PROGRAMACIÓN DEL SHELL

Bibliografía

Sarwar, S. M.; Koretsky, R.; Sarwar, S. A. El libro de LINUX. Addison Wesley, 2005.
Sobell, Mark G. Manual práctico de Linux. Comandos, editores y programación Shell. Anaya Multimedia,
2008 Sánchez Prieto, S. UNIX y LINUX. Guía práctica (Tercera edición). Ra-Ma, D.L. 2004.

1. CONCEPTO DE *SCRIPT*

El shell de Linux posee un lenguaje de programación bastante potente, que incluye instrucciones de decisión, aritméticas, bucles, etc. Las estructuras que proporciona no son una variedad aleatoria, más bien, se han escogido para proporcionar la mayoría de las características que están presentes en otros lenguajes estructurados, como C. Los programas escritos en el lenguaje del shell se denominan *scripts* o *guiones de comandos*.

Un *script* es un programa escrito con comandos del shell e instrucciones condicionales, bucles y variables. Debe estar contenido en un fichero de texto creado mediante un editor (vi, pico, emacs, etc.). Este archivo contendrá las órdenes según el orden en que el shell va a ir interpretando y ejecutando. La primera línea (opcional) se emplea para especificar el tipo de shell que va a interpretarlo (si dicho shell está definido y en estado activo). En nuestro caso, el intérprete de comandos es bash.

Shell Comando
C #!/bin/csh
bash #!/bin/bash
Korn #!/bin/ksh

En concreto, los caracteres #! le indican al sistema que el argumento que le sigue en la línea es el programa que se ha de usar para ejecutar el contenido de este archivo.

Los comentarios se declaran con el carácter #. El shell ignora todo lo que hay en una línea de comandos a continuación de dicho carácter.

El nombre del fichero que contiene un script no debe coincidir con los comandos de Linux, y es aconsejable que tenga la extensión .sh para reconocer fácilmente el tipo de archivo.

Existen varias formas de ejecutar un script:

- Dándole permiso de ejecución al fichero que contiene el script, y ejecutándolo poniendo el nombre del fichero en la línea de comandos.

  $ nombre_script.sh

- Sin darle permiso de ejecución y desde la línea de comandos ejecutar el intérprete de comandos pasándole como argumento el nombre del script. Por ejemplo, si se usa el shell bash sería:

  $sh nombre_script.sh **o $**bash nombre_script.sh

Ejemplo:

Los pasos a seguir para escribir y ejecutar un script son:

1. usando el editor vi se escribe el script y se guarda en un fichero, por ejemplo *primerscript.sh:*

```
#!/bin/bash #Indica que es un script que será interpretado por el shell
bash #Primer script en Linux
echo "Primer script"
```

2. se le da permiso de ejecución:

```
$ chmod u+x primerscript.sh
```

3. se ejecuta:

```
$ primerscript.sh o $ ./primerscript.sh
```

2. VARIABLES

Variables locales

El identificador de una variable debe comenzar siempre con una letra o un carácter de subrayado seguido de cualquier número de caracteres. Estos caracteres son necesariamente letras, dígitos numéricos o caracteres de subrayado (excluir letras acentuadas y la ñ). No se permiten los espacios en blanco. Se distingue entre mayúsculas y minúsculas.

En bash no se definen tipos de variables y se crean cuando son asignadas o leídas. Por defecto, todas las variables se consideran y se guardan como cadenas, aunque se les asignen valores numéricos. Para asignar un valor a una variable se utiliza el signo = (igual) y no puede haber espacios en blanco alrededor del signo igual. Cuando se desea obtener el valor de una variable, se escribe su nombre comenzando con el carácter **$** (dólar). De esta forma, el shell interpreta la cadena de caracteres siguiente al dólar como una variable y utiliza el valor de la misma.

