Práctica 1: **Shell Scripting**

Sistemas Operativos

Febrero del 2017

Índice

1 Introducción

1.1 Unix y GNU/Linux

En ésta sección se introducen de forma breve el concepto de sistema operativo. Se verá con detalle en clase de teoría.

Un sistema **sistema operativo** es un programa o conjunto de programas que maneja y administra los recursos de un ordenador, y cuyas funciones son: (i) gestión del tiempo de CPU, (ii) control del uso de la memoria, y (iii) control de operaciones de entrada/salida.

1.2 Editores de texto

Existen multitud de editores de texto que se utilizan por consola. Algunos ejemplos son `vi` o `emacs`. Si bien no es imprescindible, **sí es muy recomendable** usar uno de estos editores con soltura. La razón es que algunas veces os será necesario editar ficheros (de configuración, por ejemplo) sin disponer de una interficie gráfica. Esto puede ocurrir si os conectáis de forma remota a otro ordenador, como se realizará en la práctica 3.

En Internet hay multitud de tutoriales de estos u otros editores de texto. Un sencillo manual de `vi` se puede encontrar en: http://www.tutorialesytrucos.com/tutoriales/tutorial-unix-linux/26-tutorial-bsico-del-editor-vi.html

2 El intérprete de comandos

El *intérprete de comandos*, o simplemente la *shell*, es una pieza software implementada en la mayoría de los sistemas operativos, y su función es proporcionar una interfaz de usuario para interpretar las órdenes introducidas por éste y acceder de esta manera a los distintos servicios proporcionados por el sistema operativo.

En general, distinguimos entre dos tipos de shell: *interfaz por línea de comandos* (o CLI: command-line interface) e *interfaz de usuario gráfica* (o GUI: graphical user interface).

En los sistemas operativos Linux, se pueden utilizar distintas shells CLI, como por ejemplo *sh*, *ksh* o *ash*. Sin embargo, probablemente la más conocida y utilizada en la actualidad es ***bash*** (acrónimo de Bourne-Again Shell), y es el intérprete de comandos por defecto en la mayoría de distribuciones de Linux, así como en OSX. En cuanto a GUIs, algunas conocidas son GNOME, KDE o LXDE.

En Windows, la CLI es el intérprete de comandos nativo de MS-DOS, y la GUI es el *Explorador de Windows* y algunos complementos asociados (el escritorio, el menú de inicio y la barra de tareas). En la actualidad en Windows la CLI ha sido mejorada con la PowerShell.

Si bien una GUI puede ser útil para diferentes tareas, las CLIs (en concreto ***bash***) es la herramienta aconsejable para otras muchas tareas, desde controlar servidores remotos hasta realizar operaciones en nuestra propia máquina, con especial énfasis en aquellas operaciones nativas para interactuar con el sistema operativo. En esta práctica nos centraremos en el uso de ***bash***.

Ejemplo 1. Abre un intérprete de comandos. ¿Qué información aparece en pantalla? Habitualmente, encontrarás la secuencia `user@hostname:dir`, seguida del carácter $ (y finalmente de un cursor parpadeando). Por ejemplo, la secuencia `jesus@lab:~` nos indica que el usuario actual es `jesus`, la máquina para la que el bash interpreta los comando se llama `lab` y el directorio actual

es el *home* del usuario `jesus`. La forma en la que se muestra esta secuencia se puede cambiar en el fichero `~/.bashrc` o `~/.profile`.
El *hostname* no tiene porque coincidir con la máquina en la que estamos trabajando. Eso puede ocurrir en conexiones remotas con `ssh`, tal y como veremos en la práctica 3.
El símbolo $ nos indica que el usuario actual es un usuario regular. En caso de superusuario (usuario *root*), se muestra el símbolo #.

2.1 Algunas teclas interesantes

Para interactuar con el intérprete de comandos, hay una serie de teclas que son indispensables conocer. Otras, en cambio, no son imprescindibles pero nos harán la vida más fácil:

- La tecla *enter*: Confirma la ejecución del comando escrito en el terminal.
- Las teclas ↑ y ↓ : Se usan para navegar en el historial de comandos introducidos (éstos quedan guardados en el fichero `~/.bash_history`).
- La tecla *tab*: Autocompletado (según contexto).
- El carácter ';' : Concatena varios comandos en una sola línea.
- El carácter '&' : Ejecuta el comando en background.
- El carácter '*' : Equivalente a cero o más caracteres en el nombre de los archivos.
- El carácter '?' : Equivalente a exactamente un carácter en el nombre de los archivos.
- El conjunto "[lista]" : Equivalente a exactamente un carácter del conjunto *lista*, en el nombre de los archivos.
- La combinación "Ctrl + C" : Interrumpe/Mata el proceso (aplicación) que se está ejecutando[1].

Ejemplo 2. Veamos algunos ejemplos con usando las teclas anteriores (más adelante veremos otros ejemplos con ellas).

