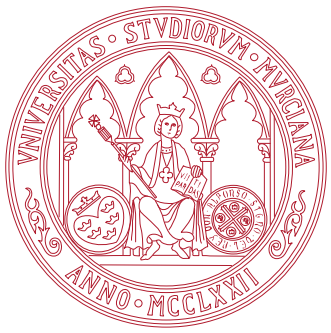

Universidad de Murcia

Facultad de Informática

Departamento de Ingeniería y Tecnología de Computadores

Área de Arquitectura y Tecnología de Computadores

PRÁCTICAS DE I.S.O.

2º de Grado en Ingeniería Informática

Boletín de prácticas 4 – Programación avanzada de *shell scripts* en Linux

Curso 2017/2018

Índice

1. Variables. Uso avanzado

Las variables se dividen en dos tipos:

- **Variables locales**: son heredadas por los hijos creados por el shell actual, pero sus valores desaparecen en el momento en que éstos realizan una llamada al sistema `exec`.
- **Variables de entorno**: son heredadas por los procesos hijos del shell actual, manteniendo sus valores incluso cuando éstos ejecutan una llamada al sistema `exec`.

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.

Además de las variables que puede definir el programador, un shell tiene definidas, por defecto, una gran número de variables, muchas de ellas de entorno. Las más importantes son:

- `LOGNAME`: nombre del usuario.
- `HOME`: directorio de trabajo del usuario actual que la orden `cd` toma por defecto.
- `PWD`: directorio actual.
- `PATH`: contiene un conjunto de rutas de directorios, separadas por ":". Cuando el usuario invoca una orden o programa a ejecutar indicando solo su nombre (sin ruta ni absoluta, ni relativa), el shell buscará el fichero ejecutable correspondiente en este conjunto de rutas, inspeccionándolas en orden secuencial.

  Así, si por ejemplo tenemos:

  ```
  $ echo $PATH
  /usr/bin:/usr/sbin:/home/alumno/misejecutables
  ```

  y ejecutamos:

  ```
  $ programilla
  ```

  el shell buscará el ejecutable `programilla` en primer lugar en `/usr/bin`. Si lo encuentra ahí, lo ejecuta. En caso contrario, lo buscará en `/usr/sbin` de igual forma y, finalmente, si sigue sin encontrarlo, lo buscará en `/home/alumno/misejecutables`.

  Obviamente, también podemos ejecutar cualquier orden o programa si especificamos su ruta absoluta o relativa, aunque éste se encuentre en un directorio no incluido en la variable `PATH`.

2. 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, por lo que cuando es llamada desde un script no se ejecuta el fichero `/bin/echo`. Realmente, muchas de estas órdenes también son externas (tienen su ejecutable en `/bin/`), pero sus códigos han sido incorporados al del `/bin/bash` para acelerar sus ejecuciones. De esta forma, por ejemplo, cuando invocamos la orden `echo` directamente, se ejecuta el código de ésta que tiene incorporado en `/bin/bash`, mientras que si ejecutamos `/bin/echo` estaremos llamando al ejecutable externo.

Algunas 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
  ```

  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 encerrar la cadena a mostrar entre comillas dobles:

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

  Otra opción muy utilizada de la orden `echo` es "`-n`". Con esta opción, no se realiza el salto de línea automático tras mostrar el texto por pantalla.

  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
  ```

