Introducción al shell scripting

Tabla de contenidos

GOBIERNO DE ESPAÑA · MINISTERIO DE EDUCACIÓN, CULTURA Y DEPORTE · SECRETARÍA DE ESTADO DE EDUCACIÓN, FORMACIÓN PROFESIONAL Y UNIVERSIDADES · Unión Europea · Fondo Social Europeo · El FSE invierte en tu futuro

1 Introducción

Si ya has practicado un poco con el shell bash, te habrás dado cuenta de que se pueden hacer muchas cosas interesantes con él. Aunque hay veces, que para conseguir determinado resultado necesitas introducir, por ejemplo, 3 o 4 comandos. Otras veces tenemos que evaluar una condición manualmente para determinar si ejecutamos o no un comando. Por ejemplo, supón que quieres borrar un fichero y no estás seguro de que esté en un directorio concreto. Primero con el comando "`ls`" compruebas si está, y si es así, a continuación, lo borras con el comando rm. Los scripts de shell nos dan la posibilidad de automatizar series de comandos y ejecutar algunos de ellos en función del resultado de otros o de alguna condición a comprobar sin intervención directa y continua del usuario.

Un script (guión en castellano) es como se llama a un archivo (de texto) que contiene instrucciones (en nuestro caso comandos de bash), y que necesita de un programa ayudante para ejecutarse (en nuestro caso el mismo bash será el programa ayudante). Un script tiene cierta similitud con un programa, pero no es estrictamente lo mismo. Generalmente, los programas se compilan y generan un fichero con instrucciones ejecutables directamente por el procesador (lenguaje máquina), mientras que los scripts son archivos que contienen -en formato texto- los comandos o instrucciones que la aplicación ayudante ejecutará, interpretándolos uno a uno conforme va leyéndolos del fichero que escribe el programador.

Aunque en principio esto puede resultar un tanto abstracto, confuso o innecesario, vamos a verlo despacio y con ejemplos, porque tiene implicaciones importantes. Si no has programado nunca, en este capítulo encontrarás el aprendizaje del shell-scripting en bash como una sencilla introducción a la programación, que te ayudará a comprender después los fundamentos de otros lenguajes de programación. No te preocupes, porque lo vamos a explicar todo de forma sencilla y no te quedará ninguna duda.

2 Nuestro primer script en bash

Si has practicado con bash y con algún editor de texto, ya estás preparado para crear tu primer script bash. Un script es un archivo de texto plano con comandos; así que, para empezar, abre un editor y prepárate para escribir un fichero:

```
$ gedit miscript.sh
```

("gedit" es un editor, puedes usar otro editor de texto si quieres, por ejemplo "nano", "geany", "kate", "pluma", etc.)

Una vez abierto el editor de texto, escribe o copia lo siguiente (lo explicaremos detenidamente a continuación):

```
#!/bin/bash
# Esta línea será ignorada
# Ésta también

echo "Hola"
echo "Soy un script de shell."
```

El texto anterior -que habrás escrito en el editor- debe ser guardado en un fichero. La extensión (.sh) en los archivos en Linux no es tan importante como en windows puede ser ".exe" o ".bat". Es por convenio general, por lo que a los scripts de shell se les suele llamar con la extensión " `.sh`" (de SHell). De este modo identificaremos fácilmente nuestros scripts entre el resto de archivos de un directorio. Busca la opción de guardar en tu editor y guarda el fichero.

Observemos la primera línea: `#!/bin/bash`. Esta línea es un tanto especial, y es característica de todos los scripts en Linux, no solamente los de bash. Tras `#!` indicamos la ruta a la aplicación ayudante, la que interpretará los comandos del archivo. En nuestro caso es bash, así que ponemos ahí `/bin/bash`, que es la ruta hacia el intérprete de comandos bash. Si este punto te resulta confuso, ignoralo. Esta línea es la menos importante, déjala y no te calientes la cabeza ahora con ella.

No importa que pongamos líneas en blanco, pues serán ignoradas. Podemos ponerlas después de cada bloque de comandos que tengan cierta relación, para dar claridad y legibilidad al conjunto de comandos del script.

Las dos siguientes líneas comienzan por el carácter "#". En los scripts de bash, las líneas -a continuación de la primera- que comienzan con este signo son ignoradas, y se llaman "comentarios". Podemos usarlas como anotaciones o comentarios, para ayudarnos a recordar qué hace el grupo de comandos siguiente tiempo después de haberlo escrito o como aclaración para otros programadores. En este tutorial, a veces, se sustituye un bloque de comandos por un comentario, porque son comandos de ejemplo y no queremos que ocupen innecesariamente en los ejemplos.

Las dos últimas líneas son dos comandos. El intérprete de comandos examina y ejecuta línea por línea o comando por comando. Como sabemos, el comando `echo` saca por la salida

estándar (por defecto por pantalla) lo que le pasemos como argumento, en este caso dos frases.

Una vez guardado el script, salgamos de gedit (o tu editor). Ahora lo ejecutaremos. La primera forma de hacerlo es como sigue:

`$ bash miscript.sh`	← esta línea la escribes tú (pulsa intro al final)
`Hola`	← esta línea se muestra como resultado de la ejecución
`Soy un script de shell`	← también se muestra como resultado de la ejecución
`$`	← vuelve a verse el "prompt". El script ha acabado ya

Como ves, todos los comandos del script han sido ejecutados. "bash" ha tomado el fichero escrito, ha interpretado su contenido, y lo ha ejecutado. Hay una forma más directa de conseguir lo mismo.

Primero le damos permisos de ejecución al fichero:

```
$ chmod +x miscript.sh
```

Una vez el fichero tenga permisos de ejecución, se podrá ejecutar el script de forma más fácil invocando sólo el fichero de la siguiente forma

```
$ ./miscript.sh
Hola
Soy un script de shell
```

A partir de ahora, simplemente has de escribir la ruta (relativa o absoluta) del fichero o script y se ejecutará. Podemos ir más allá y ya que nuestro script es un programa como otro cualquiera, podemos incluir el directorio donde está dentro de la variable de entorno $PATH. Esta variable tiene una lista de directorios donde están los programas ejecutables que el intérprete de comandos analiza en busca de un comando que es tecleado por el usuario. Es decir, que si tecleamos el comando "gedit", la variable $PATH contiene los directorios donde se busca el ejecutable llamado "gedit". El directorio donde estás trabajando haciendo este script no está dentro de esa variable y ello nos obliga a escribir la ruta ("./miscript.sh").

Recordatorio:
Observa que los comandos ejecutados siguen las mismas reglas que existen cuando se escriben en el terminal. Puedes experimentar algún comando probando primero directamente en el terminal, ahorrándote tiempo respecto a crear un script y ejecutarlo.

Los espacios se utilizan, al igual que en el terminal para separar argumentos y repeticiones de espacios no significan nada, así, por ejemplo el comando

```
$ echo              juan              perico  andres
```

es equivalente a :

```
$ echo juan perico andres
```

y produce la misma salida

```
juan perico andres
```

Si se desea dar formato con espacios, hay que usar las comillas dobles (también simples, pero con cuidado) para incluir espacios en los parámetros

```
$ echo "juan              perico                andres"
juan           perico               andres
```

3 Variables y parámetros

3.1 Introducción

Una variable es un elemento que tiene un nombre (elegido por el programador) y que almacena un valor. Así, se relaciona el nombre con un valor almacenado. El valor puede cambiar (decimos entonces que "la variable actualiza su valor"). El nombre de la variable siempre es el mismo, no cambia. El siguiente script muestra cómo se le da valor a una variable y cómo se accede a ese valor

```
1  #!/bin/bash
2
3  nombrePersona="Juan Garcia Martinez"
4  echo "Hola $nombrePersona" ;
```

- La variable se llama "`nombrePersona`"
- En la línea 3 se crea dicha variable y se le asigna un valor. (¡Ojo! no dejes espacios antes y después del símbolo "=")
- El valor o contenido de la variable es "Juan Garcia Martinez". Si pruebas a quitar las comillas ocurrirán errores extraños. No es el momento ahora de analizarlos. ¡ Usa las comillas, Luke !
- En la línea 4 se recupera el contenido de la variable. Dicho valor se utiliza para generar un mensaje que aparecerá por la salida (gracias al comando echo)

- En la línea 4 el símbolo $ se ha utilizado para indicar que la palabra "nombrePersona" era el nombre de una variable. Es un símbolo especial que viene a indicar que la palabra siguiente es una variable y no una palabra tal cual, literal. Sin el símbolo "$", se habría mostrado la siguiente salida :

```
Hola nombrePersona
```

3.2 Características de las variables en bash

Todos los lenguajes de programación admiten el uso de variables. Aunque cada cual tiene algunas particularidades que los diferencian enormemente. Veamos qué características tienen las variables de shell script:

- Las variables no se declaran antes de su uso (como en C). "*Declarar*" una variable significa anunciar que se desea usar una y ponerle nombre. Algo así como reservar un nombre. En la línea 3 anterior, directamente hemos asignado un valor a una variable que no existía y a partir de esa línea ya existe y tiene un valor.
- Las variables no tienen tipo (no están predestinadas a contener por ejemplo, números exclusivamente, o cadenas de texto, o carácteres). Realmente, todo lo que bash almacena en las variables son cadenas de texto (excepto con “declare”).
- Si una variable no existe y se intenta consultar su valor, no ocurre ningún error, simplemente se sustituye por la cadena vacía

  Muchos lenguajes tienen las características diferentes o contrarias a éstas de bash.

3.3 Asignación de valor a una variable

Como shell script no es un lenguaje orientado a expresiones, sino a ejecución de comandos[1], cuando bash analiza una línea, para diferenciar un comando de una variable se utilizan algunas reglas particulares:

- En la asignación, se usa el símbolo `'='` para definir el nombre de la variable y el valor que se almacena dentro de ella (por ejemplo: `alturaEdificio=234`)
- A la hora de recuperar el valor, el símbolo $ se pone justo delante de la variable (por ejemplo `echo "la altura del edificio es` **`$alturaEdificio`**`"`)

  Observe la siguiente secuencia de comandos y respuestas capturados de un terminal

[1] Bash, en principio, cree que cada línea es un comando, y hay pocas formas de indicarle que una línea no es un comando.

```
1   $ num_veces = 7
2   bash error: num_veces: command not found
3   $ num_veces=7
4   $ echo num_veces
5   num_veces
6   $ echo $num_veces
7   7
8   $echo $variable_no_existente
9   $
```

(No hay que confundirse con el símbolo $ del prompt y el de la variable, son situaciones y significados distintos).

Analicemos línea a línea lo que ocurre en este ejemplo capturado de un terminal

1. Esta línea será interpretada por bash como un comando que lleva argumentos:
    a. "`num_veces`" es un comando
    b. "=" es un argumento a pasar a dicho comando
    c. " 7 " es el segundo argumento

La clave para que bash interprete la línea como un comando es que encuentra distintas palabras separadas por espacios y deduce que eso es un comando con argumentos… eso es lo que NO queremos que pase ahora.

2. Esta línea es la respuesta de bash a la ejecución de la línea 1: no encuentra el comando `num_veces`. Si en vez de escribir "`num_veces`" hubieses escrito el nombre de un comando, éste se habría ejecutado (o se habría intentado)
3. Esta línea se parece a la primera, pero hay algo que lo cambia todo: no hay espacios y por tanto bash cambia su forma de entender la línea. Además, bash le da mucha importancia al símbolo "=" y entiende que esto es una asignación: un valor que debe guardarse en una variable. Por ello, bash destripa la línea escrita de la siguiente manera:
    a. "`num_veces`" es el nombre de una variable
    b. "=" se indica una asignación
    c. "7" es el valor a almacenar en la variable
4. "`echo num_veces`" La línea anterior no tiene ninguna respuesta. Esta línea 4 es un comando escrito por el usuario. Aunque hayamos asignado un valor a la variable num_veces en la instrucción anterior, aquí la palabra "`num_veces`" aparece como argumento literal. Se mostrará por la salida estándar exactamente "num_veces" tal cual

se ha escrito en el comando. Aunque exista una variable llamada igual ("num_veces"), bash no interpreta eso como una variable

5. En esta línea se observa la salida del comando anterior.
6. Esta línea es otra vez, un comando introducido por el usuario. Se parece a la línea 4 pero la pequeña diferencia "$" causa un resultado muy diferente. Bash hace una cosa especial con dicha línea antes de ejecutarla: primero sustituye la variable "`$num_veces`" por el valor que contiene (que es un 7 de la asignación anterior).

   Bash realiza dos pasadas en la misma línea, en la primera sustituye todo lo que lleve un $antepuesto. Y lo sustituye por el valor que contiene cada variable. Cada palabra con un $antepuesto se toma como una variable que debe sustituirse.

   Has de imaginar que la línea 6 se reescribe sobre la marcha. Después de la primera pasada donde se sustituyen variables por valores, viene una segunda pasada en la que se ejecuta la línea . Durante un momento esa línea será reescrita como

   ```
   echo 7
   ```

7. Aquí se observa el resultado de la ejecución de la línea anterior. El valor mostrado es el que almacena la variable.
8. (y 9) En este caso usamos una variable que no existe. Lo significativo es que no hay ningún error, simplemente no aparece valor alguno.

Recuerde la regla para las asignaciones: en la construcción correcta, **no hay espacios ni antes ni después** del '=' y por ello, el shell sabe que eso es una asignación.