1. Ejecuta el comando `hostname`, que muestra el nombre de la máquina.
2. Ejecuta el comando `hostinfo`, que muestra información sobre la máquina.
3. Ejecuta el comando `who`, que muestra información de los usuario conectados.
4. Usa las teclas ↑ y ↓ para navegar por el historial.
5. Ejecuta el comando `hostname; hostinfo`. ¿Qué ocurre en este caso?
6. Escribe en el prompt "`hostna`" y pulsa la tecla *tab*. ¿Qué ocurre?
7. Ahora escribe "`host`" y pulsa la tecla *tab*. ¿Qué ocurre en este caso?
8. Ejecuta el comando `sleep 2`, que *duerme* el proceso durante 2 segundos. ¿Qué ocurre en el prompt?

[1] Técnicamente diremos que enviamos la señal SIGINT al proceso. Eso se verá en teoría

9. Ahora ejecuta el comando `sleep 2 &`. ¿Qué ocurre en este caso?

10. Ejecuta el comando `sleep 100`. Interrumpe esta última orden para volver tener el control del prompt.

2.2 El sistema de ayuda: el comando *man*

El sistema de ayuda se implementa de una forma estándar, y provee información sobre cada comando. Para invocarlo:

```
man [options] command
```

donde `command` es el comando del cual se requiere la información.

Ejemplo 3. ¿Cuáles son las opciones del comando `ls`? Para ello teclea `man ls`. Usa las teclas ↑ y ↓ para desplazarte por el contenido, y usa la tecla '`q`' para salir y volver al *prompt*. Hay además otras teclas equivalentes al editor de texto `vi`. Por ejemplo, utiliza la tecla `G` para desplazarte al final del fichero, la `g` para desplazarte al inicio del fichero o la tecla '`/`' para realizar una búsqueda de una cadena.

La información de cada comando se organiza en secciones. Cada sección se almacena en un subdirectorio diferente en el disco. Se puede acceder a cada una de las secciones mediante la opción `section` del comando `man`.

Ejemplo 4. El comando `man 1 printf` hace referencia a la función `printf` de *bash* mientras que `man 3 printf` hace referencia a la implementación de esa función en C por la librería `stdio.h`.

Otra forma de invocar la ayuda es usando palabras clave de búsqueda, `man -k key`, donde `key` es la palabra a buscar.

Ejemplo 5. Imagina que quieres compilar un fichero `java` pero no recuerdas el nombre del compilador de Java. Usa el comando anterior para averiguarlo. Ejecutando `man -k java` se obtiene la respuesta: *javac(1) - Java compiler*. El comando es, por tanto, `javac`.

3 El sistema de ficheros

El sistema de ficheros son las estructuras lógicas así como los métodos para gestionarlas que utiliza el sistema operativo en la gestión y organización de los ficheros dentro de un soporte físico. Estos métodos que utiliza el sistema operativo para pasar del soporte físico (disco duro) al soporte lógico (particiones), y viceversa, lo hace de una forma transparente al usuario final (y además, no es parte del temario de esta asignatura ...).

En el caso de los sistemas Linux (como en muchos otros), el sistema de ficheros está organizado jerárquicamente en forma de árbol. Esto significa que los archivos son agrupados en directorios, los cuales también pueden estar contenidos en otros directorios (siendo subdirectorios de éste), hasta un directorio raíz que es "/". A partir de él, se organizan los directorios más importantes:

- /bin : programas que estarán disponibles en modos de ejecución restringidos (bash, cat, ls, ps, ...).

- /boot : kernel (núcleo del sistema) y ficheros de arranque.
- /dev : dispositivos externos.
- /etc : archivos de configuración.
- /home : usuarios del sistema.
- /lib : librerías indispensables y otros módulos.
- /proc : directorio virtual para el intercambio de ficheros.
- /root : administradores (superusuarios, o usuarios *root*) del sistema.
- /sbin : programas que estarán disponibles para usuarios *root* en modos de ejecución restringidos.
- /tmp : archivos temporales (el directorio es vaciado en el arranque).
- /usr : programas accesibles a todos los usuarios (y algunos ficheros no modificables que usan estos programas).
- /var : archivos que son modificados frecuentemente por otros programas.

Ejemplo 6. Muestra el contenido del directorio raíz ejecutando el comando:

```
ls /
```

¿Qué ocurre al mostrar el contenido del `/home`? Para ello, ejecuta el comando:

```
ls /home
```

La **ruta** es la cadena de texto única que identifica a cada fichero. Se construye concatenando los nombres de los directorios, en orden jerárquico, hasta llegar al directorio que contiene el fichero, y concatenando finalmente el nombre del fichero correspondiente.

3.1 Rutas relativas y absolutas

Las rutas se pueden especificar de forma relativa o de forma absoluta.