Ejemplos:

```
num=4
nombre=ana
saludo="buenos días"
a=35
echo "$a" # visualiza el valor de la variable a
```

En el tercer ejemplo hay que "acotar" el contenido de la variable porque tiene espacios en blanco, sino, el shell asociaría el valor *buenos* a la variable saludo e interpretaría *días* como un comando.

Parámetros de entrada

Un script soporta argumentos o parámetros de entrada. Éstos se sitúan a la derecha del nombre del script en la línea de órdenes.

$ nombre_script.sh argumento1 argumento2 ...

Los parámetros de entrada de los scripts están separados por caracteres en blanco. Si se desea que un parámetro contenga uno o más caracteres en blanco, se debe poner el parámetro entre comillas dobles.

Se puede hacer referencia a estos argumentos, también llamados parámetros de posición, desde el propio script como cualquier otra variable, de la siguiente forma:

$n

donde ***n*** representa la posición del argumento dentro de la lista de argumentos de entrada. nombre_script.sh argumento1 argumento2 ... argumentoN

$0 $1 $2 $N

Es necesario tener en cuenta que el argumento cero devuelve el nombre del script, y que a partir del parámetro nueve, la posición debe ir encerrada entre llaves; es decir,

$0 Representa al parámetro cero o nombre del programa
$1 Representa al parámetro uno
$2 Representa al parámetro dos
…
$9 Representa al parámetro nueve
${10} Representa al parámetro diez
${11} Representa al parámetro once
…

Ejemplo:

El siguiente script lleva a cabo las mismas acciones que el comando copy de MS-DOS, (de esta forma los usuarios acostumbrados a ese comando no tienen que aprenderse otro). 1. se edita un script en el fichero llamado copy, en el que se introduce la siguiente línea: cp $1 $2
2. después, se da permiso de ejecución al fichero copy: $ chmod u+x copy 3. ya se pueden copiar dos ficheros usando el script creado: $ copy factoriales prueba El shell ejecuta el comando cp, sustituyendo $1 por factoriales, y $2 por prueba

Variables especiales

Estas variables informan del estado de un proceso o de los argumentos recibidos por el

mismo. **$*** Representa todos los argumentos excepto el argumento 0.

$@ Representa la cadena de argumentos entera (excluyendo el argumento 0), pero como una lista de cadenas, a diferencia de $* que obtiene todos los argumentos en una única cadena.

$# Representa el número de argumentos sin contar el argumento 0 (el nombre del script).

$$ Devuelve el identificador del proceso (PID) que se está ejecutando.

$? Devuelve un código según el resultado de la operación anterior (estado de salida). Esto puede ser utilizado para controlar si se ha producido algún error.

$! Devuelve el identificador de proceso (PID) del último proceso lanzado como tarea de fondo.

$LINENO Devuelve el número de línea del script donde aparece. Sólo tiene sentido en la fase de depuración.

Variables de entorno

Un servicio del shell es el mantenimiento de las variables de entorno. Una variable de entorno puede ser cualquier cadena de caracteres que no incluya el signo **$** (dólar) y que no tenga espacios en blanco.

> ***Nota:*** *El utilizar las mayúsculas para el nombre de las variables de entorno no es necesario, pero se trata de un convenio muy extendido (y por tanto, aconsejable) entre los programadores del shell.*

Las variables de entorno definidas se pierden al finalizar la sesión. Sin embargo, existen una serie de variables de entorno "permanentes" asociadas con cada login. Dichas variables se pueden visualizar con el comando **env**. Entre estas variables están por ejemplo: HOME (ruta del directorio de inicio), PWD (ruta del directorio de trabajo actual), SHELL (tipo de shell por defecto), PATH (lista de directorios de búsqueda de órdenes), etc.

Estas variables se pueden usar dentro de un programa del shell y no es aconsejable cambiarles el valor, ya que pueden ser utilizadas por órdenes o aplicaciones existentes. Si la variable de entorno existe, el shell cambiará su valor por el nuevo valor que le proporcionamos. Si no existe esa variable, la crea.