Si en algún momento se necesita desactivar el efecto que produce, se debe escapar el carácter $, típicamente anteponiendo una \.

```
$ num_veces=7
$ echo "--> \$num_veces es igual a $num_veces"
--> $num_veces es igual a 7
$
```

En ocasiones, pueden aparecer conflictos a la hora de recuperar el valor de dos variables cuyo nombre empieza igual (el siguiente ejemplo no es real):

```
$ area=7
$ aream=8
$ echo "el área es de $aream cuadrados "
                ¿Qué sale a continuación:
            "el área es de 7m cuadrados"  ???
             "el área es de 8 cuadrados"  ???
```

Para solucionar esto, se debe utilizar el par de llaves (“{“ , ”}”) para delimitar el nombre de la variable. Así, el ejemplo anterior se podría reescribir

```
$ num=7
$ numv=8
$ echo “hay ${numv}acas en el rebaño”
hay 8acas en el rebaño ???
$ echo “hay ${num}vacas en el rebaño”
hay 7vacas en el rebaño  ???
```

3.3.1 Ejemplo de script que usa variables

Vamos a crear un script que emite un saludo junto con al fecha del día. El saludo es personalizado y lleva el nombre del usuario. Dicho nombre será almacenado en una variable

```
1  #!/bin/bash
2
3  nombre="Juan Garcia Martinez"
4  echo "Hola $nombre" ;
5  echo "la fecha de hoy es: "
6  date
```

Observa que en la línea 3 se asigna un texto a la variable "nombre". Y más tarde, en la línea 4, se usa esta variable para acceder a su contenido y mostrarlo por la pantalla gracias al comando `echo`.

3.4 Indirección de Variables "Variable indirection"

(tema avanzado no se incluye aquí)

3.5 Parámetros de un shell script

Recuerda que a muchos comandos practicados hasta ahora, se les pasaba argumentos (por ejemplo, el comando "`ls -l -t`" lleva dos argumentos: "`-l`" y "`-t`", "`ls` " es el comando).

Se puede invocar a un shell pasándole una lista de argumentos (y/o opciones), como a cualquier otro comando. Desde nuestro script será posible acceder a dichos argumentos y actuar de forma distinta según su contenido. El orden de aparición de cada argumento en la lista es importante, puesto que desde el script vamos a acceder a cada uno de los argumentos en base a dicho orden. Es decir, podremos recuperar 'el primer' argumento, 'el tercer' argumento, etc.

Observe en el siguiente script, cómo las variables $0, $1 y $2, representan el primer, segundo y tercer parámetro respectivamente, tomando el nombre del propio script invocado como el primer parámetro ($0)

```
#!/bin/bash

echo "\$0 contiene $0"
echo "\$1 contiene $1"
echo "\$2 contiene $2"

echo "En total hay $# parametros"
```

Cuando se invoque el script, se le puede pasar una lista de argumentos con el siguiente resultado:

```
$ ./args.sh buenos dias
$0 contiene ./args.sh
$1 contiene buenos
$2 contiene dias
En total hay 2 parametros
```

Existen tres variables más, asociadas a la manipulación de parámetros. Una de ellas ($#) ya aparece en el script anterior e indica el número total de parámetros pasados (sin incluir $0, o el nombre del script). Las otras dos variables contienen la lista entera de argumentos y son: `$@` , $* . Tenga precaución porque el significado de ambas variables no es idéntico. Existen diferencias sutiles que aquí no comentamos. Veámos un ejemplo de uso de $@

```
#!/bin/bash
# arg2.sh muestra lista de argumentos

echo "has pasado: $@"
```

Al invocar el script, observamos la siguiente salida

```
$ args2.sh pipi caca culo pedo pis
has pasado: pipi caca culo pedo pis
```

4 Evaluación de expresiones

Una expresión es una secuencia de palabras y símbolos que se ha de analizar y computar, para que al final se obtenga un resultado equivalente. Una expresión podría ser "2 + 2", que la gran mayoría de personas transforma automáticamente en 4 porque saben que es una expresión de una operación suma.

4.1 Valor de retorno de un comando

Cuando un programa termina de ejecutarse, *devuelve* o *retorna* un valor. Este dato devuelto se le denomina "valor de retorno" o "código de salida" y es **numérico**. El valor es cero (0) si el programa finalizó con éxito o distinto de cero si el programa no finalizó con éxito (quizá estés acostumbrado a pensar lo contrario como ocurre en otros lenguajes).

El caso es que ¡ El valor de retorno no se imprime por la pantalla! es decir, cuando un programa termina el usuario no ve este *valor de retorno* en la pantalla. No confundas el valor de retorno con los mensajes de error. Un programa puede terminar bien, pero mostrar mensajes de error y al revés. Aunque no salga por pantalla y pase desapercibido para un usuario normal, podemos conocer cuál es ese valor consultando una variable muy especial. En el shell bash (y por extensión, en nuestros scripts) el valor de retorno del último comando o programa ejecutado queda almacenado en la variable especial **$?** (Y es especial, porque al consultarla, ya se está ejecutando otro comando (`echo $?`) y su valor se actualiza)

```
nacho@Toshibatil:~$ rm ficheronoexiste
rm: no se puede borrar «ficheronoexiste»: No existe el
archivo o el directorio
nacho@Toshibatil:~$ echo $?
1
nacho@Toshibatil:~$ echo $?
0
```

En la primera línea, el usuario ejecuta el comando `rm` (borrar un fichero) sobre un fichero que no existe. El comando responde por la salida de error, mostrando una descripción del problema. El comando no ha tenido éxito y ello se puede saber también consultando la variable $? . Esa es la intención del comando `echo` (del primero de los dos "echos"). El valor que contiene esa variable es 1, y por tanto indica un error. Observa que la segunda vez que se ejecuta el comando echo $?, se está viendo el resultado de la ejecución del anterior comando `echo`. Realiza una prueba similar, pero usando un comando que tenga éxito, por ejemplo "ls"

Prueba el siguiente ejemplo:

```
#!/bin/bash

echo "Listando archivo existente..."
ls archivo_existente
echo "El valor de retorno fue: $?"

echo " "
```

```
echo "Listando archivo inexistente..."
ls archivo_inexistente
echo "El valor de retorno fue: $?"

exit
```

La utilidad de conocer el valor de retorno en nuestros scripts será evidente un poco más adelante.

4.2 Evaluación de expresiones lógicas

Las expresiones lógicas son similares a una pregunta que se hace el script. Por ejemplo, las siguientes son expresiones lógicas formuladas como preguntas:

- `¿ 4 < 7 ?  : " ¿Es 4 menor que 7 ? "`
- `¿ 5 > $a ?  : " ¿la variable $a tiene un valor menor que 5?`
- `¿ $pal == "hola" ? : " ¿ La variable $pal guarda "hola" ?`

La expresión lógica -o pregunta- se analiza, y se evalúa. Se obtiene entonces una respuesta que puede ser afirmativa o negativa, verdadero o falso true o false, 0 ò 1, etc. Se dice que "una expresión lógica devuelve un resultado".

El comando que nos permite evaluar expresiones lógicas y obtener respuestas es **`test`**. Este comando analiza una expresión y devuelve el resultado de la evaluación en el valor de retorno, que puede ser:

- cero (0) si la expresión es VERDADERA (TRUE),
- uno (1) si la expresión que le pasamos a test es FALSA (FALSE).

¡Cuidado, porque estos valores son contrarios a los de muchos otros lenguajes de programación!

Lo que queremos evaluar recibe el nombre de EXPRESIÓN. `test` recibe las expresiones de un modo un tanto especial. En este caso, el comando no usa el valor de retorno para indicar si ha habido un error. En el siguiente ejemplo comprobamos si dos números son iguales o no:

```
$ test 7 -eq 7
$ echo $?
0
$ test 7 -eq 8
$ echo $?
1
```

`7 -eq 7` es la expresión. De esta forma tan peculiar le decimos a test que compruebe si 7 es igual a 7 (`-eq`, abreviatura de *'equals'*, 'es igual a'). Así que ese primer comando quiere decir: ¿Es 7 igual a 7? Como ves, test no ha producido ninguna salida por la pantalla, sin embargo, si comprobamos su valor de retorno, vemos que es cero (0), lo cual indica que la expresión es verdadera, 7 es igual a 7, efectivamente.

En el siguiente comando estamos comprobando si 7 es igual a 8. Como era de esperar, el valor de retorno de test no es cero (1), lo cual indica que la expresión es falsa; 7 no es igual a 8.

Ahora supón que antes de todos aquellos comandos hacemos:

```
$ a=7
$ b=8
$ test $a -eq $a
# Aqui seguiríamos con las dos variables
```

y que repetimos todo lo anterior, pero en vez de usar 7 y 8, usamos las variables `$a` y `$b`. No hay cambios, test reemplaza las variables por su valor y después efectúa la comprobación, siendo el resultado final el mismo.

Existe además otra forma equivalente de llamar a `test`, y que en nuestros scripts puede ayudarnos a que con un sólo golpe de vista nos demos cuenta de cuál es la expresión que se evalúa. Esta otra forma consiste en poner entre corchetes (y con espacio) la expresión a evaluar:

```
$ [ 3 -gt 2 ]
$ echo $?
0
```

El comando test sirve también para realizar otras muchas comprobaciones que nada tienen que ver con comparar números y cantidades.

Diferencias entre test ([ ]) y “new test” ([[ ]]) : http://mywiki.wooledge.org/BashFAQ/031

4.3 Evaluación de expresiones aritméticas

Una expresión aritmética es una operación matemática con números que da como resultado otro número (si se devolviese verdadero o falso, sería una “expresión lógica”).

En los shell scripts se pueden realizar cálculos numéricos. Por desgracia, hay que aprender a realizarlos de diversas maneras, ya que puede haber necesidades diferentes según

la situación. También existen diferencias entre cálculos con números enteros y cálculos con números fraccionarios (que no trataremos).

4.3.1 Comando expr

El comando **expr** tiene una similitud con el comando **test**. En ambos se incluyen unos parámetros que se evalúan y se ofrece una respuesta. **test** tan solo responde con verdadero o falso (mediante el valor de retorno), mientras que **expr** muestra un resultado numérico por la salida estándar.

expr evalúa una expresión aritmética (sólo con enteros) y devuelve un resultado numérico.

```
$ expr 5 - 1
4
```

4.3.2 $[ ]

El bash lleva integrado un evaluador de expresiones que se puede utilizar en sustitución del comando **expr**.

En este caso se antepone un $ al corchete '[' y se cierra la expresión con ']'

`expr 2 + 2` viene a equivaler a `$[ 2 + 2 ]`

Sin embargo no son intercambiables siempre: `expr` es un comando y "`$[ ]`" es evaluado y sustituido por el bash antes de la ejecución del comando (de forma similar a cómo sustituye variables por sus valores). Observa cómo se puede percibir la diferencia en el ejemplo siguiente:

```
$[ 0 + 1 ]
1: command not found
$ expr 0 + 1
1
$ NUM=$[ 0 + 1 ]
$ echo $NUM
1
$
```

4.3.3 $(( ))

Los caracteres ' $((' y ')) ' se utilizan para englobar una expresión aritmética que el propio intérprete de comandos resuelve y expande antes de seguir dentro de la línea de comandos donde está. Seguidamente se ejecuta el comando.

```
$ echo dos y dos son $(( 2 + 2 ))

dos y dos son 4
```

Como la expresión es y debe ser siempre aritmética, no cabe lugar a cadenas de texto y por tanto bash realiza automáticamente la sustitución de variables, aunque se puede explicitar libremente con $

```
$ dos=2
$ resultado=$(( dos + dos ))
$ echo "dos y dos son $resultado"
dos y dos son 4
$ seis=$(( resultado + $dos ))
$ echo "cuatro y dos son $seis"
cuatro y dos son 6
```

4.3.4 let y (( ))

`let` y (( )) son equivalentes. '`let`' evalúa la expresión que le sigue (en '(( ))' la expresión está entre ambos grupos de caracteres). La particularidad es que las expresiones pueden tener la sintáxis similar al lenguaje C. Por ejemplo, se pueden incluir espacios entre los símbolos, se pueden utilizar operadores de autoincremento (++), etc.

```
#!/bin/bash
#script para hacer el factorial del número 5

numero=5

anterior=(( $numero - 1 ))
POR TERMINAR
let "factorial = ${numero} \* ${cuatro}

anterior=(( anterior-- ))
factorial=`expr $factorial * $anterior`

anterior=`expr $numero - 1`
```

```
    factorial=`expr $factorial * $anterior`

    echo el factorial es
```

4.3.5 Resumen:

```
    #TODO LO DE A CONTINUACIÓN, FUNCIONA
    #    elementoActual=` expr $elementoActual + 1`
    #    elementoActual=$( expr $elementoActual + 1 )
         elementoActual=$(( $elementoActual + 1 ))
    #    suma=$(( suma + notas[j] ))
    #    suma=$(( $suma + ${notas[j]} ))
    #    elementoActual=$[ $elementoActual + 1 ]
    #    (( elementoActual++ ))
    #    let elementoActual++
    #    let suma=2+2
    #       let "suma = 2 + 3 "    # AL HABER ESPACIOS HAY QUE
AGRUPAR
    #    let unomas=suma+1
```

4.3.6 declare

4.4 Ejecución de un comando dentro de la línea de argumentos de otro comando

En ocasiones es deseable utilizar el resultado que sale por salida estándar de la ejecución de un comando para guardarlo o utilizarlo desde otro comando. El caso más habitual es el de asignar un resultado de una operación aritmética obtenida con `expr` o `cal` a una variable. Por ejemplo, suponga que quiere sumar dos variables y guardar el resultado en otra variable

```
$ var=9
$ num=6
$ expr $var + $num
15
```

Como puede ver el comando `expr` se ejecuta solitariamente sin problema, pero su resultado sale por la salida estándar. Deseamos almacenar ese resultado en otra variable. Probemos en principio a realizar las siguientes operaciones

```
$ var=9
$ num=6
$ expr $var + $num
15
$ result=expr $var + $num #esto da errror
$ result="expr $var + $num" #esto no da error
$ echo $result
expr 9 + 6
$
```

En las líneas en negrita, NO se ha ejecutado el comando expr, ya que no se interpreta como comando, sino como una palabra más.