Cuando hablamos de **rutas absolutas**, nos referimos a la concatenación completa de todos los directorios y subdirectorios desde "/" hasta el fichero al que apunte la ruta correspondiente.

En el caso de **rutas relativas**, se hace únicamente referencia a la jerarquía de subdirectorios a partir del directorio actual.

Existen también algunos caracteres reservados:

- Carácter "." : referencia el directorio actual.
- Cadena ".." : referencia al directorio donde está contenido el directorio actual.
- Carácter ~ : referencia al *home* del usuario.

Ejemplo 7. Hemos iniciado sesión con el usuario `esthercolero` y estamos en su *home* (`/home/esthercolero/`). Veamos los siguientes ejemplos:

- La ruta relativa `~/foo.txt`
  apunta a la ruta absoluta `/home/esthercolero/foo.txt`
- La ruta relativa `./foo2.txt`
  apunta a la ruta absoluta `/home/esthercolero/foo2.txt`
- La ruta relativa `./../esthercolero`
  apunta a la ruta absoluta `/home/esthercolero`
- ¿A qué ruta absoluta apunta la ruta relativa
  `~/.././esthercolero/./././dir1/../`?

4 Comandos para el manejo de ficheros

En esta sección veremos los comandos básicos para el manejo de ficheros. Aquí se explican en su forma básica; pero todos incluyen opciones adicionales que pueden ser consultadas accediendo a la ayuda del comando correspondiente.

4.1 El comando *pwd*

El comando `pwd` imprime el nombre del directorio de trabajo actual.

```
pwd
```

En general, por defecto la shell inicializa su directorio de trabajo actual al *home* del usuario.

4.2 El comando *cd*

El comando `cd` cambia el directorio de trabajo a la ruta especificada por argumentos:

```
cd <path>
```

Cuando se utiliza sin argumentos, cambia el directorio de trabajo al *home* del usuario. Es decir, el comando `cd` es equivalente al comando `cd ~`.

4.3 El comando *mkdir*

El comando `mkdir` crea el directorio especificado por argumentos:

```
mkdir <path>
```

4.4 Los comandos *cp* y *mv*

Los comandos `cp` y `mv` respectivamente copian o mueven el contenido de `<source>` a `<target>`:

```
cp <source> <target>
mv <source> <target>
```

4.5 Los comandos *rm* y *rmdir*

El comando `rm` elimina el fichero especificado por argumentos:

```
rm <path>
```

En el caso de que el fichero especificado en `<path>` sea un directorio, es necesario invocar al comando `rmdir`. Sin embargo, para que su eliminación sea satisfactoria, es necesario que el directorio esté vacío.

```
rmdir <path>
```

4.6 El comando *ls*

El comando `ls` muestra el contenido de la ruta especificada por argumentos:

```
ls <path>
```

En caso de no introducir ningún argumento, este comando muestra el contenido del directorio actual. Es decir, el comando `ls .` es equivalente al comando `ls`

Ejemplo 8. Razona el comportamiento de ejecutar los siguientes comandos (en dicho orden):

1. `cd`
2. `cp foo.txt myfile.pdf`
3. `mv foo2.txt otherfile.jpg`
4. `mkdir ejemplo`
5. `cd ejemplo`
6. `cp ../myfile.pdf .`
7. `cd ..  ; mv otherfile.jpg ./foo2.txt`
8. `rmdir ejemplo`
9. `rm ejemplo/myfile.pdf ; rmdir ejemplo`

5 Permisos, usuarios y grupos

Una parte muy importante del sistema operativo es la gestión de los permisos de los ficheros. En los sistemas GNU/Linux, se establecen 3 tipos de permisos (lectura, escritura y ejecución) en tres categorías (el propietario del fichero, grupo del propietario del fichero, y cualquier usuario).

Recalcar que los usuarios se organizan en grupos, de forma que un grupo puede contener múltiples usuarios. Además, el SO permite crear nuevos grupos y la gestión de los usuarios en los distintos grupos es sencilla para el administrador del sistema (pero eso no lo veremos en esta práctica ...).

Para ver cómo se organizan los permisos de un fichero, analicemos la siguiente secuencia de 9 caracteres:

```
r w x r - x r - -
```

Cada grupo de tres caracteres representan los permisos de: (i) el usuario propietario (es decir, los permisos `rwx`), (ii) de todos los usuarios del mismo grupo que el propietario (es decir, los permisos `r-x`), y (iii) de cualquier otro usuario (es decir, los permisos `r--`). Estas tres categorías de usuarios las declararemos con los caracteres `u` (*user*), `g` (*group*) y `o` (*others*). Además, el carácter `a` (*all*) nos permitirá referirnos a las tres categorías a la vez.

En cuanto a las ternas de tres caracteres, éstas representan los permisos de lectura `r` (*read*), escritura `w` (*write*) y ejecución `x` (*execute*), del grupo correspondiente. El carácter '-' representa la ausencia del permiso correspondiente.

