Prácticas de Fundamentos del Software

Módulo I. Órdenes UNIX y Shell Bash

Sesión Nº6: Programación del shell

7-Nov-2011

1 Objetivos principales

- Aprender a utilizar las órdenes de control de flujo para alterar la ejecución secuencial de órdenes y cómo podemos pasar información a un guion desde el teclado.
- Saber definir y usar funciones dentro del bash shell.
- Ampliar el repertorio de órdenes de Unix visto hasta el momento.

Además, se verán las siguientes órdenes:

Órdenes Linux			
:	for	while	until
case	select	seq	read
break	true	continue	return
tar	gzip	cut	

Tabla 1. Órdenes de la sesión.

En esta sesión, veremos cómo construir guiones más complejos, introduciendo nuevas órdenes, que si bien pueden ser usadas de modo interactivo (como vimos que ocurría con `if` en la Sesión 5), muestran su gran potencia en programas shell. Las órdenes de control de flujo que veremos son:

- **`for`** ejecuta una lista de declaraciones un número fijo de veces.
- **`while`** ejecuta una lista de declaraciones cierto número de veces mientras cierta condición se cumple.
- **`until`** ejecuta una lista de declaraciones repetidamente hasta que se cumpla cierta condición.
- **`case`** ejecuta una de varias listas de declaraciones dependiendo del valor de una variable.

2 Orden for

El bucle for permite repetir cierto número de veces las declaraciones especificadas. Su sintaxis es:

```
for nombre [in lista]
do
      declaraciones que pueden usar $nombre
done
```

Podemos ver de qué tipo son todos los archivos de nuestro directorio de trabajo con el siguiente guion:

```
for archivo in $(ls)
do
   file $archivo
done
```

En el siguiente ejemplo generamos los 5 primeros impares mediante la orden **`seq`** genera una secuencia de números entre el valor indicado como primer argumento (este será el valor inicial), tomando como incremento el segundo, hasta el valor final dado como tercer argumento.

```
for NUM in `seq 0 1 4`;
do
 let "NUM=$NUM * 2 + 1"
 printf "Número impar %d\n" $NUM
done
```

Podemos construir un bucle clásico de C `(())`, como muestra el ejemplo siguiente:

```
#!/bin/bash
# forloop.sh: Cuenta desde 1 a 9
for (( CONTADOR=1; CONTADOR<10; CONTADOR++ )) ; do
  printf "Contador vale ahora %d\n" $CONTADOR
done
```

En este bucle se declara e inicializa la variable `CONTADOR=1` y el bucle finaliza cuando la variable vale `10 (CONTADOR<10)`. En cada iteración, la variable se incrementa en `1 (CONTADOR++)`. Es posible que la condición tenga varias variables.

Ejercicio 1. Escriba un guion que acepte dos argumentos. El primero será el nombre de un directorio y el segundo será un valor entero. El funcionamiento del guion es el siguiente: deberán anotarse en un archivo denominado `archivosSizN.txt` aquellos archivos del directorio dado como argumento y que cumplan la condición de tener un tamaño menor al valor aportado en el segundo argumento. Se deben tener en cuenta las comprobaciones sobre los argumentos, es decir, debe haber dos argumentos, el primero deberá ser un directorio existente y el segundo un valor entero.

Ejercicio 2. Escriba un guion que acepte el nombre de un directorio como argumento y muestre como resultado el nombre de todos y cada uno de los archivos del mismo y una leyenda que diga "Directorio", "Enlace" o "Archivo regular", según corresponda. Incluya la comprobación necesaria sobre el argumento, es decir, determine si el nombre aportado se trata de un directorio existente.