- `read`: lee una línea 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 su nombre:"
  read nombre
  echo "Hola $nombre !"
  ```

- `cd`: cambia de directorio, tal como se vio en el primer boletín.
- `pwd`: devuelve el nombre del directorio actual. Equivale a leer el valor de la variable `PWD`.
- `let arg` [`arg`]: cada `arg` es una expresión aritmética a ser evaluada, tal como se vio en el boletín anterior.
- `test`: permite evaluar si una expresión es verdadera o falsa, tal como se vio en el boletín anterior.
- `export`: hace que el valor de una variable esté disponible para todos los procesos hijos del shell, tal como se vio en la sección anterior.
- `exit`: finaliza la ejecución del guión, tal como se vio en el boletín anterior.
- `true` y `false`: devuelven como código de retorno 0 y 1, respectivamente.

  Nota: En general, en Linux, el valor 0 se corresponde con `true`, y cualquier valor distinto de 0 con `false`, tal como vimos en el boletín anterior acerca del valor devuelto tras la ejecución de una orden del shell (recogido en la variable `$?`), donde un 0 indica que la orden se ha ejecutado con éxito y otro valor indica que ha habido errores.

3. Órdenes simples, listas de órdenes y órdenes compuestas

3.1. 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.

Un operador de control es uno de los siguientes símbolos:

`||`	`&`	`&&`	`;`	`;;`	`(`	`)`	`|`	`|&`	<nueva-línea>

La primera palabra especifica la orden a ser ejecutada. Las palabras restantes se pasan como argumentos a la orden dada.

Por ejemplo, si tenemos un shell script llamado `programa`, podemos ejecutar la siguiente orden:

```
$ a=9 programa 34 234 > fic.out
```

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. Además, `programa` recibe dos argumentos: 34 y 234. Por último, la salida estándar de `programa` se redireccionará al fichero `fic.out`.

Tal como vimos en el boletín anterior, el valor devuelto de una orden simple es su estado de salida.

Si una orden se termina mediante el operador de control `&`, el shell ejecuta la orden en segundo plano en un proceso hijo. El shell no espera a que la orden acabe y el estado devuelto es 0.

3.2. Listas de órdenes

Tal como se explicó en el boletín 2, una tubería es una secuencia de una o más órdenes separadas por un operador de control | o `|&` (el operador `|&` conecta la salida de error de la orden que le precede a su salida estándar, y, a continuación, redirecciona ésta última a la entrada estándar de la orden que le sigue mediante una tubería; es una abreviación para `2>&1|`).

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

Las órdenes separadas por un `;` se ejecutan secuencialmente; el shell espera que cada orden termine antes de ejecutar la siguiente. 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, es decir, si tiene éxito en su ejecución.

Una lista O tiene la forma:

```
orden1 || orden2
```

`orden2` se ejecuta si y sólo si `orden1` devuelve un estado de salida distinto de 0, o sea, si no tiene éxito en su ejecución.

El estado de salida de las listas Y y O es el de la última orden de la lista que se ha ejecutado (tal como hemos visto, la última orden ejecutada de una lista puede coincidir o no con la última orden que aparece en dicha 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, en paralelo, sin que una orden espere a otra.

3.3. Órdenes compuestas

Las órdenes se pueden agrupar formando órdenes compuestas:

- `{ orden_1 ; ... ; orden_n; }`: las n órdenes se ejecutan simplemente en el entorno del shell en curso, sin crear un nuevo shell. Esto se conocen como una **orden de grupo**.
- `( orden_1 ; ... ; orden_n; )`: las n órdenes se ejecutan en un nuevo shell en un proceso hijo, o sea, tras un `fork`.

SINTAXIS. Agrupación de órdenes: uso de espacios en blanco.
En el caso de las llaves, es obligatorio dejar un espacio en blanco al empezar la lista, así como un punto y coma para acabarla. En el caso de los paréntesis, ambas normas son opcionales.

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

Ejemplo	`#!/bin/bash`
	`cd /usr`
	`{ cd bin; ls; }`
	`pwd`
		`#!/bin/bash`
		`cd /usr`
		`( cd bin; ls; )`
		`pwd`

Resultado	1.- Entra al directorio `/usr` 2.- Lista el directorio `/usr/bin` 3.- Directorio actual: `/usr/bin`	1.- Entra al directorio `/usr` 2.- Lista el directorio `/usr/bin` 3.- Directorio actual: `/usr`

En el caso de la izquierda, el cambio de directorio de trabajo con la orden `cd bin` se ha producido en el propio shell que está ejecutando el script, por lo que la instrucción `pwd` muestra el directorio al que se había cambiado. Por el contrario, en el ejemplo de la derecha, la orden de cambio de directorio de trabajo se ha ejecutado en un proceso hijo del script que está ejecutando el guión, con lo que cuando se acaba la ejecución de este proceso hijo (al salir de los paréntesis) y volver al proceso padre, éste sigue teniendo como directorio de trabajo el mismo que tenía antes de ejecutarse el hijo.

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
```

4. 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 guión shell, sólo hay que escribir su nombre.