Ejemplo 9. En el ejemplo anterior, el propietario tiene permisos de lectura, escritura y ejecución; el grupo tiene permisos de lectura y ejecución, y el resto de usuarios sólo tiene permisos de lectura.

Para obtener esta información basta ejecutar el comando `ls -l` (es decir, en formato largo). Cada fichero aparece en una fila. Para cada uno, el primer carácter representa el tipo de fichero (directorio, enlaces, ficheros regulares, ...), y los 9 siguientes representan los permisos. También aparece otra información como el propietario y el grupo, así como el tamaño del fichero.

5.1 El comando *chmod*

El comando `chmod` sirve para cambiar los permisos del fichero pasado por argumentos:

```
chmod <permissions> <file>
```

de forma que `<permissions>` tiene la siguiente sintaxis:

```
<permissions> := <sentence> [,<sentence>]*
<sentence>    := <group> ('+' | '-' | '=' (<type>)+)+
<group>       := 'u' | 'g' | 'o' | 'a'
<type>        := 'r' | 'w' | 'x'
```

Hay que remarcar que el anterior comando no modifica aquellos permisos a los que no se haga referencia.

Ejemplo 10. El siguiente comando asigna para el propietario permisos de lectura y ejecución y se los quita para la escritura, para el grupo asigna permisos de ejecución, y para los otros les deja únicamente permisos de lectura (es decir, en caso de tener previamente permisos de escritura y ejecución, se los quita):

```
chmod u+rx-w,g+x,o=r foo
```

Otra forma de usar el comando `chmod` es introduciendo los permisos en formato octal, de forma que para tripleta de permisos, se calcula un número como suma de:

```
r w x
4 2 1
```

De esta forma, los permisos de lectura+escritura+ejecución tienen un valor de 7, los permisos de lectura+ejecución tienen un valor de 5, los permisos de lectura+escritura tienen un valor de 6, etc...

En el comando `chmod` necesita como argumento un número de tres dígitos, que corresponden con los permisos del usuario, grupo y otros, sucesivamente.

Ejemplo 11. Similar al ejemplo anterior podemos ejecutar el siguiente comando:

```
chmod 514 foo
```

¿Cuál es la diferencia con el ejemplo anterior¿

6 Comandos para visualizar texto por pantalla

6.1 El comando *echo*

El comando `echo` escribe el contenido de los argumentos por la salida estándar:

```
echo <string>
```

Ejemplo 12. Ejecuta `echo "Hola"`

6.2 Los comandos *cat* i *less*

El comando `cat` escribe el contenido del fichero pasado por argumento (por la salida estándar):

```
cat <file>
```

Es interesante también el comando `less`. Este comando es muy útil para ficheros de texto largos. Coge cualquier fichero de texto grande y ejecuta

```
less <file>
```

Puedes utilizar las teclas ↑, ↓, `<espacio>` para poder navegar por el fichero. Igual que con el comando `man`, hay otras teclas equivalentes al editor de texto `vi`: utiliza la tecla `G` para desplazarte al final del fichero, la `g` para desplazarte al inicio del fichero o la tecla '`/`' para realizar una búsqueda de una cadena.

Ejemplo 13. Ejecuta el siguiente comando y busca la cadena "benea" en el fichero. Puedes utilitzar las teclas `n` y `N` para buscar hacia adelante o atrás. El fichero `btowe10.txt` se incluye con la práctica.
`less btowe10.txt`

6.3 Los comandos *head* y *tail*

Los comandos `head` y `tail` respectivamente muestran las primeras/últimas líneas de un fichero (por la salida estándar que por defecto es la pantalla):

```
head <file>
tail <file>
```

Utilizar el comando `man` para obtener detalles sobre el funcionamiento de ambos comandos.

Ejercicio 1. Realiza los siguientes ejercicios con los comandos `head` y `tail` con el fichero `btowe10.txt`.

1. Imprime por pantalla las primeras 15 líneas del fichero.

2. Imprime por pantalla las últimas 20 líneas del fichero.

3. Imprime por pantalla todo el fichero excepto las primeras 50 líneas del fichero.

Los comandos `head` y `tail` pueden ser utilizados para coger una versión reducida de un fichero. En este ejemplo se utilizan técnicas que todavía no se han explicado en teoría y que se utilizaran con más detalle en la pràctica 3.

Ejemplo 14. El siguiente comando redirige el contenido que se imprime por pantalla al fichero `tmp.txt`

```
head -n 20 btowe10.txt > tmp.txt
```

6.4 El comando *clear*

Este comando nos sirve para limpiar la pantalla del terminal.

Ejemplo 15. Escribe el comando:

```
for i in {1..10}; do echo $i; done
```

que imprimirá por pantalla los números del 1 al 10. Tras eso, haz una limpieza del terminal usando el comando `clear`.