3 Orden case

`Case` compara una variable contra una serie de valores o patrones. Si el valor o el patrón coinciden, se ejecutan las declaraciones correspondientes. La sintaxis para esta orden es:

```
case expresión in
      patron1 )
            declaraciones;;
      patron2 )
            declaraciones;;
      …
esac
```

Como ejemplo, vamos a construir un fragmento de código que permite al usuario elegir entre dos opciones. Si elegimos la primera, borramos un archivo; si es la segunda, hacemos una copia de seguridad de él. Se pedirá al usuario que introduzca de forma interactiva mediante el teclado (sección 6, página 6).

```
printf "%s -> " "1 = borrar, 2 = copiar. Elija una opción"
read REPLY
case "$REPLY" in
      1) rm "$TEMPFILE" ;;
      2) mv "$TEMPFILE" "$TEMPFILE.old" ;;
      *) printf "%s\n" "$RESPUESTA no es una de las opciones" ;;
esac
```

En la línea 6 hemos puesto el patrón asterisco `(*)` que se iguala en cualquier otro caso: si no se ha dado una de las respuestas anteriores se ejecutarán las declaraciones de esta parte.

Ejercicio 3. Escriba un guion en el que, a partir de la pulsación de una tecla, detecte la zona del teclado donde se encuentre. Las zonas vendrán determinadas por las filas. La fila de los números `1, 2, 3, 4, ...` será la fila 1, las teclas donde se encuentra la `Q, W, E, R, T, Y,...` serán de la fila 2, las teclas de la `A, S, D, F, ...` serán de la fila 3 y las teclas de la `Z, X, C, V, ...` serán de la fila 4. La captura de la tecla se realizará mediante la orden `read`. (vea sección 6 en página 6).

Ejercicio 4. Escriba un guion que acepte como argumento un parámetro en el que el usuario indica el mes que quiere ver, ya sea en formato numérico o usando las tres primeras letras del nombre del mes y muestre el nombre completo del mes introducido. Si el número no está comprendido entre 1 y 12 o las letras no son significativas del nombre de un mes, el guion deberá mostrar el correspondiente mensaje de error.

4 Órdenes while y until

Las órdenes **`while`** y **`until`** permiten repetir una serie de declaraciones. La orden **`while`** ejecuta las declaraciones comprendidas entre **`do`** y **`done`** mientras que la expresión sea *true* (`expresión` puede ser una condición o una expresión de control). Si `expresión` falla en el primer intento, nunca se ejecutan las declaraciones del cuerpo de **`while`**. La sintaxis para **`while`** es:

```
while expresión;
do
      declaraciones …
done
```

En el ejemplo siguiente hemos incluido en la línea 2 la orden **`read`** que hará que se almacene en la variable `$ARCHIVO` lo que se escriba en el teclado mostrándonos el texto `"Archivo ?"`. Es importante que comprenda que este bucle encontrará falsa su condición cuando demos <control-d> (que siempre tendrá el significado de fin de archivo). Observe la potencialidad de este lenguaje en que con pocas instrucciones estamos consiguiendo una funcionalidad grande. Una vez que se almacene en la variable `$ARCHIVO` un valor, en la línea 3 la orden **`test`** chequea si existe o no dicho archivo cuyo nombre ha introducido el usuario por teclado.

```
printf "%s\n" "Introduzca los nombres de archivos o <control-d> para finalizar"

while read -p "Archivo ?" ARCHIVO ;
do
   if test -f "$ARCHIVO" ;
   then
      printf "%s\n" "El archivo existe"
   else
      printf "%s\n" "El archivo NO existe"
   fi
done
```

Podemos crear un bucle infinito utilizando la orden **`true`**. Dado que true siempre tiene éxito, el bucle se ejecuta indefinidamente:

```
printf "%s\n" "Introduzca los nombres de archivos o teclea FIN"

while true ;
do
   read -p "Archivo ?" ARCHIVO
   [ "$ARCHIVO" = "FIN" ] ;
   then
      break
   elif test -f "$ARCHIVO" ;
   then
       printf "%s\n" "El archivo existe"
   else
       printf "%s\n" "El archivo NO existe"
   fi
done
```

Hemos utilizado la orden **`break`** para finalizar el bucle si se escribe `FIN`. Cuando el shell encuentra esta orden, sale del bucle (orden **`while`**) y continúa ejecutando la orden siguiente. La orden **`break`** puede venir seguida de un número que indica cuántos bucles anidados romper, por ejemplo, `break 2`.

