



Universidad de Murcia

Facultad de Informática

Departamento de Ingeniería y Tecnología de Computadores

Área de Arquitectura y Tecnología de Computadores

PRÁCTICAS DE SS.OO.

I.I./I.T.I. SISTEMAS/I.T.I. GESTIÓN

Práctica 2 – Programación de *shell scripts* en Linux

NOVIEMBRE DE 2006

Índice

1. CONCEPTO DE SHELL EN LINUX	4
2. FUNCIONAMIENTO DEL SHELL	5
3. VARIABLES Y PARÁMETROS	5
3.1. Variables	5
3.2. Parámetros	7
3.3. Reglas de evaluación de variables	8
3.4. Arrays	9
4. CARACTERES ESPECIALES Y DE ENTRECORNILLADO	10
5. ESTRUCTURAS DE CONTROL	11
5.1. Condiciones: <code>if</code> y <code>case</code>	11
5.2. Bucles condicionales: <code>while</code> y <code>until</code>	12
5.3. Bucles incondicionales: <code>for</code> y <code>seq</code>	13
5.4. Menús de opciones: <code>select</code>	13
5.5. Ruptura de bucles: <code>break</code> y <code>continue</code>	14
6. ENTRADA/SALIDA ESTÁNDAR Y REDIRECCIÓN	14
7. FILTROS	16
7.1. Introducción	16
7.2. Expresiones regulares	17
7.3. Ejemplos de uso de filtros	18
8. ÓRDENES INTERNAS DE BASH	19
9. EVALUACIÓN ARITMÉTICA	20
10. LA ORDEN <code>test</code>	22
11. ÓRDENES SIMPLES, LISTAS DE ÓRDENES Y ÓRDENES COMPUESTAS	24
11.1. Órdenes simples	24
11.2. Listas de órdenes	24
11.3. Órdenes compuestas	25
12. FUNCIONES	25
12.1. Ejemplo de funciones	26
12.2. Ejemplo de funciones con parámetros	26
13. DEPURACIÓN	26
14. PATRONES DE USO DEL SHELL	27
14.1. Comprobación de cadena vacía	27
14.2. Uso de <code>xargs</code>	27
14.3. Leer un fichero línea a línea	28
14.4. Comprobar si una determinada variable posee un valor numérico válido	28
14.5. Leer argumentos opcionales de la línea de comandos	28
15. EJEMPLOS DE GUIONES SHELL	29

16. EJERCICIOS PROPUESTOS

34

17. BIBLIOGRAFÍA

38

1. CONCEPTO DE SHELL EN LINUX

Un **shell** es un intérprete de órdenes, y un **intérprete de órdenes** es un programa que procesa todo lo que se escribe en el terminal. Básicamente, permite a los usuarios interactuar con el sistema operativo para darle órdenes. En otras palabras, el objetivo de cualquier intérprete de órdenes es procesar los comandos o ejecutar los programas que el usuario teclea.

El **prompt** es una indicación que muestra el intérprete para anunciar que espera una orden del usuario. Cuando el usuario escribe una orden, el intérprete la ejecuta. Dicha orden puede ser **interna** o **externa**. Las órdenes internas son aquellas que vienen incorporados en el propio intérprete, como, por ejemplo, **echo**, **cd**, o **pwd**. Las externas, por el contrario, son programas separados como, por ejemplo, todos los programas que residen en los directorios **/bin** (como **ls**), **/usr/bin** (como **cut**), **/sbin** (como **fsck**) o **/usr/sbin** (como **lpc**).

En el mundo UNIX/Linux existen tres grandes familias de shells: **sh**, **csh** y **ksh**. Se diferencian entre sí, fundamentalmente, en la sintaxis de sus órdenes y en su interacción con el usuario. En estas prácticas nos centraremos en el uso del shell **bash**, una variante *libre* de la familia **sh**.

Por defecto, cada usuario tiene asignado un shell, establecido en el momento de creación de su cuenta, y que se guarda en el fichero `/etc/passwd`. En los laboratorios de prácticas se puede consultar ese fichero con la orden `ypcat passwd`. El shell asignado a un usuario se puede cambiar de dos maneras: editando manualmente dicho fichero (lo cual sólo puede hacer el administrador del sistema), o bien con el programa `chsh` (que lo puede ejecutar el propio usuario). Los shells están en los directorios `/bin` y `/usr/bin`¹. Por ejemplo, para hacer que el shell por defecto sea `/bin/bash` se ejecutaría:

```
chsh -s /bin/bash
```

Una de las principales características del shell es que puede programarse usando ficheros de texto a partir de órdenes internas y programas externos. Además, el shell ofrece construcciones y facilidades para hacer más sencilla su programación. Estos ficheros de texto se llaman **scripts**, **shell scripts** o **guiones shell**. Tras su creación, estos guiones shell pueden ser ejecutados tantas veces como se desee. Una posible definición podría ser la siguiente: “*Un guión de shell es un fichero de texto ejecutable que contiene una secuencia de órdenes ejecutables por el shell*”.

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

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

La programación de guiones shell es una de las herramientas más apreciadas por todos los administradores y muchos usuarios de UNIX/Linux ya que permite automatizar tareas complejas y/o repetitivas, y ejecutarlas con una sola llamada al script, incluso de manera automática a una hora preestablecida sin intervención humana. A continuación se muestran una serie de ejemplos de distintas tareas que se suelen automatizar con scripts:

- Tareas administrativas definidas por el propio sistema o por su administrador. Por un lado, algunas partes de los sistemas UNIX/Linux son guiones shell. Para poder entenderlos y modificarlos es necesario tener alguna noción sobre la programación de scripts. Por otro lado, el administrador del sistema puede necesitar llevar a cabo tareas de mantenimiento del sistema de manera regular. El uso de guiones shell permite automatizarlas fácilmente en la mayor parte de los casos.
- Tareas tediosas, incluso aquellas que sólo se van a ejecutar una o dos veces, para las que no importa el rendimiento obtenido durante su realización pero sí que se terminen con rapidez.
- Integración de varios programas independientes para que funcionen como un conjunto de forma sencilla.
- Desarrollo de prototipos de aplicaciones más complejas que posteriormente se implementarán en lenguajes más potentes.

¹El fichero `/etc/shells` contiene una lista de los shells disponibles.

Conocer a fondo el shell aumenta tremendamente la rapidez y productividad a la hora de utilizarlo, incluso sin hacer uso de guiones (es decir, utilizándolo simplemente desde el *prompt* del sistema).

Los guiones shell pueden utilizar un sin número de herramientas como:

- Órdenes del sistema internas o externas, por ejemplo, órdenes como `echo`, `ls`, etc.
- Lenguaje de programación del shell, por ejemplo, construcciones como `if/then/else/fi`, etc.
- Programas y/o lenguajes de procesamiento en línea como `sed` o `awk`.
- Programas propios del usuario escritos en cualquier lenguaje de programación.

Si un guión shell no resulta suficiente para lo que queremos hacer, existen otros lenguajes interpretados mucho más potentes como **Perl**, **TCL** o **Python**.

El intérprete de órdenes seleccionado para realizar estas prácticas es el **Bourne-Again Shell** o **bash**, cuyo ejecutable es `/bin/bash`. El resto del contenido de este documento está centrado en este intérprete de órdenes.

2. FUNCIONAMIENTO DEL SHELL

Suponiendo que tenemos el siguiente guión shell,

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

al ejecutarse el proceso que se sigue es el siguiente:

1. El shell `/bin/bash` hace un `fork`.
2. El proceso padre espera mientras no termina el nuevo proceso hijo.
3. El proceso hijo hace un `fork` y un `exec` para ejecutar la orden `clear`, y a continuación ejecuta un `wait` para esperar a que termine la ejecución de `clear`.
4. Una vez que ha terminado la orden `clear`, el proceso hijo repite los mismos pasos pero esta vez ejecutando la orden `date`.
5. Si quedasen órdenes por ejecutar se seguiría el mismo procedimiento.
6. Cuando finaliza el proceso hijo, el proceso padre reanuda su ejecución.

3. VARIABLES Y PARÁMETROS

3.1. Variables

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

```
variable=valor
```

Nótese que no puede haber espacios entre el nombre de la variable, el signo `=` y el valor. Por otra parte, si se desea que el valor contenga espacios, es necesario utilizar comillas.

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

```
echo $variable
```

Un ejemplo del uso de las variables sería:

```
$ mils="ls -l" # Se crea una nueva variable
$ mils        # No hace nada porque busca el ejecutable
              # mils que no existe
$ $mils       # Ejecuta la orden "ls -l"
$ echo $mils  # Muestra el contenido de la variable mils,
              # es decir, "ls -l"
```

Las variables se dividen en dos tipos:

- **Variables locales:** no son heredadas por los procesos hijos del shell actual cuando se realiza un `fork`.
- **Variables de entorno:** heredadas por los procesos hijos del shell actual cuando se ejecuta un `fork`.

La orden **export** convierte una variable local en variable de entorno:

```
$ export mils      # Convierte la variable mils en variable de entorno
$ export var=valor # Crea la variable, le asigna "valor"
                  # y la exporta a la vez
```

La orden **set** muestra todas las variables (locales y de entorno) mientras que la orden **env** muestra sólo las variables de entorno. Con la orden **unset** se pueden restaurar o eliminar variables o funciones. Por ejemplo, la siguiente instrucción elimina el valor de la variable `mils`:

```
$ unset mils
```

Además de las variables que puede definir el programador, un shell tiene definidas, por defecto, una serie de variables. Las más importantes son:

- **PS1:** prompt primario. El siguiente ejemplo modifica el prompt, utilizando diferentes para el nombre del usuario y el host, y el directorio actual:

```
$ PS1='[\033[31m\]\u@\h[\033[0m\]:[\033[33m\]\w[\033[0m] $ '
```

- **PS2:** prompt secundario.
- **LOGNAME:** nombre del usuario.
- **HOME:** directorio de trabajo (`home`) del usuario actual que la orden `cd` toma por defecto.
- **PWD:** directorio actual.
- **PATH:** rutas de búsqueda usadas para ejecutar órdenes o programas. Por defecto, el directorio actual no está incluido en la ruta de búsqueda. Para incluirlo, tendríamos que ejecutar `PATH =PATH: . :`
- **TERM:** tipo de terminal actual.
- **SHELL:** shell actual.