7 Variables, condiciones y bucles

Cuando se está usando la shell, a menudo es necesario hacer uso de variables, condiciones y bucles, los cuales nos permiten ejecutar comandos de una forma más apropiada para nuestros propósitos.

En el caso de las variables, distinguiremos entre variables locales (también conocidas como *User Defined Variables*), que son creadas y gestionadas por el usuario, y las variables de entorno (o *System Variables*), que son creadas y gestionadas por el sistema operativo.

7.1 Variables locales

En general, estas variables comienzan por letras minúsculas (para diferenciarlas de las variables de entorno). La forma de definirlas es la siguiente:

```
variableName=value
```

donde `value` es una cadena. Algunos aspectos interesantes a hacer notar de la definición anterior:

- El nombre de la variable `variableName` sólo acepta caracteres alfanuméricos y la barra baja '_'.
- Los nombre de las variables son sensibles a mayúsculas/minúsuclas (*case-sensitive*).
- Los espacios no son admitidos en la definición de una variable.
- Una variable puede recibir el valor `NULL` definiéndola como:
  `var=` o `var=""`

Ejemplo 16. Razona el comportamiento de ejecutar los siguientes comandos (en dicho orden):

1. `a=1`
2. `A=6`
3. `b = 30`

Para usar su valor en la invocación a otros comandos, hay que añadir el carácter '$' antes del nombre de la variable. En este ejemplo vemos como se pueden visualizar valores de variables por pantalla.

Ejemplo 17. ¿Qué diferencia hay entre las siguientes líneas?

1. a=10 ; b=3; c=$a
2. echo a es $a y b $b
3. echo 'a es $a y b $b'
4. echo "a es $a y b $b"
5. c=$a; sleep $c; echo "hello!" ; sleep $c ; echo "bye!"

Dado que el `value` asignado a una variable se interpreta como una cadena, es necesario el uso de comandos como `define`, `let` o `expr`, para realizar operaciones aritméticas.

7.2 Variables de entorno

Estas variables son proporcionadas por el sistema, y su utilidad puede ser distinta dependiendo de cada una de ellas. Por ejemplo, algunas son útiles para no tener que escribir mucho al utilizar un programa; otras son usadas por la propio *shell*, etc...

Aquí listamos **algunas** de las más importantes:

- HOME : Muestra la ruta al *home* del usuario.
- USER : Muestra el nombre del usuario.
- PATH : Muestra el contenido del *path* del usuario. El *path* son los directorios donde la shell buscará ficheros ejecutables.
- SHELL : Muestra la *shell* por defecto del sistema.
- ...

Al igual que las variables locales, las variables de entorno pueden cambiar su valor (con los comandos `set`, `export`). Sin embargo, asignar un valor no adecuado a algunas de estas variables puede crear problemas en el sistema.

Ejemplo 18. Observa el comportamiento de ejecutar los siguientes comandos:

1. `echo $HOME`
2. `echo $PATH`

Ejercicio 2. Imprime el siguiente mensaje:
`Está en el directorio <dir> y su nombre de usuario es <user>.`
donde `<dir>` y `<user>` deben ser correctamente completadas.

7.3 Valor de retorno de las aplicaciones (exit status)

El *exit status* de un comando es el valor que éste devuelve a la *shell*, y que nos indica si la ejecución de dicho comando fue satisfactoria o no.

Ejemplo 19. Crear el siguiente fichero `prueba.c`:

```
#include <string.h>
int main(int argc, char* argv[]){
        if(argc >= 2){
                if(!strcmp(argv[1], "ten")){
                        return 10;
                }else if (!strcmp(argv[1], "twenty")){
                        return 20;
                }
        }
        return 100;
}
```

y compílalo para generar el ejecutable `prueba`:

```
gcc prueba.c -o prueba
```

En la variable `$?` queda almacenado el *exit status* del último comando ejecutado. En la ejecución de una aplicación hace referencia al valor entero devuelto por la función `main`. Observa el resultado de ejecutar los siguientes comandos:

1. `./prueba 10; echo $?`
2. `./prueba 20; echo $?`
3. `./prueba; echo $?`

Se pueden consultar los posibles valores de exit status de un comando mediante el `man`. En general, una ejecución satisfactoria devuelve el valor 0.

Ejemplo 20. El comando `cat foo` devolverá un *exit status* con valor 0 si el fichero `foo` existe; en caso contrario devolverá > 0. ¿Qué devuelven los siguientes comandos?

1. `cat foo; echo $?`
2. `cat prueba.c; echo $?`

7.4 Captura de la salida de un comando

Otra posibilidad muy interesante es guardar todo aquello que un comando imprime por pantalla (técnicamente, todo aquello que imprime por la salida estándard; ya se verá en clase de teoría). Por ejemplo, imagina que necesitamos guardar en alguna variable los ficheros del directorio de trabajo actual. Para guardarlo en una variable, necesitamos ejecutar el comando deseado, `ls`, entre los operadores `$(...)`.