Otra orden ligada a los bucles es la orden **`continue`**, que permite iniciar una nueva iteración del bucle saltando las declaraciones que haya entre ella y el final del mismo. Por ejemplo, podemos generar hasta nueve números aleatorios siempre y cuando sean mayores de 20000. Para ello, utilizamos las variable `$RANDOM` que devuelve un entero diferente entre 0 y 32676 cada vez que la consultamos.

```
for n in {1..9} ## Ver uso de llaves en sesiones anteriores
do
   x=$RANDOM
   [ $x -le 20000 ] && continue
   echo "n=$n x=$x"
done
```

La orden **`until`** es idéntica a **`while`** excepto que se repite el cuerpo mientras la condición sea cierta, esencialmente es lo mismo que **`while`**. La sintaxis para **`until`** es:

```
until expresión;
do
      declaraciones …
done
```

Ahora, vamos a ver un ejemplo donde expresión es una condición en lugar de una orden, como ocurría en los ejemplos anteriores. El ejemplo genera los 10 primeros números:

```
n=1
until [ $n -gt 10 ]
do
   echo "$n"
   n=$(( $n + 1 ))
done
```

Ejercicio 5. Escriba un guion que solicite un número hasta que su valor esté comprendido entre 1 y 10. Deberá usar la orden **`while`** y, para la captura del número, la orden **`read`** (ver página 6).

Algo bastante frecuente es la construcción de menús, a continuación vemos un ejemplo. En la línea 1 aparece como expresión del bucle while la orden `:` que es equivalente a "no operación".

```
while :
do
   printf "\n\nMenu de configuración:\n"
   printf "\tInstalar ...  [Y, y]\n"
   printf "\tNo instalar [N, n]\n"
   printf "\tPara finalizar pulse 'q'\n"

   read -n1 -p "Opción: " OPCION
   case $OPCION in
       Y | y ) printf "\nHas seleccionado %s \n" $OPCION;;
       N | n ) printf "\nHas seleccionado %s \n" $OPCION;;
       q ) printf "\n"
           break;;
       * ) printf "\n Selección invalida: %s \n";;
   esac
   sleep 2  # duerme durante 2 segundos
done
```

5 Funciones

Una de las características de Bash es que nos permite definir funciones. Éstas, a diferencia de los guiones, se ejecutan dentro de la memoria del proceso bash que las utiliza, por lo que su invocación es más rápida que invocar a un guion u orden de Unix. Definimos una función con la orden **`function`**:

```
function nombre_fn {
      declaraciones
}
```

De forma inversa, podemos borrar una función con **`unset -f nombre_fn`**. Podemos ver qué funciones tenemos definidas junto con su definición con `declare -f`, y solo su nombre con `declare -F`. ¿Qué ocurre si una función tiene el mismo nombre que un guion o una orden? El shell siempre utiliza el orden de preferencia que se cita a la hora de resolver un símbolo:

1. Alias
2. Palabra clave (como if, function, etc.)
3. Funciones
4. Órdenes empotradas
5. Guiones y programas ejecutables

Como norma de estilo, para evitar la confusión de nombres entre funciones y órdenes, podemos nombrar las funciones con un signo "_" delante del nombre (por ejemplo, `_mifuncion`). De todas formas, volviendo a la pregunta, y según la ordenación citada, una función con el mismo nombre que una orden empotrada se ejecutaría antes que la orden.

Para invocar una función dentro de un guion solamente debemos escribir su nombre seguido de los argumentos correspondientes, si los hubiera. En este sentido, las funciones utilizan sus propios parámetros posicionales. Una función puede tener variables locales.

