# PROGRAMACIÓN DE SCRIPTS EN SHELL

## INTRODUCCIÓN:

Los programas shell (*Guiones del Shell, guiones, programas Shell, Shell Scripts o Scripts*) son un conjunto de órdenes del shell escritas en un fichero ordinario de Linux en formato texto puro que son interpretadas por el núcleo del sistema. Los scripts son muy potentes y eficaces, entre sus características podemos incluir:

- Interpretado
- Permite programación estructurada.
- Interactivo (read , echo)
- Sentencias de Control de flujo del programa (If..., Case..., While..., Until..., For.....)
- Tratamiento y operaciones con Variables del Sistema y de Usuario.
- Uso de argumentos de entrada.
- Funciones.
- *Sintaxis compleja.*
- *Poca capacidad para cálculo numérico (no están diseñados para este uso).*

## CREACIÓN Y EJECUCIÓN DE UN SCRIPT:

Para crear un Guión solo es necesario disponer de un Editor que disponga de modo texto puro como es el ***vi***. Si estamos trabajando desde un terminal en red se puede usar un editor externo desde otro S. Operativo (Windows por ejemplo) y transferir el fichero vía FTP o usar las opciones copiar-pegar.

Las formas de ejecutar un Guión son las siguientes:

1. **$ . /nombre_script**

2. **$ exec nombre_script**

*Método 1: $ . / nombre_script*

- El punto ha de ir separado por un blanco del nombre del Script.
- Es necesario que el fichero Script tenga permiso de ejecución (chmod u+x).
- El programa se ejecuta en el entorno actual y no se crea ningún subshell.

*Método 24: $ exec nombre_Script*

- Es necesario que el fichero Script tenga permiso de ejecución (chmod u+x).
- El programa se ejecuta en el entorno actual y no se crea ningún subshell., al finalizar, nos saca a lógin.
- *Se recomienda no usar este método ya que exec tiene otros usos muy importantes.*

## COMENTARIOS EN EL SHELL:

Siempre es recomendable añadir comentarios a nuestros programas en cualquier lenguaje, una buena norma es además de los comentarios específicos donde sean necesarios, usar una cabecera con el Nombre del programa y las principales características.

En los scripts, una línea de comentario debe empezar por el símbolo almohadilla (#)

```
# Búsqueda de cadenas
#  Fecha: 18/02/2003
#  Autor: J.A. Sánchez
#  Modificado por: A.S. Romero.
#  Este programa sirve para ...utiliza ...
#  .....
```

## VARIABLES DE USUARIO:

Es posible definir variables por parte del usuario, las normas y significado de las mismas son las siguientes:

1. Para definirlas se admiten Mayúsculas, minúsculas, números y carácter subrayado. Deben comenzar por un carácter Alfabético o Subrayado.

2. No es necesario declararlas previamente a su uso, si se usa una variable que no ha sido creada o asignada, su valor es nulo.

3. No tienen tipo, todo lo que se escriba en ellas, tiene significado de carácter. Los números se almacenan como si fuesen cadenas de texto y no en formato binario como puede ser en un lenguaje orientado a proceso de datos.

4. Para su asignación se usa el símbolo igual sin blancos a los lados:

   - **Variable1=valor1 [ variable2=valor2 variable3=valor3 ....]**

     *nombre=pepe*

5. Se pueden crear y usar variables en Shell fuera de los Script

6. Para utilizar el contenido (operar, visualizar, mover..) de una variable se antepone delante del nombre el símbolo $

   $ echo Tu nombre es $nombre

7. Las variables son locales al shell donde se han creado. En el caso de querer poder usadas en el Shell actual y todos los subshell que se utilicen, hay que usar la orden **export**:

   - **export**
   - **export lista_variables**

   *nombre=pepe*
   *edad=18*

   ← Las variables son locales al Shell actual

   *nombre=pepe*
   *edad=18*
   *export nombre edad*

   ← Las variables son globales al shell actual y subshells.

   Las variables que se exportan, mantiene su valor del shell padre a los hijos, etc. Pero nunca al contrario, una variable exportada no se exporta al shell padre del actual donde se ha creado.

   Las variables exportadas se las llama a veces variables globales o variables de entorno (aunque este último nombre a veces se reserva las variables del Sistema).

