 10 ]; do
echo Contador-Until $contador
let contador-=1
done
```

En este caso vemos la equivalencia de la sentencia y además otra forma de hacer operaciones sobre las mismas variables con el "-=".

Resumen

Resumen: las sentencias repetitivas se pueden resumir como:

```
for name [ [ in [ word ... ] ] ; ] do list ; done
for (( expr1 ; expr2 ; expr3 )) ; do list ;
while list-1; do list-2; done
until list-1; do list-2; done
```

Como en la sentencia de **if** se debe tener en cuenta los ";" y las palabras clave de cada instrucción.

2.5. Funciones, select, case, argumentos y otras cuestiones

Las funciones son la forma más fácil de agrupar secciones de código que se podrán volver a utilizar. La sintaxis es *function nombre { mi_código }*, y para llamarlas solo es necesario que estén dentro de mismo archivo y escribir su nombre.

```
#!/bin/bash
function salir {
exit
}
function hola {
echo Hola UOC!
}
hola
salir
echo Nunca llegará a esta línea
```

Como se puede ver tenemos dos funciones, *salir* y *hola*, que luego se ejecutarán (no es necesario declararlas en un orden específico) y como la función *salir* ejecuta un `exit` en *shell* nunca llegará a ejecutar el `echo` final. También podemos pasar argumentos a las funciones, que serán leídos por orden (de la misma forma como se leen los argumentos de un *script*) por $1, el primero, $2 el segundo y así sucesivamente:

```
#!/bin/bash
function salir {
exit
}
function hola-con-parametros {
echo $1
}
hola-con-parametros Hola
hola-con-parametros UOC
salir
```

```
    echo Nunca llegará a esta línea
```

El `select` es para hacer opciones y leer desde teclado:

```
    #!/bin/bash
    OPCIONES="Salir Opción1"
    select opt in $OPCIONES; do
    if [ "$opt" = "Salir" ]; then
    echo Salir. Adiós.
    exit
    elif [ "$opt" = "Opción1" ]; then
    echo Hola UOC
    else
    clear
    echo opción no permitida
    fi
    done
```

El `select` permite hacer menús modo texto y es equivalente a la sentencia `for`, solo que itera sobre el valor de la variable indicada en el `select`. Otro aspecto importante es la lectura de argumentos, como por ejemplo:

```
    #!/bin/bash
    if [ -z "$1" ]; then
    echo uso: $0 directorio
    exit
    fi
    A=$1
    B="/home/$USER"
    C=backup-home-$(date +%d%m%Y).tgz
    tar -cZf $A$B $C
```

Si el número de argumentos es 0, escribe el nombre del comando ($0) con un mensaje de uso. El resto es similar a lo que hemos visto anteriormente excepto por el primer argumento ($1). Mirad con atención el uso de " y la sustitución de variables. El `case` es otra forma de selección, miremos este ejemplo que nos alerta en función del espacio de disco que tenemos disponible:

```
#!/bin/bash
space=`df -h | awk '{print $5}' | grep % \
| grep -v Use | sort -n | tail -1 | cut -d "%" -f1 -`

   case $space in
   [1-6]*)
   Message="todo ok."
   ;;
   [7-8]*)
```

```
    Message="Podría comenzar a borrar algo en $space %"
    ;;
    9[1-8])
    Message="Uff. Mejor un nuevo disco. Partición $space % a tope."
    ;;
    99)
    Message="Pánico! No tiene espacio en $space %!"
    ;;
    *)
    Message="NO tienen nada..."
    ;;
    esac

    echo $Message
```

Además de prestar atención a la sintaxis del `case`, podemos mirar cómo escribir un comando en dos líneas con "\", el filtrado secuencia con varios greps y | y un comando interesante (filtro) como el `awk` (el mejor comando de todos los tiempos).

Como resumen de las sentencias antes vistas tenemos:

```
select name [ in word ] ; do list ; done
case word in [ [(] pattern [ | pattern ] ... ) list ;; ] ... esac
```

Un ejemplo de la sentencia `break` podría ser el siguiente: que busca un archivo en un directorio hasta que lo encuentra:

```
    #!/bin/bash
    for file in /etc/*
    do
    if [ "${file}" == "/etc/passwd" ]
    then
         nroentradas=`wc -l /etc/passwd | cut -d" " -f 1`
         echo "Existen ${nroentradas} entradas definidas en ${file}"
         break
   fi
    done
```

El siguiente ejemplo muestra cómo utilizar el `continue` para seguir con el siguiente bucle del lazo:

```
    #!/bin/bash
    for f in `ls`
    do
         # if el archivo .org existe leo la siguiente
         f=`echo $f | cut -d"." -f 1`
```

```
        if [ -f ${f}.org ]
then
  echo "Siguiente $f archivo..."
  continue  # leo la siguiente y salto el comando cp
fi
        # no tenemos el archivo .org entonces lo copio
        cp $f $f.org
  done
```

En muchas ocasiones, puede querer solicitar al usuario alguna información, y existen varias maneras para hacer esto:

```
#!/bin/bash
echo Por favor, introduzca su nombre
read nombre
echo "Hola $nombre!"
```

Como variante, se pueden obtener múltiples valores con `read`:

```
#!/bin/bash
echo Por favor, introduzca su nombre y primer apellido
read NO AP
echo "Hola $AP, $NO!"
```

Si hacemos `echo 10 + 10` y esperabais ver 20 quedaréis desilusionados, la forma de evaluación directa será con `echo $((10+10))` o también `echo $[10+10]` y si necesitáis operaciones como fracciones u otras podéis utilizar `bc`, ya que lo anterior solo sirve para números enteros. Por ejemplo, `echo $[3/4]` dará 0 pero **echo 3/4|bc -l** sí que funcionará. También recordemos el uso del `let`, por ejemplo, `let a=75*2` asignará 150 a la variable a. Un ejemplo más:

```
RS@debian:~$ echo $date
20042011
RS@debian:~$ echo $((date++))
20042011
RS@debian:~$ echo $date
20042012
```

Es decir, como operadores podemos utilizar:

- VAR++ VAR-- variable post-incremento y post-decremento.
- ++VAR –VAR ídem anterior pero antes.
- + - operadores unarios.
- ! ~ negación lógica y negación en *strings*.
- ** exponente.
- / + - % operaciones.

- << >> desplazamiento izquierda y derecha.
- <= >= < > comparación.
- == != igualdad y diferencia.
- & ^ | operaciones and, or exclusivo y or.
- && || operaciones and y or lógicos.
- expr ? expr : expr evaluación condicional.
- = *= /= %= += -= <<= >>= &= ^= |= operaciones implícitas.
- , separador entre expresiones.

Para capturar la salida de un comando, es decir, el resultado de la ejecución, podemos utilizar el nombre del comando entre comilla hacia la izquierda (` `).

```
#!/bin/bash
A=`ls`
for b in $A ;
do
file $b
done
```

Para la comparación tenemos las siguientes opciones:

- *s1 = s2* verdadero si s1 coincide con s2.
- *s1 != s2* verdadero si s1 no coincide con s2.
- *s1 < s2* verdadero si s1 es alfabéticamente anterior a s2.
- *s1 > s2* verdadero si s1 es alfabéticamente posterior a s2.
- *-n s1* s1 no es nulo (contiene uno o más caracteres).
- *-z s1* s1 es nulo.

Por ejemplo, se debe ir con cuidado porque si S1 o S2 están vacíos, nos dará un error de interpretación (*parse*):

```
#!/bin/bash
S1='cadena'
S2='Cadena'
if [ $S1!=$S2 ];
then
echo "S1('$S1') no es igual a S2('$S2')"
fi
if [ $S1=$S1 ];
then
echo "S1('$S1') es igual a S1('$S1')"
fi
```

En cuanto a los operadores aritméticos y relacionales, podemos utilizar:

a) Operadores aritméticos:

- + (adición).
- - (sustracción).
- * (producto).
- / (división).
- % (módulo).

b) Operadores relacionales:

- -lt (<).
- -gt (>).
- -le (<=).
- -ge (>=).
- -eq (==).
- -ne (!=).

Hagamos un ejemplo de *script* que nos permitirá renombrar ficheros S1 con una serie de parámetros (prefijo o sufijos) de acuerdo a los argumentos. El modo será p [prefijo] ficheros... o si no, s [sufijo] ficheros.. o también r [patrón-antiguo] [patrón-nuevo] ficheros.. (y como siempre decimos, los *scripts* son mejorables):