Para ello, debemos declararlas dentro de la función con el modificador `local`. Por ejemplo:

```
#!/bin/bash
# file.sh: guion shell para mostrar el uso de funciones
# ------------------------------------------------------
# definimos la funcion uso()
uso()
{
   echo "Uso: $0 nombre_archivo"
   exit 1
}
# ------------------------------------------------------
# definimos la función si_existe_file
# $f -> almacena los argumentos pasados al guion
si_existe_file()
{
   local f="$1"
   [[ -f "$f" ]] && return 0 || return 1
}
# invocamos uso
# llamamos a uso() si no se da el nombre de archivo
[[ $# -eq 0 ]] && uso

# invocamos a si_existe_file()
if ( si_existe_file "$1" )
then
   echo "El archivo existe"
else
   echo "El archivo NO existe"
fi
```

Es posible utilizar los valores devueltos por una función. En el ejemplo siguiente, la función devuelve el doble de un número que se pasa como argumento; en la linea 7 recogemos en la variable `$resultado` el valor devuelto por `dbl`:

```
#!/bin/bash
# usamos echo para devolver un valor

function dbl
{
   read -p "Introduzca un valor: " valor
   echo $[ $valor * 2 ]
}

resultado=`dbl`
echo "El nuevo valor es $resultado"
```

6 Lectura del teclado

En nuestros guiones podemos leer desde el teclado usando la orden **`read`**, que detiene la ejecución del guion y espera a que el usuario teclee algo de forma que el texto tecleado es asignado a la(s) variable(s) que acompañan a la orden. Su sintaxis es:

`read [-ers] [-a array] [-d delim] [-i text] [-n nchars] [-p prompt] [-t timeout] [-u fd] [name …]`

Podéis consultar las opciones con `help read`. Por ejemplo, podemos leer el nombre de un archivo de la forma:

```
printf "Introduzca el nombre del archivo:"
read ARCHIVO_INPUT
```

O bien:

```
read -p "Introduzca el nombre del archivo:" ARCHIVO_INPUT
```

En ambos casos, la variable `$ARCHIVO_INPUT` contiene los caracteres tecleados por el usuario.

También podemos leer de un archivo como muestra el ejemplo siguiente:

```
#!/bin/bash
# Leer datos de un archivo

count=1
cat test | while read linea ;
do
       echo "Linea $count: $linea"
       count=$[ $count + 1]
done
echo "Fin de procesamiento del archivo"
```

Otra forma de hacer lo mismo utilizando redirecciones, sería:

```
while read line ;
do
   echo "Line $count: $line"
   count=$[ $count + 1]
done < test
echo "Finished processing the file"
```

7 Ejemplos realistas de guiones

El objetivo de este apartado es mostrar algunos ejemplos de guiones completos para ilustrar todo su potencial.

7.1 Elimina directorios vacíos

El siguiente guion podrá ser llamado con argumentos o sin ellos. Si hay argumentos, deseamos que se borren los directorios vacíos que cuelguen de cada uno los argumentos indicados (que deben ser nombres de directorios existentes). Si no hay argumentos, deseamos que la operación se realice sobre el directorio actual. Observe la forma en que ésto se ha programado en el guion siguiente.

```
# rmemptydir - remove empty directories
# Heiner Steven (heiner.steven@odn.de), 2000-07-17
#
# Category: File Utilities

[ $# -lt 1 ] && set -- .

find "$@" -depth -print  -type d |
    while read dir
    do
       [ `ls "$dir" | wc -l` -lt 1 ] || continue
       echo >&2 "$0: removing empty directory: $dir"
       rmdir "$dir" || exit $?
    done
exit 0
```

En la línea 8 la orden **`find`** alude con "`$@`" a todos los parámetros posicionales ("`$@`" será sustituido por `$1`, `$2`, … hasta el final); pero si no se han puesto parámetros, ¿cómo conseguimos que únicamente con esta orden se aluda a `.`?