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

- **\$?:** esta variable contiene el valor devuelto por la última orden ejecutada que es útil para saber si una orden ha finalizado con éxito o ha tenido problemas. Un 0 indica que la orden se ha ejecutado con éxito, otro valor indica que ha habido errores.
- **\$!:** identificador de proceso (PID) de la última orden ejecutada en segundo plano.
- **\$\$:** el PID del shell actual (comúnmente utilizado para crear nombres de ficheros únicos).

- **\$-**: las opciones actuales suministradas para esta invocación del shell.
- **\$***: todos los argumentos del guión shell comenzando por el \$1. Cuando la expansión ocurre dentro comillas dobles, se expande a una sola palabra con el valor de cada parámetro separado por el primer carácter de la variable especial **IFS** (habitualmente un espacio). En general, **\$*** es equivalente a `$1c$2c . . .`, donde `c` es el primer carácter del valor de la variable **IFS**. Si **IFS** no está definida, el carácter `c` se sustituye por un espacio. Si **IFS** es la cadena vacía, los parámetros se concatenan sin ningún separador.
- **\$@**: igual que la anterior excepto cuando va entrecomillada. Cuando la expansión ocurre dentro de comillas dobles, cada parámetro se expande a una palabra separada, esto es, **\$@** es equivalente a `$1 $2 . . .`

3.2. Parámetros

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

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

La variable `$0` contiene el nombre con el que se ha invocado el script, `$1` contiene el primer parámetro, `$2` contiene el segundo parámetro, . . .

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

```
#!/bin/bash
echo El nombre del programa es $0
echo El primer parámetro recibido es $1
echo El segundo parámetro recibido es $2
echo El tercer parámetro recibido es $3
echo El cuarto parámetro recibido es $4
```

La orden **shift** mueve todos los parámetros una posición a la izquierda, esto hace que el contenido del parámetro `$1` desaparezca, y sea reemplazado por el contenido de `$2`, que `$2` sea reemplazado por `$3`, etc.

La variable **\$#** contiene el número de parámetros que ha recibido el script. Como se indicó anteriormente **\$*** o **\$@** contienen todos los parámetros recibidos. La variable **\$@** es útil cuando queremos pasar a otros programas algunos de los parámetros que nos han pasado.

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

```
#!/bin/bash
echo El nombre del programa es $0
echo El número total de parámetros es $#
echo Todos los parámetros recibidos son $*
echo El primer parámetro recibido es $1
shift
echo El segundo parámetro recibido es $1
shift
echo El tercer parámetro recibido es $1
echo El cuarto parámetro recibido es $2
```

3.3. Reglas de evaluación de variables

A continuación se describen las reglas que gobiernan la evaluación de las variables de un guión shell:

- **`$var`**: valor de `var` si está definida, si no nada.
- **`${var}`**: igual que el anterior excepto que las llaves contienen el nombre de la variable a ser evaluada.
- **`${var-thing}`**: valor de `var` si está definida, si no `thing`.
- **`${var=thing}`**: igual que el anterior excepto cuando `var` no está definida en cuyo caso el valor de `var` pasa a ser `thing`.
- **`${var?message}`**: valor de `var` si está definida, si no imprime el mensaje en el terminal.
- **`${var+thing}`**: `thing` si `var` está definida, si no nada.

El siguiente ejemplo muestra cómo podemos usar una variable asignándole un valor en caso de que no esté definida:

```
$ echo El valor de var1 es ${var1}
# No está definida, no imprimirá nada

$ echo El valor de la variable es ${var1=5}
# Al no estar definida, le asigna el valor 5

$ echo Su nuevo valor es $var1
# Su valor es 5
```

Pero si lo que queremos es usar un valor por defecto, en caso de que la variable no esté definida, sin inicializar la variable:

```
$ echo El valor de var1 es ${var1}
# No está definida, no imprimirá nada

$ echo El valor de la variable es ${var1-5}
# Al no estar definida, utiliza el valor 5

$ echo El valor es $var1
# Su valor sigue siendo nulo, no se ha definido
```

Por otro lado, si lo que queremos es usar el valor de la variable y, en caso de que no esté definida, imprimir un mensaje:

```
$ echo El valor de var1 es ${var1}
# No está definida, no imprimirá nada

$ echo El valor de la variable es ${var1? No está definida...}
# Al no estar definida, se muestra en pantalla el mensaje

$ echo El valor es $var1
# Su valor sigue siendo nulo, no se ha definido
```

Este último ejemplo nos muestra cómo utilizar un valor por defecto si una variable está definida:

```
$ var1=4 # Le asigna el valor 4

$ echo El valor de var1 es ${var1}
# El valor mostrado será 4

$ echo El valor de la variable es ${var1+5}
# Al estar definida, se utiliza el valor 5

$ echo El valor es $var1
# Su valor sigue siendo 4
```

3.4. Arrays

El shell permite que se trabaje con arrays (o listas) unidimensionales. Un array es una colección de elementos del mismo tipo, dotados de un nombre, y que se almacenan en posiciones contiguas de memoria. El primer subíndice del primer elemento del array es 0, y si no se utiliza subíndice, se considera también que se está referenciando a dicho elemento. No hay un tamaño máximo para un array, y la asignación de valores se puede hacer de forma alterna.

La sintaxis para crear e inicializar un array es la siguiente:

```
nombre_array=(val1 val2 val3 ...) # Crea e inicializa un array
nombre_array[x]=valor # Asigna un valor al elemento x
```

Para acceder a un elemento del array se utiliza la siguiente sintaxis:

```
${nombre_array[x]} # Para acceder al elemento x
${nombre_array[*]} # Para consultar todos los elementos
${nombre_array[@]} # Para consultar todos los elementos
```

La diferencia entre usar `*` y `@` es que `${nombre_array[*]}` crea una única palabra con todos los elementos del array mientras que `${nombre_array[@]}` crea palabra distintas para cada elemento del array.

Para conocer el tamaño en bytes de un elemento dado del array se utiliza la sintaxis `${#nombre_array[x]}`, donde `x` es un índice del array. De hecho, esa misma expresión vale también para saber la longitud de una simple variable normal (por ejemplo, `${#var}`). Si lo que nos interesa, por el contrario, es saber el número total de elementos del array, entonces emplearemos las expresiones `${#nombre_array[*]}` o `${#nombre_array[@]}`.

Nótese la diferencia entre las siguientes órdenes:

```
$ aux=`ls`
$ aux1=(`ls`)
```

En el primer caso, la variable `aux` contiene la salida de `ls` como una cadena de caracteres. En el segundo caso, al haber utilizado los paréntesis, `aux1` es un array, y cada uno de sus elementos es uno de los nombres de fichero devueltos por la orden `ls`. Si en el directorio actual tenemos los ficheros `a.latex`, `b.latex`, `c.latex`, `d.latex`, `e.latex` y `f.latex`, observe el resultado de ejecutar las órdenes anteriores:

```
$ ls
a.latex b.latex c.latex d.latex e.latex f.latex
$ aux=`ls`
$ echo $aux
a.latex b.latex c.latex d.latex e.latex f.latex
$ echo ${aux[0]}
a.latex b.latex c.latex d.latex e.latex f.latex
$ aux1=(`ls`)
$ echo ${aux1[0]}
a.latex
```

4. CARACTERES ESPECIALES Y DE ENTRECORNILLADO

Los mecanismos de protección se emplean para quitar el significado especial para el shell de ciertos caracteres especiales o palabras reservadas. Pueden emplearse para que caracteres especiales no se traten de forma especial, para que palabras reservadas no sean reconocidas como tales, y para evitar la evaluación de variables.

Los metacaracteres (`*` `$` `|` `&` `;` `(` `)` `<` `>` **espacio tab**) tienen un significado especial para el shell y deben ser protegidos o entrecornillados si quieren representarse a sí mismos. Hay 3 mecanismos de **protección**: el **carácter de escape**, las **comillas simples** y las **comillas dobles**².

Una **barra inclinada inversa (o invertida) no entrecornillada** (`\`) es el **carácter de escape**, (no confundir con el código ASCII cuyo valor es 27 en decimal), el cual preserva el valor literal del siguiente carácter que lo acompaña, con la excepción de `<nueva-línea>`. Si aparece la combinación `\<nueva-línea>` y la barra invertida no está entre comillas, la combinación `\<nueva-línea>` se trata como una continuación de línea (esto es, se quita del flujo de entrada y no se tiene en cuenta). Por ejemplo, sería equivalente a ejecutar `ls -l`:

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

Encerrar caracteres entre **comillas simples** (`'` `'`) preserva el valor literal de cada uno de ellos entre las comillas. Una comilla simple no puede estar entre comillas simples, ni siquiera precedida de una barra invertida.

```
$ echo 'caracteres especiales: *, $, |, &, ;, (, ), {, }, <, >, \, ", `'  
caracteres especiales: *, $, |, &, ;, (, ), {, }, <, >, \, ", `'
```

Encerrar caracteres entre **comillas dobles** (`"` `"`) preserva el valor literal de todos los caracteres, con la excepción de `$`, `'`, y `\`. Los caracteres `$` y `'` mantienen su significado especial dentro de comillas dobles. La barra invertida mantiene su significado especial solamente cuando está seguida por uno de los siguientes caracteres: `$`, `'`, `"`, o `<nueva-línea>`. Una comilla doble puede aparecer entre otras comillas dobles precedida de una barra invertida.

```
$ echo "caracteres especiales: *, \$, |, &, ;, (, ), {, }, <, >, \\, \", \`"  
caracteres especiales: *, $, |, &, ;, (, ), {, }, <, >, \, ", `'
```

Los parámetros especiales `$*` y `$@` tienen un significado especial cuando están entre comillas dobles (véanse los apartados 3.1 y 3.2).

