Capítulo 6

Programación en el *shell* bash Parte II

6.10. ¿Qué es un *script*?

El *shell* bash, al igual que otros, proporciona un lenguaje de programación. Los programas que se realizan utilizando el lenguaje del *shell* se denominan *shell scripts*, o simplemente *scripts* (aunque a veces se puede encontrar la traducción literal "guiones").

Un *script* es un fichero de texto en el que se introducen órdenes que son interpretadas por el *shell* y por el sistema operativo. Se suelen usar para automatizar procesos de administración de sistemas, tales como realización de copias de seguridad, mantenimiento de servidores, migraciones, etc.

El fichero del *script* puede crearse con cualquier editor de textos, escribiendo una orden en cada línea. Una vez creado, podemos ejecutarlo de cualquiera de las dos formas siguientes:

1. `$ bash` *script* [*parámetro* ...]
2. `$` *script* [*parámetro* ...]

En el primer caso, no es necesario activar previamente el permiso de ejecución del fichero. En el segundo caso, sí lo tenemos que activar. Además la mayoría de sistemas LINUX no incorporan el directorio punto (`.`) en el PATH, por lo que es necesario indicar la dirección absoluta (o relativa comenzando por punto) del fichero para ejecutar el *script*. Por otro lado, hay que tener en cuenta que para poder ejecutar un *script*, además del permiso de ejecución,

se necesita el permiso de lectura (porque es un lenguaje interpretado). Los permisos SUID y SGID no tienen efecto sobre los ellos.

La forma en que se ejecuta un *script* es similar a cualquier otro programa: el shell ejecutará la primera orden. Cuando termine pasará a ejecutar la segunda y así sucesivamente. Esta sucesión se rompe con las órdenes de control de flujo (`if`, `while`, etc) que veremos más adelante.

A la hora de ejecutar un *script* podemos pasarle argumentos que van a ser utilizados durante su ejecución, son los denominados **parámetros posicionales**.

Si ejecutamos un *script* de cualquiera de las dos formas anteriores, se estará creando un proceso hijo y por tanto se estará ejecutando en un ambiente separado. Esto significa que las variables del ambiente actual no están disponibles para los *scripts* a menos que se hayan exportado explícitamente, asimismo las variables definidas dentro del *script* no serán heredadas por el *shell* padre.

Es bastante usual que el *script* dé errores al ejecutarlo; para averiguar dónde están es conveniente depurarlo, y para ello lo ejecutaremos de la siguiente forma:

`$ bash -xv` *script* [*parámetro* ...]

Cuando se llama al *shell* bash con la opción `-x`, se van mostrando las órdenes que se ejecutan junto a sus argumentos. La opción `-v` muestra las líneas de órdenes leídas por el *shell* bash. De esta forma, podemos ver lo que va interpretando el *shell*, así como los resultados tras ejecutar cada una de las órdenes del *script*.

6.11. Parámetros en el *shell* bash

Se denominan parámetros todas aquellas entidades que almacenan valores en el *shell* bash. Podemos distinguir diversos tipos de parámetros:

Variables Los parámetros que tienen un nombre se denominan variables. Éstas pueden tener atributos que son establecidos por el usuario.

Parámetros posicionales Estos parámetros se denotan por uno o más dígitos. Mantienen los argumentos que se le pasan al *script* o a una función.

Parámetros especiales El *shell* establece automáticamente su valor y se denotan por los caracteres

`* @ # ? - $ !`

6.11.1. Expansión básica de parámetros

Expansión de parámetros es el término que se usa para designar a la capacidad de acceder y manipular los valores de las variables y parámetros. La expansión básica se hace precediendo el nombre de la variable o parámetro con el carácter `$`. En algunos casos es necesario usar la forma `${`*parámetro*`}`. ¿Cuándo es necesario usar esta segunda forma?

- Cuando tenemos más de nueve parámetros posicionales, para acceder al décimo y siguientes, tendremos que encerrarlos entre llaves. Ejemplo: `echo ${12}`.
- Cuando queremos extraer el valor de un elemento de una variable de tipo vector. Esto se verá más adelante.
- Cuando queremos concatenar el valor de una variable a una cadena.