Para forzar a que antes de la ejecución de la línea, se evalúe un comando existente dentro de ella, se utilizan el carácter comilla (obtenido como el acento abierto situado al lado de la tecla P en el teclado español '`'). Si en una línea, aparece un bloque entrecomillado con ellos, entonces se evalúa esa parte en primer lugar, se ejecuta el comando y se sustituye el bloque entrecomillado por el resultado de la ejecución del comando. Así, el ejemplo quedaría como sigue:

```
$ result=`expr $var + $num`
$ echo $result
15
$
```

Si los caracteres comilla no son fáciles de obtener o la visualización no es clara, se puede utilizar, al igual que ocurría con en el comando test, otra forma de contener el comando interno. En este caso utilizamos `$( comando )`. Así las siguientes líneas son equivalentes

```
$ result=`expr $var + $num`
$ result=$( expr $var + $num )
```

Ojo con errores en expr. El error se indica por la salida de error, pero no se asigna nada si estamos asignando a una variable el resultado de la operación

```
$ NUM=`expr $NUM +1 `
expr: error de sintaxi
```

```
$ echo $NUM

$
```

Un ejemplo más elaborado completo.

```
#!/bin/bash
#script tonto para hacer el factorial del número 5

numero=5

anterior=`expr $numero - 1 `
factorial=`expr ${numero} * ${anterior}`

anterior=`expr $numero - 1`
factorial=`expr $factorial * ${anterior}`

anterior=`expr $numero - 1`
factorial=`expr $factorial * $anterior`

echo " el factorial es ${factorial} "
```

5 Estructuras de control

Muchas veces nos será necesario hacer que nuestro script se comporte de una forma u otra según el usuario introduzca uno u otro argumento, o según una variable tenga tal o cual valor. En esta sección pasamos a ver los comandos y estructuras que nos permiten hacer esto.

5.1 Estructuras condicionales con if

Hemos visto cómo evaluar condiciones con test. Con este comando obtenemos una respuesta verdadero/falso a una pregunta. Si nuestro script debe realizar acciones o comandos diferentes en función de la respuesta a la pregunta, vamos a necesitar utilizar la estructura de control llamada "if". En su versión más simple tiene esta forma

```
if comando_pregunta; then
        comando_accion1
        comando_accion2
        comando_accion3
fi
```

5.1.1 Explicación:

Esta estructura de control, ejecutará el comando contenido entre `if` y `then` (el que aparece en azul como "comando_pregunta"). El resultado de ese comando determinará (decidirá) si se ejecutan todos y cada uno de los comandos contenidos entre `then` y `fi`. (los comandos en morado comando_accion). En nuestro caso este comando será casi siempre test.

Esto es una ejecución condicional que viene a decir "ejecuta estos comandos si y sólo si este otro comando se ejecuta con éxito". O lo que es lo mismo "*ejecuta el comando_pregunta y si el resultado es ok, entonces ejecuta la lista de comando_accion* "

El siguiente es un pequeño guión del juego de adivinar un número secreto. El usuario lanzará el script y pasará un número por argumento para probar suerte. El número secreto a adivinar está establecido en el propio guión en la variable `numSecreto`:

```
numSecreto=4
if test $numSecreto -eq $1 ; then
     echo "has acertado el número secreto"
fi
```

Es posible que en el caso de que el comando no tenga éxito se desee realizar otro comando. Por ejemplo, en el caso anterior, si el número secreto no se acierta, entonces se debería mostrar un mensaje indicando el fallo. En dicho caso se puede especificar la partícula "else" que indica cuál es el comando a ejecutar si el comando test falla.

```
numSecreto=4
if test $numSecreto -eq $1 ; then
      echo "has acertado el número secreto"
else
      echo "Has fallado el número secreto"
fi
```

5.1.2 Definición "seria" de if:

La estructura condicional simple tiene este aspecto:

```
if comando1; then
        # Bloque
fi
```

"`# Bloque`" Hace referencia a un bloque de comandos encuadrados entre el "`then`" y el "`fi`"

La estructura condicional con else tiene este aspecto:

```
if comando1; then
        # Bloque 1
else
        # Bloque 2
fi
```

La estructura condicional con todas las opciones tiene este aspecto:

```
if comando1; then
        # Bloque 1
elif comando2 ; then
        # Bloque 2
else
        # Bloque 3
fi
```

Este es el modelo más general con todas las opciones posibles. Si el valor de retorno de comando1 es cero, las líneas de comandos de "Bloque 1" serán ejecutadas. El resto del bloque if será ignorado en ese caso.

Si el valor de retorno de comando1 no es cero, pero el de comando2 sí que es cero, entonces el "Bloque 1" no será ejecutado, mientras que el "Bloque 2" sí que será ejecutado. El "Bloque 3" será ignorado. Podemos poner tantos `elif`'s como necesitemos.

Si ni comando1 ni comando2 ni cualquier otra de otros elif's que pongamos retornan cero, entonces el "Bloque 3" será ejecutado.

Usamos fi para terminar la estructura condicional.

Como habrás supuesto, los `comandoN` pueden ser comandos con test como se ve en este ejemplo:

```
#!/bin/bash
input=$1

if [ $input -lt 5 ]; then
        echo "El numero es menor que 5"
elif [ $input -eq 5 ]; then
        echo "El numero es 5"
elif [ $input -gt 5 ]; then
        echo "El numero es mayor que 5"
else
        echo "No introdujo un número"
fi
```

En este sencillo ejemplo capturamos el argumento pasado al script que queda guardado en la variable `$input`. Dependiendo de si el número es menor, igual o mayor que 5, el script se comporta de una forma u otra (podemos añadir más líneas de comandos después de los echo's que aparecen ahí). Si el usuario no introduce nada o introduce una cadena de texto, lo que hay tras else es lo que será ejecutado, pues ninguna de las condiciones anteriores se cumple.

De todos modos, los `elif` y los `else` son opcionales, podemos crear una estructura condicional sin ellos si no los necesitamos. Como muestra, el siguiente ejemplo:

```
#!/bin/bash


if [ $# -ne 2 ]; then

        echo "Necesito dos argumentos, el primero"

        echo "es el fichero donde debo buscar y"
        echo "el segundo es lo que quieres que"
        echo "busque."
        echo " "
        echo "Uso: $0 <fichero> <patron_busqueda>"
        echo " "

        exit
fi

FICHERO=$1
BUSQUEDA=$2

if [ ! -e $FICHERO ]; then
        echo "El fichero no existe"
```

```
        exit
fi

NUM_VECES=`cat "$FICHERO" | grep --count "$BUSQUEDA"`

if [ $NUM_VECES -eq 0 ]; then
        echo "El patron de busqueda \"$BUSQUEDA\" no fue encontrado"
        echo "en el fichero $FICHERO "
else
        echo "El patron de busqueda \"$BUSQUEDA\" fue encontrado"
        echo "en el fichero $FICHERO $NUM_VECES veces"
fi
```

Este ejemplo ya se va pareciendo a un programa de los buenos. Estudiémoslo en detalle:

En primer lugar comprobamos que nos han pasado dos argumentos (`if [ $# -ne 2 ]; then`) : el fichero donde buscar y la cadena de búsqueda. Si no es así, le indicamos al usuario cómo debe ejecutar el script y salimos.

A continuación comprobamos que el fichero existe. Si no es así, advertimos convenientemente y salimos.

Ahora viene la línea que hace todo el trabajo:

`NUM_VECES=`cat "$FICHERO" | grep --count "$BUSQUEDA"`.`

Ya sabemos que al entrecomillar se provoca la ejecución de un comando cuya salida se puede asignar a una variable o asignar a otro comando. Por tanto en esta línea se consigue, mediante cat, el pipe y grep contar el número de veces que el valor de `$BUSQUEDA` está en el fichero, que nos queda guardado en `$NUM_VECES`.

Lo siguiente habla por sí solo.

5.2 Comprobar el valor de una variable con case

Escribiendo scripts, es muchas veces necesario comprobar si el valor de una variable coincide con alguno de los que nosotros estábamos esperando, y si así es, actuar de una forma u otra dependiendo de este valor.

Por ejemplo, imagina un script para acceder a una base de datos de personas. El script se puede usar para manipular los registros de personas, de forma que el usuario puede pasar por argumento las siguientes palabras: "alta", "baja" y "modificar". El script deberá analizar el

primer argumento para descubrir qué operación hay que hacer. Una manera posible es la siguiente:

```
#!/bin/bash

if [ $1 == "alta" ]; then
     #   comandos para realizar un alta
fi
if [ $1 == "baja" ]; then
     #   comandos para realizar una baja
fi
if [ $1 == "modificar" ]; then
     #   comandos para realizar una modificacion
fi
```

Para estos casos viene mejor usar una estructura de control "CASE" que hace más corto realizar esta función.

CASE comprueba valores contra **patrones**. A continuación tienes la descripción de esta estructura y después el mismo ejemplo de antes pero usando CASE

```
#!/bin/bash
case $VARIABLE in
        valor1)
                # Bloque 1
                ;;
        valor2)
                # Bloque 2
                ;;
.
.
.
        *)
                # Ultimo Bloque
                ;;
esac
```

`valor1` y `valor2` son valores que nosotros esperamos que la variable pueda contener, y Bloque 1 y Bloque 2 son los bloques de comandos con lo que queremos que se haga en cada caso. Podemos poner tantos valores reconocibles como queramos. El `*` (asterisco) es un comodín que encaja con cualquier

cosa que no haya coincidido en alguna de las de arriba. Usar el comodín "*" es opcional. No obstante, en el ejemplo siguiente te darás cuenta de su utilidad.

¡ Sí, Los dos ";;" son necesarios !

```
#!/bin/bash
# case_arg.sh

case $1 in
        alta )
        # comandos para realizar un alta
        ;;

        baja )
   #    comandos para realizar una baja
        ;;

        modificar )
   #    comandos para realizar una baja
        ;;

        fin)
                exit;
        ;;
        *)
        echo "Parametro no reconocido"
        ;;
esac
```

Dentro de cada uno de los bloques de comandos se pueden poner estructuras condicionales con if, o incluso otro case.

5.2.1 Patrones:

En la estructura case, la variable examinada puede ser comparada con uno o varios patrones. Es decir una opción puede activarse si la variable vale "A" o "a"; o si la variable es un número de tres cifras:

Para ello se pueden usar los mismos patrones que se usan en otros lugares de linux -y que nosotros no hemos visto:

```
case "$1" in
 +(start|run) ) /usr/app/startup-script ;;
 @([Ss])top ) /usr/app/stop-script ;;
 esac

?(pattern1 | pattern2 | ... | patternn)
```

```
zero or one occurrence of any pattern

*( pattern1 | pattern2 | ... | patternn)
zero or more occurrences of any pattern

@( pattern1 | pattern2 | ... | patternn)
exactly one occurrence of any pattern

+( pattern1 | pattern2 | ... | patternn)
one or more occurrence of any pattern

!( pattern1 | pattern2 | ... | patternn)
all strings except those that match any pattern

read opcion


case $opcion in
        [a,A])
        echo "has pulsado la letra a o A"
        ;;

        [0-9][0-9][0-9])
        echo "el número elegido es menor de 1000 "
        ;;

        b|c|d|e)
        echo "b, c, d, e: son opciones no soportadas"
....
```

5.3 Ejecución de comandos condicional con && y ||

Cuando dentro del cuerpo del if o del else se pretende ejecutar un sólo comando, existe una alternativa más compacta. Se usan unos operadores que bash reconoce. Estos operadores son are `&&` (léase como "y" o AND) y || (léase como "o" ó OR).

5.3.1 Operador &&:

Sintaxis

```
comando1 && comando2
```

`comando2` se ejecutará si y sólo si `comando1` tiene éxito y devuelve un valor de retorno de 0. Si el comando1 falla, bash entiende que ya no vale la pena seguir ejecutando comandos,

porque en una operación AND, cuando uno de los dos operandos es false, toda la operación es false también sin importar el valor de los otros operandos.

5.3.2 Operador ||

La sintaxis es similar

```
comando1 || comando2
```

El comando2 se ejecutará si y sólo sí el comando1 falla devolviendo un valor de retorno diferente de cero. Se puede entender que bash considera que no vale la pena seguir ejecutando comando2 cuando el comando1 ha devuelto un true, ya que en una operación OR, si uno de los operandos es true, toda la operación es true sin importar el valor de los otros opeandos

Se pueden combinar, a sabiendas que la operación && tiene preferencia (se ejecuta antes) y se dejan el operador || para el final:

```
comando1 && comando  || comando
```

`comando1 && comando2` **`if exit status is zero`** `||` `comando3` **`if exit status is non-zero`**

if `comando1` is executed successfully then shell will run `comando2` and if comando1 is not successful then `comando3` is executed. Para entender este ejemplo, pensemos que "bash quiere devolver un valor de retorno exitoso" mientras no lo consigue, lo sigue intentando hasta donde la línea de comandos le permite, pero en cuanto consigue un valor de retorno sin errores (exitoso), entonces deja de ejecutar más comandos.

Example:

```
rm myf && echo "File successfully removed" || echo "File not
removed"
```

If file (`myf`) is removed successful (exit status is zero) then "*echo File is removed successfully*" statement is executed, otherwise "*echo File is not removed*" statement is executed (since exist status is non-zero)

6 Bucles

Un bucle es una estructura que consigue repetir un bloque de comandos varias veces. Escribamos una aproximación y la comentaremos después

bucle

```
        comando1;
        comando2;
        comando3;
fin_bucle
```

El bloque de comandos anterior (del comando1 al comando3) se ejecutará varias veces. Al bucle le falta Existen diferentes tipos de bucles en todos los lenguajes.

Los bucles en bash-scripting difieren un poco de lo que son en la mayoría de los lenguajes de programación, pero vienen a cumplir una misión similar; a solucionar el problema de, por ejemplo:

1. "Quiero repetir algo varias veces"
2. "Quiero que mi script haga esto mientras se dé esta condición"
3. "Para cada uno de los elementos siguientes, quiero que mi script haga esto otro".
4. “Quiero que mi script haga esto otro a la vez que una variable entera cambia de valor entre uno de inicio y uno de fin, incrementándose o decrementándose cada iteración una determinada cantidad”

Así, tenemos dos tipos de bucles diferentes en bash para estos tres bucles: `for` y `while`.