Ejemplo 21. Prueba ejecutar estos comandos:

```
myFiles=$(ls); echo $myFiles
```

7.5 Condiciones

Las condiciones tienen la siguiente sintaxis:

```
if condition
then
       commands....
else
       commands....
fi
```

donde la condición se evalúa a `true` cuando una expresión es cierta o cuando recibe 0 como *exit status* de un comando. En general la condición será una expresión. Distinguimos entre expresiones aritméticas, expresiones sobre cadenas de texto, expresiones sobre ficheros y expresiones lógicas.

7.5.1 Expresiones aritméticas

Para evaluarlas se usan los operadores `x -eq y` (x==y), `-ne` (!=), `-lt` (<), `-le` (<=), `-gt` (>) y `-ge` (>=).

7.5.2 Expresiones sobre strings

Para evaluarlas se usan los operadores `s1 = s2` (mismo valor), `s1 != s2` (distinto valor), `s1` (no definido o no NULL), `-n s1` (existe y no NULL) y `-z s1` (existe y NULL).

7.5.3 Expresiones sobre ficheros

Para evaluarlas se usan los operadores `-s f` (fichero `f` no vacío), `-f f` (`f` existe), `-d d` (`d` es directorio), `-w f` (`f` tiene permisos de escritura), `-r f` (`f` tiene permisos de lectura), y `-x f` (`f` tiene permisos de ejecución).

7.5.4 Expresiones lógicas

En las expresiones, se pueden usar los operadores '!' (NOT), '-a' (AND) y '-o' (OR).

También es posible usar los operadores "&&" (AND) y "||" (OR) para separar expresiones.

Ejemplo 22. Veamos el siguiente ejemplo donde usamos todas las expresiones anteriores. Observar que las **expresiones** deben **escribirse entre corchetes** y con los espacio necesarios.

```
a=5;
if [ $a -le 6 -a $USER == estercolero -a -f foo.txt ]
       && [ -d ejemplo ];
then
       echo "hello!";
fi
```

Veamos aquí otro ejemplo en que se utiliza en exit status de una aplicación

Ejemplo 23. ¿Cuál es el resultado de ejecutar cada uno de los siguientes comandos? ¿Por qué?

- ```
  ./prueba twenty; e=$?;
  if [ $e == 10 ]; then echo "ten";
  elif [ $e == 20 ]; then echo "twenty";
  else echo "unknown"; fi
  ```

- ```
  ./exit twenty;
  if [ $?  == 10 ]; then echo "ten";
  elif [ $?  == 20 ]; then echo "twenty";
  else echo "unknown"; fi
  ```

7.6 Bucles

7.6.1 Bucle FOR y listas

Tienen la siguiente sintaxis:

```
for <varName> in <list>
do
          commands....
done
```

Ejemplo 24. En el siguiente ejemplo se muestran dos bucles anidados:

```
for n in 1 2 3 4 5 6 7 8 9 10
do
    for i in 1 2 3 4 5 6 7 8 9 10
    do
         echo "$n * $i = $(expr $i \* $n)"
    done
done
```

Ejercicio 3. Muestra el contenido de todos los directorios del directorio actual (no de los subdirectorios). Usa bucles y condicionales.

Ejercicio 4. 1. Ejecuta el siguiente comando:
`for i in {1..100}; do echo $i >> numbers; done`
que crea el fichero `numbers`, que contiene 100 líneas, cada una de las cuales tiene el número de línea del fichero.

2. Imprime el valor de cada línea multiplicado por 10.

7.6.2 Bucle FOR y expresiones

Tienen la siguiente sintaxis:

```
for (( expr1; expr2; expr3 ))
do
          commands....
done
```

de forma que `expr1` es evaluada en la primera iteración (usualmente para inicializar las variables del bucle), `expr2` es evaluada en todas las demás (condición de fin), y `expr3` se evalúa al final de cada iteración (usualmente para incrementar el valor de las variables del bucle).

Ejemplo 25. En el siguiente ejemplo se muestran dos bucles anidados:

```
for (( i = 1; i <= 5; i++ ))
do
    for (( j = 1 ; j <= 5; j++ ))
    do
         echo -n "$i "
    done
    echo ""
done
```

7.6.3 Bucle WHILE y expresión

Tienen la siguiente sintaxis:

```
while [ condition ]
do
          commands....
done
```

donde `condition` tiene la misma sintaxis que para los *if-else-fi*.