Se pueden usar otros tipos de expansión para devolver la longitud del parámetro, porciones del mismo, etc. Esto se verá más adelante.

6.11.2. Parámetros posicionales y especiales

Los parámetros posicionales sirven para almacenar los argumentos que se le pasan al *script* cuando se ejecuta. Los parámetros posicionales tienen los nombres `1`, `2`, `3`, etc. Esto significa que sus valores son denotados como `$1`, `$2`, `$3`, etc. Hay también un parámetro posicional `0`, cuyo valor es el nombre del *script*.

El parámetro `#` contiene el número de los posicionales (como una cadena de caracteres). Los parámetros especiales `*` y `@` contienen todos los parámetros posicionales, excepto el `0`. ¿Cuál es la diferencia entre ellos? Más adelante la contaremos.

Otros parámetros especiales son `?` y `$`. El parámetro `?` contiene el status de salida de la última orden ejecutada, mientras que `$` contiene el identificador del proceso (PID) correspondiente al *shell* actual.

Todos estos parámetros son de sólo lectura, es decir, no se les pueden asignar nuevos valores dentro de un *script*.

Ejemplo:

Escriba el *script* 6.1 en un fichero. Utilice la orden `chmod` para dotarle de los permisos adecuados. Ejecute el *script* pasándole tres parámetros posicionales cualesquiera.

***Script* 6.1** Parámetros posicionales

```
#!/bin/bash
# ejemplo1: Ejemplo de manejo de parámetros posicionales

echo "El nombre de este script es $0"
echo "El primer parámetro posicional es $1"
echo "El segundo parámetro posicional es $2"
echo "El tercer parámetro posicional es $3"
echo "El numero de parámetros pasados es $#"
echo "Todos los argumentos pasados al script son: $*"
```

Observaciones:

- Se ha puesto como primera línea del *script* la siguiente:

  `#!/bin/bash`

  Esto se hace para indicar qué *shell* es el que va a interpretar nuestro *script*. Si no se pone esa línea, el *shell* que se use depende del que estemos usando en interactivo, y del sistema. Los caracteres `#!` deben ser los dos primeros del fichero.
- La línea que comienza por `#` es una línea de comentario.
- La orden `echo` muestra en la salida estándar los parámetros que se le pasan. Más adelante la estudiaremos en mayor profundidad.

Desplazamiento de los parámetros posicionales

Los parámetros posicionales pueden ser reasignados con la orden `shift`. Ésta desplaza el valor de los parámetros posicionales el número de posiciones que se le indica mediante el parámetro n. Si no se le indica ningún parámetro se desplaza una posición.

Formato:

`shift` $[n]$

Ejemplo:

El *script* 6.2 es una modificación del 6.1. Ejecútelo pasándole los mismos argumentos que antes.

Sustituya en el *script* 6.2, la línea que contiene la orden `shift` por `shift 2`, y vuelva a ejecutarlo pasándole los mismos argumentos.

***Script* 6.2** Desplazamiento de parámetros posicionales

```
#!/bin/bash

# ejemplo3: Ejemplo de manejo de los parámetros
# posicionales junto con la orden shift.

echo "El nombre de este script es $0"
echo "El primer parámetro posicional es $1"
echo "El segundo parámetro posicional es $2"
echo "El tercer parámetro posicional es $3"
echo "El numero de parámetros pasados es $#"
echo "Todos los argumentos pasados al script son: $*"
shift
echo "El número de argumentos que quedan es $#"
echo "Los argumentos que quedan son $*"
echo "Arg1=$1, Arg2=$2, Arg3=$3"
```

6.11.3. Variables

En la elaboración de un *script* el programador puede hacer uso de las variables que considere necesarias. Los nombres de éstas pueden contener cualquier combinación de:

- Letras (mayúsculas o minúsculas).
- Dígitos.
- Subrayados (_).

Una limitación que se impone es que no pueden comenzar por un dígito. Sin embargo no se impone ninguna limitación a la longitud del nombre de la variable. Hay que tener en cuenta que los caracteres en mayúsculas y en minúsculas son distintos en los nombres de las variables. Así, `LORO`, `Loro` y `loro` son tres variables distintas.