Veamos un primer ejemplo de script que contiene una función:

```
#! /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`.

Veamos un ejemplo ahora con una función que usa parámetros y devuelve un valor de retorno:

```
#!/bin/bash

function muestra {
       echo $1
       return 55
}

muestra Hola
echo "Valor devuelto por la función: $?"
```

En este script la función `muestra` escribe en pantalla el primer argumento que recibe, `$1`, y devuelve el valor 55, que quedará recogido en la variable `$?`. Como podemos ver, los argumentos y el valor de retorno, de las funciones, son tratados de manera similar que los argumentos suministrados y el estado de salida del script. (Véase el apartado "Variables, parámetros y código de salida" del boletín anterior).

5. Depuración

Una buena idea para depurar un guión shell es usar la opción `-x` con el ejecutable bash. Por ejemplo, si nuestro programa se llama `prac1`, se podría invocar como:

```
$ bash -x prac1
```

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:

```
$ bash -v prac1
```

Ambas opciones pueden ser utilizadas de forma conjunta:

```
$ bash -xv prac1
```

Por otro lado, una opción de depuración que resulta también muy útil es `-u`. Esta opción sirve para detectar variables o parámetros que son utilizados sin estar inicializados previamente. Así por ejemplo, en este guión shell, llamado `prueba_u`, que parece estar correcto a priori:

```
#!/bin/bash
operador1=100
operador2=200
let resultado=operado1+operador2
echo "El resultado de sumar $operador1 y $operador2 es $resultado"
```

la ejecución nos mostraría algo extraño:

```
$ prueba_u
El resultado de sumar 100 y 200 es 200
```

pero, si usamos la opción `-u` podríamos detectar fácilmente el error:

```
$ bash -u prueba_u
prueba_u: line 4: operado1: unbound variable
```

Es decir, en la suma realizada con la instrucción `let` estamos usando erróneamente la variable no definida `operado1`, en lugar de `operador1`. Cuando en una operación aritmética aparece una variable no definida, ésta se toma con valor 0.

6. 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 fácil solución, y a lo largo del tiempo la gente ha ido recopilando las mejores soluciones para ellos.

6.1. 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
```

6.2. 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 seleccionar un conjunto de líneas, no permiten actuar sobre ellas una a una. La orden `read` ya se usó para leer desde el teclado un valor para asignárselo a una variable, pero gracias a la redirección se puede utilizar para leer un fichero. Un ejemplo de este patrón lo podemos ver en el siguiente guión shell (`leer_fichero.sh`):

```
#!/bin/bash
while read linea
do
        echo "Línea: $linea"
        # Procesar $linea (línea actual)
done < $1
```

El bucle termina cuando la función `read` llega al final del fichero de forma automática.
Se puede usar una tubería en lugar del direccionamiento de esta manera:

```
#!/bin/bash
cat $1 | while read linea
do
        echo "Línea: $linea"
        # Procesar $linea (línea actual)
done
```

Si se amplía el ejemplo anterior para ir contando las líneas del fichero de la siguiente forma:

```
#!/bin/bash
contador=0
cat $1 | while read linea
do
        let contador=$contador+1
        echo "Línea $contador: $linea"
        # Procesar $linea (línea actual)
done
echo "Numero total de lineas del fichero: $contador"
```

su ejecución producirá un salida de este tipo:

```
$ ./leer_fichero.sh clientes
Línea 1: Pepe Perez
Línea 2: Juan Lopez
Línea 3: Jesus Sanchez
Numero total de lineas del fichero: 0
```

Este error a la hora de mostrar el número de líneas del fichero ha ocurrido porque las órdenes enlazadas mediante tuberías se ejecutan en sus respectivos procesos, todos ellos hijos del proceso principal que está interpretando el guión shell. Por lo que la actualización de la variable `contador` dentro del segundo proceso de la tubería (el bucle `while`) afecta únicamente a la copia de esta variable en dicho proceso, pero no a la copia de esta variable en el proceso padre. Por tanto, en el proceso padre, la variable `contador` se ha mantenido a 0 desde su inicialización, y eso es lo que se muestra en la última línea.