Es necesario, en los bucles pensar :

- En el caso general, en una iteración cualquiera que ocurre después de la primera y antes de la última. A veces hay que pensar en dos o tres iteraciones conjuntamente
- En el caso inicial. En la primera vez que se itera en el bucle. Las circunstancias pueden ser distintas (por ejemplo una variable debe ser inicializada antes de la primera vez)
- En el caso final. ¿Cómo termina el bucle y sus variables?

6.1 Bucles for

La estructura de control for se puede utilizar de dos maneras diferentes.

- Para recorrer una lista de palabras, una a una
- Para usar una variable numérica que va contando desde un valor inicial hasta uno final.

En ambos casos, con el bucle for, hay una variable que automáticamente va recibiendo un valor distinto a cada iteración.

6.1.1 “for para recorrer una lista de elementos”

El objetivo principal de for es recorrer una lista de valores e iterar sobre cada uno de dichos elementos, asignándolos a la variable de control. Veamos algunos ejemplos de cómo recorrer los elementos de una lista:

```
$ for nombre in pedro pepe juan jose; do
> echo "Hola $nombre"
> done
Hola pedro
Hola pepe
Hola juan
Hoja jose
```

En primer lugar apreciamos que todo lo que se puede hacer en un script también se puede hacer en bash. La estructura de for es la siguiente:

```
for nombre_var in LISTA ;  do
     comando 1
     comando 2
     # Y asi los que queramos
done
```

nombre_var es el nombre de una variable (el que nosotros queramos, y sólo el nombre, no precedida de $). **LISTA** es una lista de elementos. Podemos escribir una lista de palabras, una variable que ya contenga una lista de elementos (en este caso sí que la precederíamos de $, para que sea reemplazada por los elementos de la lista que contiene), o un comando que se ejecute ahí mismo y genere una lista.

Con este bucle lo que conseguimos es: **comando 1**, **comando 2**, etc. se ejecutan tantas veces como elementos haya en LISTA. En el siguiente ejemplo se ejecutará 4 veces.

```
for nombre in pedro pepe juan jose; do

     echo "Hola $nombre"

done
```

Para la primera vez, `$nombre` valdrá el primer elemento de la lista (pedro) , la segunda vez que se ejecuten los comandos `$nombre` valdrá el segundo elemento de la lista (pepe) , y asi sucesivamente hasta llegar al último elemento en el que el bucle habrá terminado (y el script seguirá su flujo de ejecución normal hacia abajo). El resultado es:

```
Hola pedro
Hola pepe
Hola juan
Hoja jose
```

LISTA puede ser una lista de archivos dada implícitamente. Más concretamente, podemos usar los wildcards que ya conocemos para referirnos a varios archivos con un nombre parecido:

```
#!/bin/bash
# Listador de archivos y directorios ocultos

DIR=$1
if [ -z $DIR ]; then
        echo "El primer argumento debe ser un directorio"
        exit 1
fi

if [ ! -d $DIR ]; then
        echo "El directorio debe existir"
        exit 1
fi

for i in $DIR/.*; do
   echo "Encontré el elemento oculto $i"
done

echo "Saliendo..."
```

`.*` hace referencia a todos los archivos y directorios ocultos de $DIR. Al detectar for el wildcard, sabrá que los elementos de la lista son cada uno de los archivos coincidentes. Ejecuta el ejemplo anterior si quieres ver cómo funciona. Puedes hacer uso con este bucle de lo que ya sabes de wildcards referidos a archivos.

El elemento LISTA del bucle for, puede ser un comando entrecomillado con comillas laterales; un comando tal que su salida sea una lista de elementos. Así, sería lo mismo:

```
for i in *; do
```

que:

```
for i in `ls`; do
        echo "el fichero es $i"
done
```

En este punto, conviene entender que la ejecución del bucle for se realiza en varias fases o etapas (es una característica de este lenguaje). Usaremos el siguiente ejemplo:

```
#suma todos los números comprendidos entre los dos parámetros

for num in $( seq $1 $2 ) ; do

      suma=$(( $suma + $num ))

done
```

Cuando bash examina la línea "for.. " anterior, descubre que hay un comando en su interior que debe ser ejecutado primero. Esto lo realiza "aparte", el comando es el siguiente:

```
seq $1 $2
```

Pero este mismo comando seq lleva en sus argumentos variables. Antes de lanzarlo a ejecutar, bash debe sustituir las variables por sus valores, (suponga que el script se ha lanzado pasando 2 y 6 como primer y segundo argumento respectivamente). La sustitución genera el siguiente comando

```
seq 2 6
```

Este comando se ejecuta y se genera el siguiente resultado:

```
2 3 4 5 6
```

El cual marca el final de la ejecución del comando interno. Es hora de reemplazar el comando interno por su resultado

```
for num in $( seq $1 $2 ) 2 3 4 5 6 ; do

    echo "num es $num"

done
```

… y sigue con normalidad la ejecución

6.1.2 “for” para usar con una variable numérica incrementalmente

Existe una alternativa al uso del `for` que imita el estilo de uso de bucles for en el lenguaje “C” y otros. En este tutorial no se usará apenas

```
$ for ((i=0;i<=4;i++))
>do
>    echo $i
>done
0
1
2
3
```

6.2 Bucles con while

El otro tipo de bucle posible en bash-scripting es el bucle while. Este bucle viene a cumplir con el propósito de "quiero que se haga esto mientras que se dé esta condición". Su estructura es la siguiente:

```
while comando; do
        ...
        # Comandos a ejecutar
        ...
done
```

Donde el "comando" que aparece después del while es ejecutado y su valor de retorno se utiliza para iterar o volver a repetir los comandos que hay dentro del while. El comando más habitual para usar ahí es el `test`.

Como ya sabemos usar el bucle anterior, este lo entenderemos con algunos pocos ejemplos:

```
#!/bin/bash
j=0
while [ $j -lt 10 ]; do
            echo "j vale $j"
            j=$[ $j + 1 ]
done

# Aquí sigue el flujo de ejecución normal del script
```

Este ejemplo es interesante. La condición para que el bucle se siga ejecutando es que j valga menos de 10. Observa la forma mediante la cual en cada ciclo del bucle sumamos 1 a la variable j.

Recuerda, en la construcción de un bucle hay cuatro dificultades clave:

- Delimitar correctamente el bloque de comandos a repetir

- Especificar la pregunta adecuada para la condición de continuación
- Entrar la primera vez en el bucle con la inicialización correcta
- Calcular la condición de parada si se usan variables numéricas de valor incremental para la condición de continuación

6.2.1 Romper la iteración en los bucles

Tanto en el bucle for como en el while, podemos usar **break** para indicar que queremos salir del bucle en cualquier momento:

```
#!/bin/bash

MINUM=8

while [ 1 ]; do
        echo "Introduzca un número: "
        read USER_NUM

        if [ $USER_NUM -lt $MINUM ]; then
                echo "El número intrododucido es menor que el mío"
                echo " "
        elif [ $USER_NUM -gt $MINUM ]; then
                echo "El número introducido es mayor que el mío"
                echo " "
        elif [ $USER_NUM -eq $MINUM ]; then
                echo "Acertaste: Mi número era $MINUM"
                break
        fi
done

# Comandos inferiores...

echo "El script salió del bucle. Terminando..."
exit
```

La condición es que el bucle se ejecute siempre (`test 1` siempre retornará TRUE, y el bucle se seguirá ejecutando). Pero, si el número introducido es el mismo que el valor de `$MINUM`, break hará que se salga de el bucle y continúe ejecutando los comandos inferiores.

6.3 shift y bucles

La instrucción `shift` realiza un desplazamiento en el valor almacenado en la secuencia de argumentos, haciendo que los valores de `$1`, `$2`, `$3`, etc. se actualicen. $1 pasa a valer el valor que tenía $2, $2 el que tenía $3

```
#!/bin/bash

echo "este script suma los tres primeros argumentos"
echo "El primer argumento es $1"
suma=$1

shift
echo "El segundo argumento es $1"
suma=$[ $suma + $1 ]

shift
echo "El tercer argumento es $1"
suma=$[ $suma + $1 ]

echo "la suma de los tres primeros argumentos es $suma"
```

`shift` se vuelve especialmente útil cuando lo usamos dentro de un bucle while, porque logramos recorrer toda la lista de argumentos,

```
while [ $1 != ""] ; do
      #se usa $1 de alguna forma
      shift
done
```

7 Funciones y ejecuciones externas

Si bien las funciones en bash scripting difieren de lo que son en la mayoría de los lenguajes de programación, sí comparten, en cambio, su razón de ser básica. Una función es un conjunto de comandos agrupados a los que -como grupo- se les da un nombre especial y pueden ser ejecutados desde cualquier punto del script mediante el uso de ese nombre especial.

El objetivo primero, es evitar que se repitan secuencias de comandos que deben ejecutarse en diferentes puntos de un script. No se considera una buena práctica repetir en distintos sitios las mismas líneas de código. Así, se agrupan en una función, y llamando a la función por su nombre, esas líneas son ejecutadas dondequiera que la llamemos. Las funciones se preparan o implementan en un lugar del fichero del script (generalmente al principio, antes del script en sí), y después son invocadas o llamadas desde el script principal. Otra forma de verlo es pensar que una función es un script hecho dentro de otro script.

Una función se DEFINE, "implementa" o "programa"de la siguiente forma:

```
function nombre_de_la_funcion () {
        ## Aquí van los comandos que se ejecutarán
        ## cuando llamemos a la función
}
```

`nombre_de_la_funcion` lo elegimos nosotros, y le podemos dar el nombre que queramos, con las mismas condiciones que el nombre de una variable (no espacios, no caracteres extraños, etc.). Entre las dos llaves ({ y }) se introducen los comandos que se quiere que se ejecuten cuando la función sea llamada.

Un ejemplo sencillo podría ser el siguiente:

```
#!/bin/bash
function uso () {
        echo "Este script recibe dos argumentos."
        echo "El primero debe ser --opc1 u --opc2 ,"
        echo "y el segundo debe ser un fichero existente."
        echo "--"
}

if [ $# -ne 2 ]; then
        uso
        exit 1
fi

case $1 in
        --opc1)
                if [ -e $2 ]; then
                        echo "El script terminó con éxito"
                        exit 0
                else
                        uso
                        exit 1
                fi
        ;;
        --opc2)
                if [ -e $2 ]; then
                        echo "El script terminó con éxito"
                        exit 0
                else
                        uso
                        exit 1
                fi
        ;;
```

```
        *)
                uso
                exit 1
        ;;
esac
```

Se necesita definir la función antes de llamarla. Se llama a la función simplemente con el nombre que le dimos; ese nombre se convierte en algo similar a un comando más.

De no haber usado aquí las funciones, tendríamos que haber reescrito los `echo` que indican cómo se usa el script una y otra vez en cada sitio donde los hubiésemos necesitado. En este ejemplo, mientras no se ejecute el script con los argumentos adecuados, llamamos a la función uso que explica al usuario cómo se debe llamar el script. Esta estrategia es muy frecuente y útil en scripts más complejos con muchas opciones.

7.1 Paso de parámetros a funciones

Es bastante habitual que las funciones sean aprovechadas para realizar cálculos repetidos o especializados. Para ello, los comandos dentro de las funciones deben acceder a datos. Dichos datos se pueden "pasar" a las funciones cada vez que son invocadas. Una de las posibilidades de "pasar" datos a las funciones es como argumentos en la misma línea en la que son invocados, de forma idéntica a cómo le pasamos argumentos a un script.

Dentro de las funciones, los argumentos recibidos son accedidos de la misma forma que lo hace un script, por ej: $1 $2 se corresponden con el primer y el segundo parámetro pasado a la función respectivamente. Evidentemente $1 y $2, dentro de la función no son los argumentos pasados al script. Cada ámbito tiene valores diferentes para $1, $2, $3, $#, $@, etc.

Al convertirse la función en un "comando posible" dentro del script, los parámetros se le pasan igual que a cualquier otro comando. Prueba el siguiente script y revisa su ejecución:

```
function lista_parametros () {
        echo "Se ha llamado a lista_parametros, con $#"
        echo "parámetros. Mostrando los parámetros:"
        echo " "
        b=7

        numparms=$#
        local j=1
        while [ $j -le $numparms ]; do
                echo "El parámetro $j contiene $1"
                shift
```

```
                    local j=$[$j+1]
            done
    }

    lista_parametros a b c d dsfafd d

    echo $b
    echo $j
```

La función `lista_parametros` en este ejemplo recibe varios parámetros del mismo modo que cualquier otro comando dentro del script, y se procesan igual que ya conocemos para los scripts.

7.1.1 Visibilidad de las variables.

Para terminar con este ejemplo, nos fijamos en "**`local j=...`**", . La palabra "local" indica que la variable `j` sólo existe dentro de la función. Por ello, cuando hacemos echo $j fuera de la función, no obtenemos ningún valor. La variable `$b`, al no ser declarada explícitamente local, sino global, existe dentro y fuera de la función. Cualquier otra variable con valor en el flujo principal del script existe dentro de las funciones que definamos.

Hemos dicho que cada función se convierte en una especie de "comando posible" dentro del script donde está definida. La consecuencia es que la función se asemeja a un comando cualquiera, y al igual que hacíamos con `exit 0` o `exit1` en el flujo principal del script, podemos hacer que la llamada a una función genere un valor de retorno de éxito o bien de error. Lo conseguimos con el comando `return`. Se usa exactamente igual que `exit` en el script, pero usaremos return dentro de las funciones (si se usase exit, se terminaría todo el script)

Meter un ejemplo de comprobación de cosas dentro de una función que aproveche el return

7.2 Retorno de valores desde las funciones.

En la mayoría de lenguajes, las funciones pueden devolver un valor de retorno de una forma fácil. En muchos lenguajes, las funciones se declaran de antemano indicando el tipo de valor que pueden devolver. Así por ejemplo, la siguiente función en C recoge dos valores enteros y devuelve un valor entero. (nótese que sólo se va a ejecutar un comando return)

```
// esto es lenguaje C
     int minimo (int a, int b) {
          if (a < b) return a;
          return b;
     }
     .
```

```
.
int i;
i = minimo(3,4);
```

En shell script el valor de retorno no es el mismo que en otros lenguajes. La palabra `return` devuelve un valor numérico de forma similar a cómo los procesos devuelven un código numérico de error. No sirve para devolver valores de otro tipo (cadenas, etc). Esta restricción la podemos superar de varias maneras si nuestra intención es que al llamar a una función se recoja un valor de retorno devuelto por la función.