```
#!/bin/bash -x
# renom: renombra múltiples ficheros de acuerdo con ciertas reglas
# Basado en original de F. Hudson Enero - 2000
# comprueba si deseo renombrar con prefijo y dejo solo los nombres de
# archivos
if [ $1 = p ]; then
   prefijo=$2 ; shift ; shift

   # si no hay entradas de archivos termino.
   if [ "$1" = '' ]; then
      echo "no se especificaron ficheros"
      exit 0
   fi
   # Interación para renombrar
   for fichero in $*
      do
      mv ${fichero} $prefijo$fichero
      done
   exit 0
fi
# comprueba si deseo renombrar con prefijo y dejo solo los nombres de
# archivos
if [ $1 = s ]; then
   sufijo=$2 ; shift ; shift
   if [ "$1" = '' ]; then
      echo "no se especificaron ficheros"
```

```
      exit 0
   fi
   for fichero in $*
      do
      mv ${fichero} $fichero$sufijo
      done
   exit 0
fi
# comprueba si es una sustitución de patrones
if [ $1 = r ]; then
shift
# se ha incluido esto como medida de seguridad
   if [ $# -lt 3 ] ; then
      echo "uso: renom r [expresión] [sustituto] ficheros... "
      exit 0
   fi
   VIEJO=$1 ; NUEVO=$2 ; shift ; shift
   for fichero in $*
      do
      nuevo=`echo ${fichero} | sed s/${VIEJO}/${NUEVO}/g`
      mv ${fichero} $nuevo
      done
   exit 0
fi
# si no le muestro la ayuda
echo "uso:"
echo " renom p [prefijo] ficheros.."
echo " renom s [sufijo] ficheros.."
echo " renom r [patrón-antiguo] [patrón-nuevo] ficheros.."
exit 0
```

Especial atención se debe prestar a las "", '', `` {}etc., por ejemplo:

```
RS@debian:~$ date=20042010
RS@debian:~$ echo $date
20042010
RS@debian:~$ echo \$date
$date
RS@debian:~$ echo '$date'
$date
RS@debian:~$ echo "$date"
20042010
RS@debian:~$ echo "`date`"
Sat Apr 17 14:35:03 EDT 2010
RS@debian:~$ echo "lo que se me ocurre es: \"Estoy cansado\""
lo que se me ocurre es: "Estoy cansado"
RS@debian:~$ echo "\"
```

```
>
(espera más entradas: hacer Crtl-C)
remo@debian:~$ echo "\\"
\
RS@debian:~$ echo grande{cit,much,poc,ningun}o
grandecito grandemucho grandepoco grandeninguno
```

Una combinación de entrada de teclado y operaciones/expresiones para calcular un año bisiesto:

```
#!/bin/bash
clear;
echo "Entre el año a verificar (4 digits), y después [ENTER]:"
read year
if (( ("$year" % 400) == "0" )) || (( ("$year" % 4 == "0") \
&& ("$year" % 100 !="0") )); then
   echo "$year es bisiesto."
else
   echo "$year este año NO es bisiesto."
fi
```

Un comando interesante combinado de `read` con delimitador en una expresión (está puesto todo en una línea y por eso la sintaxis está separada por ;). El resultado será "interesante" (nos mostrará todos los *string* dados por el `find` en un línea y sin /:

```
find "$PWD" -name "m*" | while read -d "/" file; do echo $file; done
```

Un aspecto interesante para trabajar co *string* (ver manual de bash: `man bash`) es ${VAR:*OFFSET*:*LENGTH*}.