Para evitar esta problema podemos forzar a que el bucle `while` y la siguiente línea de código se ejecuten dentro del mismo proceso formando una orden compuesta:

```
#!/bin/bash
contador=0
cat $1 | ( while read linea
         do
                let contador=$contador+1
                echo "Línea $contador: $linea"
                # Procesar $linea (línea actual)
        done;
echo "Numero total de lineas del fichero: $contador" )
```

O bien, podemos volver al esquema original, mediante redireccionamiento:

```
contador=0
while read linea
        do
                let contador=$contador+1
                echo "Línea $contador: $linea"
                # Procesar $linea (línea actual)
        done < $1
echo "Numero total de lineas del fichero: $contador"
```

6.2.1. Recorrer la salida de una orden

En el código de un guión shell frecuentemente se invoca la ejecución de una orden o conjunto de órdenes, cuyo resultado se precisa utilizar más adelante en diferentes instrucciones del script. Una manera de conseguir este objetivo es almacenando la salida de esta orden en una variable y, posteriormente, realizar los tratamientos oportunos del contenido de esta variable allí donde se precise a lo largo del script.

Un ejemplo:

En el siguiente ejemplo se muestra el uso de este patrón para mostrar el tamaño de todos los ficheros cuyo nombre acabe en `".pdf"` y que se encuentren a partir del directorio actual.

```
#!/bin/bash
tam_tot=0
find . -name "*.pdf" -printf "%f %s \n" | (while read linea ; do
        nom_fic=$(echo $linea | cut -f1 -d" ")
        tam_fic=$(echo $linea | cut -f2 -d" ")
        let tam_tot=$tam_tot+$tam_fic
        echo "Fichero: $nom_fic --> Tamaño: $tam_fic"
done;
echo "Tamaño total de los ficheros acabados en '.pdf': $tam_tot")
```

donde observamos como, entubando la salida de la orden `find`, que es la que hace la labor de búsqueda de toda la información requerida, con el bucle `while`, vamos leyendo, línea a línea, el resultado de esa búsqueda, al estilo del esquema de lectura de un fichero visto en la subsección anterior. Dentro del bucle ahora tocaría ir filtrando, para cada línea resultado del `find`, los diferentes campos de interés (nombre del fichero y tamaño). Pues bien, eso se hace extrayendo de la variable `linea` los campos 1 y 2, respectivamente, en las dos primeras instrucciones del interior del bucle.

Otra forma de conseguir este mismo resultado sería enviar la salida de la orden a un fichero temporal y, tras ello, ir leyendo este fichero línea a línea. De esta manera, evitamos la creación de procesos hijos como en el ejemplo visto y, además, tenemos los datos almacenados por si queremos tratarlos de nuevo más allá del fin del bucle.

6.3. 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, para números enteros positivos:

```
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
```

La orden `grep` realiza la búsqueda de un entero positivo en el texto que le llega por su entrada estándar a través de la tubería, procedente de la orden `echo`. Este texto corresponde al contenido del primer parámetro del guión, `$1`. Para ello:

- El patrón de búsqueda del `grep` consiste en la aparición de un dígito (conjunto posibles de valores `[0-9]`) una o más veces (operador de repetición '+').
- Con la opción `-x` se indica que, para cada línea de la entrada, la búsqueda será exitosa si toda la línea coincide con el patrón. Normalmente la salida de `echo $1` será una única línea, por lo que el primer parámetro debe cumplir exactamente el patrón de búsqueda.
- Con la opción `-q` se evita que la orden `grep` muestre nada en pantalla.