3. INSTRUCCIONES

shift

El comando **shift** permite desplazar hacia la izquierda todos los argumentos del script, exceptuando el argumento 0 que no se ve afectado. Esta orden es muy utilizada dentro de los bucles. **Sintaxis:** *shift* n

Por defecto, el desplazamiento (n) tiene el valor 1, por lo que cada vez que se ejecute esta orden se pierde el primer argumento del script. Si se especifica un desplazamiento distinto de 1, el número de argumentos que se pierden vendrá especificado por n.

echo

Se encarga de enviar a la pantalla los caracteres que siguen a dicho comando y terminados en una nueva línea.

Sintaxis: *echo* [-ne] texto a visualizar

El texto puede ir entre comillas o sin comillas.

Opciones:

1. La opción **–n** no introduce el salto de línea.
2. La opción **– e** activa la interpretación de caracteres de barra invertida. En este caso, el texto tiene que ir entre comillas.

Código	Significado
\b	Retroceso de espacio
\f	Salto de página
\n	Salto de línea
\r	Retorno de carro
\t	Tabulador horizontal
\v	Tabulador vertical
\a	Alerta (pitido)
\"	Comillas dobles

```
Ejemplos:
        echo -n "esto es una prueba"
        echo -e "esto \n es una \t prueba"
        echo $PATH
```

read

Esta orden sirve para leer información desde el teclado (entrada estándar) y asignársela a una variable. Cada instrucción **read** lee una línea del teclado. Su sintaxis es:

Sintaxis: *read* var1 [var2 var3 .....]

La instrucción asigna a *var1* el primer argumento de la línea introducida, a *var2* el segundo y así sucesivamente. Si se introducen más argumentos que variables hay, todos los datos que sobran por la derecha se asignan a la última variable de la lista. Los argumentos se separan por blancos o tabuladores.

exit

Se puede utilizar esta orden dentro de los scripts para finalizar inmediatamente su ejecución. Esta orden puede llevar un argumento, que se convierte en el valor que el script devuelve al shell que lo invocó. Cuando se utilizan estos valores de retorno, se debe seguir la convención estándar, es decir, la terminación correcta debe devolver cero. Los valores distintos de cero deben estar reservados para las distintas condiciones de error.

if

Sintaxis: *if* orden
then
orden (o varias órdenes)
else
orden (o varias órdenes)
fi

Las palabras-claves **if**, **then**, **else y fi** deben colocarse al principio de la línea. No obstante, si se prefiere poner **then** en la misma línea que **if**, se debe añadir un punto y coma para separar la orden del **if** de la palabra clave **then**.

En primer lugar se ejecuta la orden que va detrás de **if**, si esta ha sido realizada satisfactoriamente (estado de salida cero) se ejecutan las órdenes que aparecen entre **then** y **else**, en caso contrario se ejecutan las que van entre **else** y **fi**.

La parte **else** es opcional: *if* orden
then
orden
.
fi

Si no se realiza satisfactoriamente la orden que va detrás de **if**, se salta al final y no se ejecuta ninguna orden. Se pueden encadenar órdenes **if** empleando la siguiente sintaxis:

if orden1
then
orden
.
elif orden2
then
orden
.
else
orden
.
fi

Si la ejecución de orden1 es correcta, entonces se ejecutan las órdenes que siguen al **then**; en caso contrario (que no se ejecute correctamente orden1) y si la ejecución de orden2 es correcta, entonces se ejecutarán las órdenes que siguen al segundo **then**; y si no lo es, serán ejecutadas las órdenes siguientes a **else**. Es importante tener en cuenta que por cada **if** tiene que aparecer un **fi** y viceversa. Además los **elif** no se cierran con un **fi** adicional.

test

Esta orden permite evaluar expresiones lógicas.