Vea lo que se hace en la línea 6: si el número de argumentos es <1 entonces deseamos que los parámetros posicionales sean "reemplazados" por ., y esto es justamente lo que hace **`set -- .`** (Consulte la ayuda de bash para ver la explicación completa y detallada de las órdenes empotradas (*shell builting commands*), en concreto `set`) .

¿Cuál es la entrada estándar de la orden `read` que aparece en la línea 9? la salida de la orden `find`, por esa razón la variable `$dir`, en el cuerpo del bucle irá tomando los sucesivos valores que va proporcionando `find`, y el bucle terminará cuando acabe esta lista.

Una vez visto esto, el tratamiento para cada ruta encontrada por `find` es simple: en la línea 11 se chequea el número de hijos, si NO ES < 1 no hacemos nada con este, pasamos al siguiente valor (orden `continue`).

En la línea 13, si ha dado error `rmdir` entonces finalizamos devolviendo el mismo código de error que nos haya devuelto `rmdir`. Si no hemos pasado por aquí, entonces la orden de la línea 15 devolverá el valor 0 indicativo de que no ha habido error.

7.2 Muestra información del sistema en una página html

En el ejemplo siguiente se construye el archivo `status.html` que después podrá abrir con un navegador. En este archivo se va a ir llevando la salida de órdenes como `hostname`, `uname`, `uptime`, `free`, `df`, `netstat` y `w` (use la ayuda para ver qué información proporcionan). En la línea 4 se le da valor a la variable `_STAT` (simplemente el literal `status.html`) y el resto del guion consiste en órdenes que escriben en el archivo cuyo nombre está almacenado en dicha variable.

```
#! /bin/bash
# statgen -- Luke Th. Bullock - Thu Dec 14 11:04:18 MET 2000
#

_STAT=status.html
echo "<html><title>`hostname` Status</title>" > $_STAT
echo "<body bgcolor=white><font color=slategray>" >> $_STAT
echo "<h2>`hostname` status <font size=-1>(updated every 1/2 hour)
</h2></font></font>" >> $_STAT
echo "<pre>" >> $_STAT
echo "<b>Date:</b> `date`" >> $_STAT
echo >> $_STAT
echo "<b>Kernel:</b>" >> $_STAT
uname -a >> $_STAT
echo >> $_STAT
echo "<b>Uptime:</b>" >> $_STAT
uptime >> $_STAT
echo >> $_STAT
echo "<b>Memory Usage:</b>" >> $_STAT
free >> $_STAT
echo >> $_STAT
echo "<b>Disk Usage:</b>" >> $_STAT
df -h >> $_STAT
echo >> $_STAT
echo "<b>TCP connections:</b>" >> $_STAT
netstat -t >> $_STAT
echo >> $_STAT
echo "<b>Users logged in</b> (not showing processes):" >> $_STAT
w -hus >> $_STAT
echo "</pre>" >> $_STAT
```

Para ver el resultado podemos abrir el archivo `status.html` con el navegador. Este tipo de archivo se denomina *CGI* (Common Gateway Interface) que es una tecnología web que permite a un cliente web (navegador) solicitar información de un programa ejecutado en el servidor web.

7.3 Adaptar el guion al sistema operativo donde se ejecuta

La orden **uname** nos da el nombre del sistema en uso. Con ella, podemos construir un guion de la forma que se mostrará a continuación con la idea de conseguir el propósito de que nuestro guion emplee unas órdenes u otras dependiendo del sistema operativo en el que nos encontremos. Puede sustituir los puntos suspensivos del ejemplo por un simple mensaje de texto.

```
SO=`uname`
case $SO in
       Linux)
            # ordenes exclusivas de Linux
            … ;;
       AIX)
            # ordenes exclusivas de AIX
            … ;;
       SunOS)           # ordenes de Solaris
            …;;
       *)   echo "Sistema Operativos no soportado\n"
            exit 1
       ;;
esac
```