7.2.1 Retornar un valor con echo

Para ilustrar las posibles maneras, vamos a suponer una función llamada "minimo" que recibe dos números y devuelve el número más pequeño de los dos.

La primera forma es realizar un `echo` del resultado dentro de la función y recogerlo en el script principal,

```
function minimo(){
    if [ $1 -lt $2 ] ;
    then
     echo $1
    else
         echo $2
    fi
}
ciertoNumero=5
read numeroIntroducido
valorMinimo=$( minimo $ciertoNumero $numeroIntroducido )
if [ $valorMinimo -eq $numeroIntroducido ] ;
then
     echo "El número introducido es más pequeño"
else
     echo "El número introducido no es más pequeño"
fi
```

7.3 Devolver un estado con return

Otra alternativa es devolver un estado de finalización. Igual que los procesos indican al sistema operativo un estado de la finalización de su ejecución, como ocurre en el siguiente ejemplo recordatorio:

```
~$ ls kk
ls: no se puede acceder a kk: No existe el ....
```

```
~$ echo $?
2
$ [ 3 -gt 2 ]
$ echo $?
0
```

Una función también puede emitir un estado de finalización. Desde dentro de ella ejecutamos la instrucción "`return`" seguida del número que expresará el código de respuesta. Generalmente esto se utiliza para indicar estados de finalización (error, ok, fichero no encontrado, no autorizado, etc) pero podemos utilizarlo dentro de nuestros scripts con otra función

```
function esMenor(){
 if [ $1 -lt $2 ] ;
         then
            return 0
         else
            return 1
 fi
}
ciertoNumero=5
read numeroIntroducido

if esMenor $valorMinimo $numeroIntroducido  ;
   then
       echo "El número introducido es más pequeño"
    else
       echo "El número introducido no es más pequeño"
fi
```

El principal inconveniente de este método, es que sólo podemos devolver números, y que por convenio, todos los números excepto el 0 suelen expresar un error. Si nuestra función usa este mecanismo para devolver distinto de 0 valores que no son errores, estaremos usando correcta pero no convencionalmente la función

7.4 Compartir una variable

Compartir el valor de una variable. Las variables son visibles dentro y fuera de las funciones, por tanto siempre es posible depositar un valor desde dentro de la función y leerlo desde el cuerpo principal del script

```
    function esMenor(){
```

```
        if [ $1 -lt $2 ] ;
        then
         menor="SI"
        else
            menor="NO"
        fi
    }
    ciertoNumero=5
    read numeroIntroducido
    esMenor $valorMinimo $numeroIntroducido  ;
    if [ $menor = "SI" ]
    then
         echo "El número introducido es más pequeño"
    else
         echo "El número introducido no es más pequeño"
    fi
```

7.5 Recoger el nombre de una variable y escribir en ella

Esta manera es bastante compleja, requiere dominar el comando "`eval`" y está aquí tan sólo para ser consultada como referencia. The other way to return a value is to write your function so that it accepts a variable name as part of its command line and then set that variable to the result of the function:

```
function myfunc(){
              local  __resultvar=$1
              local  myresult='some value'
              eval $__resultvar="'$myresult'"
}

myfunc result
echo $result
```

Como el nombre de la variable está almacenada dentro de otra variable (`$__resultvar`), nos es imposible cambiar su valor directamente. Hemos de apoyarnos en el comando "eval" para poder cambiar el valor. "Eval" básicamente indica a bash que interprete la línea dos veces. En la primera interpretación la variable `$__resultvar` es sustituida por su valor, que es el nombre de la variable. En esta primera interpretación, las comillas simples existentes alrededor de `$myresult` hacen que no se sustituya nada y se mantenga ese `$myresult`, aunque las comillas sí desaparecen. Por tanto, en el ejemplo, al final la expresión

que se queda `result=$myresult`. En la segunda interpretación se realiza una asignación normal de variable a la variable "result".

When you store the name of the variable passed on the command line, make sure you store it in a local variable with a name that won't be (unlikely to be) used by the caller (which is why I used __resultvar rather than just resultvar). If you don't, and the caller happens to choose the same name for their result variable as you use for storing the name, the result variable will not get set. For example, the following does not work:

```
function myfunc(){
          local  result=$1
          local  myresult='some value'
          eval $result="'$myresult'"
      }

      myfunc result
      echo $result
```

The reason it doesn't work is because when eval does the second interpretation and evaluates result='some value', result is now a local variable in the function, and so it gets set rather than setting the caller's result variable.

For more flexibility, you may want to write your functions so that they combine both result variables and command substitution:

```
function myfunc()
{
    local  __resultvar=$1
    local  myresult='some value'
    if [[ "$__resultvar" ]]; then
        eval $__resultvar="'$myresult'"
    else
        echo "$myresult"
    fi
}

myfunc result
echo $result
result2=$(myfunc)
echo $result2
```

Here, if no variable name is passed to the function, the value is output to the standard output.

8 Redirección avanzada

http://wiki.bash-hackers.org/howto/redirection_tutorial

En los scripts, como en las sesiones interactivas, se puede redirigir la salida estándar y de error, así como la entrada estándar. En ocasiones, hay comandos que emiten mensajes por la salida estándar y de error, y podemos desear:

- redirigir sólo una de las dos salidas
- redirigir ambas salidas al mismo fichero
- intercambiar ambas salidas

Lo más habitual en redirecciones especiales, es escribir en la salida de error. Para que un comando escriba por la salida de error, la redirección se realiza:

```
comando >&2
```

Generalmente, la redirección se consigue con comando N>&M, donde

- N es el descriptor de salida redirigido
- y M es el que lo recibe

El descriptor es un número que identifica a una salida. La salida estándar tiene el número 1, la salida de error el número 2. Ya no hay más salidas predefinidas, pero se pueden usar otros descriptores si el usuario lo desea para emitir datos hacia otros canales.

Por ejemplo:
swap STDOUT and STDERR: "**3>&1 1>&2 2>&3**"
In a bit more detail:

- `3>&1` - moves file descriptor 1 (aka stdout) to file descriptor 3.
- `1>&2` - moves file descriptor 2 (aka stderr) to file descriptor 1.
- `2>&3` - moves file descriptor 3 to file descriptor 2 (aka stderr).

Similar to:

```
tmp    = stdout
stdout = stderr
stderr = tmp
```

and thereby swapping the standard out and the standard error. Which then allows the stderr (rather than stdout) to be piped into grep.

as in:

```
(/usr/bin/$COMMAND $PARAM 3>&1 1>&2 2>&3 | grep -v
$uninteresting_error ) 3>&1 1>&2 2>&3
```

In this case grep command would be operating on string generated as an error by the previous command. Note that there are three consecutive redirections within the same command.

Las salidas estándar y de error, y los ficheros tienen todos unos descriptores de ficheros asociados, que son simplemente números. La salida estándar tiene el número 1 y la de error el número 2 (de ahí que 2> se utilizase para redirigir la salida de error). El número M anterior se corresponde con estos descriptores. Por ejemplo ls >&2 redirige la salida 1 (standar) a su salida de error (2)

En otras ocasiones se quiere establecer una redirección para todos los comandos del script, sin tener que indicar en cada uno de ellos la misma redirección. Esto se consigue con la ejecución de :

exec < nombre_fichero

In Bash the exec built-in replaces the shell with the specified program. So what does this have to do with redirection? exec also allow us to manipulate the file descriptors. If you don't specify a program, the redirection after exec modifies the file descriptors of the current shell.

```
#!/bin/bash
# Redirecting stdin using 'exec'.

exec 6<&0          # Link file descriptor #6 with stdin.
                   # Saves stdin.

exec < data-file   # stdin replaced by file "data-file"

read a1            # Reads first line of file "data-file".
read a2            # Reads second line of file "data-file."

echo
echo "Following lines read from file."
echo "-------------------------------"
echo $a1
echo $a2

echo; echo; echo

exec 0<&6 6<&-
#  Now restore stdin from fd #6, where it had been saved,
#+ and close fd #6 ( 6<&- ) to free it for other processes to use.
#
# <&6 6<&-    also works.

echo -n "Enter data  "
read b1  # Now "read" functions as expected, reading from normal stdin.
echo "Input read from stdin."
echo "----------------------"
echo "b1 = $b1"
echo

exit 0
```

8.1 “Here Documents”

Es posible que los datos de entrada de algún comando estén en el propio script. Esto es ciertamente sorprendente y puede costar entenderlo al principio. Veámoslo con el siguiente ejemplo

```
cat << EOF
juan
perico
andres
EOF
```

El comando anterior es el cat (sólo él). La entrada estándar por defecto del comando es el teclado, (aquí cat no lleva argumentos, lo cual reforzaría esta afirmación). Sin embargo, con la redirección <<, la entrada estándar es reemplazada y las propias líneas que siguen al comando cat son las que están redirigidas al comando. Dicho de otro modo, las líneas (juan, perico, etc.) son las que se "enchufan" a la entrada del comando.

La redirección es similar a la de un fichero, pero el fichero son esas líneas del script. Queda una cuestión … ¿Todas las líneas que siguen al comando son entrada a dicho comando?

o mejor: "¿hasta qué línea se debe transferir al comando? La palabra EOF es la que tiene la clave en esta cuestión. Esa palabra la hemos elegido arbitrariamente. Se puede elegir la que se quiera, sabiendo que no debe estar entre las que se va a tratar como entrada del comando. Esa palabra marcará el final de las líneas (o final de fichero "EndOfFIle"). De forma que la siguiente línea a EOF, será ya una línea cualquiera del script.

Esto es muy útil cuando queremos generar datos desde el propio script. Por ejemplo un fichero de pruebas en el propio script que facilita la portabilidad para hacer

```
cat > ficheroPruebas << EOF
juan
perico
andres
EOF
```

8.2 "Here strings"

http://linux.die.net/abs-guide/x15683.html

A *here string* can be considered as a stripped-down form of a *here document*. It consists of nothing more than **COMMAND <<<$WORD**, where `$WORD` is expanded and fed to the stdin of *COMMAND*.

As a simple example, consider this alternative to the echo-grep construction.

```
      # Instead of:
if echo "$VAR" | grep -q txt   # if [[ $VAR = *txt* ]]

# Try:
if grep -q "txt" <<< "$VAR"

      then
   echo "$VAR contains the substring sequence \"txt\""
fi
```

8.3 Process substitution

In computing, process substitution is a form of inter-process communication that allows the input or output of a command to appear as a file. The command is substituted in-line, where a file name would normally occur, by the command shell. This allows programs that normally only accept files to directly read from or write to another program. En resumen, existe un comando que pretende leer de un fichero, pero en lugar del fichero, se ejecuta otro comando cuya salida estándar es el contenido que el primer proceso (o principal) pretende leer. En muchas situaciones este funcionamiento resulta equiparable a la concatenación de

comandos mediante '|'
(http://unix.stackexchange.com/questions/17107/process-substitution-and-pipe)

En bash, la sustitución de procesos se realiza mediante los símbolos ' <(' y ') ' que engloban al comando que se ejecuta y cuya salida aparece como fichero para el proceso principal

```
$ tail <( ls -l )
$ ls -l | tail
```

The Unix diff command normally accepts the names of two files to compare, or one file name and standard input. Process substitution allows you to compare the output of two programs directly:

```
$ diff <(sort file1) <(sort file2)
```

Without process substitution, the alternatives are:

1. Save the output of the command(s) to a temporary file, then read the temporary file(s).

```
$ sort file2 > /tmp/file2.sorted
$ sort file1 | diff - /tmp/file2.sorted

$ rm /tmp/file2.sorted
```

2. Create a named pipe (also known as a FIFO), start one command writing to the named pipe in the background, then run the other command with the named pipe as input.

```
$ mkfifo /tmp/sort2.fifo
$ sort file2 > /tmp/sort2.fifo &
$ sort file1 | diff - /tmp/sort2.fifo
$ rm /tmp/sort2.fifo
```

Process substitution can also be used to capture output that would normally go to a file, and redirect it to the input of a process. The Bash syntax for writing to a process is `>(command)`. Here is an example using the `tee`, `wc` and `gzip` commands that counts the lines in a file with `wc -l` and compresses it with `gzip` in one pass:

```
$ tee >(wc -l >&2) < bigfile | gzip > bigfile.gz
```

Atención, el process substituion NO es una redirección, el siguiente ejemplo no funciona:

```
while read linea ; do
```

```
    echo "he leído : $linea" ;
done <(ls -l)
```

Ya que "<(ls -l)" realmente se sustituye por un fichero temporal que se crea y se quedaría algo así:

```
while read linea ; do
    echo "he leído : $linea" ;
done /dev/fd/63
```

Para realizar la redirección habría que indicarle la redirección explícitamente, añadiendo el símbolo habitual.

```
while read linea ; do
    echo "he leído : $linea" ;
done < <(ls -l)
```

Aunque puede parecer demasiado extraño, el ejemplo anterior tiene más sentido que usar una tubería, ya que con la tubería se crea un subshell que tiene variables aparte. Con process substitution, en el bucle while podrían asignarse valores a variables que se recuperarían después.

8.4 Ejecuciones externas.

Comando **`eval`**

9 Lectura de datos interactiva.

El comando interno al shell `read` se encarga de leer caracteres de teclado (normalmente hasta encontrar un retorno de carro) y almacenarlos en la variable pasada al mismo.