```
RS@debian:~$ export str="unstring muy largo"
RS@debian:~$ echo $str
unstring muy largo
RS@debian:~$ echo ${str:4}
ring muy largo
RS@debian:~$ echo $str
unstring muy largo
RS@debian:~$ echo ${str:9:3}
muy
RS@debian:~$ echo ${str%largo}
unstring muy
```

La sentencia `trap` nos permitirá capturar una señal (por ejemplo, de teclado) dentro de un script.

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

```
# Para finalizar hacer desde otro terminal kill -9 pid

trap "echo ' Ja!'" SIGINT SIGTERM
echo "El pid es $$"

while : # esto es lo mismo que "while true".
      do
      sleep 10 # El script no hace nada.
done
```

2.6. Filtros: Grep

El grep es un filtro de patrones muy útil y versátil, a continuación podemos ver algunos ejemplos:

```
RS@debian:~$ grep root /etc/passwd   busca patrón root
root:x:0:0:root:/root:/bin/bash

RS@debian:~$ grep -n root /etc/passwd   además numera las líneas
1:root:x:0:0:root:/root:/bin/bash

RS@debian:~$ grep -v bash /etc/passwd | grep -v nologin

   lista los que no tienen el patrón y bash ni el patrón nologin
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
sync:x:4:65534:sync:/bin:/bin/sync
...

RS@debian:~$ grep -c false /etc/passwd   cuenta los que tienen false
7
RS@debian:~$ grep -i ps ~/.bash* | grep -v history
   busca patrón ps en todos los archivos que comienzan por .bash en el directorio home
   excluyendo los que el archivo contenga history
/home/RS/.bashrc:[ -z "$PS1" ] &amp;amp;&amp;amp; return
/home/RS/.bashrc:export HISTCONTROL=$HISTCONTROL${HISTCONTROL+,}ignoredups
/home/RS/.bashrc:# ... or force ignoredups and ignorespace
...

RS@debian:~$ grep ^root /etc/passwd   busca a inicio de línea
root:x:0:0:root:/root:/bin/bash

RS@debian:~$ grep false$ /etc/passwd   busca a final de línea
Debian-exim:x:101:105::/var/spool/exim4:/bin/false
statd:x:102:65534::/var/lib/nfs:/bin/false
```

```
RS@debian:~$ grep -w / /etc/fstab   busca /
/dev/hda1 / ext3 errors=remount-ro 0 1

RS@debian:~$ grep [yf] /etc/group   busca y o Y
sys:x:3:
tty:x:5:
....

RS@debian:~$ grep '\<c...h\>' /usr/share/dict/words   busca comenzando por c y terminando
                                        por h con un máximo de tres.
catch
cinch
cinch's
....

RS@debian:~$ grep '\<c.*h\>' /usr/share/dict/words   busca comenzando por c y terminando
                                        por h todas.
caddish
calabash
...
```

2.7. Filtros: Awk

Awk, o también la implementación de GNU gawk, es un comando que acepta un lenguaje de programación diseñado para procesar datos basados en texto, ya sean ficheros o flujos de datos, y ha sido la inspiración de Larry Wall para escribir Perl. Su sintaxis más común es `awk 'programa' archivos` ... y donde programa puede ser: `patrón {acción} patrón {acción}...`, awk lee la entrada de archivos un renglón a la vez. Cada renglón se compara con cada patrón en orden; para cada patrón que concuerde con el renglón se efectúa la acción correspondiente. Un ejemplo pregunta por el primer campo si es *root* de cada línea del */etc/passwd* y la imprime considerando como separador de campos el ":" con -F:, en el segundo ejemplo reemplaza el primer campo por *root* e imprime el resultado cambiado (en el primer comando utilizamos '= =' en el segundo solo un '=').

```
RS@debian:~$ awk -F: '$1=="root" {print}' /etc/passwd
root:x:0:0:root:/root:/bin/bash



RS@debian:~$ awk -F: '$1="root" {print}' /etc/passwd
root x 0 0 root /root /bin/bash
root x 1 1 daemon /usr/sbin /bin/sh
root x 2 2 bin /bin /bin/sh
root x 3 3 sys /dev /bin/sh
root x 4 65534 sync /bin /bin/sync
root x 5 60 games /usr/games /bin/sh
```

```
root x 6 12 man /var/cache/man /bin/sh
```

El awk divide automáticamente la entrada de líneas en campos, es decir, cadena de caracteres que no sean blancos separados por blancos o tabuladores, por ejemplo, el `who` tiene 5 campos y `awk` nos permitirá filtrar cada uno de estos campos.