Para números enteros, en general, podemos ampliar la comprobación para el caso de que el número pueda llevar signo. Para ello, se indica en el patrón de búsqueda que delante de los dígitos puede haber un + o un −, repetidos 0 o 1 veces:

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

6.4. Tratamiento de errores

En aquellos guiones que precisan algún parámetro de entrada en la propia línea de invocación, es frecuente tener que hacer un chequeo de estos parámetros para comprobar que se ajustan en número y forma a lo que precisa el script para su correcta ejecución. Este tratamiento de errores se suele implementar al principio del código del guión shell de manera que si los parámetros no son correctos se muestre el correspondiente mensaje de error en la salida estándar de error (descriptor número 2) y se aborte la ejecución, devolviendo un código de error mediante la orden `exit`. Como ya se vio anteriormente, el código de error devuelto por el guión podrá ser consultado en `$?`, donde se encontrará el código de error de la orden ejecutada justo antes.

Ejemplo: Supongamos que queremos programar un guión shell, llamado `buscaytraeme.sh`, con esta sintaxis:

```
buscaytraeme.sh dir fic
```

que busca el fichero `fic` en el árbol de directorios a partir de `dir` y si lo encuentra lo copia al directorio de trabajo. Pues bien, la comprobación de errores en los parámetros al inicio del guión shell podría tener esta forma:

```
#!/bin/bash
if [ $# -ne 2 ]
then
    echo "Numero de parametros erróneo" >&2
    echo "USO: $0 directorio_busqueda fichero_a_buscar" >&2
    exit 1
fi
if [ ! -d $1 ]
then
    echo "El primer parámetro debe ser un directorio" >&2
    echo "USO: $0 directorio_busqueda fichero_a_buscar" >&2
    exit 2
fi
if [ ! -f $1/$2 ]
then
    echo "El segundo parámetro deber ser un fichero" >&2
    echo "USO: $0 directorio_busqueda fichero_a_buscar" >&2
    exit 3
fi

####################################
# resto del codigo del guion shell #
####################################
```

Y, por tanto, diferentes intentos de utilización errónea de este guión producirían este efecto:

```
$ ./buscaytraeme.sh
Numero de parametros erróneo
USO: ./buscaytraeme.sh directorio_busqueda fichero_a_buscar

$ echo $?
1

$ ./buscaytraeme.sh directorioerroneo fic
El primer parámetro debe ser un directorio
USO: ./buscaytraeme.sh directorio_busqueda fichero_a_buscar

$ echo $?
2

$ ./buscaytraeme.sh directoriocorrecto ficheroerroneo
El segundo parámetro deber ser un fichero
USO: ./buscaytraeme.sh directorio_busqueda fichero_a_buscar

$ echo $?
3
```

De igual forma, la comprobación de errores en el resto de código del script se debe implementar de forma similar, de manera que si se produce alguna situación de error que conlleve que no se puede continuar la ejecución con normalidad, se ponga en funcionamiento el protocolo de tratamiento de errores: mostrar el mensaje oportuno en la salida estándar de error y abortar la ejecución devolviendo el código de error establecido para cada caso.

6.5. Utilización de ficheros temporales

En la implementación de algunos guiones shell puede ser oportuno la utilización de algún fichero temporal donde almacenar resultados parciales para, a continuación, ir consultando este fichero de cara a realizar una segunda parte del procesamiento previsto. Con el objetivo de asegurarnos de que no haya ningún problema de permisos a la hora de crear este fichero temporal, es conveniente utilizar la orden `mktemp` que, por defecto, crea este fichero temporal en el directorio `/tmp`.

Ejemplo: Supongamos que queremos escribir un guión llamado `cambiapalabra.sh` que cambie en el contenido de un fichero de texto, cuyo nombre se pasa como primer parámetro, cada aparición de una palabra (segundo parámetro) por otra palabra (tercer parámetro). Sintaxis:

```
cambiapalabra.sh fichero palabra_old palabra_new
```