   Usando *export* sin argumentos se muestra la lista de variables exportadas.

8. Cuando se usen variables en expresiones de comparación se recomienda ponerlas entrecomilladas, el no hacerlo es una fuente de problemas.

9. Si la variable va a contener cadenas con caracteres en blanco hay que usar comillas para asignarla:

*nombre=Jose Luis* ← Erroneo

*nombre="Jose Luis"* ← Correcto

10. Las variables incluidas en líneas echo con comillas dobles son interpretadas correctamente:

```
ciudad=mostoles
echo "Vives en
$ciudad"
```

11. Se puede asignar a una variable la salida de una orden haciendo uso de los acentos graves (acento francés):

```
fecha=`date`
```

12. Para separar una variable de carácter situados al lado se deben usar corchetes o comillas dobles:

```
$ nombre=maria
$ echo nombre
 maria
$ echo ${nombre}no
  mariano
$ echo "$nombre"no
 mariano
```

13. Se puede hacer que una variable sea de sola lectura con la instrucción **readonly** (es la forma de este lenguaje de declarar constantes):

    - **readonly**
    - **readonly lista_variables**

```
$ instituto=Arboleda
$ ciudad=Mostoles
$ readonly insituto ciudad
$insitituto=Mirasierra
 instituto: This variable is read only
$
```

    Usando readonly sin argumentos, se muestran las variables que no son modificables junto con sus valores.

14. La asignación de variables desde teclado en un programa se efectúa con la instrucción read como veremos próximamente.

15. Se puede restaurar (vaciar) el contenido de una variable haciendo uso de la orden unset:

    - **unset lista_variables**

```
$ unset ciudad instituto
```

    Otra forma de hacerlo es no asignarle nada a la variable:

```
$ insitituo=
```

**VARIABLES DEL SISTEMA:**

Son variables globales creadas por el sistema (ya están exportadas). Algunas son modificables por el usuario (PATH, PS1,...) y otras no lo son (LOGNAME). A continuación se muestran algunos valores de variables del Sistema:

- CDPATH Nombre del path usado por la orden cd, si no tiene nada examina el de trabajo
- EDITOR Editor predetermindo que usaran los programas.
- HOME Directorio del usuario.
- LOGNAME Nombre de usuario.
- MAIL nombre del archivo del buzón de correo del usuario.
- MAILCHECK Segundos que usa el shell para mirar si examinar el correo y mandar aviso.
- PATH Ruta de búsqueda del usuario, donde el shell busca ls ordenes externas.
- PPID Identificación del proceso padre del actual.
- PS1 y PS2 Indicadores del prompt primario y secundario.
- SHELL Ruta absoluta del Shell que estamos usando.
- TERM Tipo de terminal que estamos usando.

## ORDENES SET, ENV y EXPORT SIN ARGUMENTOS:

### Orden set

La orden set sin argumentos informa los nombres y los valores todas las variables locales y del entorno, sean del usuario o del sistema.

### Orden env

La orden env sin argumentos informa de los nombres de las variables de entorno, pero no incluye a las del usuario. En general da menos información que set.

### Orden export

La orden export sin argumentos informa los nombres y los valores todas las variables exportadas en el shell actual.

## ARGUMENTOS PARA LOS SCRIPTS:

Se puede invocar un script con argumentos de forma que internamente se reciban en unas variables denominadas parámetros posiciónales.

- **nombre_script [argu1 argu2 argu3.....argu9]**

  *$ agenda Lopez Mostoles 34 916642334*

Los argumentos se escriben separados por blancos o tabuladores generalmente.

Las variables o parámetros posiciónales no son modificables y se denominan $1, $2..... $9 (Se pueden usar hasta 9 parámetros en la línea de órdenes, este límite se puede ampliar con la orden **shift** aunque es inusual programas con un nº tan elevado de parámetros).

*$ mi_agenda Lopez Mostoles 34 916642334*

En este ejemplo internamente las variables posiciónales se cargan con:

*$1* ← *Lopez*
*$2* ← *Mostoles*
*$3* ← *34*
*$4* ← *916642334*

Otras variables internas relacionadas con los argumentos de un script son:

- $# Nº de argumentos de la línea de órdenes.
- $* Contiene la totalidad de los argumentos.
- $@ Idem, pero si se usa como "$@" los argumentos van entre comillas.
- $0 Contiene el nombre del Script.