Las expresiones de la forma `$'cadena'` se tratan de forma especial. Las secuencias de escape con barra invertida de cadena, si están presentes, son reemplazadas según especifica el estándar ANSI/ISO de C, y el resultado queda entre comillas simples:

- `\a`: alerta (campana)
- `\b`: espacio-atrás
- `\e`: carácter de escape (ESC)
- `\f`: nueva página
- `\n`: nueva línea
- `\r`: retorno de carro
- `\t`: tabulación horizontal
- `\v`: tabulación vertical
- `\\`: barra invertida
- `\xnnn`: carácter cuyo código es el valor hexadecimal `nnn`

²Las comillas simples y dobles son las que aparecen en la tecla que hay a la derecha de la ñ y en la tecla del 2, respectivamente.

Encerrar una cadena entre **comillas invertidas** (' '), o bien entre paréntesis precedida de un signo \$, supone forzar al shell a ejecutarla como una orden y devolver su salida:

```
`orden`  
ō  
$(orden)
```

Este proceso se conoce como **sustitución de órdenes**. A continuación se muestran varios ejemplos:

```
$ aux=`ls -lai` # Ejecuta ls -lai y almacena el resultado en aux  
$ echo $aux    # Muestra el contenido de aux  
$ fecha=$(date) # Ejecuta date y almacena el resultado en fecha  
$ echo $fecha  # Muestra el contenido de fecha
```

Téngase en cuenta que el shell, antes de ejecutar una orden, procesa todos los caracteres especiales (en función de los mecanismos de protección), expande las expresiones regulares³, y realiza la sustitución de órdenes:

```
$ echo *        # Muestra todos los ficheros del directorio actual  
$ var=`ls`     # Primero ejecuta ls -la y luego almacena el resultado en var  
$ echo $var    # Muestra el contenido de var, esto es, equivale a echo *  
$ echo '$var'  # Imprime $var  
$ echo "$var"  # Muestra el contenido de var, esto es, equivale a echo *  
$ echo `date`  # Primero se ejecuta date y luego echo
```

5. ESTRUCTURAS DE CONTROL

5.1. Condiciones: if y case

En un guión shell se pueden introducir condiciones, de forma que determinadas órdenes sólo se ejecuten cuando éstas se cumplen. Para ello se utilizan las órdenes **if** y **case**, con la siguiente sintaxis:

```
if [ expresión ] # Habitualmente un test  
then  
    órdenes a ejecutar si se cumple la primera condición  
elif [ expresión ]  
then  
    órdenes a ejecutar si se cumple la segunda condición  
    # (el bloque elif y sus órdenes son opcionales)  
...  
else  
    órdenes a ejecutar en caso contrario  
    # (el bloque else y sus órdenes son opcionales)  
fi
```

La expresión a evaluar por if puede ser un test, una lista de órdenes (usando su valor de retorno), una variable o una expresión aritmética, esto es, básicamente cualquier orden que devuelva un código en \$?.

Un ejemplo del funcionamiento de la orden if sería:

```
if grep -q main prac.c  
then  
    echo encontrada la palabra clave main  
else  
    echo no encontrada la palabra clave main  
fi
```

³La generación de nombres de ficheros se basa habitualmente en expresiones regulares tal y como se describe en la utilización de los comandos find y grep que se realiza en el documento “Ejemplos de uso de distintas órdenes”, proporcionado con la primera práctica.

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

```
case $var in
v1)      ..
..
;;
v2|v3)   ..
..
;;
*)       ..                # Caso por defecto
;;
esac
```

`v1`, `v2` y `v3` son expresiones regulares similares a las utilizadas como comodines para los nombres de los ficheros.

Un ejemplo de su funcionamiento podría ser:

```
case $var in
1)      echo La variable var es un uno
;;
2)      echo La variable var es un dos
;;
*)      echo La variable var no es ni un uno ni un dos
;;
esac
```

5.2. Bucles condicionales: `while` y `until`

También es posible ejecutar bloques de órdenes de forma iterativa dependiendo de una condición. La comprobación puede realizarse al principio (**`while`**) o al final (**`until`**). La sintaxis es la siguiente:

```
while [ expresión ] # Mientras la expresión sea cierta...
do
...
done

until [ expresión ] # Mientras la expresión sea falsa...
do
...
done
```

Un ejemplo del funcionamiento de ambas órdenes sería:

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

# También muestra todos los parámetros
until [ -z $1 ]
do
    echo $1
    shift
done
```

5.3. Bucles incondicionales: for y seq

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

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

Por ejemplo:

```
for i in 10 30 70
do
    echo Mi número favorito es $i # toma los valores 10, 30 y 70
done
```

Aunque la lista de valores del **for** puede ser arbitraria (incluyendo no sólo números, sino cualquier otro tipo de cadena o expresión), a menudo lo que queremos es generar secuencias de valores numéricos al estilo de la instrucción *for* de los lenguajes de programación convencionales. En este caso, el comando **seq**, combinado con el mecanismo de sustitución de órdenes (véase el apartado 4) puede resultarnos de utilidad. Por ejemplo:

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

5.4. Menús de opciones: select

Con la orden **select** podemos solicitar al usuario que elija una opción de una lista. La sintaxis de la orden **select** es:

```
select opcion in [ lista ] ;
do
    # bloque de órdenes
done
```

select genera una lista numerada de opciones al expandir la lista *lista*. A continuación, presenta un prompt (#?) al usuario pidiéndole que elija una de las posibilidades, y lee de la entrada estándar la opción elegida. Si la respuesta dada es uno de los números de la lista presentada, dicho número se almacena en la variable **REPLY**, la variable *opcion* toma el valor del elemento de la lista elegido, y se ejecuta el bloque de órdenes. Si la respuesta es no válida, se vuelve a interrogar al usuario, y si es EOF, se finaliza. El bloque de órdenes se ejecuta después de cada selección válida, mientras no se termine, bien con **break** o bien con EOF. El valor de salida de **select** será igual al valor de la última orden ejecutada.

Un ejemplo sería el siguiente:

```
select respuesta in "Ver contenido directorio actual" \
    "Salir"
do
    echo Ha seleccionado la opción: $respuesta
    case $REPLY in
    1) ls .
    ;;
    2) break
    ;;
    esac
done
```

En pantalla aparecería:

```
1) Ver contenido directorio actual
2) Salir
#?
```

Si se selecciona la primera opción, 1, se mostraría el mensaje: “Ha seleccionado la opción: Ver contenido directorio actual”, se ejecutaría la orden `ls` en el directorio actual, y volvería a pedir la siguiente selección. Si por el contrario se pulsase un 2, seleccionando la segunda opción, aparecería el mensaje: “Ha seleccionado la opción: Salir”, y se saldría del `select`.

5.5. Ruptura de bucles: `break` y `continue`

Los órdenes **`break`** y **`continue`** sirven para interrumpir la ejecución secuencial del cuerpo de un bucle. La orden `break` transfiere el control a la orden que sigue a `done`, haciendo que el bucle termine antes de tiempo. La orden `continue`, por el contrario, transfiere el control a `done`, haciendo que se evalúe de nuevo la condición, es decir, la ejecución del bucle continúa en la siguiente iteración. En ambos casos, las órdenes del cuerpo del bucle siguientes a estas sentencias no se ejecutan. Lo normal es que formen parte de una sentencia condicional.

Un par de ejemplos de su uso serían:

```
# Muestra todos los parámetros, si encuentra una "f" finaliza
while [ $# -gt 0 ]
do
    if [ $1 = "f" ]
    then
        break
    fi
    echo Parámetro: $1
    shift
done

# Muestra todos los parámetros, si encuentra una "f"
# se lo salta y continúa el bucle
while [ $# -gt 0 ]
do
    if [ $1 = "f" ]
    then
        shift
        continue
    fi
    echo Parámetro: $1
    shift
done
```

6. ENTRADA/SALIDA ESTÁNDAR Y REDIRECCIÓN

La filosofía de UNIX/Linux es en extremo modular. Se prefieren las herramientas pequeñas que realizan tareas puntuales a las macro-herramientas que realizan todo. Para completar el modelo es necesario proporcionar el mecanismo para ensamblar estas herramientas en estructuras más complejas. Esto se realiza por medio del redireccionamiento de las entradas y las salidas.

Todos los programas tienen por defecto una **entrada estándar** (teclado) y dos salidas: la **salida estándar** (pantalla) y la **salida de error** (pantalla). En ellos se puede sustituir el dispositivo por defecto por otro dispositivo. Con esto se consigue que los datos de la entrada estándar para un programa se puedan leer de un archivo, y los de la salida (estándar o error) se puedan enviar a otro archivo.

La entrada estándar, la salida estándar y la salida de error se asocian a los programas mediante tres ficheros con los cuales se comunican con otros procesos y con el usuario. Estos tres ficheros son:

- **stdin** (entrada estándar): a través de este descriptor de fichero los programas reciben datos de entrada. Normalmente `stdin` está asociado a la entrada del terminal en la que está corriendo el programa, es decir, al **teclado**. Cada descriptor de fichero tiene asignado un número con el cual podemos referirnos a él dentro de un script, en el caso de **stdin** es el 0.
- **stdout** (salida estándar): es el descriptor de fichero en el que se escriben los mensajes que imprime el programa. Normalmente estos mensajes aparecen en la pantalla para que los lea el usuario. Su descriptor de fichero es el número 1.
- **stderr** (salida de error): es el descriptor de fichero en el que se escriben los mensajes de error que imprime el programa. Normalmente coincide con **stdout**. Tiene como descriptor de fichero el número 2.

Estos tres descriptores de fichero pueden redireccionarse, consiguiendo comunicar unos procesos con otros, de forma que trabajen como una unidad, haciendo cada uno una tarea especializada, o simplemente almacenando los datos de salida en un fichero determinado, o recibiendo los datos de entrada de un fichero concreto. Hay varios operadores para redireccionar la entrada y las salidas de un programa de distintas maneras:

- `>` : redirecciona **stdout** a un fichero, si el fichero existe lo sobrescribe:

```
$ who > usuarios.txt; less usuarios.txt
```

- `>>` : redirecciona **stdout** a un fichero, si el fichero existe añade los datos al final del mismo.
- `2 >` : redirecciona **stderr** a un fichero, si el fichero existe lo sobrescribe:

```
$ find / -type d -exec cat {} \; 2>errores.txt
```

- `2 >>` : similar a `>>` pero para la salida de error.
- `n>&m` : redirecciona el descriptor de fichero `n` al descriptor de fichero `m`, en caso de que `n` se omita, se sobrentiende un 1 (**stdout**):

```
$ cat file directorio >salida.txt 2>&1
# Redirecciona stdout al fichero salida.txt, y stderr a stdout
```

- `<` : lee la entrada estándar de un fichero:

```
$ grep cadena < fichero.txt # busca "cadena" dentro de fichero.txt
```

- `|` : redirecciona la salida estándar de una orden a la entrada estándar de otra orden:

```
$ who | grep pilar
```

Éste último tipo de redirección es quizás el más importante, puesto que se usa para integrar diferentes órdenes y programas, mediante la interconexión de sus entradas y salidas estándar. Más concretamente, con una tubería o *pipe* (símbolo `|`) hay varias órdenes que se ejecutan sucesivamente, de manera que la salida estándar de la primera se envía a la entrada estándar de la siguiente, y así hasta que se ejecuta la última:

```
$ orden1 | orden 2 | ... | orden n
```

En la siguiente sección veremos una serie de comandos cuyo principal cometido es procesar el texto que les llega por la entrada estándar, y volcar el resultado de dicho procesamiento en la salida estándar, en la forma de texto *filtrado*. Estos comandos, por tanto, se prestan al uso intensivo del mecanismo de redirección a través de tuberías.

7. FILTROS

7.1. Introducción

Como comentábamos anteriormente, para explotar la estructura de interconexión de comandos mediante tuberías, UNIX dispone de una serie de programas, conocidos como “filtros”, cuyo principal cometido es realizar algún tipo de procesamiento básico sobre un texto de entrada. En la siguiente tabla resumimos los más importantes, junto con sus opciones más utilizadas. Como siempre, se recomienda la consulta de sus respectivas páginas del manual para obtener más detalles, así como el documento “*Ejemplos de uso de distintas órdenes*”, proporcionado en la primera práctica, donde se presentan de forma tutorizada algunos de sus usos más típicos.

Filtro	Función y opciones comúnmente utilizadas
sort	Ordena las líneas de la entrada estándar. Opciones: -n: emplear orden numérico. -d: emplear orden alfabético (por defecto). -f: ignorar mayúsculas/minúsculas. -r: ordenación inversa. -c: no ordenar, simplemente chequear si ya hay orden o no en la entrada. -k: ordenar por un determinado campo. -t: define carácter de separación de campo (un espacio por defecto).
cut	Corta secciones de cada línea de la entrada (por columnas) Opciones: -c: cortar por columnas de caracteres. -f: cortar por campos (combinar con -d). -d: define carácter de separación de campo. Nota: Las opciones -c y -f admiten rangos numéricos bastante completos, como “-c4-8,13,18”, o “-f1,8-”, por ejemplo.
grep	Busca en la entrada que se le especifica líneas que concuerden o coincidan con un determinado patrón (puede ser una <i>expresión regular</i> ; ver más adelante). Opciones: -i: ignorar mayúsculas/minúsculas. -n: numerar las líneas. -c: no mostrar la salida, sólo contar número de coincidencias. -h: suprimir nombre de fichero coincidente (sólo mostrar línea). -l: suprimir la línea coincidente (sólo mostrar nombres de ficheros). -v: invertir la salida (buscar líneas NO coincidentes con el patrón). -q: no mostrar nada en la salida, sólo comprobar si existe coincidencia. Nota: Aunque suele usarse más como filtro (leyendo de la entrada estándar), la orden se usa también a menudo para procesar varios ficheros de entrada, pasados como parámetros.
head	Obtiene las primeras líneas (por defecto 10) de la entrada. Opciones: -n: obtener las primeras n líneas.
tail	Obtiene las últimas líneas (por defecto 10) de la entrada. Opciones: -n: obtener las últimas n líneas.

(Continúa...)

Tabla de filtros (Continuación)

Filtros	Función y opciones comúnmente utilizadas
tr	Cambia, borra o comprime caracteres. Opciones: "set1" "set2": va cambiando cada caracter de <i>set1</i> por el correspondiente de <i>set2</i> . -d: borrar caracteres indicados. -s: comprimir a uno sólo secuencias repetidas de los caracteres indicados. -c: complementar el conjunto de caracteres indicados. Nota: Esta orden es muy útil para comprimir espacios antes de usar <code>cut -f -d ' '</code> .
wc	Cuenta caracteres, palabras y/o líneas. Opciones: -w: contar palabras. -c: contar caracteres. -l: contar líneas.
uniq	Elimina líneas repetidas. Nota: Esta orden suele emplearse combinada con un <i>sort</i> previo.
tac	Concatena e imprime archivos invertidos línea a línea.
tee	Lee de la entrada estándar y escribe tanto en la salida estándar como en los ficheros que se le indican.
od	Convierte la entrada a formas octal y hexadecimal, entre otras.

7.2. Expresiones regulares

Algunas órdenes que reciben patrones como parámetros (la más común es la orden *grep*, pero no es la única.) admiten *expresiones regulares* en el patrón a buscar. Una expresión regular no es más que una cadena que sirve para expresar, de forma compacta, un conjunto (posiblemente infinito) de cadenas que cumplen un determinado patrón, mediante un sencillo repertorio de normas sintácticas. A continuación se enumeran las más importantes:

- . : Vale por cualquier carácter.
- * : La expresión anterior se repite 0 o más veces.
- + : La expresión anterior se repite 1 o más veces.
- { *n* , *m* } : La expresión anterior se repite entre *m* y *n* veces.
- [. . .] : Un subconjunto determinado de caracteres (admite rangos, p.e. [a - z]).
- [^ . . .] : El complemento del subconjunto de caracteres indicado.
- ^ : carácter especial (indica comienzo de línea).
- \$: carácter especial (indica final de línea).

He aquí algunos ejemplos:

- a . * s : cadenas que empiecen por "a" y terminen por "s" (p.e., as, abs, aduanas, aaaasss, etc.)
- [0 - 9] { 1 , 5 } : Números enteros de entre 1 y 5 dígitos.
- ^ root : La secuencia "root" al comienzo de una línea.
- final \$: La secuencia "final" al final de una línea.

7.3. Ejemplos de uso de filtros

A continuación se muestran unos cuantos ejemplos del encadenamiento de algunos de estos programas mediante tuberías:

- La siguiente orden cuenta cuántas entradas hay en el directorio actual, contando las líneas de la salida de un *ls*:

```
$ ls | wc -l
```

- Esta otra orden muestra el número de directorios que hay en el directorio actual, filtrando y contando posteriormente sólo aquellas líneas que comienzan con una “d”:

```
$ ls -l | grep "^d" | wc -l
```

- Usando las órdenes *who*, *tr*, *cut*, *sort* y *uniq* podemos saber qué usuarios están trabajando en el sistema en este momento, ordenando el listado y eliminando elementos repetidos:

```
$ who | tr -s " " | cut -f1 -d " " | sort | uniq
```

- La siguiente orden elimina todos los espacios en blanco del fichero “*basura.txt*”, guardando la salida en otro fichero llamado “*basura.sinblancos*”, al tiempo que muestra el resultado por pantalla:

```
$ cat basura.txt | tr -s " " | tee basura.sinblancos
```

- Esta otra orden lista todos los procesos que se están ejecutando en el sistema en orden numérico inverso de su PID (segundo campo de la salida del “*ps aux*”):

```
$ ps aux | sort -k 2 -n -r
```

- En el siguiente caso estamos interesados en mostrar un listado largo del directorio actual, en el que todos los números son sustituidos por el carácter “X”.

```
$ ls -l | tr "0-9" "X"
```

- En el último ejemplo mostramos una orden que obtiene el PID de todos los procesos ejecutados por el usuario “*pilar*”, y los muestra en una única línea:

```
$ ps aux | grep ^pilar | tr -s " " | cut -f2 -d " " | tr "\n" " "
```

Algunos filtros han llegado a ser tan complejos que son en sí un lenguaje de procesamiento de texto, de búsqueda de patrones, de construcción de scripts, y muchas otras posibilidades. Entre ellos podemos mencionar herramientas tradicionales en UNIX/Linux como *awk* y *sed* y otras más modernas como *perl*. A título de ejemplo de la potencia de alguna de estas herramientas, y para cerrar este apartado, he aquí un útil comando *sed* que busca y sustituye todas las ocurrencias de un número entero de entre 3 y 5 dígitos por la cadena “PRUEBA”:

```
$ ls -l | sed -e "s/[0-9]\{3,5\}/PRUEBA/g"
```

8. ÓRDENES INTERNAS DE BASH

Una orden interna del shell es una orden que el intérprete implementa y que ejecuta sin llamar a programas externos. Por ejemplo, *echo* es una orden interna de bash y cuando se llama desde un script no se ejecuta el fichero */bin/echo*. Algunos de las órdenes internas más utilizadas son:

- *echo*: envía una cadena a la salida estándar, normalmente la consola o una tubería. Por ejemplo:

```
echo El valor de la variable es $auxvar
```

- *read*: lee una cadena de la entrada estándar y la asigna a una variable, permitiendo obtener entrada de datos por teclado en la ejecución de un guión shell:

```
echo -n "Introduzca un valor para var1: "  
read var1  
echo "var1 = $var1"
```

La orden *read* puede leer varias variables a la vez. También se puede combinar el uso de *read* con *echo*, para mostrar un *prompt* que indique qué es lo que se está pidiendo. Hay una serie de caracteres especiales para usar en *echo* y que permiten posicionar el cursor en un sitio determinado:

- *\b*: retrocede una posición (sin borrar)
- *\f*: alimentación de página
- *\n*: salto de línea
- *\t*: tabulador

Para que *echo* reconozca estos caracteres es necesario utilizar la opción “-e” y utilizar comillas dobles:

```
$ echo -e "Hola \t ¿cómo estás?"  
hola      como estás
```

Una orden alternativa a *echo* para imprimir en la salida estándar es la orden *printf*. Su potencial ventaja radica en la facilidad para formatear los datos de salida al estilo del *printf* del lenguaje C. Por ejemplo, la orden:

```
printf "Número: \t%05d\nCadena: \t%s\n" 12 Mensaje
```

produciría una salida como la siguiente:

```
Número:      00012  
Cadena:      Mensaje
```

- *cd*: cambia de directorio
- *pwd*: devuelve el nombre del directorio actual, equivale a leer el valor de la variable *\$PWD*.
- *pushd* / *popd* / *dirs*: estas órdenes son muy útiles cuando un script tiene que navegar por un árbol de directorios:
 - *pushd*: apila un directorio en la pila de directorios.
 - *popd*: lo desapila y cambia a ese directorio.
 - *dirs*: muestra el contenido de la pila.
- *let arg [arg]*: cada *arg* es una expresión aritmética a ser evaluada (véase el apartado 9):

```
$ let a=$b+7
```

Si el último arg se evalúa a 0, let devuelve 1; si no, devuelve 0.

- *test*: permite evaluar si una expresión es verdadera o falsa, véase el apartado “La orden test”.
- *export*: hace que el valor de una variable esté disponible para todos los procesos hijos del shell.
- *[.source] nombre_fichero argumentos*: lee y ejecuta órdenes desde *nombre_fichero* en el entorno actual del shell y devuelve el estado de salida de la última orden ejecutada desde *nombre_fichero*. Si se suministran argumentos, se convierten en los parámetros cuando se ejecuta *nombre_fichero*. Cuando se ejecuta un guión shell precediéndolo de “.” o *source*, no se crea un shell hijo para ejecutarlo, por lo que cualquier modificación en las variables de entorno permanece al finalizar la ejecución, así como las nuevas variables creadas.
- *exit*: finaliza la ejecución del guión. Recibe como argumento un entero que será el valor de retorno. Este valor lo recogerá el proceso que ha llamado al guión shell.
- *fg*: reanuda la ejecución de un proceso parado, o bien devuelve un proceso que estaba ejecutándose en segundo plano al primer plano.
- *bg*: lleva a segundo plano un proceso de primer plano o bien un proceso suspendido.
- *wait*: detiene la ejecución hasta que los procesos que hay en segundo plano terminan.
- *true* y *false*: devuelven 0 y 1 siempre, respectivamente.

Nota: El valor 0 se corresponde con *true*, y cualquier valor distinto de 0 con *false*.

9. EVALUACIÓN ARITMÉTICA

El shell permite que se evalúen expresiones aritméticas, bajo ciertas circunstancias. La evaluación se hace con enteros largos sin comprobación de desbordamiento, aunque la división por 0 se atrapa y se señala como un error. La lista siguiente de operadores se agrupa en niveles de operadores de igual precedencia, se listan en orden de precedencia decreciente.

- , +	Menos y más unarios
~	Negación lógica y de bits
**	Exponenciación
*, /, %	Multiplicación, división, resto
+, -	Adición, sustracción
<<, >>	Desplazamientos de bits a izquierda y derecha
<=, >=, <, >	Comparación
==, !=	Igualdad y desigualdad
&	Y de bits (AND)
^	O exclusivo de bits (XOR)
	O inclusivo de bits (OR)
&&	Y lógico (AND)
	O lógico (OR)
expre?expre:expre	Evaluación condicional
=, +=, -=, *=, /=, %=, &=, ^=, = <<=, >>=	Asignación: simple, después de la suma, de la resta, de la multiplicación, de la división, del resto, del AND bit a bit, del XOR bit a bit, del OR bit a bit, del desplazamiento a la izquierda bit a bit y del desplazamiento a la derecha bit a bit.

Por ejemplo, la siguiente orden muestra en pantalla el valor 64

```
$ echo $((2**6))
```

Se permite que las variables del shell actúen como operandos: se realiza la expansión de parámetro antes de la evaluación de la expresión. El valor de un parámetro se fuerza a un entero largo dentro de una expresión. Una variable no necesita tener activado su atributo de entero para emplearse en una expresión.

Las constantes con un 0 inicial se interpretan como números octales. Un 0x ó 0X inicial denota un número en hexadecimal. De otro modo, los números toman la forma $[base\#]n$, donde *base* es un número en base 10 entre 2 y 64 que representa la base aritmética, y *n* es un número en esa base. Si *base* se omite, entonces se emplea la base 10. Por ejemplo:

```
$ let a=6#10+1
$ echo el valor de a es $a
El valor de a es 7
```

Los operadores se evalúan en orden de precedencia. Las subexpresiones entre paréntesis se evalúan primero y pueden sustituir a las reglas de precedencia anteriores.

Existen tres maneras de realizar operaciones aritméticas:

1. Con *let lista_expresiones*, como se ha dicho anteriormente, se pueden evaluar las expresiones aritméticas dadas como argumentos. Es interesante destacar que esta orden no es estándar, sino que es específica del *bash*. A continuación se muestra un ejemplo de su uso:

```
$ let a=6+7
$ echo El resultado de la suma es $a
El resultado de la suma es: 13

$ let b=7%5
$ echo El resto de la división es: $b
El resto de la división es: 2

$ let c=2#101\|2#10
$ echo El valor de c es $c
El valor de c es 7
```

2. La orden *expr* sirve para evaluar expresiones aritméticas. Puede incluir los siguientes operadores: $\backslash($, $\backslash)$, $\backslash*$, $\backslash\backslash$, $\backslash+$, $\backslash-$, donde el carácter \backslash se introduce para quitar el significado especial que pueda tener el carácter siguiente. Por ejemplo:

```
$ expr 10 \* \ ( 5 \+ 2 \ )
70

$ i='expr $i - 1'      #restará 1 a la variable i
```

3. Mediante $\$((expresión))$ también se pueden evaluar expresiones. Varios ejemplos de su uso serían:

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

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

$ echo El valor es $((2#101|2#10))
El valor de c es 7
```

10. LA ORDEN `test`

El comando `test` permite evaluar si una expresión es verdadera o falsa. Los tests no sólo operan sobre los valores de las variables, también permiten conocer, por ejemplo, las propiedades de un fichero.

Principalmente se usan en la estructura `if/then/else/fin` para determinar qué parte del script se va a ejecutar. Un `if` puede evaluar, además de un `test`, otras expresiones, como una lista de órdenes (usando su valor de retorno), una variable o una expresión aritmética, básicamente cualquier orden que devuelva un código en `$?` .

La sintaxis de `test` puede ser una de las dos que se muestran a continuación:

```
test expresión
[ expresión ]
```

¡OJO! Los espacios en blanco entre la expresión y los corchetes son necesarios.

La expresión puede incluir operadores de comparación como los siguientes:

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

<code>-eq</code>	Igual a
<code>-ne</code>	Distinto de
<code>-lt</code>	Menor que
<code>-le</code>	Menor o igual que
<code>-gt</code>	Mayor que
<code>-ge</code>	Mayor o igual que

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

```
if [ $e -eq 1 ]
then
    echo Vale 1
else
    echo No vale 1
fi
```

Por el contrario, en el siguiente ejemplo, a la variable “e” se le asigna un valor si no está definida, por lo que sí funcionaría:

```
if [ ${e=0} -eq 1 ]
then
    echo Vale 1
else
    echo No vale 1
fi
```

- Para **caracteres alfabéticos o cadenas**:

<code>-z cadena</code>	Verdad si la longitud de cadena es cero.
<code>-n cadena</code>	Verdad si la longitud de cadena no es cero.
<code>cadena1 == cadena2</code>	Verdad si las cadenas son iguales. Se puede emplear = en vez de ==.
<code>cadena1 != cadena2</code>	Verdad si las cadenas no son iguales.
<code>cadena1 < cadena2</code>	Verdad si <code>cadena1</code> se ordena lexicográficamente antes de <code>cadena2</code> en la localización en curso.
<code>cadena1 > cadena2</code>	Verdad si <code>cadena1</code> se clasifica lexicográficamente después de <code>cadena2</code> en la localización en curso.