Su implementación podría ser:

```
#!/bin/bash
fichero_temporal=$(mktemp)

while read linea
do
    palabra="*"
    nuevalinea=""
    while [ ! -z "$palabra" ]
    do
            palabra=$(echo $linea | cut -f1 -d" ")
            linea=$(echo $linea | cut -f1 -d" " -s --complement)
            if [ "$palabra" == "$2" ]
            then
                    palabra=$3
            fi
            nuevalinea="$nuevalinea $palabra"
    done
    echo $nuevalinea >> $fichero_temporal
done < $1

mv $fichero_temporal $1
```

donde observamos cómo el script va leyendo, línea a línea, el contenido del fichero pasado como primer parámetro (bucle while). Por cada línea del fichero realiza entonces una búsqueda de la palabra pasada como segundo parámetro, sustituyendo cada ocurrencia por la palabra pasada como tercer parámetro (bucle while interno). La línea resultante es escrita en un fichero temporal. Al acabar el recorrido del fichero original es cuando el script renombra el fichero temporal, poniéndole el nombre del fichero original, para que el efecto del script quede reflejado en este último y, a la vez, se libere el espacio ocupado en disco por el fichero temporal.

6.6. Recorrido de un directorio

Para recorrer un directorio cuyo nombre, por ejemplo, se ha pasado como parámetro al guión shell, tenemos básicamente dos planteamientos:

- Un planteamiento simple, orientado principalmente a un recorrido plano, sin recursividad, de un directorio, se podría realizar mediante un bucle `for` que itere sobre el contenido de dicho directorio. Por ejemplo, el siguiente guión shell, llamado `cambia_tex_latex.sh`, cambia el sufijo de todos los archivos `*.tex` a `*.latex` del directorio indicado como primer parámetro.

```
#!/bin/bash

for fic in $1/*.tex
do
    echo "nombre actual fichero = $fic "
    new_fic=$(basename "$fic" ".tex")
    echo "nuevo nombre fichero = $new_fic"

    new_fic="$new_fic.latex"
    echo "nuevo nombre fichero con sufijo nuevo = $new_fic"

    ruta=$(dirname "$fic")
    echo "ruta del fichero: $ruta"
    new_fic="$ruta/$new_fic"
    echo "nuevo nombre fichero con ruta = $new_fic"


    mv "$fic" "$new_fic"
done
```

En este guión, a cada fichero del directorio se le extrae su nombre, sin ruta y sin el sufijo `tex`, usando la orden `basename`, explicada anteriormente. A continuación se le añade el nuevo sufijo `latex` y la ruta del fichero original, que se obtiene usando la orden `dirname`. Finalmente, se realiza el cambio de nombre, con la orden `mv`. Se han usado comillas dobles para manejar variables que contienen nombres de ficheros (`fic`, `new_fic`) de cara a evitar errores en el caso de que dichos nombres de ficheros contengan algún espacio en blanco.

- Un planteamiento de recorrido más elaborado y potente, tanto en el filtrado de los ficheros a tratar, como en el grado de profundización recursiva por el árbol de subdirectorios, se tendría que abordar con la orden `find`. Un ejemplo lo vimos en el script de la sección 6.2.1:

```
#!/bin/bash
tam_tot=0
find -name "*.pdf" -printf "%f %s \n" | (while read linea ; do
        nom_fic=$(echo $linea | cut -f1 -d" ")
        tam_fic=$(echo $linea | cut -f2 -d" ")
        let tam_tot=$tam_tot+$tam_fic
        echo "Fichero: $nom_fic --> Tamaño: $tam_fic"
done;
echo "Tamaño total de los ficheros acabados en '.pdf': $tam_tot")
```

Este guión muestra el tamaño de todos los ficheros que se encuentren a partir del directorio actual (profundizando recursivamente en todos sus subdirectorios) cuyo nombre acabe en `".pdf"`.

7. Ejercicios

1. Escribe un guión shell para realizar una copia de un fichero. Este guión debe comprobar que recibe exactamente dos argumentos: el nombre del fichero origen y el destino. Si es así, entonces, antes de hacer la copia, en el caso de que ya exista un fichero con el nombre y ruta indicada para el fichero destino debe preguntar al usuario si desea sobreescribirlo con esta operación de copia.