Sintaxis: *test* expresión_lógica o [expresión_lógica]

Si la expresión es verdadera devuelve un estado de salida 0 y si es falsa un estado diferente. Las expresiones lógicas hacen referencia a cadenas, enteros y ficheros. Se suele utilizar a menudo conjuntamente con la orden **if** permitiendo bifurcar nuestros programas en base a una condición, que resulta de evaluar expresiones lógicas. Las expresiones se construyen utilizando *primitivas*, que se pueden referir a cadenas, enteros y ficheros.

Primitivas referidas a cadenas:

-n cadena Devuelve verdadero si la longitud de la cadena no es cero.
-z cadena Devuelve verdadero si la longitud de la cadena es cero
S1 = S2 Compara las dos cadenas S1 y S2. Devuelve verdadero si son iguales. **S1 !=**
S2 Compara las cadenas S1 y S2. Devuelve verdadero si son diferentes.

Primitivas referidas a enteros:

n1 -eq n2 Devuelve verdadero si ambos enteros son iguales numéricamente. **n1 -ne**
n2 Devuelve verdadero si ambos enteros son diferentes numéricamente. **n1 -gt n2**
Devuelve verdadero si numéricamente n1 es mayor que n2.
n1 -ge n2 Devuelve verdadero si numéricamente n1 es mayor o igual que n2. **n1**
-lt n2 Devuelve verdadero si numéricamente n1 es menor que n2.
n1 -le n2 Devuelve verdadero si numéricamente n1 es menor o igual que n2.

Primitivas referidas a ficheros:

-d fichero Devuelve verdadero si el fichero existe y es un directorio.
-f fichero Devuelve verdadero si el fichero existe y no es un directorio.
-s fichero Devuelve verdadero si el fichero existe y tiene un tamaño mayor que cero (no está vacío).
-r fichero Devuelve verdadero si el fichero existe y tenemos permiso de lectura. **-w fichero** Devuelve verdadero si el fichero existe y tenemos permiso de escritura. **-x fichero** Devuelve verdadero si el fichero existe y tenemos permiso de ejecución. **fichero1 -nt fichero2** Devuelve verdadero si fichero1 es más reciente que fichero2. **fichero1 -ot fichero2** Devuelve verdadero si fichero1 es más antiguo que fichero2. **fichero1 -ef fichero2** Devuelve verdadero si los dos ficheros (fichero1 y fichero2) poseen el mismo inode.

Las primitivas anteriores se pueden combinar con los siguientes operadores booleanos: **!** Operador unario *negación*.
-a Operador binario *and*.
-o Operador binario *or*.
\(expresión \) Se usan los paréntesis para realizar agrupamientos. Las barras invertidas son necesarias para evitar una incorrecta interpretación de los paréntesis.

Cuando se comprueba el contenido de una variable, es preferible encerrar entre comillas dobles para evitar un error en la sintaxis del comando **test** si la variable no está definida.

```
Ejemplos:
        [ "$1" -eq 0 ]
        test "$1" = "$2"
        test "$a" -ne 3 -a "$a" -ne 5
        test \( "$1" -eq 0 \) -o \( "$1" -eq 5 \)
        [ -f "$1" ]
```

OPERADORES: && y ||

El shell posee también estos operadores para la ejecución condicionada de las instrucciones:

- operador **&&**, cuya sintaxis es: orden1 **&&** orden2
  Se ejecuta la primera orden (*orden1*), si devuelve un estado de salida 0 se ejecuta la segunda orden (*orden2*).

- operador ||, cuya sintaxis es: orden1 || orden2
  Se ejecuta la primera orden (*orden1*), si devuelve un estado de salida distinto de cero, se ejecuta la segunda orden.

Se puede observar que en ocasiones estos operadores son equivalentes a la orden **if**.

```
Ejemplo:
       El siguiente código: if cat nombre_fichero
                                    then
                                          echo correcto
                                    else
                                          echo incorrecto
                                    fi
                es equivalente a: cat nombre_fichero && echo correcto || echo incorrecto
```

case

Esta orden permite comparar el valor de una variable frente a una serie de valores posibles.