```
RS@debian:~$ who
RS tty7 2010-04-17 05:49 (:0)
RS pts/0 2010-04-17 05:50 (:0.0)
RS pts/1 2010-04-17 16:46 (:0.0)
remo@debian:~$ who | awk '{print $4}'
05:49
05:50
16:46
```

El awk llama a estos campos $1 $2 ... $NF donde NF es una variable igual al número de campos (en este caso NF=5). Por ejemplo:

```
ls -al | awk '{print NR, $0}   Agrega números de entradas (la variable NR cuenta el
                                    número de líneas, $0 es la línea entera.
awk '{printf "%4d %s\n", NR, $0}'  significa un número decimal (NR) un string ($0)
                                    y un new line
awk -F: '$2 == "" ' /etc/passwd   Dará los usuarios que en el archivo de passwd no
                                    tengan puesto la contraseña
```

El patrón puede escribirse de varias formas:

```
$2 == "" si el segundo campo es vacío
$2 ~ /^$/ si el segundo campo coincide con la cadena vacía
$2 !~ /./ si el segundo campo no concuerda con ningún carácter (! es negación)
length($2) == si la longitud del segundo campo es 0, length es una función interna del awk
~ indica coincidencia con una expresión
!~ significa los contrario (no coincidencia)
NF % 2 != 0 muestra el renglón solo si hay un número par de campos
awk 'lenght ($0) 32 {print "Línea", NR, "larga", substr($0,1,30)} ' /etc/passwd
   evalúa las líneas de /etc/passwd y genera un substring de esta
```

Existen dos patrones especiales BEGIN y END. Las acciones BEGIN se realizan antes que el primer renglón se haya leído; puede usarse para inicializar las variables, imprimir encabezados o posicionar separadores de un campo asignándoselos a la variable FS por ejemplo:

```
awk 'BEGIN {FS = ":"} $2 == "" ' /etc/passwd   igual que el ejemplo de antes
awk 'END { print NR } ' /etc/passwd   imprime el número de líneas procesadas al
            final de la lectura del último renglón.
```

El `awk` permite operaciones numéricas en forma de columnas, por ejemplo, para sumar todos los números de la primera columna:

```
{ s=s + 1} END { print s}   y para la suma y el promedio END { print s, s/NR}
```

Las variables se inicializan a cero al declararse, y se declaran al utilizarse, por lo que resulta muy simple (los operadores son los mismos que en C):

```
{ s+=1} END {print s} Por ejemplo, para contar renglones, palabras y caracteres:
{ nc += length($0) +1 nw +=NF } END {print NR, nw, nc}
```

Existen variables predefinidas en el awk:

- FILENAME nombre del archivo actual.
- FS carácter delimitador del campo.
- NF número de campos del registro de entrada.
- NR número del registro del entrada.
- OFMT formato de salida para números (%g default).
- OFS cadena separadora de campo en la salida (blanco por default).
- ORS cadena separadora de registro de salida (*new line* por default).
- RS cadena separador de registro de entrada (*new line* por default).

Operadores (ídem C):

- = += -= *= /= %= || && ! >> >>= << <<= == != ~ !~
- + - * / %
- ++ --

Funciones predefinidas en awk: cos(), exp(), getline(), index(), int(), length(), log(), sin(), split(), sprintf(), substr().

Además soporta dentro de acción sentencias de control con la sintaxis similar a C:

```
if (condición)
proposición1
else
proposición2

for (exp1;condición;exp2)

while (condición) {
proposición
expr2
}
```

```
continue   sigue, evalúa la condición nuevamente
break   rompe la condición
next   lee la siguiente línea de entrada
exit   salta a END
```

También el `awk` maneja arreglos, por ejemplo, para hacer un `head` de */etc/passwd*:

```
awk '{ line[NR] = $0} \
END { for (i=NR; i>2; i--) print line[i]} ' /etc/passwd
```

Otro ejemplo:

```
awk 'BEGIN { print "Usuario UID Shell\n------- --- -----" } $3 >= 500 { print $1, $3,
 $7 | "sort -r"}' FS=":" /etc/passwd
Usuario UID Shell
------- --- -----
RS 1001 /bin/bash
nobody 65534 /bin/sh
debian 1000 /bin/bash
RS@debian:~$ ls -al | awk '
BEGIN { print "File\t\t\tOwner" }
{ print $8, "\t\t\t", \
$3}
END { print "done"}
'
File Owner
. RS
.. root
.bash_history RS
.bash_logout RS
.bashrc RS
.config RS
...
```

2.8. Ejemplos complementarios

1) Un ejemplo completo para borrar los logs (borra */var/log/wtmp* y se queda con 50 líneas –o las que pase el usuario en línea de comando– de */var/log/messages*).

```
   #!/bin/bash
   #Variables
   LOG_DIR=/var/log
   ROOT_UID=0 # solo lo pueden ejecutar el root
   LINES=50 # Líneas por defecto.
   E_XCD=86 # NO me puedo cambiar de directorio
```

```
E_NOTROOT=87 # Salida de No-root error.

if [ "$UID" -ne "$ROOT_UID" ] # Soy root?
   then
   echo "Debe ser root. Lo siento."
   exit $E_NOTROOT
   fi

if [ -n "$1" ] # Número de líneas a preservar?
   then
   lines=$1
   else
   lines=$LINES # valor por defecto.
   fi

cd $LOG_DIR

if [ `pwd` != "$LOG_DIR" ]
# también puede ser if [ "$PWD" != "$LOG_DIR" ]
   then
   echo "No puedo ir a $LOG_DIR."
   exit $E_XCD
   fi #

# Otra forma de hacerlo sería :
# cd /var/log || {
# echo "No puedo !" >&2
# exit $E_XCD;
# }

tail -n $lines messages > mesg.temp # Salvo en temporal
mv mesg.temp messages # Muevo.

cat /dev/null > wtmp #Borro wtmp.
echo "Logs Borrados."
exit 0
# Un cero indica que todo Ok.
```

2) Utilización del `expr`:

```
#!/bin/bash
echo "Ariméticos"
echo
a=`expr 5 + 3`
echo "5 + 3 = $a"
a=`expr $a + 1`
echo
```

```
echo "a + 1 = $a"
echo "(incremento)"
a=`expr 5 % 3`
# módulo
echo
echo "5 mod 3 = $a"

# Lógicos
# 1 si true, 0 si false,
#+ opuesto a normal Bash convention.

echo "Lógicos"
x=24
y=25
b=`expr $x = $y` # por igual.
echo "b = $b" # 0 ( $x -ne $y )
a=3
b=`expr $a \> 10`
echo 'b=`expr $a \> 10`, por lo cual...'
echo "If a > 10, b = 0 (false)"
echo "b = $b" # 0 ( 3 ! -gt 10 )
b=`expr $a \< 10`
echo "If a < 10, b = 1 (true)"
echo "b = $b" # 1 ( 3 -lt 10 )
echo
# Cuidado con los operadores de escape \.
b=`expr $a \<= 3`
echo "If a <= 3, b = 1 (true)"
echo "b = $b" # 1 ( 3 -le 3 )
# Se utiliza un operador "\>=" operator (greater than or equal to).

echo "String"
echo
a=1234zipper43231
echo "String base \"$a\"."
b=`expr length $a`
echo "Long. de \"$a\" es $b."
# index: posición del primer caracter que satisface en el string
#
b=`expr index $a 23`
echo "Posición numérica del \"2\" en \"$a\" es \"$b\"."
# substr: extract substring, starting position &amp;amp; length specified
b=`expr substr $a 2 6`
echo "Substring de \"$a\", comenzando en 2,\
y de 6 chars es \"$b\"."
# Using Regular Expressions ...
b=`expr match "$a" '[0-9]*'` # contador numérico.
```

```
    echo Número de digitos de \"$a\" es $b.
    b=`expr match "$a" '\([0-9]*\)'` # cuidado los caracteres de escape
    echo "Los dígitos de \"$a\" son \"$b\"."
    exit 0
```

3) Un ejemplo que muestra diferentes variables (con lectura desde teclado), sentencias y ejecución de comandos:

```
    #!/bin/bash
    echo -n "Introducir el % de CPU del proceso que más consume a supervisar, el intervalo[s],
             Nro  repeticiones [0=siempre]: "
    read lim t in
    while true
    do
            A=`ps -e -o pcpu,pid,comm | sort -k1 -n -r | head -1`
            load=`echo $A | awk '{ print $1 }'`
            load=${load%.*}
            pid=`echo $A | awk '{print $2 }'`

            if [ $load -gt $lim ]
            then
               # verifico cada t segundos la CPU load
               sleep $t
               B=`ps -e -o pcpu,pid,comm | sort -k1 -n -r | head -1`
               load2=`echo $B | awk '{ print $1 }'`
               load2=${load2%.*}
               pid2=`echo $B | awk '{print $2 }'`
               name2=`echo $B | awk '{print $3 }'`
               [ $load2 -gt $lim ] && [ $pid = $pid2 ] && echo $load2, $pid2, $name2
      let in--
               if [ $in == 0 ]; then  echo "Fin"; exit 0; fi
  fi
    done
```

4) Muchas veces en los *shell scripts* es necesario hacer un menú para que el usuario escoja una opción. En el primer código se debe instalar el paquete **dialog (apt-get install dialog)**.

a. Menú con el comando dialog:

```
    #!/bin/bash
    # original http://serverfault.com/questions/144939/multi-select-menu-in-bash-script

    cmd=(dialog --separate-output --checklist "Select options:" 22 76 16)
    options=(1 "Opción 1" off    # Si se desea cualquiera
                                            # se puede poner en on
             2 "Opción 2" off
```

```
            3 "Opción 3" off
            4 "Opción 4" off)
choices=$("${cmd[@]}" "${options[@]}" 2>&1 >/dev/tty)
clear
for choice in $choices
do
    case $choice in
        1)
            echo "Primera opción"
            ;;
        2)
            echo "Segunda Opción"
            ;;
        3)
            echo "Tercera Opción"
            ;;
        4)
            echo "Cuarta Opción"
            ;;
    esac
done
```

b. Menú solo con sentencias:

```
#!/bin/bash
# Basado en original de D.Williamson
# http://serverfault.com/questions/144939/multi-select-menu-in-bash-script

toogle () {
    local choice=$1
    if [[ ${opts[choice]} ]]
    then
        opts[choice]=
    else
        opts[choice]="<-"
    fi
}

PS3='Seleccionar opción: '
while :
do
    clear
    options=("Opción A ${opts[1]}" "Opción B ${opts[2]}" "Opción C ${opts[3]}" "Salir")
    select opt in "${options[@]}"
    do
        case $opt in
            "Opción A ${opts[1]}")
```

```
                    toogle 1
                    break
                    ;;
               "Opción B ${opts[2]}")
                    toogle 2
                    break
                    ;;
               "Opción C ${opts[3]}")
                    toogle 3
                    break
                    ;;
                "Salir")
                    break 2
                    ;;
                *) printf '%s\n' 'Opción no válida';;
            esac
        done
    done

    printf '%s\n' 'Opciones selecionadas:'
    for opt in "${!opts[@]}"
    do
        if [[ ${opts[opt]} ]]
        then
            printf '%s\n' "Opción $opt"
        fi
    done
```

5) En ocasiones es necesario filtrar una serie de argumentos pasados en la línea de comandos y `getop` o `getopts` pueden ayudar. Estos comandos son utilizados para procesar y validar argumentos de un *shell script* y son similares pero no idénticos. Además, la funcionalidad puede variar en función de su implementación, por lo cual es necesario leer las páginas del manual con detalle.

En el primer ejemplo utilizaremos `getopts` y filtrará `nombre_cmd -a valor`:

```
    #!/bin/bash
    # Más opciones en
    while getopts ":a:" opt; do
      case $opt in
        a)
            echo "-a fue introducido, Parámetro: $OPTARG" >&2
            ;;
        \?)
           echo "opción in´valida: -$OPTARG" >&2
           exit 1
           ;;
```