   Solución:

```
if [ $# -ne 2 ]
then
    echo "USO: $0 fichero_origen fichero_destino"
    exit 1
fi

desde=$1
hasta=$2
if [ -f "$hasta" ]
then
    echo "$hasta ya existe, ¿desea sobreescribirlo (s/n)?"
    read respuesta
if [ "$respuesta" != "s" ]
then
     echo "$desde no copiado"
     exit 0
fi
fi
cp $desde $hasta
```

2. Escribir un guión shell que ponga el atributo de ejecutable a los ficheros pasados como argumentos en la línea de órdenes. Para aquellos argumentos que se le pasen que no sean ficheros regulares el guión no debe hacer nada.

   Solución:

```
for fich in $@
do
     if test -f "$fich"
     then
          chmod 744 "$fich"
     fi
done
```

3. Escribe un guión shell que haga múltiples copias de ficheros a pares, es decir, copia el primer fichero en el segundo, el tercero en el cuarto, etc.

   Solución:

```
let cociente=$#%2
if [ $cociente -eq 1 ]
then
    echo "$0: el número de argumentos debe ser par" >&2
    exit 1
fi
#!/bin/bash
while test "$2" != ""
do
    echo $1 $2
    cp $1 $2
    shift; shift
done
```

4. Escribe un guión shell llamado `ldir` que liste el contenido de los directorios existentes en el directorio actual.

   Solución:

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

5. Escribe un guión shell que recibe el nombre de un fichero como único parámetro. El guión debe indicar si cada una de las líneas de este fichero contiene única y exclusivamente un valor numérico entero positivo.

   Solución:

```
#!/bin/bash
echo "Leyendo el fichero $1:"

while read linea
do
        if echo $linea | grep -x -q "[0-9]\+"
        then
             echo "La línea \"$linea\" es un número natural"
        else
             echo "La línea \"$linea\" NO es un número natural"
        fi

done < $1
```

6. Escribe un guión shell, llamado `cambia_tex_latex_recursivo.sh`, que cambie el sufijo `*.tex` por `*.latex` a todos los ficheros que se encuentren en el directorio indicado como parámetro, así como en cualquiera de sus subdirectorios.

   NOTA: ver el ejemplo `cambia_tex_latex.sh` en la sección 6.6.

7. Escribe un guión shell que muestre el nombre completo, incluyendo su ruta absoluta, y el nombre del usuario propietario de todos los ficheros que se encuentren a partir del directorio actual y que hayan sido modificados en la última semana.

   NOTA: ver sección 6.2.1.

8. Escribe un guión shell que modifique el contenido de un fichero de texto cuyo nombre se pasa como primer parámetro. La modificación del fichero consiste en introducir en su contenido un salto de línea tras cada aparición de la palabra que se indica como segundo parámetro del guión. Al acabar deberá mostrar el pantalla el número total de saltos de líneas que ha introducido.

   NOTA: ver sección 6.5.

8. Bibliografía

- Página de manual del intérprete de órdenes Bash (`man bash`).
- *El libro de UNIX*, S. M. Sarwar *et alt*, ISBN: 8478290605. Addison-Wesley, 2005.
- *Linux: Domine la administración del sistema*, 2ª edición. Sébastien Rohaut. ISBN 9782746073425. Eni, 2012.
- *Shell & Utilities: Detailed Toc.* The Open Group Base Specifications. http://pubs.opengroup.org/onlinepubs/9699919799/utilities/contents.html.
- *Unix shell patterns* (http://wiki.c2.com/?UnixShellPatterns), J. Coplien *et alt.*

- *Programación en BASH - COMO de introducción* (http://es.tldp.org/COMO-INSFLUG/COMOs/Bash-Prog-Intro-COMO/), Mike G. (traducido por Gabriel Rodríguez).
- *Shell & Utilities: Detailed Toc* (http://pubs.opengroup.org/onlinepubs/9699919799/utilities/contents.html), The Open Group Base Specifications.
- *Espacio Linux. Portal y comunidad GNU* (http://www.espaciolinux.com/).
- *Linux Shell Scripting Tutorial (LSST) v2.0* (https://bash.cyberciti.biz/guide/Main_Page), Vivek Gite *et alt*.