- con la opción '-r' usará la variable REPLY si no se especifica ninguna.
- La opción '-p' permite mostrar un mensaje de prompt.
- will use Readline to obtain the line when given the `-e' option.
- The read builtin also has additional options to control input: the `-s' option will turn off echoing of input characters as they are read,
- the `-t' option will allow read to time out if input does not arrive within a specified number of seconds,
- the `-n' option will allow reading only a specified number of characters rather than a full line, and the `-d' option will read until a particular character rather than newline.

Este comando se aprende a base de ejemplos.

9.1 Ejemplos más básicos

9.1.1 Lectura de variables simple

El ejemplo más básico consiste en leer una variable directamente del teclado. Esto constituye la primera y principal manera de leer datos del teclado. Observa el siguiente

ejemplo y observa cómo suele emitirse antes un mensaje indicando al usuario lo que se espera que escriba

```
echo "escribe el número de factura"
read numFact

nombreFich="factura_${numFact}"
if [ -f $nombreFich ] ; then
    echo "no se encuentra esa factura"
fi
```

9.1.2 Usando read para validar la entrada

Es posible que se desee preguntar al usuario su conformidad con lo que se obtiene de resultado o se quiere esperar a que esté listo para continuar. En este caso se desea leer de teclado aunque no se espera que se introduzca dato alguno, tan sólo es importante la pulsación de una tecla. En estos casos no se desea que el comando read espere a la pulsación de la tecla "intro". De hecho lo que se debe hacer es simplemente especificar la longitud de la cadena esperada, que será de 1 para el caso que nos trae.

```
    ...
    echo "Ponga papel en la impresora y después, pulse una tecla"
    read -n1 varInutil

    imprimir;
```

¿Por qué se ha nombrado a la variable "varInutil"? Porque el contenido de la variable no importa, tan sólo se ha puesto ahí por completar totalmente el comando read y que se guarde lo pulsado en una variable, que no vamos a usar.

9.1.3 Evitar el eco de las pulsaciones

En el caso anterior, el carácter de la tecla pulsada aparecerá en la pantalla. Esto no resulta de utilidad e incluso puede confundir. Podemos indicar al comando que no haga echo de las pulsaciones de teclas y haga una entrada silenciosa. Lo mismo podría ser necesario, por ejemplo, al introducir una contraseña, en la que nunca se quiere ver lo que se está escribiendo.

```
    ...
    echo "Escribe tu código de impresión de cinco caracteres"
    read -n5 -s codigoImpresion
```

```
      if imprimir $codigoImpresion ; then
          echo " Se ha enviado el trabajo a imprimir
satisfactoriamente"
      fi
```

9.2 Ejemplos elaborados

9.2.1 Rudimentary cat replacement

A rudimentary replacement for the cat command: read lines of input from a file and print them on the terminal.

```
opossum() {
  while read -r; do
    printf "%s\n" "$REPLY"
  done <"$1"
}
```

Note: Here, read -r and the default REPLY is used, because we want to have the real literal line, without any mangeling. printf is used, because (depending on settings), echo may interpret some baskslash-escapes or switches (like -n).

9.2.2 Press any key...

Recuerdas el comando "pause" del MS-DOS. Pues aquí hay algo similar

```
pause() {
  local dummy
  read -s -p "Press any key to continue..." -n 1 dummy
}
```

9.2.3 Reading Columns.

`read` puede ser utilizado para leer una línea con varias palabras y asignar cada palabra a una variable diferente.

```
var="one two three"
read -r col1 col2 col3 <<< "$var"
printf "col1: %s col2: %s col3 %s\n" "$col1" "$col2" "$col3"
```

Cuidado! No se pueden usar tuberías:

```
echo "$var" | read col1 col2 col3 # does not work!
printf "col1: %s col2: %s col3 %s\n" "$col1" "$col2" "$col3"
```

¿Por qué? Los comandos de una tubería se ejecutan en un "subshell" que no pueden modificar el shell padre. Como resultas, las variables col1, col2 y col3 del shell padre no son modificadas (see article: Bash and the process tree).

If the variable has more fields than there are variables, the last variable get the remaining of the line:

```
read col1 col2 col3 <<< "one two three four"
printf "%s\n" "$col3" #prints three four
```

9.2.4 Changing The Separator

By default reads separates the line in fields using spaces or tabs. You can modify this using the special variable IFS, the Internal Field Separator.

```
IFS=":"
read -r col1 col2 <<< "hello:world"
printf "col1: %s col2: %s\n" "$col1" "$col2"
```

Here we use the `var=value` command syntax to set the environment of read temporarily. We could have set IFS normally, but then we would have to take care to save its value and restore it afterward `(OLD=$IFS IFS=":"; read ....;IFS=$OLD)`.
The default IFS is special in that 2 fields can be separated by one or more spaces. When you set IFS to something else, the fields are separated by exactly one character:

```
IFS=":" read -r col1 col2 col3 <<< "hello::world"
printf "col1: %s col2: %s col3 %s\n" "$col1" "$col2" "$col3"
```

See how the :: in the middle infact defines an additional empty field.
The fields are separated by exactly one character, but the character can be different between each field:

```
IFS=":|@" read -r col1 col2 col3 col4 <<< "hello:world|in@bash"
printf "col1: %s col2: %s col3 %s col4 %s\n" "$col1" "$col2" "$col3" "$col4"
```

9.2.5 Are you sure?

```
asksure() {
        echo -n "Are you sure (Y/N)? "
        while read -r -n 1 -s answer; do
          if [[ $answer = [YyNn] ]]; then
            [[ $answer = [Yy] ]] && retval=0
            [[ $answer = [Nn] ]] && retval=1
            break
          fi
        done

        echo # just a final linefeed, optics...

        return $retval
}
```

```
### using it
if asksure; then
  echo "Okay, performing rm -rf / then, master...."
else
  echo "Pfff..."
fi
```

Ask for a path with a default value
Note: The -i option was introduced with Bash 4

```
read -e -p "Enter the path to the file: " -i "/usr/local/etc/"
FILEPATH
```

The user will be prompted, he can just accept the default, or edit it.

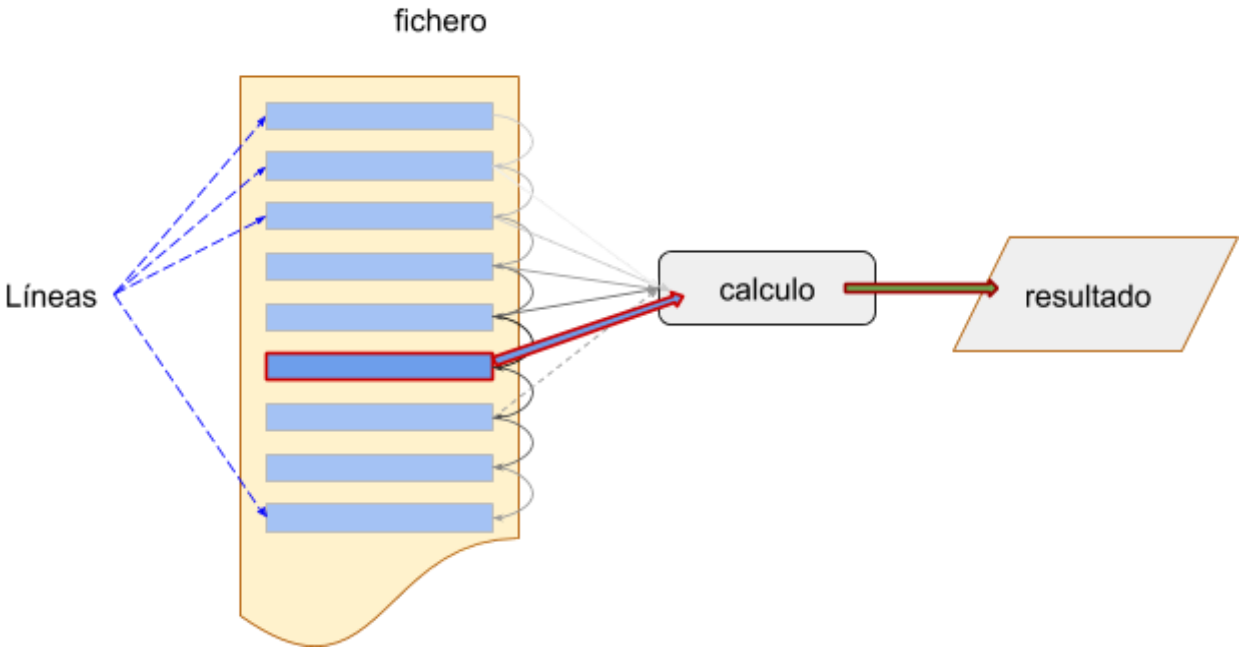
9.2.6 Lectura de datos interactiva usando interfaz gráfica

ver zenity

9.3 Procesado de ficheros línea a línea

En muchos problemas, hay que leer las líneas de algún fichero y procesarlas una a una. Por ejemplo: imagina un fichero en el que cada línea contiene un número y deseas sacar la cantidad de números pares e impares que hay. Se necesita leer tratar cada línea una a una y hacerlo con todas, ya que no existe ningún comando que haga esa tarea.

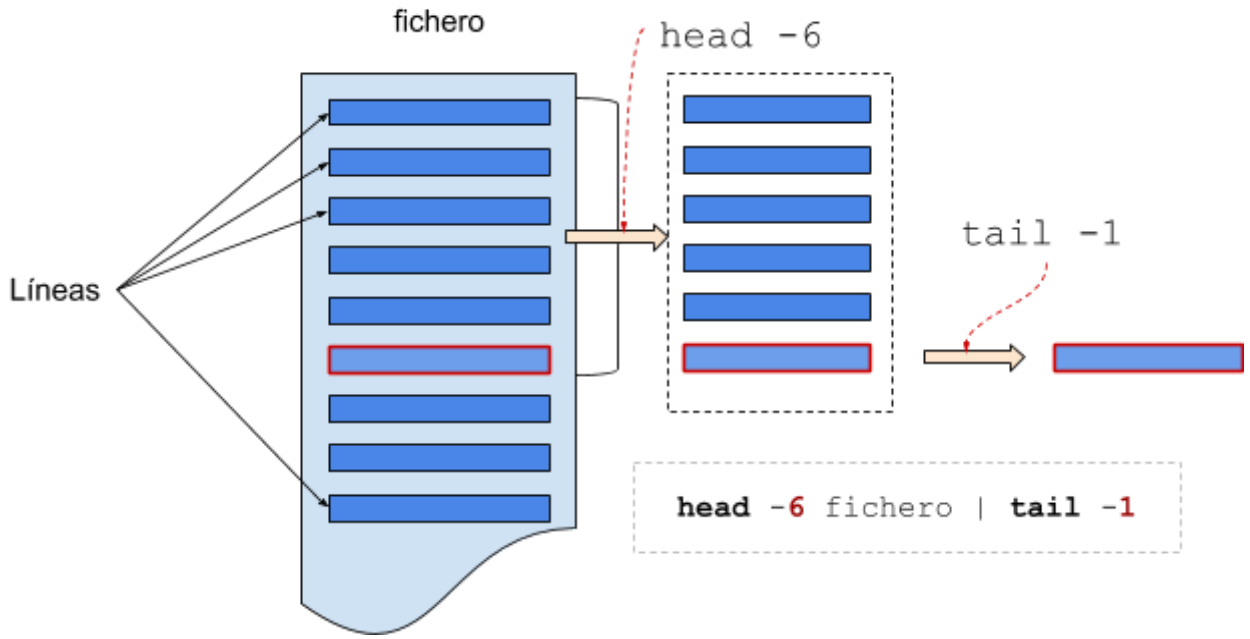
Existen varias alternativas para cumplir este objetivo, pero todas pasan por **implementar un bucle** que repita una y otra vez algún cálculo o procesamiento de una sola línea. Así, línea a línea se terminará tratando todo el fichero.



Una manera primitiva de conseguir este bucle es extraer cada vez una línea diferente. Para ello, usando el comando head y `tail`, y combinándolos con la concatenación de comandos, conseguiremos quedarnos con una sola línea a partir del fichero entero. Dado que podemos elegir qué línea es la que surge de combinar ambos comandos, podremos ir pasando por todas las líneas.

La idea es recortar el fichero quedándonos con una parte inicial (o del principio) del mismo. Seguidamente, de esta parte inicial, nos quedamos siempre con la última línea. Para seleccionar una parte inicial usamos el comando head y el resultado de ese comando es tratado por el comando "`tail -1`" que "recorta" y se queda con la última línea.

En el ejemplo siguiente, pretendemos quedarnos con la línea 6. Primero aplicamos el comando head con el parámetro -6 sobre el fichero y después pasamos el resultado por una tubería al comando "`tail -1`". El resultado combinado de ambos es la línea 6:



Sin embargo, este procedimiento es feo, poco elegante y muy ineficiente. Veamos la alternativa que usaremos habitualmente. Recapitulemos nuestras dos necesidades inmediatas_

- Leer línea a línea
- Leer de un fichero

El único comando que lee de forma nativa una línea es "`read`". Éste lee de la entrada estándar (teclado habitualmente) hasta que el usuario pulsa la tecla "`enter`", lo cual marca el final de línea. Casi lo tenemos, read lee una línea, pero no de un fichero, sino de la entrada estándar. … Y ahí es donde entra la redirección: con la redirección de entrada, podemos leer de un fichero lo que de normal sería leído del teclado. Así:

```
read unaLinea < fich_datos;
```

leería una sóla línea del fichero "fich_datos" y almacenaría ese dato en la variable "unaLinea"

Sería interesante repetir este procedimiento tantas veces como líneas tenga el fichero, lo ideal sería algo así:

```
while [ no se termine el fichero] ;  do
      read unaLinea < fich_datos
      echo $unaLinea
done
```

Sin embargo, el bucle anterior falla estrepitosamente, ya que la instrucción

```
read unaLinea < fich_datos
```

que se repite una y otra vez dentro del bucle, siempre lee la primera línea del fichero y por tanto no recorremos todas las líneas del mismo.