```
        :)
           echo "Opción -$OPTARG requiere un argumento." >&2
           exit 1
           ;;
      esac
    done
```

Y con `getopt`:

```
    #!/bin/bash
    args=`getopt abc: $*`
    if test $? != 0
         then
             echo 'Uso: -a -b -c valor'
             exit 1
    fi
    set -- $args
    for i
    do
      case "$i" in
              -c) shift;echo "Opción c como argumento con $1";shift;;
              -a) shift;echo "Opción a como argumento";;
              -b) shift;echo "Opción b como argumento";;
      esac
    done
```

Actividades

1. Cread un script interactivo que calcule el número de días entre dos fechas introducidas por el usuario en el siguiente formato día/mes/año. También deberéis calcular el número de horas si el usuario introduce hora de inicio y hora de final en el siguiente formato hh:mm. Si introduce solo una hora, se calculará hasta las 23:59 del día dado como final del período a calcular.

2. Cread un script que se haga una copia (recursiva) de los archivos en */etc*, de modo que un administrador del sistema pueda modificar los archivos de inicio sin temor.

3. Escribid un script llamado *homebackup* que automatice el `tar` con las opciones correctas y en el directorio */var/backups* para hacer una copia de seguridad del directorio principal del usuario con las siguientes condiciones:
a. Haced un test del número de argumentos. El script debe ejecutarse sin argumentos y si hay, debe imprimir un mensaje de uso.
b. Determinad si el directorio de copias de seguridad tiene suficiente espacio libre para almacenar la copia de seguridad.
c. Preguntad al usuario si desea una copia completa (todo) o una copia incremental (solo aquellos archivos que hayan cambiado). Si el usuario no tiene una copia de seguridad completa, se deberá hacer, y en caso de una copia de seguridad incremental, solo hacerlo si la copia de seguridad completa no tiene más de una semana.
d. Comprimid la copia de seguridad utilizando cualquier herramienta de compresión.
e. Informad al usuario que se hará en cada momento, ya que esto puede tardar algún tiempo y así se evitará que el usuario se ponga nervioso si la salida no aparece en la pantalla.
f. Imprimid un mensaje informando al usuario sobre el tamaño de la copia de seguridad sin comprimir y comprimida, el número de archivos guardados/actualizados y el número de directorios guardados/actualizados.

4. Escribid un script que ejecute un simple navegador web (en modo texto), utilizando `wget` y `links -dump` para mostrar las páginas HTML en un terminal.

El usuario tiene 3 opciones: introducir una dirección URL, entrar **b** para retroceder (back) y **q** para salir. Las 10 últimas URL introducidas por el usuario se almacenan en una matriz, desde donde el usuario puede restaurar la URL utilizando la funcionalidad **b** (back).

Bibliografía

Es recomendable complementar la lectura en la que se ha basado parte de esta documentación y sobre todo para aspectos avanzados, ya que aquí solo se ha visto un breve resumen de lo esencial:

Barnett, Bruce. "AWK" http://www.grymoire.com/Unix/Awk.html. © General Electric Company.

"Bash Reference Manual" http://www.gnu.org/software/bash/manual/bashref.html.

Cooper, Mendel. "Advanced Bash-Scripting Guide" http://tldp.org/LDP/abs/html/index.html.

Garrels, Machtelt. "Bash Guide for Beginners" http://tldp.org/LDP/Bash-Beginners-Guide/html/index.html.

Mike, G.. "Programación en BASH - COMO de introducción" http://es.tldp.org/COMO-INSFLUG/COMOs/Bash-Prog-Intro-COMO/.

Robbins, Arnold. "UNIX in a Nutshell" (3.ª ed.) http://oreilly.com/catalog/unixnut3/chapter/ch11.html.

"The GNU awk programming language" http://tldp.org/LDP/Bash-Beginners-Guide/html/chap_06.html.

"The Shell Scripting Tutorial" http://bash.cyberciti.biz/guide/Main_Page.

van Vugt, Sander. "Beginning Ubuntu Server Administration: From Novice to Professional" (Disponible en Safari Books).