8 Manipulación de strings y otros operadores

Otra opción que también nos permite *bash* es manipular fácilmente los strings (es decir, las variables). Para usarlos, encerraremos entre llaves { y }. Es decir, la variable `$var` la escribiremos como `${var}`. Veamos algunas de las manipulaciones que podemos conseguir:

La **longitud** de un string se puede extraer con el operador '#':

Ejemplo 26. Longitud de un string:

```
var=abcABC123ABCabc
echo ${#var}      #15
```

Se pueden extraer **substrings** con el operador ':' e indicando la posición inicial y la longitud:

Ejemplo 27. Substring:

```
var=abcABC123ABCabc
echo ${var:0}      #abcABC123ABCabc
echo ${var:6}      #123ABCabc
echo ${var:9:3}    #ABC
```

También se pueden remover **substrings** usando patrones:

- `${var#pattern}` : elimina de `var` el *string* más corto igual a `pattern` desde el principio.

- `${var##pattern}` : elimina de `var` el *string* más largo igual a `pattern` desde el principio.
- `${var%pattern}` : elimina de `var` el *string* más corto igual a `pattern` desde el final.
- `${var%%pattern}`: elimina de `var` el *string* más largo igual a `pattern` desde el final.

Ejemplo 28. Substring:

```
var=abcABC123ABCabc
echo ${var#a*C}       # 123ABCabc
echo ${var##a*C}      # abc
echo ${var%b*c}       # abcABC123ABCa
echo ${var%%b*c}      # a
```

9 Bash Script Files

Todo el contenido de esta práctica (comandos, condiciones, bucles, ...) puede usarse de forma conjunta y guardarse en un único fichero, cuyo nombre es *bash script*. La utilidad de estos ficheros es guardar una secuencia de operaciones que a menudo se realizará de forma conjunta, y de esta forma, ahorramos tener que escribirlos una y otra vez.

Habitualmente, estos ficheros tienen la extensión `sh`, y para ejecutarlos deben tener los permisos adecuados.

9.1 Argumentos en scripts

En la *shell*, podemos referirnos a los argumentos pasados por parámetros usando el operador `$i`, donde $i \geq 0$. En concreto, el nombre del programa se encuentra en `$0`, y el resto de parámetros en `$1`, `$2`, ...

Ejemplo 29. En este ejemplo usaremos el fichero `test.sh`, cuyo contenido es el siguiente:

```
if [ $1 == suma ]
then
       echo "$2 + $3 = $(expr $2 + $3)";
elif [ $1 == resta ]
then
       echo "$2 - $3 = $(expr $2 - $3)";
else
       echo "No entiendo arg1 = $1"
       exit 1
fi
exit 0
```

Una vez creado el fichero, modifica los permisos de este fichero para que sea ejecutable por el usuario que haya creado el fichero.
¿Qué producen las siguientes llamadas?

1. `./test.sh suma 10 15`
2. `./test.sh resta 10 3`
3. `./test.sh divide 10 15`

9.2 Exit status en scripts

Un script, al finalizar, finaliza con el exit status del último comando ejecutado. En caso que se quiera salir con un exit status diferente, hay que utilizar

```
exit <num>
```

donde `<num>` es el exit status a devolver.

Ejemplo 30. Ejecuta los comandos

1. `./test.sh suma 10 15; echo $?`
2. `./test.sh resta 10 3; echo $?`
3. `./test.sh divide 10 15; echo $?`

Ejercicio 5. Crea un fichero `script.sh` que tiene dos argumentos

```
./script <arg1> <arg2>
```

y realize los siguientes pasos

1. El primer argumento es un fichero. En caso que el el fichero no exista, salir del script con un exit status de 1.
2. El segundo argumento es un directorio. En caso que el directorio no exista, hay que crearlo.
3. Copiar el fichero especificado en el primer argumento al directorio especificado en el segundo argumento.
4. Salir con un exit status de 0.

10 Ejercicios a entregar para la Práctica 1: Shell Scripting

10.1 Instrucciones de la entrega

La práctica consta de una serie de problemas. Para cada problema únicamente se pueden utilizar los comandos introducidos en esta práctica (cualesquiera que sean sus opciones). No se pueden utilizar otros comandos (en particular `xargs` o `basename`). En caso que se quieran utilizar otros comandos, hay que razonar el porqué mediante comentarios en los scripts.

La práctica se realizará **individualmente**. Se entregará cada ejercicio en un fichero separado de forma que pueda ser ejecutado por el profesor. Hay que asegurarse que los scripts se ejecuten correctamente en el aula en el que se realizan las prácticas. El hecho que no sea así es motivo de penalización.

10.2 Test adicional

El día siguiente a la fecha de entrega (es decir, en la primera sesión de la práctica 2) se realizará un cuestionario tipo test, que obligatoriamente será realizado en las aulas de prácticas y de forma individual.

10.3 Plagio

En caso de copia parcial o completa de la entrega de esta práctica entre varios estudiantes, la práctica quedará calificada como 0, para todas las personas involucradas (es decir, tanto para el copiador como para el que se dejó copiar).