El *shell* bash reconoce tres tipos de datos:

Cadenas Por omisión, todas las variables son de tipo cadena, a no ser que se indique otra cosa. No es necesario especificar su longitud cuando se declaran.

Enteros La declaración de una variable como entero permite realizar operaciones aritméticas.

Vectores Por omisión, los elementos del vector son declarados de tipo cadena. Cada elemento se referencia mediante un índice, correspondiéndole al primero el índice cero. Los vectores no tienen ningún límite en su tamaño, y sus elementos no tienen que estar de forma contigua.

Se pueden especificar atributos para todos los tipos de variables.

Asignación de valores

Para asignar un valor a una variable podemos hacerlo de varias formas:

1. *variable*=*valor*
2. `declare` [±*atributo*] *variable*=*valor*
3. `local` *variable*=*valor*

Si lo hacemos mediante la orden `declare` podremos además especificar al mismo tiempo atributos para las variables. La orden `local` sólo se puede utilizar dentro de una función, en este caso la variable será local a la función donde se encuentre. De esto hablaremos más adelante.

En el caso de una variable de tipo vector, cuando queremos asignarle un valor a un elemento hemos de indicar su índice correspondiente:

variable`[`*índice*`]=`*valor*

Si no se especifica *índice* se utilizará el empleado en la última asignación más uno, y si no se hubiera asignado ningún elemento se tomará el índice cero. Igualmente se pueden realizar asignaciones a distintos elementos. Los diferentes modos de asignar valores a este tipo de variable se pueden ver en el siguiente ejemplo.

Ejemplos:

1. `nombres[42]=Alicia`

`nombres[96]=Antonio`
Se asignan los valores `Alicia` *y* `Antonio` *a los elementos 42 y 96 del vector* `nombres`.

2. `asignaturas=([10]=SSOO [8]=MTPII [15]=BD)`
Asigna los valores `SSOO`, `MTPII` *y* `BD` *a los elementos 10, 8 y 15 del vector* `asignaturas`, *respectivamente.*
3. `profesores=([3]=Fernando Eugenio Antonia Juan)`
Asigna los valores `Fernando`, `Eugenio`, `Antonia` *y* `Juan`, *a los elementos 3, 4, 5 y 6 del vector* `profesores`, *respectivamente.*

El índice de un elemento de un vector debe ser entero, y puede ser el resultado de una operación aritmética. No obstante, si el índice es `*` o `@`, el resultado equivale a todos los elementos del vector, separados por espacio. Pero, si el vector está encerrado entre comillas dobles, el resultado es diferente, pues el `*` se expande a cada uno de los valores de cada elemento separados por el primer carácter de la variable IFS.

Ejemplo:

Ejecute el *script* 6.3 y observe las diferencias.

***Script* 6.3** Utilización de una variable vector

```
#!/bin/bash

declare -a dos=(pepe antonio juan)
IFS=:
echo ${dos[*]} con \*
echo ${dos[@]} con \@
echo "${dos[*]}" con \* y comilla
echo "${dos[@]}" con \@ y comilla
```

Una característica de este tipo de dato es que los elementos no tienen por qué estar en posiciones contiguas, y pueden existir elementos vacíos.

Atributos de las variables

Se pueden establecer atributos para las variables con la orden `declare`,

`declare` -*atributo variable*

En la tabla 6.1 se muestran los atributos que se pueden especificar para las variables.

Atributo	Significado
`i`	Tipo entero.
`r`	Sólo lectura.
`x`	Se exportará.
`a`	Será un vector de tipo cadena.

Cuadro 6.1: Atributos para las variables

Ejemplo:

```
$ declare -ai notas
```

La variable `notas` *se declara como un vector de enteros.*

Excepto en el caso del atributo de sólo lectura, los demás pueden ser eliminados con la orden:

`declare` +*atributo variable*

Otras operaciones con variables

- Se les puede asignar el valor de otra variable:

  Ejemplo:

  ```
  $ X=$HOME
  $ Y=$X
  $ echo $Y
  ```

- Se pueden concatenar.