Sintaxis: *case* valor *in*
patron1) serie de mandatos 1;;
patron2) serie de mandatos 2;;
.....
*) serie de mandatos n;;
esac

donde **esac** es la palabra-clave para el final de la instrucción **case**, y ***)** representa el caso por defecto. Esta instrucción *case* orienta las operaciones dependiendo de si *valor* es igual a *patron1*, *patron2*,... o distinto. Si
el *valor* = *patron1*, entonces se ejecuta la serie de mandatos 1. Si el *valor* = *patron2*, se ejecuta la serie de mandatos 2, y así sucesivamente. Un doble punto y coma (;;) sirve para finalizar cada elección de **case**.

Los patrones pueden ser cualquier expresión regular. Una lista de patrones se establece al separarlos por el símbolo "|". Por ejemplo, *[aA]*e|jj|u** es una lista de tres patrones.

El uso de **case** puede ser equivalente a usar operadores lógicos o varios **if**. Sin embargo, la utilización de *case* puede hacer más legible el script, además permite hacer comparaciones con patrones y no solamente con valores determinados.

expr

Esta orden permite realizar algunas operaciones aritméticas con enteros. Las operaciones aritméticas que se pueden realizar son: suma (**+**), resta (**-**), multiplicación (*****), división (**/**) y módulo (%). Es necesario separar los operandos de los operadores con un espacio. Para que el asterisco sea interpretado como operador de multiplicación es necesario anteponerle la barra invertida (\) eliminando su función como carácter especial.

Sintaxis: *expr* arg1 op arg2 [op arg3 …]

```
Ejemplo:
       expr 4 + 5 # devolvería 9 por la salida estándar
```

Si se quiere asignar el resultado de una expresión a una variable, se debe encerrar la expresión entre comillas invertidas simples (`) o entre paréntesis anteponiéndole el símbolo $.

```
Ejemplos:
       NETO=`expr 100 - 80` o NETO=$(expr 100 - 80) echo $NETO #muestra 20
       echo `expr 100 - 80` o echo $(expr 100 - 80) #muestra 20
       a=$(expr 5 \* 2) o a=`expr 5 \* 2 ) #asigna 10 a la variable a
```

Si se utilizan las comillas dobles, lo considera texto y no ejecuta la orden **expr**, por tanto, la instrucción *echo "expr 100 – 80*" muestra por pantalla la cadena: *expr 100 - 80*

let

Esta orden, al igual que la orden **expr**, también permite realizar operaciones aritméticas con enteros. **Sintaxis:** *let* expresión_aritmética o ((expresión_ aritmética))

Los operadores aritméticos son: suma (**+**), resta (**-**), multiplicación (*****), división entera (**/**), módulo o resto de la división entera (**%**) y potencia (******). Se puede usar el operador de asignación (**=**) para asignar el resultado obtenido a una variable.

```
Ejemplo:
       let c=3+5*2 o ((c=3+5*2)) #asigna el valor 13 a la variable c
```

Además, es posible especificar varias expresiones separadas por una coma (sin dejar espacios en blanco) en una misma llamada al comando.

```
Ejemplo:
       ((a=$1/$2,b=$3**$4)) #asigna a la variable a el resultado de la división entera
                            entre los dos primeros argumentos de posición y a la
                            variable b el
                       resultado de elevar el tercer argumento al cuarto argumento de
                            posición.
```

La sintaxis $(()) devuelve el resultado de la expresión.

```
Ejemplo:
       echo $((3+9)) #muestra el entero 12
```

El comando **let** también ofrece los operadores aritméticos de postincremento (var++), postdecremento (var--), preincremento (++var) y predecremento (--var).

```
Ejemplos:
        z=3; echo "z vale $z y ahora vale $((z--))" #muestra: z vale 3 y ahora
        vale 3 z=3; echo "z vale $z y ahora vale $((--z))" #muestra: z vale 3 y
        ahora vale 2
```

Los operadores lógicos son: y lógico (&&), o lógico (||), negación lógica (!), menor que (<), menor o igual (<=), mayor que (**>**), mayor o igual (>=), igual (==) y distinto (!=).