La forma adecuada de recorrer el fichero es realizar la redirección de forma que permanezca mientras dure el bucle y cada lectura que se haga, sea de una línea distinta del fichero (ya que la redirección se empieza y mantiene). Por ello, esta redirección afecta a todo el bucle while.

```
while read linea ;  do
      echo $linea
done < nombreFichero
```

Un ejemplo de procesado del fichero (para contar las líneas, en este caso) es:

```
numLinea=1
while read linea ;  do
      echo "mostrando la línea" $numLinea
      echo $linea
      numLinea=$[ numLinea + 1 ]
done < nombreFichero
```

La idea central del bucle anterior es el la pareja compuesta por el comando "read" y la redirección de entrada. El comando read está situado dentro de un bucle while, que es tratado como un bloque compuesto internamente de más comandos. La redirección situada

después de "done" afecta a todo el bucle y por tanto al comando read, el cual lee líneas del fichero en lugar de la entrada estándar. El comando read lee una línea cada vez, porque el carácter de final de línea o "retorno de carro" es equivalente a pulsar la tecla "intro" cuando lo usamos desde la entrada estándar. El comando read falla después de leer la última línea, cuando ya no hay más datos y el fichero ha llegado a su fin, esto hace que bucle finalice

El bucle anterior, sin embargo, presenta tres problemas:

- El comando read elimina espacios iniciales y finales de cada línea. Quizá haya que mantenerlos en la cadena que almacena la variable.
- Si la última línea no tiene un retorno de carro al final de ella, el comando falla en esa última línea y no se procesa.
- Como todo el bloque while está afectado por la redirección, ningún otro comando que lea de la entrada estándar podrá hacerlo. En su lugar leerá del fichero

El siguiente bucle soluciona todos estos problemas

```
nombreFichero=$1
numLinea=1
while IFS='' read -r linea || [[ -n "$linea" ]] ;  do
#while read  linea  ;  do
      echo "mostrando la línea" $numLinea
      echo "$linea"
      numLinea=$[ numLinea + 1 ]
      read -n 1 -p "pulsa para seguir" <&1
done < $nombreFichero
```

- `IFS` establece el separador de campos al caracter vacío, de forma que el espacio no se toma como separador y sí como un caracter más.
- La última línea del fichero de datos, si no tiene retorno de carro es leída por read y asignada a la variable "linea", sin embargo, el comando read devuelve una condición de error (porque se termina el fichero de repente). Esa última vez la comprobación adicional `[[ -n "$linea" ]]` logra hacer entrar al bucle una vez más (pese a la condición de retorno de read), ya que `"$linea"` tiene valor y con eso es suficiente. La próxima vez el fichero estará totalmente leído per además, `$linea` ya no tendrá valor.
- Observa cómo en mitad del bucle hay un read más, que pretende leer de la entrada estándar una pulsación de una tecla. Este read está redirigido desde la entrada estándar, de forma que se "escapa" a la redirección que afecta a todos los demás comandos del while. Se ha puesto aquí únicamente como demostración más completa, pero nada tiene que ver con el problema tratado.

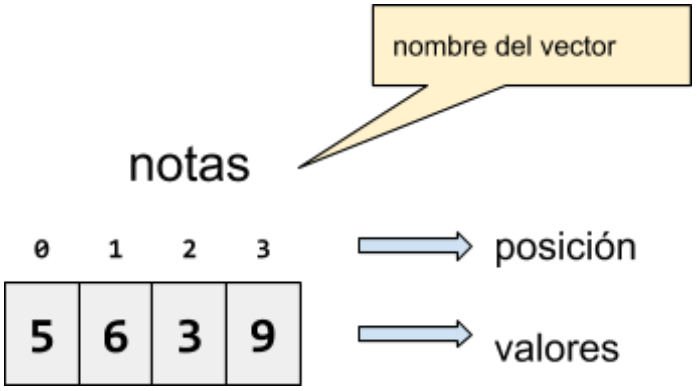
10 Arrays y vectores

Un array o vector, es una estructura de datos que podríamos imaginar como una variable (con un nombre -como cualquier variable-) que contiene varios valores. Cada valor está identificado, dentro de la variable con un número que indica la posición del valor dentro del vector (similar a cómo identificamos las casas de una calle con un número)

10.1 Declaración y asignación de valores

Directamente se utilizan, no hay que declarar nada, como con las variables normales. No estamos obligados a utilizar índices consecutivos. En el siguiente ejemplo se declara un array y se le asigna valores a cuatro elementos

```
ejercicio$ notas[0]=5
ejercicio$ notas[1]=6
ejercicio$ notas[5]=3
ejercicio$ notas[3]=9
```



Es posible inicializar todos los elementos de un array a partir de una lista:

```
ejercicio$ notas=( 5 6 3 9 )
ejercicio$ echo notas[2]
3
ejercicio$
```

Añadir elementos a un vector

Añadir un elemento al final de un vector es una operación habitual en los programas. En bash, se puede hacer de la siguiente manera

```
ejercicio$ notas+=(8)
ejercicio$ echo ${notas[4]}
```

```
    8
    ejercicio$
```

Nota: Si en la primera línea se omiten los paréntesis, bash entiende que **notas** es una variable normal y realiza una concatenación de caracteres tomando **notas** como una línea

Esta misma técnica se puede usar para añadir varios elementos:

```
    ejercicio$ notas=( 5 6 3 9 )
    ejercicio$ notas+=( 2 5 3 8 )
    ejercicio$ echo ${notas[6]}
    3
    ejercicio$
```

10.2 Acceso a la información de un vector.

La información del vector comprende todos los valores o elementos almacenados dentro de él. Nuestra primera necesidad es poder acceder a sólo uno de los elementos, tanto para leer su valor como para cambiarlo.

Para acceder a los elementos del vector, hay que referenciar dos elementos (sic). Por una parte el nombre del array y de otra parte el índice o posición. Dado que el índice, seguramente estará en una variable, esto se traduce en que hay que usar dos variables para acceder a un elemento del array

```
ejercicio$ echo $notas[0] <<-poco recomendable
5
ejercicio$ echo ${notas[0]}  <<-- preferible
5
ejercicio$ indice=2
ejercicio$ echo ${notas[${indice}]}  <<-- "{}" necesario
3
ejercicio$ indice=3
ejercicio$ notas[${indice}]= 8  <<-- "{}" necesario
ejercicio$ echo ${notas[${indice}]}  <<-- "{}" necesario
8
```

Del vector se pueden obtener más datos:

- El número de elementos que contiene:
  `echo ${#notas[@]}`
- El array entero:
  `echo ${notas[@]}`

Se puede copiar un vector a otro

```
vect2=(${array[@]})
```

10.3 Pasar un vector a una función.

En ocasiones se debe pasar un vector a una función. Esto plantea problemas, porque a una función sólo se le pasan argumentos como una lista de parámetros. La manera más directa y fácil es transformar el vector en una lista de valores y pasarlos a la función usando `${array[@]}`

Por ejemplo, supongamos el siguiente vector

```
edades=(  10 15 25 23 14 24 12 )
```

Se quiere sacar la media usando una función "`calculaMedia`". Ésta, requiere que se pase por argumentos los números a tomar:

```
function calculaMedia() {
      suma=0
      cuantos=0;
      for valor in $@ ; do
            suma=$(( $suma + $valor ))
            cuantos=$[ $cuantos + 1 ]
      done
      let "media =  suma / cuantos"
      echo $media
}
```

La llamada quedaría

```
media=$( calcularMedia ${edades[@]} )
```

http://tldp.org/LDP/abs/html/arrays.html
http://tldp.org/LDP/Bash-Beginners-Guide/html/sect_10_02.html

Sin embargo, si pese a todo se quiere pasar un vector y manipular un vector dentro de la función , se puede transformar la lista de parámetros rápidamente en un vector

```
function calculaMedia() {
      numeros=("$@")
      suma=0
      i=0;
      while [ $i -lt ${#numeros[@]}] ; do
            suma=$(( $suma + ${numeros[$i]} ))
      done;
```

```
        cuantos=${#numeros[@]}
        let "media =  suma / cuantos"
        echo $media"
    }
```

10.4 Cargar un fichero, línea a línea, en un vector

El siguiente código es (casi) capaz de leer un fichero línea a línea y guardar cada línea en un elemento consecutivo de un vector.

```
numLinea=0
while read linea ;  do
            vector[$numLinea]=$linea
          let numLinea++
done < nombreFichero
```

Extender un vector

Before ending I want to point out another feature that I just recently discovered about bash arrays: the ability to extend them with the += operator. This is actually the thing that lead me to the man page which then allowed me to discover the associative array feature. This is not a new feature, just new to me:

```
aa=(hello world)
aa+=(b c d)
```

After the += assignment the array will now contain 5 items, the values after the += having been appended to the end of the array. This also works with associative arrays.

```
aa=([hello]=world)
aa+=([b]=c)           # aa now contains 2 items
```

Note also that the += operator also works with regular variables and appends to the end of the current value.

```
aa="hello"
aa+=" world"          # aa is now "hello world"
```

10.5 Convertir una cadena en un vector

```
String="This is a string of words."

read -r -a Words <<< "$String"
#  The -a option to "read"
```

```
#+ assigns the resulting values to successive members of an array.

echo "First word in String is:    ${Words[0]}"    # This
echo "Second word in String is:   ${Words[1]}"    # is
echo "Third word in String is:    ${Words[2]}"    # a
echo "Fourth word in String is:   ${Words[3]}"    # string
echo "Fifth word in String is:    ${Words[4]}"    # of
echo "Sixth word in String is:    ${Words[5]}"    # words.
echo "Seventh word in String is:  ${Words[6]}"    # (null)
                                                  # Past end of
$String.
```

10.6 Arrays asociativos (No en primero de DAM)

En la versión 4 de bash, se permite usar nativamente Arrays Asociativos (algunas veces llamados mapas). Los arrays asociativos son similares a los vectores, pero la indexación (la posición del elemento) se puede realizar con cadenas. Es decir, mientras que para acceder un elemento de vector se debe realizar algo así:

```
pos=4
notas[$pos]=8;
echo ${notas[${pos}]}
```

Con los arrays asociativos podemos indexar con una cadena y asignar valores a elementos del array de la siguiente manera:

```
#asignación clave explícita:
apellidos[Juan]=Albert
#asignación usando una variable como clave
     nombre="Nacho"
apellidos[${nombre}]=Martínez
```

Y recuperar los valores, usando una llave de forma similar a los vectores

```
nombre="Nacho"
echo "El apellido de nacho es ${apellidos[${nombre}]}"
echo "El apellido de Juan es ${apellidos[Juan]}
```

Para poder utilizarlos, hay que declarar el array antes. Esto es así porque es una característica reciente de bash y ha de "activarse"

```
declare -A aa
aa[hello]=world
aa[ab]=cd
```

Es la opción `-A` la que indica que `aa` es un array asociativo. Las asignaciones se realizan colocando la *llave* entre corchetes. Se pueden asignar varios elementos en una sola línea:

```
declare -A aa
aa=([hello]=world [ab]=cd)
```

You can also use keys that contain spaces or other "strange" characters:

```
aa["hello world"]="from bash"
```

10.6.1 Recorrido de un array asociativo.

Respecto de los arrays o vectores normales, los arrays asociativos plantean un problema importante. ¿Cómo se recorren? ¿Cómo se accede a todos sus valores? Con un vector normal es más o menos fácil, puedes usar un contador que se inicialice a 0 (o al primer índice del vector) y se va incrementando de uno en uno para acceder a cada elemento consecutivo del vector.

Con un array asociativo, no se conocen a priori las claves y por tanto no se puede establecer un bucle controlado. La manera de acceder es la de generar una lista de valores a partir del array, y recorrer la lista. Suponga un array asociativo llamado "miArray"

```
for variable in "${!miArray[@]}"
do
      echo "$k"
done
```

The keys are accessed using an exclamation point: `${!array[@]}`, the ***values*** are accessed using `${array[@]}`.

You can iterate over the key/value pairs like this:

```
for i in "${!array[@]}"
do
  echo "key  : $i"
  echo "value: ${array[$i]}"
done
```

11 Manipulación de texto

Vamos a estudiar algunos comandos que resultan necesarios para realizar acciones básicas sobre el contenido de los ficheros.

11.1 Longitud de un texto

Para obtener el tamaño de una cadena almacenada en una variable se recurre a esta construcción especial que se puede ver en el siguiente ejemplo

```
$ var=”HOLA”
$ echo ${#var}
4
```

11.2 Encontrar subcadenas

```
string='My string';

if [[ $string =~ .*My.* ]]
then
   echo "It's there!"
fi
```

11.3 Extraer subcadenas:

11.4 Cortar líneas con el comando cut

El comando cut toma una línea cada vez y devuelve un trozo de dicha línea. El siguiente ejemplo muestra los caracteres del quinto al décimo de la línea pasada

```
$ echo “Mi carro me lo robaron” | cut -c 5-10
arro m
$
```

cut puede funcionar en dos modos:

- **selección delimitada a columnas** Cada columna empieza en una posición y tiene un rango (ejemplo:"desde el carácter quinto hasta el octavo 5-8 o desde el 5 hasta el décimo del ejemplo anterior).

```
     $ ls -l | tail -2
-rwxr-xr-x 1 lliurex lliurex  473 feb  4 13:36 sumaN.sh
-rwxr-xr-x 1 lliurex lliurex  399 feb 19 11:56 torneo.sh
     $ ls -l | tail -2 | cut -c2-11,50-
rwxr-xr-x sumaN.sh
rwxr-xr-x torneo.sh
$... PONER Más opciones
```

En este ejemplo, se han "cortado" desde el caracter segundo hasta el 11 y en la misma línea desde el caracter 50 hasta el final de la línea (la coma "," separa dos rangos)

Otro ejemplo: Un fichero tiene nombres de usuario y contraseñas con el siguiente formato:

USUARIO(10)	CONTRASEÑA(20)
juan	santoysenya

El comando cut que extrae el usuario es "cut -c1-10" y el que extrae la contraseña "cut -c11-

- **selección delimitada por separadores** Mediante caracteres como el espacio en blanco, los dos puntos, la coma, etc se delimitan y numeran columnas. Después se puede seleccionar con un número la columna deseada.

```
$ echo "Mi carro me lo robaron" | cut -d" " -f2

carro

$ POINER un EJEMPLO A PARTIR DE ls -l
```

En este modo, -d define el delimitador y -f2 elige el segundo campo. .