  Ejemplos:

  1. Concatenación de dos variables

     ```
     $ X=hola
     $ Y=adios
     $ Z=$X$Y
     $ echo $Z
     ```

  2. Concatenación de una variable y una cadena

```
$ M=man
$ M=${M}zana
$ echo $M
```

Observe que hemos encerrado el nombre de la variable entre `{}`. ¿Qué obtendríamos si hubiéramos dado `M=$Mzana`?

- Se les puede asignar la salida de una orden. ¿De qué forma?

  variable=‘*orden*‘ (compatible con el *shell* de Bourne)

  o bien

  variable=`$(`*orden*`)`

  Ejemplo:

```
$ quien=$(who)
$ echo $quien
```

- Se pueden desestablecer:

  `unset` *variable*

  Ejemplo:

```
$ unset quien
$ echo $quien
```

- Se puede calcular su longitud.

  `${#`*variable*`}`

  Ejemplo:

```
$ X=abcd
$ echo ${#X}
4
```

6.12. Funciones

La noción de función en un *script* es similar a la que tienen otros lenguajes de programación: una serie de instrucciones que solucionan un problema concreto. Pueden recibir parámetros que modifiquen su comportamiento y devolver un valor al programa que las invocó.

Las funciones son ideales para organizar *scripts* largos en bloques de código modulares, que son más fáciles de desarrollar y mantener.

Para definir una función podemos usar una de las dos formas siguientes:

```
function nombre-función () {
  órdenes
}
```

o bien:

```
nombre-función () {
  órdenes
}
```

Cuando definimos una función, le decimos al *shell* que almacene su nombre y definición (es decir, las órdenes que contiene) en memoria. Si queremos ejecutar la función más tarde, sólo tendremos que teclear su nombre seguido de los argumentos que requiera, como si se tratara de un *script*.

Para pasar el control de una función al *script* que la ha llamado y pasarle un valor de salida se utiliza la orden `return`. La sintaxis de `return` es:

`return` [n]

donde n es el valor de retorno que se devuelve al *script* (este valor debe ser un número entero). Para poder utilizar este valor en el programa que llamó a la función se emplea el parámetro especial `?`.

6.13. Ámbito de los parámetros

Cualquier variable definida dentro de un *script* tiene ámbito global. Sin embargo, es posible hacer que una variable sea local a una función utilizando las órdenes `declare` o `local` dentro de su definición. Si éstas no se emplean también serán globales.

Al igual que un *script* utiliza los parámetros posicionales para manejar los argumentos que se le pasan, cada función también maneja los suyos propios mediante éstos. Es decir, los parámetros posicionales son locales al *script* y a cada una de las funciones. Igual ocurre con los parámetros especiales `@`, `*` y `#`.

***Script* 6.4** Manejo de parámetros posicionales y funciones

```
#!/bin/bash
# ejemplo2: Manejo de parámetros posicionales dentro y
# fuera de funciones.

function fun1(){
  echo "En la función: $1 $2"
  var1="dentro de la función"
}

var1="fuera de la función"
echo "El valor de var1 es: $var1"
echo $0: $1 $2
fun1 hola adios
echo "El valor de var1 es: $var1"
echo $0: $1 $2
```

Ejemplo:

Si ejecutamos el *script* 6.4 como se indica, los resultados serán:

```
$ ejemplo2 rosa clavel
El valor de var1 es: fuera de la función
ejemplo2: rosa clavel
En la función: hola adios
El valor de de var1 es: dentro de la función
ejemplo2: rosa clavel
```

Como se puede observar, la función fun1 cambia el valor de la variable var1, y este cambio es conocido fuera de la función, mientras que `$1` *y* `$2` *tienen valores diferentes en la función y en el script. Modifique el ejemplo anterior definiendo* var1 *como una variable local a fun1.*

Cuando hablamos de los parámetros posicionales hicimos referencia a algunos parámetros especiales como `*` y `@`. Vamos ahora a explicar la diferencia

entre `$*` y `$@`. `"$*"` es una cadena simple que consta de todos los parámetros posicionales, separados por el primer carácter de la variable de ambiente IFS, que por omisión es el carácter espacio, TAB o `Nueva-Línea`. `"$@"` es igual a `"$1" "$2" ...` `"$`N`"`, donde N es el número de parámetros posicionales.