Ventajas del comando **let** respecto al comando **expr**:

- Es más rápido porque está integrado en el shell y no requiere la creación de un subproceso. • Los operadores utilizados son los que se emplean habitualmente en matemáticas y en otros lenguajes de programación.
- No es necesario insertar un espacio entre cada elemento en una expresión, ni preceder con la barra invertida ciertos operadores.
- El trabajo con variables se simplifica porque es posible asignar una variable dentro de una expresión y los nombres de variables no van precedidos de un carácter $.
- Existe un mayor número de operadores para el cálculo aritmético.

for

Sintaxis: *for* variable *in* lista de valores (valor1 valor2 ...)

do

orden1
orden2
.
.

done

Las palabras-clave **for**, **do** y **done** deben colocarse al principio de la línea. La variable de iteración puede tener cualquier nombre. La serie de instrucciones entre **do** y **done** se ejecuta para cada elemento de la lista de valores.

```
Ejemplos:
 Usando cadenas fijas como lista de valores para la variable que controla
el for 1. for i in 1 2 3 4 5
        do
                echo "el valor de la variable i en este paso es: $i"
        done
2. for i in 1 a hola 45
        do
```

```
                echo "el valor de la variable i en este paso es: $i"
        done
3. for i in "1 2 3 4 hola"
        do
                echo "el valor de la variable i en este paso es: $i"
        done
```

Cuando no se establecen de antemano los valores que va a tomar la variable del bucle, se asume que van a ser los argumentos del script, es decir, el parámetro de posición $@ que representa la cadena de argumentos entera excluyendo el nombre del programa.

Cuando se programa un script, pocas veces se va a conocer de antemano los elementos que componen la lista por la que se itera. Normalmente esta lista se genera dinámicamente (cuando se va a ejecutar el bucle) de una de estas dos maneras:

- Utilizando las comillas invertidas simples (o los paréntesis precedidos por el carácter dólar) para generar la lista a partir de la salida de un comando.

```
Ejemplo:
            Para iterar por una lista que contiene el nombre de los elementos del

                    directorio raíz: for x in `ls /` o for x in $(ls / )
```

- Utilizando el comando **seq** que genera secuencias de números. En este comando se debe indicar: el primer número, el incremento y el valor máximo.

```
Ejemplo:
       for x in `seq 1 1 100`
```

while

Sintaxis: *while*lista_órdenes

do

orden1
orden2
.

done

Se ejecuta la lista de órdenes (separadas por **;**) que aparece detrás de **while**. Si la última devuelve un estado de salida 0 (sin errores en la ejecución de la orden o devuelve "true" una orden **test**), se ejecutarán las órdenes entre **do** y **done**. A continuación, se ejecuta de nuevo la lista de órdenes y si la última devuelve un estado de salida 0 se ejecutan las órdenes entre **do** y **done** otra vez. Esto se repite indefinidamente hasta que la última orden de la lista devuelva un estado de salida distinto de 0.

```
Ejemplo:
       cont=0
       while [ $cont -lt 10 ]
       do
              echo El contador es $cont
              cont=`expr $cont + 1`
       done
```

until

Sintaxis: *until*lista_órdenes

do

orden1
orden2
.

done

Se ejecuta la lista de órdenes (separadas por ;) que aparece detrás de **until**. Si la última devuelve un estado de salida distinto de 0 se ejecutan las órdenes entre **do** y **done**. A continuación, se ejecuta de nuevo la lista de órdenes y si la última devuelve un estado de salida distinto de 0 se ejecutan las órdenes entre **do** y **done** otra vez. Esto se repite indefinidamente hasta que la última orden de la lista devuelva un estado de salida igual a 0.

```
Ejemplo:
        cont=20
        until [ $cont -eq 10 ]
        do
                echo "El contador es $cont"
                cont=`expr $cont - 1`
        done
```

Se puede decir, por tanto, que las órdenes **while** y **until** son complementarias.