7.4 Mostrar una raya girando mientras se ejecuta una orden

A continuación vamos a construir un guion bash que muestra una raya girando mientras se ejecuta una orden que consume mucha CPU. En nuestro caso simulamos la orden que consume CPU mediante un bucle for, pero esta orden podría ser por ejemplo una larga compilación. La idea es que estemos visualizando el giro de una línea mientras se ejecuta el programa de larga duración para que no pensemos que el sistema no nos está respondiendo.

```
#!/bin/bash
# rotor -- Randal M. Michael – Mastering Unix Shell Scripting, Wiley, 2003
#
function rotar_linea {
  INTERVAL=1            # Tiempo a dormir entre giro
  TCOUNT="0"            # Para cada TCOUNT la linea gira 45 grados
  while true            # Bucle infinito hasta que terminamos la funcion
  do
    TCOUNT=`expr $TCOUNT + 1` # Incrementa TCOUNT
    case $TCOUNT in
    "1" ) echo -e "-""\b\c"
    sleep $INTERVAL
    ;;
    "2" ) echo -e '\\'"\b\c"
    sleep $INTERVAL
    ;;
    "3" ) echo -e "|""\b\c"
    sleep $INTERVAL
    ;;
    "4" ) echo -e "/""\b\c"
    sleep $INTERVAL
    ;;
    * ) TCOUNT="0" ;; # Pone a cero TCOUNT
  esac
done
}
######## Cuerpo principal ############
```

```
rotar_linea &     # Ejecuta la funcion rotar_linea de fondo:
                  # rotar-linea se ejecuta simultáneamente al resto del guion
ROTAR_PID=$!      # Captura el PID del último proceso de fondo

# Simulamos la ejecución de una orden que consume mucha CPU
# durante la cual mostramos la linea rotando

for ((CONT=1; CONT<400000; CONT++ )) ;
do
   :
done

# Paramos la funcion rotar_linea


kill -9 $ROTAR_PID # provoca la terminación del proceso cuyo pid es $ROTAR_PID

# Limpiamos el trazo que queda tras finalizar
echo -e "\b\b"
```

La función `rotar_línea` se encarga de representación en pantalla la barra que gira; esta función se construye a partir de la línea 3, está en un bucle infinito en que pasa por escribir en la pantalla la sucesión de caracteres

- \ | / - \ | / - \ | / - \ | / …….

A modo de dibujos animados, al situarlos en el mismo lugar de la pantalla logramos la ilusión óptica del giro de una barra. Observe que en la línea 33 se lanza la ejecución de `rotar-linea` como actividad "de fondo" gracias al metacarácter &; esto es esencial pues provoca que tengamos dos actividades concurrentes: `rotar-linea` y el las instrucciones siguientes del guion. Mientras hemos dejado la barrita girando, en las líneas siguientes se entra en un bucle de larga duración que al terminar deberá parar el movimiento de la barra; para ello se utiliza la orden

```
kill -9 <pid>
```

que finaliza el proceso cuyo pid se pasa como argumento.

Ejercicio 6. Copie este ejercicio y pruébelo en su sistema para ver su funcionamiento. ¿Qué podemos modificar para que el giro se vea más rápido o más lento? ¿Qué hace la opción `-e` de las órdenes `echo` del guion?

Consulte la ayuda de bash, observe que aquí hay información muy interesante sobre las órdenes empotradas (*shell builting commands*) como por ejemplo `echo`, y sobre sus secuencias de escape.

Podemos encontrar ejemplos útiles de guiones en diferentes páginas web, entre las que podemos citar la página de Heiner Steven cuya dirección es `http://www.shelldorado.com/scripts/`

8 Archivos de configuración

Existen diferentes archivos de configuración que son leídos por el shell cuando se lanza, son los *archivos de arranque*. Estos guiones tienen como objetivo establecer la configuración del Shell definiendo variables, funciones, alias, etc.