¿Por qué los elementos de `"$*"` están separados por el primer carácter de IFS en vez de sólo por espacios? Para dar flexibilidad a la salida. Veamos un ejemplo simple: supongamos que queremos imprimir la lista de todos los parámetros posicionales separados por comas. Esto lo podríamos conseguir introduciendo en nuestro *script*:

```
IFS=,
echo "$*"
```

¿Por qué `"$@"` es equivalente a los N parámetros por separado? Para permitirnos usarlos otra vez como valores separados. El script 6.5 muestra un ejemplo para diferenciarlos, junto con la utilización del parámetro especial `?`.

***Script* 6.5** Parámetros especiales

```
#!/bin/bash

function cuenta-argumentos() {
  return $#
}

cuenta-argumentos "$@"
echo "Con \@ hay $? argumentos"
cuenta-argumentos "$*"
echo "Con \* hay $? argumentos"
```

Si al ejecutar el *script* 6.5 le pasamos los argumentos `manzana`, `pera` y `melón`, el resultado que obtendremos será:

```
Con \@ hay 3 argumentos
Con \* hay 1 argumentos
```

6.14. Leer de la entrada estándar: la orden `read`

El mandato incorporado `read` se usa para leer la entrada de la terminal o de un fichero.

Formato:

`read` *variable* ...

La orden `read` lee una línea de la entrada estándar y la parte en palabras delimitadas por cualquiera de los caracteres del valor de la variable de ambiente IFS (por omisión espacio, tabulador y `Nueva-Línea`) asignándole cada palabra de la entrada a cada una de las variables. Si no hay suficientes variables para todas las palabras de la entrada, la última contendrá las palabras restantes.

Ejemplos:

1. ```
   $ read X Y Z
   manzana naranja pomelo pera
   $ echo $X
   manzana
   $ echo $Y
   naranja
   $ echo $Z
   pomelo pera
   ```
2. ```
   $ IFS=:
   $ read PAL1 PAL2 PAL3 PAL4
   manzana:naranja:pomelo:pera
   $ echo $PAL1 $PAL2 $PAL3 $PAL4
   manzana naranja pomelo pera
   ```
   Es útil cuando queremos leer datos que no están separados por espacios.

Si a `read` sólo se le pasa como argumento una variable se le asignará a ésta la línea completa.

Ejemplo:

```
$ read LINEA
naranja manzana pera
$ echo $LINEA
naranja manzana pera
```

Si se omiten todos los nombres de variables, se asigna la línea completa a la variable REPLY.

Ejemplo:

```
$ read
Cualquier cosa
$ echo $REPLY
Cualquier cosa
```

Leyendo de un fichero

La orden `read` también puede leer de un fichero. Por sí mismo, `read` sólo lee una línea de la entrada; por tanto, para leer el fichero completo habrá que introducir la orden de lectura dentro de un bucle.

Ejemplo:

En el script 6.6 tenemos un ejemplo de cómo leer las líneas de un fichero mediante la orden `read`. *En él se han introducido algunas sentencias que no se han estudiado todavía, tales como* `case` *y* `while`. *La sentencia* `case` *nos permite realizar distintas acciones dependiendo del valor que tenga una determinada variable (*`$#` *en nuestro caso). El bucle* `while` *nos permitirá leer todas las líneas del fichero. Más adelante veremos más detenidamente cómo utilizar ambas. También se ha realizado una operación aritmética con la variable* `LNUM`.

***Script* 6.6** Lectura de un fichero

```
#!/bin/bash

# ejemplo4: Muestra el uso de la orden read.
# Este script admite un parámetro posicional, que
# debería ser el nombre de un fichero. Si no se le da,
# lo preguntará. Una vez le hemos dado el nombre del
# fichero, nos lo muestra con las líneas numeradas.

case $# in
0)   echo -n "Introduzca un nombre de fichero: "
     read FICHERO;;
*)   FICHERO=$1;;
esac

declare -i LNUM=1

while read  LINEA
do
     echo "$LNUM: $LINEA"
     LNUM=$((LNUM+1))
done < $FICHERO
```

6.15. Operaciones aritméticas

El *shell* bash proporciona varias formas de realizar operaciones aritméticas:

1. La orden `let`, cuya forma de uso es:

   `let "`*expresión_ aritmética*`"`

   la *expresión_ aritmética* puede contener constantes, operadores y variables del *shell* bash. Se deben usar comillas para encerrar la expresión aritmética siempre que ésta contenga espacios u operadores que tengan un significado especial para el *shell*.