4. ENTRECOMILLADO

En Linux existen caracteres con significados especiales: $, *, <, >, etc. Si se desea eliminar el significado especial de estos caracteres, se necesita usar un mecanismo especial denominado **entrecomillado**.

- Backslash (\): elimina el significado especial al carácter que precede.
- Comillas simples ('Cadena'): conserva el valor literal de cada uno de los caracteres de la *Cadena*.
- Comillas dobles ("Cadena"): conserva el valor de literal de la *Cadena*, excepto para los caracteres dólar, comilla simple, comilla doble, backslash y comilla invertida simple.
- Comillas invertidas simples (`Cadena`): ejecuta la orden representada por *Cadena*.

```
Ejemplos:
        x=3
        echo $3 #muestra 3
        echo \$x #muestra $x
        echo '$x' #muestra $x
        echo "$x" #muestra 3
        echo `ls -l` #muestra el listado largo del directorio de trabajo actual
```

5. FUNCIONES

El shell de Linux permite escribir funciones en nuestros scripts para agrupar cierta parte de código que tiene una funcionalidad especial.

Las funciones se deben declarar antes de ser usadas siguiendo la sintaxis que se describe a continuación:

[function] nombre_funcion *( )*
{
código de la función
}

Cuando se declara una función se puede omitir la palabra reservada *function* o los paréntesis, pero no las dos cosas a la vez.

Para invocar a una función es necesario poner el nombre de dicha función seguido, opcionalmente, de uno o más argumentos:

nombre_funcion arg1 arg2 …argN

Aspectos a tener en cuenta a la hora de usar funciones:

1. A una función se le pueden pasar como argumentos: un valor constante, una variable, cualquiera de los argumentos recibidos por el script donde está definida la función (incluido todos) o ningún argumento. No es necesario declarar los parámetros en la declaración de la función, basta usar las variables $1, $2, etc. dentro del cuerpo de la función para referirse a los parámetros en su orden correspondiente.

Ejemplo:

```
suma1y2argumentos ( )
{
        echo "Suma: $(expr $1 + $2) "
}

sumatodosargumentos ( )
{
        suma=0
        for i
        do
                suma=$(expr $suma + $i)
        done
        echo "Suma total: $suma"
}

sinargumentos ( )
{
        echo Nombre del fichero:
        read fichero
        [ -r "$fichero" ] && more $fichero || echo No es un fichero o no
        tiene permiso de lectura
}

echo "Los argumentos recibidos son: $*"
b=22
suma1y2argumentos $1 $b
suma1y2argumentos $1 10
suma1y2argumentos $1 $3
sumatodosargumentos $*
sinargumentos
```

2. La posición de los argumentos que recibe una función comienzan a enumerarse desde 1

Ejemplo:

```
resta3y4argumentos ( )
{
        echo "Resta: $(expr $1 - $2) "
}
echo "Los argumentos recibidos son: $*"
resta3y4argumentos $3 $4
```

3. Las funciones pueden ser invocadas desde cualquier punto del script o desde otra función.

Ejemplo:

```
suma1y2argumentos ( )
{
        echo "Suma: $(expr $1 + $2) "
}

sumaenteros ( )
{
        echo Introduce dos enteros:
        read x y
 suma1y2argumentos $x $y
}
```

4. Una vez finalizada la ejecución de una función, los argumentos del script siguen siendo los mismos que los que existían antes de la llamada a la función.