Los archivos de arranque ejecutados van a depender del tipo de shell. El shell que aparece cuando arrancamos la máquina se denomina *login shell* (desde el que ejecutamos `startx`). Este shell utiliza los archivos:

- `/etc/profile`
- `$HOME/.bash_profile`

- `$HOME/.bash_login`
- `$HOME/.profile`
- `$HOME/.bashrc`
- `$HOME/.bash_logout`

El primero de ellos es un archivo del sistema y solo puede ser modificado por el administrador. El resto pueden ser modificados por el usuario para adaptarlos a sus necesidades.
Para shell interactivos que no son el *login shell*, por ejemplo, el que usamos desde una ventana de terminal, se utiliza el archivo *.bashrc*.

El último tipo de shell, es el *shell no interactivo* que es el que lanza el sistema para ejecutar los guiones y que no necesita indicador de órdenes. Este shell no tiene archivo de inicio pero utiliza la variable `$BASH_ENV` para adaptar su ejecución.

También puede existir un archivo `.bash_logout` que se ejecutará al finalizar el shell y que generalmente hace labores de limpieza del entorno o se imprime algún mensaje.

Ejercicio 7. Escriba un guion que admita como argumento el nombre de un tipo de shell (por ejemplo, csh, sh, bash, tcsh, etc.) y nos dé un listado ordenado alfabéticamente de los usuarios que tienen dicho tipo de shell por defecto cuando abren un terminal. Dicha información del tipo de shell asignado a un usuario se puede encontrar en el archivo `/etc/passwd` y para poder filtrar la información que nos interesa nos será útil la orden siguiente:

```
cut -d':' -f1
```

que aplicada de forma encauzada con cualquier orden para mostrar el contenido del citado archivo, cortará por la columna 1 y hasta que aparezca el delimitador ":".

Ejercicio 8. Dos órdenes frecuentes de Unix son `tar` y `gzip`. La orden `tar` permite almacenar/extraer varios archivos de otro archivo. Por ejemplo, podemos almacenar el contenido de un directorio en un archivo con

```
tar -cvf archivo.tar directorio
```

(la opción `-x` extrae los archivos de un archivo `.tar`).

La orden `gzip` permite comprimir el contenido de un archivo para que ocupe menos espacio. Por ejemplo, `gzip archivo` comprime `archivo` y lo sustituye por otro con el mismo nombre y con la extensión `.gz`. La orden que hace lo contrario es `gunzip`.

Dadas estas órdenes construya un guion, denominado `cpback`, que dado un directorio o lista de archivos como argumento(s) los archive y comprima en un archivo con nombre `copiaYYMMDD`, donde `YY` corresponde al año, la `MM` al mes y la `DD` al día, dentro de un directorio denominado `CopiasSeguridad`. El guion debe realizar las comprobaciones oportunas: los argumentos existen, el directorio de destino existe y si no, lo crea.

Ejercicio 9. Hacer un script en Bash denominado `newdirfiles` con los siguientes tres argumentos:

- `<dirname>` Nombre del directorio que, en caso de no existir, se debe crear para alojar en el los archivos que se han de crear.
- `<num_files>` Número de archivos que se han de crear.
- `<basefilename>` Será una cadena de caracteres que represente el nombre base de los archivos.

Ese guión debe realizar lo siguiente:

- Comprobar que el número de argumentos es el correcto y que el segundo argumento tenga un valor comprendido entre 1 y 99.
- Crear, en caso de no existir, el directorio dado en el primer argumento a partir del directorio donde se esté situado y que posea permisos de lectura y escritura para el usuario `$USER`.
- Dentro del directorio dado en el primer argumento, crear archivos cuyos contenidos estarán vacíos y cuyos nombres lo formarán el nombre dado como tercer argumento y un número que irá desde 01 hasta el número dado en el segundo argumento.