2. Encerrando la expresión aritmética de la siguiente forma:

   `$((`*expresión*`))`

3. Indicando la expresión aritmética directamente, sin poner ningún espacio entre los operadores y operandos.

Las siguientes órdenes son equivalentes:

```
let "X=X + 1"
X=$((X+1))
X=X+1
```

Hay que hacer notar que en este caso no hay que preceder las variables del carácter `$`. Las siguientes órdenes son equivalentes:

```
let "X=X + 1"
let "X=$X + 1"
```

Antes de asignar a una variable el resultado de una operación aritmética debemos declarar dicha variable de tipo entero; si no, podemos llevarnos una sorpresa con el resultado obtenido (especialmente usando la tercera nomenclatura (`X=X+1`).

Ejemplos:

1. ```
   $ declare -i numero
   $ numero=12*2
   $ echo $numero
   24
   ```
2. ```
   $ producto=12*2
   $ echo $producto
   ```
   ¿Qué resultado obtendremos? Haga la prueba.

En la tabla 6.2 pueden verse los operadores aritméticos del *shell* en orden de precedencia.

`-`	Menos unario.
`!`	Negación lógica.
`* / %`	Multiplicación, división, módulo.
`+ -`	Adición, sustracción.
`<= <`	Menor o igual, menor que.
`>= >`	Mayor o igual, mayor que.
`==`	Igual a.
`!=`	Distinto a.
`&&`	Y lógico.
`||`	O lógico.
`=`	Asignación.
`+= -= *= /= %=`	Operadores con asignación.

Cuadro 6.2: Operadores aritméticos en orden de precedencia

Hay que comentar que los operadores `+= -= *= /= %=` realizan una operación aritmética y almecenan el resultado en el operando a su izquierda. Por ejemplo:

Ejemplo:

La siguiente sentencia

```
A=A+8;
```

Equivale a

```
A+=8;
```

Los operadores tienen el comportamiento esperado en cualquier lenguaje de programación.

Ejemplo:

Créese un fichero que contenga el *script* 6.7 y otro con valores numéricos, uno por línea. Ejecute el *script*.

***Script* 6.7** Operaciones aritméticas

```
#!/bin/bash

# ejemplo5: Ejemplo de uso de expresiones aritméticas.
# Lee valores enteros de un fichero y calcula la suma,
# el producto y la media de todos los valores.

case $# in
0)  echo -n "Nombre del fichero de datos: "
    read FICH;;
*)  FICH=$1;;
esac

declare -i CONT=0 SUMA=0 PROD=1 MEDIA=0

while read X
do
   CONT=$((CONT + 1))
   SUMA=$((SUMA + X))
   PROD=$((PROD * X))
done < $FICH

MEDIA=$((SUMA / CONT))

echo "La suma de todos los valores es $SUMA"
echo "El producto de todos los valores es $PROD"
echo "La media de los valores es $MEDIA"
```

6.16. Ejercicios

1. Cree un *script* llamado `busca`, que acepte exactamente el mismo tipo de argumentos que `find`. La diferencia entre `busca` y `find` debe ser que la salida de `busca` esté ordenada alfabéticamente. Nota: no hace falta tener en cuenta la protección de parámetros.

2. Cree un *script* llamado `fichero`, que acepte exactamente tres parámetros. El primero es el nombre de un fichero y el segundo y el tercero son nombres de directorios. El *script* buscará en los directorios indicados (y subdirectorios) ficheros con dicho nombre, y si los encuentra, mostrará por pantalla la información larga de `ls`. Si es posible, evite que el usuario reciba mensajes de error en pantalla.