- En expresión se pueden incluir **operaciones con ficheros**, entre otras:

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

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

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

A continuación veremos distintos ejemplos de uso de la orden *test*, con el fin de aclarar su funcionamiento. Uno de los usos más comunes de la variable \$#, es validar el número de argumentos necesarios en un programa shell. Por ejemplo:

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

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

```
if test -f "$1"          # ¿ es un fichero ?
then
    more $1
elif test -d "$1"      # ¿ es un directorio ?
then
    (cd $1;ls -l|more)
else
    # no es ni fichero ni directorio
    echo "$1 no es fichero ni directorio"
fi
```

Comparando dos cadenas:

```
#!/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 determinadas versiones, esto no es buena idea, porque si $$$1$ o $$$2$ son vacíos, aparecerá un error sintáctico. En este caso, es mejor: x1=x2 o $"$1"="$2"$.

11. ÓRDENES SIMPLES, LISTAS DE ÓRDENES Y ÓRDENES COMPUESTAS

11.1. Órdenes simples

Una orden simple es una secuencia de asignaciones opcionales de variables seguida por palabras separadas por blancos y redirecciones, y terminadas por un operador de control. La primera palabra especifica la orden a ser ejecutada. Las palabras restantes se pasan como argumentos a la orden pedida. Por ejemplo, si tenemos un shell script llamado *programa*, podemos ejecutar la siguiente orden:

```
$ a=9 programa
```

La secuencia de ejecución que se sigue es: se asigna el valor a la variable *a*, que se exporta a *programa*. Esto es, *programa* va a utilizar la variable *a* con el valor 9.

El valor devuelto de una orden simple es su estado de salida, ó 128+n si la orden ha terminado debido a la señal n.

11.2. Listas de órdenes

Un **operador de control** es uno de los siguientes símbolos:

&	&&	;	::	()			<nueva-línea>
---	----	---	----	-----	--	--	---------------

Una lista es una secuencia de una o más órdenes separadas por uno de los operadores **;**, **&**, **&&**, o **||**, y terminada opcionalmente por uno de **;**, **&**, o **<nueva-línea>**. De estos operadores de listas, **&&** y **||** tienen igual precedencia, seguidos por **;** y **&**, que tienen igual precedencia.

Si una orden se termina mediante el operador de control **&**, el shell ejecuta la orden en segundo plano en un subshell. El shell no espera a que la orden acabe, y el estado devuelto es 0. Las órdenes separadas por un **;** se ejecutan secuencialmente; el shell espera que cada orden termine. El estado devuelto es el estado de salida de la última orden ejecutada.

Los operadores de control **&&** y **||** denotan listas *Y* (*AND*) y *O* (*OR*) respectivamente. Una lista *Y* tiene la forma:

```
orden1 && orden2
```

orden2 se ejecuta si y sólo si *orden1* devuelve un estado de salida 0.

Una lista *O* tiene la forma:

```
orden1 || orden2
```

orden2 se ejecuta si y sólo si *orden1* devuelve un estado de salida distinto de 0.

El estado de salida de las listas *Y* y *O* es el de la última orden ejecutada en la lista.

Dos ejemplos del funcionamiento de estas listas de órdenes son:

```
test -e prac.c || echo El fichero no existe
test -e prac.c && echo El fichero sí existe
```

La orden *test -e fichero* devuelve verdad si el fichero existe. Cuando no existe devuelve un 1, y la lista *O* tendría efecto, pero no la *Y*. Cuando el fichero existe, devuelve 0, y la lista *Y* tendría efecto, pero no la *O*.

Otros ejemplos de uso son:

```
sleep 1 || echo Hola      # echo no saca nada en pantalla
sleep 1 && echo Hola      # echo muestra en pantalla Hola
```

Un ejemplo de lista de órdenes encadenadas con ; es:

```
ls -l; cat prac.c; date
```

Primero ejecuta la orden `ls -l`, a continuación muestra en pantalla el fichero `prac.c` (`cat prac.c`), y por último muestra la fecha (`date`).

El siguiente ejemplo muestra el uso del operador de control & en una lista:

```
ls -l & cat prac.c & date &
```

En este caso, ejecuta las tres órdenes en segundo plano.

11.3. Órdenes compuestas

Las órdenes se pueden agrupar formando órdenes compuestas:

- { c1 ; ... ; cn; }: las n órdenes se ejecutan simplemente en el entorno del shell en curso, sin crear un shell nuevo. Esto se conocen como una **orden de grupo**.
- (c1 ; ... ; cn): las n órdenes se ejecutan en un nuevo shell hijo, se hace un *fork*.

¡OJO! Es necesario dejar los espacios en blanco entre las órdenes y las llaves o los paréntesis.

Para ver de forma clara la diferencia entre estas dos opciones lo mejor es estudiar qué sucede con el siguiente ejemplo:

Ejemplo	<pre>#!/bin/bash cd /usr { cd bin; ls; } pwd</pre>	<pre>#!/bin/bash cd /usr (cd bin; ls) pwd</pre>
Resultado	<pre>1.- Entrar al directorio /usr 2.- Listado del directorio /usr/bin 3.- Directorio actual: /usr/bin</pre>	<pre>1.- Entrar al directorio /usr 2.- Listado del directorio /usr/bin 3.- Directorio actual: /usr</pre>

Ambos grupos de órdenes se utilizan para procesar la salida de varios procesos como una sola. Por ejemplo:

```
$ ( echo bb; echo ca; echo aa; echo a ) | sort
a
aa
bb
ca
```

12. FUNCIONES

Como en casi todo lenguaje de programación, se pueden utilizar funciones para agrupar trozos de código de una manera más lógica, o practicar la recursión.

Declarar una función es sólo cuestión de escribir:

```
function mi_func
{ mi_código }
```

Llamar a la función es como llamar a otro programa, sólo hay que escribir su nombre.

12.1. Ejemplo de funciones

```
#!/bin/bash

# Se define la función salir
function salir {
    exit
}

# Se define la función hola
function hola {
    echo ;Hola!
}

hola          # Se llama a la función hola
salir         # Se llama a la función salir
echo petete
```

Tenga en cuenta que una función no necesita ser declarada en un orden específico.

Cuando ejecute el script se dará cuenta de que: primero se llama a la función “hola”, luego a la función “salir”, y el programa nunca llega a la línea “echo petete”.

12.2. Ejemplo de funciones con parámetros

```
#!/bin/bash
function salir {
    exit
}

function e {
    echo $1
}

e Hola
e Mundo
salir
echo petete
```

Este script es casi idéntico al anterior. La diferencia principal es la función “e”, que imprime el primer argumento que recibe. Los argumentos, dentro de las funciones, son tratados de la misma manera que los argumentos suministrados al script. (Véase el apartado “Variables y parámetros”).

13. DEPURACIÓN

Una buena idea para depurar los programas es la opción `-x` en la primera línea:

```
#!/bin/bash -x
```

Como consecuencia, durante la ejecución se va mostrando cada línea del guión después de sustituir las variables por su valor, pero antes de ejecutarla.

Otra posibilidad es utilizar la opción `-v` que muestra cada línea como aparece en el script (tal como está en el fichero), antes de ejecutarla:

```
#!/bin/bash -v
```

Otra opción es llamar al programa usando el ejecutable `bash`. Por ejemplo, si nuestro programa se llama `pract`, se podría invocar como:

```
$ bash -x pract
ó
$ bash -v pract
```

Ambas opciones pueden ser utilizadas de forma conjunta:

```
#!/bin/bash -xv
ó
$ bash -xv pract
```

14. PATRONES DE USO DEL SHELL

En esta sección se introducen patrones de código para la programación *shell*. ¿Qué es un patrón? Un patrón es una solución documentada para un problema típico. Normalmente, cuando se programa en *shell* se encuentran problemas que tienen una muy fácil solución, y a lo largo del tiempo la gente ha ido recopilando las mejores soluciones para ellos. Los patrones expuestos en este apartado han sido extraídos en su mayoría de <http://c2.com/cgi/wiki?UnixShellPatterns>, donde pueden encontrarse algunos otros adicionales con también bastante utilidad.

14.1. Comprobación de cadena vacía

La cadena vacía a veces da algún problema al tratar con ella. Por ejemplo, considérese el siguiente trozo de código:

```
if [ $a = " " ] ; then echo "cadena vacia" ; fi
```

¿Qué pasa si la variable `a` es vacía? pues que la orden se convierte en:

```
if [ = " " ] ; then echo "cadena vacia" ; fi
```

Lo cual no es sintácticamente correcto (falta un operador a la izquierda de “=”). La solución es utilizar comillas dobles para rodear la variable:

```
if [ "$a" = " " ] ; then echo "cadena vacia" ; fi
```

o incluso mejor, utilizar la construcción:

```
if [ "x$a" = x ] ; then echo "cadena vacia" ; fi
```

La “x” inicial impide que el valor de la variable se pudiera tomar como una opción.

14.2. Uso de `xargs`

Muchos de los comandos de UNIX aceptan varios ficheros. Por ejemplo, imaginemos que queremos listar todos los directorios que están especificados en una variable:

```
dirs="a b c"
for i in $dirs ; do
    ls $i
done
```

Esto tiene un problema: lanza tres (o n) subshells, y ejecuta n veces `ls`. Este comando también acepta la sintaxis:

```
dirs="a b c"
ls $dirs
```

Una alternativa a esto cuando, por ejemplo, los argumentos están en un fichero, es utilizar `xargs`. Imaginemos que el fichero «directorios» contiene los directorios a listar. El programa `xargs` acepta un conjunto de datos en la entrada estándar y ejecuta la orden con todos los parámetros añadidos a la misma:

```
cat directorios | xargs ls
```

Esto ejecuta el programa `ls` con cada línea del fichero como argumento.

14.3. Leer un fichero línea a línea

A veces surge la necesidad de leer y procesar un fichero línea a línea. La mayoría de las utilidades de UNIX tratan con el fichero como un todo, y aunque permiten separar un conjunto de líneas, no permiten actuar una a una. La orden `read` ya se vió para leer desde el teclado variables, pero gracias a la redirección se puede utilizar para leer un fichero. Éste es el patrón:

```
while read i ; do
    echo "Línea: $i"
    # Procesar $i (línea actual)
done < $fichero
```

El bucle termina cuando la función `read` llega al final del fichero de forma automática.

14.4. Comprobar si una determinada variable posee un valor numérico válido

Esto puede ser muy útil para comprobar la validez de un argumento numérico. Por ejemplo:

```
if echo $1 | grep -x -q "[0-9]\+"
then
    echo "El argumento $1 es realmente un número natural."
else
    echo "El argumento $1 no es un número natural correcto."
fi
```

14.5. Leer argumentos opcionales de la línea de comandos

Aunque puede hacerse mediante programación “convencional”, el *bash* ofrece una alternativa interesante para esta tarea. Se trata de la orden *getopts*. La mejor manera de ver cómo se utiliza es con un ejemplo:

```
while getopts t:r:m MYOPTION
do
case $MYOPTION in
    t) echo "El argumento para la opción -t es $OPTARG"
        ;;
    r) echo "El índice siguiente al argumento de -r es $OPTARG"
        ;;
    m) echo "El flag -m ha sido activado"
        ;;
    ?) echo "Lo siento, se ha intentado una opción no existente";
        exit 1;
        ;;
esac
done
```

Podemos ahora probar el efecto de una invocación del guión como ésta:

```
./guion -m -r hola -t adios -l
```

La salida sería la siguiente:

```
El flag -m ha sido activado
El índice siguiente al argumento de -r es 4
El argumento para la opción -t es adios
./guion: opción ilegal -- l
Lo siento, se ha intentado una opción no existente
```

(El mensaje de la cuarta línea es en realidad enviado a la salida estándar de error, por lo que si se quisiera se podría eliminar redireccionando con `2> /dev/null`). Como puede observarse, *getopts* comprueba si las opciones utilizadas están en la lista permitida o no, y si han sido llamadas con un argumento adicional (indicado por los `:`) en la cadena de entrada `"t:r:m"`. Las variables `$MYOPTION`, `$OPTIND` y `$OPTARG` contienen en cada paso del bucle, respectivamente, el carácter con la opción reconocida, el lugar que ocupa el siguiente argumento a procesar, y el parámetro correspondiente a la opción reconocida (si ésta iba seguida de `:` en la cadena de entrada).

15. EJEMPLOS DE GUIONES SHELL

1. El siguiente programa **llamar**, muestra su número PID y después llama a un programa llamado **num**, a través de la orden (`.`). Cuando **num** termina su ejecución, la orden (`.`) devuelve el control al programa que lo llamó, el cual muestra el mensaje.

Guión llamar

```
echo "$0 PID = $$"
. num
echo "se ejecuta esta línea"
```

Guión num

```
echo "num PID = $$"
```

Como vemos, la orden (`.`) ejecuta un proceso como parte del proceso que se está ejecutando (**llamar** y **num** tienen el mismo número de proceso). Cuando el nuevo programa termina la ejecución, el proceso actual continúa ejecutando el programa original. El programa **num** no necesita permiso de ejecución.

2. Programa que copia un fichero en otro, controlando que el número de argumento sea exactamente dos.

```
if [ $# != 2 ]
then
    echo "utilice: copia [desde] [hasta]"
    exit 1
fi
desde=$1
hasta=$2
if [ -f "$hasta" ]
then
    echo "$hasta ya existe, ¿ desea sobrescribirlo (s/n)?"
    read respuesta
    if [ "$respuesta" != "s" ]
    then
```

```

        echo "$desde no copiado"
        exit 0
    fi
fi
cp $desde $hasta

```

3. Programa que imprime el pantalla el contenido de un fichero de datos, o el contenido de todos los ficheros de un directorio.

```

if test -f "$1"
then
    pr $1|less
elif test -d "$1"
then
    cd $1; pr *|less
else
    echo "$1 no es un fichero ni un directorio"
fi

```

4. Programa que evalúa la extensión de un fichero. Si ésta se corresponde con "txt", copia el fichero al directorio ~/copias. Si es otra la extensión o no hace nada o presenta un mensaje.

```

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

```

5. Programa que borra con confirmación todos los ficheros indicados como argumentos en la línea de órdenes.

```

#!/bin/bash
while test "$1" != ""
do
    rm -i $1
    shift
done

```

6. Programa que hace múltiples copias de ficheros a pares. En cada iteración desaparecen el primer y segundo argumento.

```

while test "$2" != ""
do
    echo $1 $2
    cp $1 $2
    shift; shift
done

```

```

if test "$1" != ""
then
    echo "$0: el número de argumentos debe ser par y > 2"
fi

```

7. Ejemplo *break* y *continue*: este programa utiliza las órdenes *break* y *continue* para permitir al usuario controlar la entrada de datos.

```

while true          #bucle infinito
do
    echo "Introduce un dato "
    read respuesta
    case "$respuesta" in
        "nada") # no hay datos
                break
                ;;
        "") # si es un retorno de carro se continúa
            continue
            ;;
        *) # proceso de los datos
            echo "se procesan los datos"
            ;;
    esac
done

```

8. Ejemplo de un menú:

```

while true
do
    clear
    echo "
    Ver directorio actual.....[1]
    Copiar ficheros.....[2]
    Editar ficheros.....[3]
    Imprimir fichero.....[4]
    Salir del menú.....[5]"
    read i
    case $i in
        1)  ls -l|more; read z
            ;;
        2)  echo "Introduzca [desde] [hasta]"
            read x y
            cp $x $y
            read x
            ;;
        3)  echo "¿Nombre de fichero a editar?"
            read x;
            vi $x
            ;;
        4)  echo "¿Nombre de fichero a imprimir?"
            read x
            lpr $x
    esac
done

```

```

        ;;
5)    clear; break
        ;;
    esac
done

```

Este mismo ejercicio podría ser resuelto utilizando la orden *select*:

```

select opcion in "Ver directorio actual" \
  "Copiar ficheros" \
  "Editar ficheros" \
  "Imprimir fichero" \
  "Salir del menú"
do
  case $REPLY in
1)    ls -l|more; read z
        ;;
2)    echo "Introduzca [desde] [hasta]"
        read x y
        cp $x $y
        read x
        ;;
3)    echo "¿Nombre de fichero a editar?"
        read x;
        vi $x
        ;;
4)    echo "¿Nombre de fichero a imprimir?"
        read x
        lpr $x
        ;;
5)    clear; break
        ;;
    esac
done

```

9. Este ejemplo lee dos números del teclado e imprime su suma, (usando las órdenes *read*, *printf* y *let*).

```

#!/bin/bash
printf "Introduzca un número \n"
read numero1
printf "Introduzca otro número \n"
read numero2
let respuesta=$numero1+$numero2
printf "$numero1 + $numero2 = $respuesta \n"

```

10. Escribir un guión shell que, dado el "username" de un usuario, nos devuelva cuántas veces esa persona está conectada. (Usa: *who*, *grep*, *wc*).

```

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

```

11. Escribir un guión shell llamado **ldir** que liste los directorios existentes en el directorio actual.

```
#!/bin/bash
for archivo in *
do
    test -d $archivo && ls $archivo
done
```

12. Escribir un guión shell llamado **ver** que para cada argumento que reciba realice una de las siguientes operaciones:

- si es un directorio ha de listar los ficheros que contiene,
- si es un fichero regular lo tiene que mostrar por pantalla,
- en otro caso, que indique que no es ni un fichero ni un directorio.

```
#!/bin/bash
for fich in $*
do
    if [ -d $fich ]
    then
        echo "usando ls"
        ls $fich
    elif [ -f $fich ]
    then
        cat $fich
    else
        echo $fich no es ni un fichero ni un directorio
    fi
done
```

13. Escribir un guión shell que solicite confirmación si va a sobrescribir un fichero cuando se use la orden *cp*.

```
#!/bin/bash
if [ -f $2 ]
then
    echo "$2 existe. ¿Quieres sobrescribirlo? (s/n)"
    read sn
    if [ $sn = "N" -o $sn = "n" ]
    then
        exit 0
    fi
fi
cp $1 $2
```

14. Supongamos que queremos cambiar el sufijo de todos los archivos *.tex a .latex. Haciendo *mv *.tex *.latex* no funciona, (¿por qué?), pero sí con un guión shell.

```
#!/bin/bash
for f in *.tex
do
    nuevo=$(basename $f tex)latex
    mv $f $nuevo
done
```

15. Hacer un programa que ponga el atributo de ejecutable a los archivos pasados como argumento.

```
for fich in $@
do
    if test -f $fich
    then
        chmod u+x $fich
    fi
done
```

16. Lo siguiente es un sencillo reloj que va actualizándose cada segundo, hasta ser matado con Ctrl-C:

```
while true
do
    clear;
    echo "===== ";
    date +"%r";
    echo "===== ";
    sleep 1;
done
```

16. EJERCICIOS PROPUESTOS

1. Haga un shell script llamado **priult** que devuelva los argumentos primero y último que se le han pasado. Si se llama con:

```
priult hola qué tal estás
```

debe responder:

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

Mejorar este shell script para tratar los casos en los que se llame con 0 o 1 argumento, indicando que no hay argumento inicial y/o final.

2. Cree un shell script llamado **num_arg**, que devuelva el número de argumentos con el que ha sido llamado. Devolverá 0 (éxito) si se ha pasado algún argumento y 1 (error) en caso contrario. Mejorar este shell de forma que muestre una lista de todos los argumentos pasados o bien que indique que no tiene argumentos:

```
Los argumentos pasados son:
ARGUMENTO NÚMERO 1: X1
...
ARGUMENTO NÚMERO N: XN
ó
No se han pasado argumentos
```

3. Cree un shell script llamado **fecha_hora** que devuelva la hora y la fecha con el siguiente formato:

```
Son las hh horas, xx minutos del día dd de mmm de aaaa
donde mmm representa las iniciales del mes en letra
(ENE, FEB, MAR, ..., NOV, DIC).
```

4. Cree un shell script llamado **doble** que pida un número por teclado y calcule su doble. Debe comprobar el número introducido y antes de terminar preguntará si deseamos calcular otro doble, en cuyo caso no terminará. Ejemplo:

```
Introduzca un número para calcular el doble: 89
El doble de 89 es 178
¿Desea calcular otro doble (S/N)?
```

5. Cree un shell script llamado **tabla** que a partir de un número que se le pasará como argumento obtenga la tabla de multiplicar de ese número. Si se llama con:

```
tabla 5
```

debe responder:

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

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

6. Haga un shell script llamado **cuenta_tipos** que devuelva el número de ficheros de cada tipo que hay en un directorio, así como los nombres de estos ficheros. Tendrá un único argumento (opcional) que será el directorio a explorar. Si se omite el directorio se considerará que se trata del directorio actual. Devolverá 0 (éxito) si se ha llamado de forma correcta y 1 (error) en caso contrario. La salida será de esta forma:

```
La clasificación de ficheros del directorio XXXX es:
Hay t ficheros de texto: X1, X2, ... Xt
Hay dv ficheros de dispositivo: X1, X2, ... Xdv
Hay d directorios: X1, X2, ... Xd
Hay e ficheros ejecutables: X1, X2, ... Xe
```

(Pista: usar la orden *file*)

7. Cree un shell script llamado **instalar** al que se le pasarán dos argumentos: fichero y directorio. El shell script debe copiar el fichero al directorio indicado. Además debe modificar sus permisos de ejecución de forma que esté permitido al dueño y al grupo del fichero y prohibido al resto de usuarios. Antes de hacer la copia debe verificar los argumentos pasados, si se tiene permiso para hacer la copia, si el fichero es de texto o ejecutable, etc. Devolverá 0 (éxito) si todo ha ido bien y 1 (error) en caso contrario.
8. Cree un shell script llamado **infosis** que muestre la siguiente información:

- Un saludo de bienvenida del tipo:

```
Hola usuario uuu, está usted conectado en el terminal ttt
```

donde “uuu” y “ttt” son, respectivamente, el nombre de usuario y el terminal desde el que se ejecuta la orden.

- La fecha y la hora actuales, usando para ello el ejercicio número 3.

- Una lista con los usuarios conectados.
- Una lista de los procesos del usuario que se están ejecutando en ese momento.

9. Cree un guión shell llamado **infouser** que reciba un único parámetro (el login de un usuario) y que muestre la siguiente información:

- Login.
- Nombre completo del usuario.
- Directorio home.
- Shell que utiliza.
- Una línea que indique si el usuario está actualmente conectado o no.
- Procesos pertenecientes a dicho usuario. La información a mostrar para cada proceso debe ser el PID y la línea de órdenes que dio lugar a la creación de dicho proceso.

El guión debe comprobar:

- Si las opciones y parámetros son correctos.
- Si el usuario que se pasa como parámetro existe o no.

Además, debe permitir las siguientes opciones:

- *-p*: sólo muestra información de procesos.
- *-u*: muestra toda la información excepto la referente a los procesos.
- *--help*: muestra información de ayuda (lo que hace el guión, su sintaxis y significado de opciones y parámetros).

Los códigos de retorno deben ser:

- 0: éxito.
- 1: no se ha respetado la sintaxis de la orden.
- 2: usuario no existe.

Nota: parte de la información del usuario se puede obtener del fichero `/etc/passwd`, en las salas de prácticas ejecutando la orden `ypcat passwd`. Pueden ser de utilidad las órdenes *getopts* y *finger*.

10. Cree un shell script llamado **bustr**, al que se le pase como parámetro una cadena y una lista de 0 a n nombres de fichero. El shell script debe devolvernos los nombres de los archivos que contienen en su interior la cadena especificada. Para evitar errores sólo se hará con los archivos que sean regulares y sobre los que tengamos permiso de lectura. Por ejemplo:

```
bustr cadena fichero1 fichero2 fichero 3
```

devolvería:

La cadena "cadena" se ha encontrado en los siguientes ficheros:

```
fichero2
fichero3
```

¿Cómo podría llamar a **bustr** para que hiciera la búsqueda en todos los ficheros a partir de un directorio dado e incluyendo subdirectorios? Pista *bustr cadena '...'*

11. Construir un guión shell en Linux con la siguiente sintaxis

```
diffd [-i] directorio1 [directorio2]
```

- **Función:** debe mirar todos los nombres de fichero contenidos en el *directorio1* y en el *directorio2* (a excepción de los nombres de directorios) y mostrar las diferencias, es decir, mostrar el nombre de aquellos ficheros que aparecen en uno de los directorios pero no en el otro, indicando para cada uno de ellos el directorio en el que se encuentra.
- **Parámetros:** *directorio1* y *directorio2*: directorios entre los que se hace la comparación. Si se omite *directorio2* (que es opcional) se entenderá que queremos comparar *directorio1* con el directorio en el que nos encontremos al ejecutar **diffd**.
- **Opciones:** *-i*: invierte el funcionamiento de **diffd** haciendo que muestre los nombres de aquellos ficheros que se encuentran en los dos directorios (es obvio que en este caso no hay que indicar el directorio en el que aparece el fichero, pues aparece en ambos).

12. En RedHat se utiliza el formato de paquetes *RPM* que facilita la administración de los mismos: instalación, eliminación, verificación de integridad, dependencias, etc. El programa utilizado para llevar a cabo dicha administración se llama *rpm*. Cree un guión shell llamado **estadopaquetes** que reciba como parámetro un directorio y que, para cada uno de los paquetes que existan en ese directorio (si existen), indique su estado que puede ser uno de los 4 siguientes:

- No instalado
- Instalado
- A actualizar (cuando la versión instalada del paquete es más antigua que la que tiene el paquete)
- ¿A borrar? (cuando la versión instalada del paquete es más reciente que la que tiene el paquete).

El directorio pasado como parámetro es opcional y, si no se indica, se asumirá el directorio actual. Supondremos que el formato del nombre del fichero del paquete es:

```
nombrepaquete-versión-revisión.arquitectura.rpm
```

El nombre del paquete puede tener guiones por lo que habrá que controlar esto. Un ejemplo de paquete es *man-pages-es-0.7a-1.noarch.rpm*. No tendremos en cuenta los paquetes con código fuente, es decir, aquellos de extensión *src.rpm*.

13. Hacer un guión shell llamado **mirm** que mueva al directorio `~/ .borrados` los ficheros (nunca directorios) que se le indiquen como parámetros. Este guión shell viene acompañado por otro guión llamado **limpiezad** que se debe ejecutar en segundo plano y que cada 10 segundos compruebe si tiene que borrar ficheros del directorio `~/ .borrados` en función del espacio ocupado en disco. Ambos guiones shell hacen uso de la variable de entorno *MINLIBRE* que indica el mínimo espacio en disco que debe quedar. De tal manera que una condición que deben cumplir ambos guiones es que, tras su ejecución, el espacio libre en disco (en bloques de 1K) debe ser igual o superior a *MINLIBRE*. En el caso de que, para cumplir dicha condición, haya que borrar ficheros de `~/ .borrados`, se seguirá una política **FIFO**.

Nota: basta con que ambos guiones funcionen para la partición en la que se encuentra el directorio personal del usuario.

Además, ambos guiones deben hacer uso del fichero `~/ .listaborrados` que guardará, sólo para cada fichero presente en `~/ .borrados`, su ruta de acceso original.

Mejora: debe permitirse el borrado de ficheros con igual nombre.

14. Como complemento del ejercicio anterior, hacer un guión shell llamado **recuperar** que se utilizará para recuperar ficheros borrados. Si se le pasa un único parámetro se entenderá que se trata de un patrón y entonces mostrará todos los ficheros de `~/ .borrados` que cumplan dicho patrón. Si se le pasan dos parámetros, el primero se entenderá como antes pero el segundo debe ser el nombre de un directorio y, en

este caso, se recuperarán todos los ficheros borrados que cumplan el patrón y se dejarán en el directorio indicado como segundo parámetro.

Además, este guión debe implementar también la opción “-o <patrón>” que recuperará todos los ficheros que cumplan <patrón> y los copiará a su posición original, según lo indicado por el fichero `~/ .listaborrados`. Si el directorio original ya no existe, debe crearse siempre que se tenga permiso.

Mejora: debe permitirse la recuperación de ficheros con igual nombre.

15. Cree un shell script llamado **agenda** al que se le pasará un argumento (opcional), que será el nombre de fichero que se usará para almacenar la información (si se omite el argumento, el fichero será *agenda.dat*, creándose en blanco si no existe). Cada línea del fichero tendrá el siguiente formato:

```
nombre:localidad:saldo:teléfono
```

Cuando se ejecute el shell script mostrará un prompt para poder introducir las distintas opciones disponibles:

```
AGENDA (Introduzca opción. 'h' para ayuda) >>
```

Las opciones que debe soportar son:

- *h*: mostrará ayuda de todas las opciones.
- *q*: para salir de la agenda.
- *l*: listará el fichero de la agenda en columnas:

```
----- AGENDA -----
Nombre           Localidad        Saldo           Teléfono
-----
Juan Ruiz        Cartagena        134            968507765
Jaime López      Málaga          95             952410455
.....
Ana Martínez    Madrid          945            914678984
```

- *on*: ordenará la agenda por nombre ascendentemente. La ordenación no se mostrará y quedará en el fichero. Para ver la ordenación habrá que ejecutar *l* después.
- *os*: ordenará la agenda por saldo descendentemente (¡ojo!, numéricamente). La ordenación no se mostrará y quedará en el fichero. Para ver la ordenación habrá que ejecutar *l* después.
- *a*: añadirá una línea. Para ello el shell script debe preguntar por el nombre, localidad, saldo y teléfono, comprobando que ninguno de los campos quede en blanco. Una vez introducidos todos los datos de una nueva línea, debe añadirse al final del fichero de la agenda. Como mejora, antes de introducir la nueva línea se puede comprobar que no existe ninguna otra con el mismo nombre de persona.
- *b*: borrará una línea. Para ello el shell script debe preguntar el nombre exacto de la persona correspondiente. Una vez introducido éste se debe eliminar la línea o líneas que tengan ese nombre exactamente (pueden ser varias si en el punto anterior se permiten líneas con el mismo nombre). Antes de proceder con el borrado debe pedir confirmación.

17. BIBLIOGRAFÍA

- Página de manual del intérprete de órdenes bash.
- <http://www.insflug.org/bajar.php3?comoID=121>. *Programación en BASH - COMO de introducción*. Mike G. (traducido por Gabriel Rodríguez).
- <http://c2.com/cgi/wiki?UnixShellPatterns>. *Unix shell patterns*, J. Coplien *et al.*
- <http://learnlinux.tsf.org.za/courses/build/shell-scripting>. *Shell scripting*, Hamish Whittal.