10.4 Problemas

Problema 1. Manejo de ficheros. (**3 puntos**) Realiza las siguientes tareas usando un comando por tarea (en algunas hará falta uso de bucles, condicionales, etc...). Si es necesario, usa las opciones adicionales de los comandos estudiados en esta práctica. No "juntar" las tareas especificadas aquí en un único comando para aumentar la eficiencia.

El script a generar, denominado `problema1.sh`, tendrá dos argumentos, `<arg1>` i `<arg2>`. El primer argumento es un fichero (que se supone que existe y es un fichero de texto plano) y el segundo es un número entero.

1. Crea el directorio `problema1`; se supone que el directorio no existe previamente.
2. Copia el fichero especificado en `<arg1>` dentro del directorio creado con el nombre `fichero.txt`.
3. Entra en él directorio creado.
4. Crea tantos subdirectorios como especificados en `<arg2>` con nombres `subdir-1`, `subdir-2`, etc.
5. Genera, dentro del directorio `subdir-1`, un fichero `1.txt` que incluya únicamente la 1a línia de `fichero.txt`. Dentro del directorio `subdir-2` un fichero `2.txt` que incluya únicamente las 2 primeras líneas del fichero `fichero.txt` y así sucesivamente hasta el número especificado en `<arg2>`. Puedes suponer que el fichero `fichero.txt` tiene más lineas que el número especificado en `<arg2>`.

6. Sube al directorio padre de `problema1`

7. Borra el directorio `problema1` y todo lo que contiene.

Problema 2. Manipulacion de cadenas. (**3 puntos**)

Escribe el contenido de un script que realice las siguientes operaciones:

1. Inicialmente comprobará que el fichero pasado en el primer argumento existe, en caso contrario, mostrará un error.

2. En caso de que exista, procede con las siguientes operaciones (en caso de que no exista, termina).

3. El contenido de ese fichero es una única línea con el siguiente formato: N strings, y cada uno de ellos va seguido de una coma (también el último). Es decir:

$$<\text{string1}>,<\text{string2}>,\ldots,<\text{stringN}>,$$

   Asumiremos que todos los ficheros de entrada contienen un formato correcto.

4. Se pide imprimir una línea para cada uno de esos *strings* (es decir, `N` líneas), de forma que cada una de ellas contenga k repeticiones, separadas por un espacio, del *string* i-ésimo. Es decir:

$$<\text{string1}>^1 \sqcup <\text{string1}>^2 \sqcup \ldots \sqcup <\text{string1}>^k$$
$$<\text{string2}>^1 \sqcup <\text{string2}>^2 \sqcup \ldots \sqcup <\text{string2}>^k$$
$$\ldots$$
$$<\text{stringN}>^1 \sqcup <\text{stringN}>^2 \sqcup \ldots \sqcup <\text{stringN}>^k$$

   donde $\sqcup$ representa un espacio en blanco.

   El número k será especificado en el segundo argumento del script.

Problema 3. Manejo de entrada/salida. (**4 puntos**)

Realiza las siguientes tareas usando un comando por tarea (en algunas hará falta uso de bucles, condicionales, etc...). Si es necesario, usa las opciones adicionales de los comandos estudiados en esta práctica.

El objetivo de este script es organizar los ficheros que haya en un directorio por su extensión. En particular, se colocaran en directorios diferentes ficheros que tengan extensión diferente.

Supogamos el fichero `extensiones.txt` que tiene el siguiente contenido

```
c
cpp
bmp
jpg
png
```

Se pide realizar un script con dos argumentos

```
./problema3.sh <extensiones.txt> <directorio>
```

El primer argumento es un fichero de extensiones tal como se ha descrito. El segundo argumento es un directorio que contiene ficheros con extensiones especificadas en el primer argumento. Se supone que tanto el directorio como el fichero de extensiones existen.

El script deberá realizar las siguientes tareas (no es necesario utilizar un comando por tarea como en los anteriores problemas):

1. Crear el directorio `problema3`.
2. Entrar dentro de este directorio.
3. Guardar en una variable el contenido del fichero de extensiones.
4. Guardar en una variable el resultado de ejecutar `ls` en el directorio especificado como 2o argumento.
5. Dentro de este directorio se deberan crear todos los subdirectorios especificados en el fichero de extensiones. Para el ejemplo anterior se deberan crear los directorios `c`, `cpp`, `bmp`, `jpg` y `png`.
6. Recorrer los elementos del directorio. En caso que sea un fichero obtener su extensión y moverlo al directorio que corresponda. Observar que el fichero `prueba.ajpg` no es realmente un fichero con extensión `jpg`. Hay que considerar la subcadena desde el útimo punto.
7. Una vez realizado el paso anterior el directorio original tendrá que contener ficheros que no tiene ninguna de las extensiones especificadas en el fichero. Se moverá el directorio dentro del directorio `problema3` con el nombre `otros`.