3. Cree tres ficheros de *script* llamados `respaldo1`, `respaldo2` y `respaldo3`, que realicen una copia de seguridad (usando `tar`) del directorio `datos` (que se encuentra en su directorio casa). Para los siguientes casos, respectivamente:

   a) Sólo se almacena la última copia de seguridad.

   b) Sólo se almacena las dos últimas copias de seguridad de los dos días anteriores.

   c) Se almacenan una copia de seguridad de cada día (puede ayudarse de la orden `date`).

   El *script* se ejecuta una sola vez al día, al final de la jornada laboral.

4. Cree un *script* llamado `recuperacion1`, que restaure los datos que se almacenaron en el ejercicio anterior. El *script* copiará el contenido del directorio `datos` de su directorio de entrada en `datos_anteriores` y restaurará el contenido del fichero copia de seguridad que reciba como parámetro.

5. Realice un *script* llamado `recuperacion2` similar a `recuperacion1`, pero que reciba como segundo parámetro el directorio donde se copiará el contenido del directorio `datos`.

6. Realice un *script* llamado `alta`, que cree el directorio `public_html` dentro del directorio casa del usuario que reciba como parámetro. Además asignará como propietario del directorio a dicho usuario y le pondrá unos permisos que permitan total libertad al propietario y sólo lectura al resto de usuarios.

7. Cree un *script* llamado `segundo` que, simplemente, visualice el segundo parámetro que se le pase, independientemente del número de argumentos que le hayamos pasado. Nota no puede usarse la expresión $2.

8. Haga un *script* llamado `igrep`, que funcione de la misma forma que `grep` pero deberá pedir los parámetros interactivamente.

9. Escriba un *script* llamado `nwho` que le permita saber si un usuario está conectado o no al sistema. El nombre del usuario debe ser aceptado por `nwho` como argumento. Si el usuario está conectado al sistema el procedimiento debe darnos la línea correspondiente al usuario en cuestión que muestra la orden `who` y la cantidad de consolas que tiene abiertas. Si no está conectado no mostrará nada.

10. Realice un *script* llamado `alta`, que cree el directorio `public_html` dentro del directorio casa del usuario que reciba como parámetro. Además asignará como propietario del directorio a dicho usuario, le pondrá unos permisos que permitan total libertad al usuario y lectura y ejecución al resto de usuarios. Por último cree un fichero dentro de dicho directorio llamado `index.html` con el siguiente contenido (siendo <usuario> el nombre del usuario):

```
<HTML>
 <HEAD><TITLE>Suse Linux</TITLE></HEAD>
 <BODY>
  Bienvenido a la web de <usuario>
 </BODY>
</HTML>
```

11. Escriba un *script* que permita que le pasen cuatro parámetros posicionales como máximo. Los parámetros que se le pasan deben ser nombres de ficheros y se debe comprobar si estos ficheros existen y si el usuario que ejecuta el *script* puede leerlos. Al final debe indicar el total de ficheros que se pueden leer. Asegúrese de que la órdenes del *script* no genera mensajes de error que lleguen al usuario.

12. Créese un fichero llamado `agenda` donde va colocar una serie de nombres y los números de teléfono correspondientes, separados por el carácter `:`. Escriba un *script* llamado `busca` que reciba como argumento un nombre y responda dando el número de teléfono correspondiente a esa persona. Cuando se le dé un nombre que no esté en la agenda dé el mensaje siguiente:

    nombre `no está registrado en la agenda`

    Si no se le pasa ningún nombre debe mostrar la lista completa.

13. Cree un fichero llamado `correos`, que almacenará una dirección de correo electrónico por línea. Escriba un *scripts* llamado `dominios`, que muestre por pantalla los diferentes dominios que aparecen en el fichero sin repeticiones y ordenados alfabéticamente.

14. Cree un fichero llamado `publicidad`, pondrá un anuncio en la página personal del usuario que se le indique como parámetro. Para ello insertará como penúltima línea del fichero `public_html/index.html` de su directorio casa la siguiente expresion:

    <A HREF='http://barrapunto.com'>Visite barrapunto</A>

    Si lo desea puede usar un fichero temporal para realizar este ejercicio.

15. Cree un *script* llamado `ultimo` que, simplemente, visualice el último argumento que se le pase, independientemente del número de argumentos que le hayamos pasado.