11.5 Concatenar subcadenas:

```
nombreCompleto="${nombre} ${apellido1} ${apellido2}"
nombre+=$apellido
```

11.6 Pasar una cadena a minúsculas

```
nombre=$( echo $nombre | tr [a-z] [A-Z] )
```

12 xargs, find y grep .

Vamos a estudiar tres comandos bastante poderosos. Los estudiaremos juntos porque al final se van a combinar en un ejercicio que demuestra su capacidad por separado y de forma combinada.

12.1 Búsqueda de ficheros con find

Se utiliza este comando para buscar archivos dentro de una jerarquía de directorios. La búsqueda se puede realizar mediante varios criterios. La sintaxis de este comando es:

```
find [ruta...] [expresion]
```

La **expresión** se conforma de opciones, pruebas y acciones. En este documento no las enumeraremos todas, sino las expresiones que son más cotidianas. Algunos de los criterios de búsqueda que se pueden utilizar son:

```
# Busca en el directorio PATH un fichero con nombre ARCHIVO
$ find PATH -name ARCHIVO
# Ejemploi
$ find /home/pepe -name Factura.ods

# Igual que antes pero además se exigen ciertos permisos MODO
$ find PATH -name ARCHIVO -perm MODO
```

12.1.1 Ejemplos

Ejemplo: Carlos recuerda haber almacenado en su directorio personal una foto de su familia cuando estaban de vacaciones, y lo único que recuerda es que estaba en formato PNG, para intentar localizar dicha foto, usa el comando **find** de la siguiente forma:

```
$ find /home/carlos -name "*.png"
```

Ejemplo: El administrador de un servidor de Internet necesita realizar una auditoría de seguridad, para ello una de las pruebas que se necesita realizar es identificar aquellos archivos o directorios que poseen permisos de escritura para cualquier usuario, esto lo puede hacer como sigue:

```
$ find / -perm 777
```

Ejemplo: Este comando listará también los enlaces simbólicos, que aunque en el listado aparecen con todos los permisos activados, no significa que cualquier usuario los pueda modificar. Para evitar entonces este inconveniente, se puede ejecutar el comando de esta manera:

```
$ find / -perm 777 -follow
```

La opción `-follow` instruye a **`find`** para que en lugar de hacer la prueba con los enlaces simbólicos, la haga con los archivos apuntados por estos enlaces.

12.1.2 Salida.

En caso de encontrar algún fichero especificado en el apartado de expresiones, el comando find emitirá como resultado una línea por cada fichero encontrado. En dicha línea aparece la ruta relativa o absoluta del fichero. Veámos un ejemplo:

```
$ find ./facturas -name "*2009*.fac"
./facturas/nuevas/camion-2009.pagada.fac
./facturas/nuevas/maquina-1-2009.debida.fac
./facturas/viejas/mesa_mod_120093.pagada.fac
```

La salida de este comando puede ser utilizada en la formación de otro comando. De esta forma podríamos efectuar alguna operación sobre un conjunto de ficheros repartidos por un árbol de directorios.

```
$ rm `find . -name "*.tmp"`
```

Hasta ahora sólo podíamos actuar sobre un conjunto de ficheros reunidos en un único directorio, pero con find podemos recopilar rutas de archivos de una jerarquía entera de directorios.

12.2 Ejecución de comandos con xargs

`xargs` es un comando muy sencillo en principio y muy poderoso. xargs toma dos elementos:

- Un parámetro que será un comando o ejecutable cualquiera
- Unos datos de la entrada estándar (que normalmente está conectada a la salida de otro comando)

`xargs` toma esta entrada y ejecuta el comando pasado como argumento pasándole a éste como argumentos, lo que ha recibido por la entrada estándar. Podríamos pensar que xargs prepara una línea de comandos complicadilla y después la ejecuta. Aunque para empezar, veamos este ejemplo simple:

```
$ echo -l | xargs ls
#equivale a:
$ ls -l
```

También se puede ejecutar interactivamente. Pruebe a ejecutar el siguiente comando

```
$ xargs ls
```

Introduzca argumentos usuales para el comando ls con el teclado interactivamente. Compruebe el resultado

12.2.1 Sintaxis

`commando_previo ..... | xarg [opciones] orden`

Opciones:

- `-0` ó `–null`, se utiliza en el caso de que el nombre de los ficheros devueltos terminen con un carácter nulo, en lugar de con un espacio en blanco.

  Esta opción deberá usarse cuando utilicemos la acción –print0 con find.

- `-t` ó `–verbose`: muestra por pantalla la acción a ejecutar antes de proceder con la orden.
- `-p`: pregunta al usuario si se debe ejecutar la acción.
- `-i`: recoge el valor devuelto por find y lo utilizamos mediante los corchetes "`{}`" (similar a la sintaxis de *exec*). Muy útil para comandos del tipo mv por ejemplo. Lo vemos:

  ```
  $ find . -name '1.log' | xargs -I {} -t mv {} 1.log.old
  mv ./1.log 1.log.old
  ```

La cadena `{}` se reemplaza por el resultado devuelto por `find`, en este caso `1.log`..

- `--help`: muestra un resumen de las opciones de `xargs`.

12.2.2 Uso habitual de xargs

El comando xargs se utiliza comúnmente concatenado después de otro comando para conseguir un efecto distinto al habitual cuando se utiliza concatenación. Vamos a comparar dos casos distintos y ya estudiados frente a xargs.

12.2.3 Sin xargs y con xargs

Es habitual redirigir la salida de un comando a la **entrada** (o los datos) de otro:

```
#concatena los ficheros ".txt" en orden de creación
$ ls -t *.txt| cat
#cuenta el número de palabras total de, como máximo, cinco ficheros .txt
$ ls *.txt | head -5 | wc
```

También es posible ejecutar un comando y utilizar su resultado dentro de la línea de argumentos de otro

```
#borra los ficheros ".txt"
$ rm `ls *.txt`   #¡Vaya chorrada!
#cuenta el número de palabras total de, como máximo, cinco ficheros .txt
```

```
$ wc < `ls *.txt | head -5`
```

Pero al utilizar `xargs` buscamos que la salida de un comando sean los **argumentos** de otro.

```
# borra los ficheros ".txt"
$ ls *.txt | xargs rm

#cuenta el número de palabras total de, como máximo, cinco ficheros .txt que cuelguen en cualquier subdirectorio del actual
$ find . -name "*,txt" | head -5 | xargs cat | wc

# borra sin preguntar todos los ficheros y directorios cuyo nombre contenga el día de hoy. opción date desconocida
$ find . -name "*`date -X?X?`*" | xargs rm -fr
```

Suponga un fichero `'ficheros_facturas'` cuyo contenido es una relación de ficheros que contienen facturas. Si deseo concatenar el contenido de todas las facturas podré hacerlo simplemente con:

```
$ xargs cat < fichero_facturas > resultado_concatenado
```

Se puede usar `xargs` para renombrar un conjunto de ficheros.

```
#renombra un conjunto de imágenes anteponiendo una fecha al nombre
$ ls *.jpg | xargs -I {} mv {} 14082009-{}
foto1.jpg
foto2.jpg
mifoto.jpg
```

Esto funciona porque `{}` es un comodín que se expande con el nombre que recibe `xargs`, de forma que se puede usar más de una vez en una línea de comandos. Observe también que se ejecuta un comando `mv` por cada línea de entrada, mientras que en la mayoría de ejemplos anteriores se ejecutaba un sólo comando tomando como argumentos todas las líneas y palabras recibidas.

`-I` implica `-n1`

12.2.4 Another example:

`xargs` is particularly useful when you have a list of filepaths on stdin and want to do something with them. For example:

```
$ git ls-files "*.tex" | xargs -n 1 sed -i "s/color/colour/g"
```

Let's examine this step by step:

```
$ git ls-files "*.tex"
tex/ch1/intro.tex
```

```
tex/ch1/motivation.tex
....
```

In other words, our input is a list of paths that we want to do something to.

To find out what xargs does with these paths, a nice trick is to add echo before your command, like so:

```
$ git ls-files "*.tex" | xargs -n 1 echo sed -i "s/color/colour/g"

sed -i "s/color/colour/g" tex/ch1/intro.tex
sed -i "s/color/colour/g" tex/ch1/motivation.tex
....
```

The -n 1 argument will make xargs turn each line into a command of its own. The `sed -i "s/color/colour/g"` command will replace all occurrences of color with colour for the specified file.

Note that this only works if you don't have any spaces in your paths. If you do, you should use null terminated paths as input to xargs by passing the -0 flag. An example usage would be:

```
$ git ls-files -z "*.tex" | xargs -0 -n 1 sed -i
"s/color/colour/g"
```

Which does the same as what we described above, but also works if one of the paths has a space in it.

This works with any command that produces filenames as output such as find or locate. If you do happen to use it in a git repository with a lot of files though, it might be more efficient to use it with git grep -l instead of git ls-files, like so:

```
$ git grep -l "color" "*.tex" | xargs -n 1 sed -i
"s/color/colour/g"
```

The `git grep -l "color" "*.tex"` command will give a list of `"*.tex"` files containing the phrase `"color"`.

12.3 xargs and find

`find` y `xargs` van muy bien juntos. Se emplean, por ejemplo, para buscar ficheros que contengan alguna palabra en su interior. Estos ficheros pueden estar en varios subdirectorios colgando de uno particular y se puede seleccionar alguna característica adicional para afinar la búsqueda. Por ejemplo buscar en los archivos con extensión "`.factura`" creados esta semana aquellos que contengan la palabra "`impagado`"

Tradicionalmente, la ventaja de utilizar xargs era la de simplificar la invocación de algunos comandos que resultaban tener una línea de argumentos demasiado extensa. En ocasiones, ocurría un error por este motivo. Este comando

```
rm `find tmp -maxdepth 1 -name '*.mp3'`
```

Intenta eliminar todos los ficheros "`.mp3`" existentes en el directorio tmp e ignora todos los subdirectorios. Este otro comando es equivalente:

```
find tmp -maxdepth 1 -name '*.mp3' -maxdepth 1 | xargs rm
```

Pero evita el problema de anidar comandos uno dentro de la línea de argumentos de otro

Se puede limitar el número de argumentos que recibe un comando y lanzar varios a la vez para procesarlos todos con la opción -n .

```
find tmp -maxdepth 1 -name '*.mp3' -maxdepth 1 | xargs -n1 rm
```

pasará un argumento cada vez al comando rm. This is also useful if you're using the -p option as you can confirm one file at a time rather than all at once.

Por ejemplo, agrupar de tres en tres los ficheros "`.txt`"

```
find . -name "*,txt" | xargs -n3 cat
```

12.3.1 ¿Espacios en blancos en los argumentos pasados?

Filenames containing whitespace can also cause problems; xargs and find can deal with this, using GNU extensions to both to break on the null character rather than on whitespace:

```
find tmp -maxdepth 1 -name *.mp3 -print0 | xargs -0 rm
```

You must use these options either on both find and xargs or on neither, or you'll get odd results.

Another common use of xargs with find is to combine it with grep. For example,

```
find . -name '*.pl' | xargs grep -L '^use strict'
```

will search all the `*.pl` files in the current directory and subdirectories, and print the names of any that don't have a line starting with 'use strict'. Enforce good practice in your scripting!

12.4 Ejemplos combinados

- Renaming within the name:
  `ls -1 *old* | awk '{print "mv "$1" "$1}' | sed s/old/new/2 | sh`
  (although in some cases it will fail, as in file_old_and_old)

- remove only files:
  ```
  ls -l * | grep -v drwx | awk '{print "rm "$9}' | sh
  ```
  or with awk alone:
  ```
  ls -l|awk '$1!~/^drwx/{print $9}'|xargs rm
  ```
  Be careful when trying this out in your home directory. We remove files!
- remove only directories
  ```
  ls -l | grep '^d' | awk '{print "rm -r "$9}' | sh
  ```
  or
  ```
  ls -p | grep /$ | wk '{print "rm -r "$1}'
  ```
  or with awk alone:
  ```
  ls -l|awk '$1~/^d.*x/{print $9}'|xargs rm -r
  ```

- Be careful when trying this out in your home directory. We remove things!

```
kill `ps auxww | grep firefox | egrep -v grep | awk '{print $2}'`
```

killing processes by name (in this example we kill the process called firefox):

or with awk alone:

```
ps auxww | awk '$0~/firefox/&&$0!~/awk/{print $2}' |xargs kill
```

It has to be adjusted to fit the ps command on whatever unix system you are on. Basically it is: "If the process is called netscape and it is not called 'grep netscape' (or awk) then print the pid"Anexos .

13 Agrupando comandos. Caracteres de ambiente “(“ y “)”

Bash permite de dos maneras distintas, que una lista de comandos se ejecute como una unidad. Cuando los comandos se agrupan, la redirección se aplica a la lista entera de comandos.

```
()

( list )
```

Una lista de comandos entre paréntesis provoca que se cree un subshell y un nuevo entorno para ejecutar esta lista. Como no se ejecutan en el entorno del script, cualquier asignación de variables, se pierde una vez finalizada la ejecución de la lista de comandos.

```
{}

{ list; }
```

Una lista de comandos entre llaves provoca que la lista de comandos se ejecute en el ambiente y shell actuales. No se crea otro subshell. Se requiere un punto y coma al final del

último comando para indicar claramente que la llave que cierra no es parte del comando, sino el final de la lista

In addition to the creation of a subshell, there is a subtle difference between these two constructs due to historical reasons. The braces are reserved words, so they must be separated from the list by blanks or other shell metacharacters. The parentheses are operators, and are recognized as separate tokens by the shell even if they are not separated from the list by whitespace.

The exit status of both of these constructs is the exit status of *list*.

14 Especificar opciones generales de ejecución con set

ver "man set"