**MAS VARIABLES DEL SISTEMA:**

- $$ PID del proceso en ejecución.
- $! PID del proceso de segundo plano más reciente invocado por el shell.
- $? Código de estado del último comando ejecutado.

Cada vez que se ejecuta una orden devuelve un código entre 0 y 255. El valor 0 nos indica que la orden se ha ejecutado correctamente, mientras que un valor diferente de 0, indica que se ha producido un error. El valor de los códigos de error difiere de una orden a otra y no son de especial interés en estos apuntes.

**ASIGNAR LOS PARÁMETROS POSICONALES DEL SHELL CON SET (AVANZADO):**

Se puede usar la orden set para asignar las variables posiciónales en un programa:

- **set argumentos**

Ejemplo:

```
        $ date
    Sat  Aug  7   13:26:42  PM 1999
$ set `date`
        $ echo "$1 $2 $3 $4 $5 $6"
           Sat  Aug  7   13:26:42  PM 1999
        $ echo  "$@"
           Sat  Aug  7   13:26:42  PM 1999
        $ echo "$3 de $2 de $6"
           27 de Aug de 1999
```

**ORDEN EXIT:**

La orden exit sirve para transferir el control al proceso que hace la llamada. El argumento de la orden exit es número opcional, que se le proporciona al proceso que hace la llamada, como estado de salida del proceso que concluye.

- **exit [numero]**

Ejemplo:

```
.....
.....
.....
.....
.....
exit 0
```

En el punto anterior vimos que todas las ordenes Linux proporcionan un código de estado 0 si se ejecutan correctamente que es recogido en la variable $?.

No es necesario usar la orden exit a menos que se desee terminar un proceso desde varios puntos del código del mismo (mala práctica en programación estructurada).

Se puede terminar un shell con la orden *exit numero* como última línea si deseamos recoger el valor en la variable $?

Cuando se ejecuta una orden exit sin argumento, el núcleo de Linux asigna el valor de estado proporcionado por el guión.

Generalmente el número de salida manual es 0:

*exit 0*

## ENTRADA /SALIDA INTERACTIVA LAS ORDENES ECHO Y READ:

### Orden echo

La orden echo visualiza en el dispositivo de salida estándar (stdout) pantalla el valor de sus argumentos, que pueden ser puede ser el nombre de una variable o una expresión que contenga variable/s, o simplemente de una cadena de texto.

- **echo argumentos.....**

Ejemplos:

*$echo Tu nombre es $nombre $echo*
*"Tu nombre es $nombre"*
*$ echo 'Tu nombre es $nombre'*
*$ echo "La fecha actual es " `date`*
`

Dentro de la expresión pueden aparecer una serie de caracteres que tienen un significado especial, siendo estos los siguientes:

| | | |
|---|---|---|
| \b | → | blackspace (Retroceso) |
| **\c** | → | **imprime la línea sin salto de línea.** |
| \n | → | newline (Nueva línea). |
| \r | → | return (retorno de Carro). |
| \f | → | Salto de página |
| \t | → | tabulador. |
| \\ | → | carácter \. |
| \0num | → | código ASCII del número indicado, el cual está en octal. |

Es típico el uso de \c en echo para no avanzar línea:

*echo "El nombre de la ciudad es \c"*

Hay que tener en cuenta lo que sigue:

- Los argumentos sin comillas son interpretados literalmente como texto, excepto aquellos que sean caracteres especiales del Linux.

- Los argumentos situados entre comillas simples no son interpretados.

- Los argumentos que estén entre comillas dobles no son interpretados excepto excepto el símbolo $, el símbolo \ , y las comillas simples y dobles.
  EL símbolo \ sirve para inhibir el significado especial del próximo carácter.

- Los argumentos encerrados entre acentos graves (francés) ejecuta la orden y sitúa su salida estándar en el ámbito donde estén situados (sustitución de orden.)

**Orden read**

Si desea escribir un shell interactivo, que solicite información a través de la entrada estándar (stdin) teclado al usuario, se debe usar la orden read que sirve para asignar variables en tiempo de ejecución (similar al scanf del lenguaje C):

- **read variable1 variable2 variable3.....**

Ejemplo:

*echo "Introduce tu Nombre y Edad: \c" read nom edad*

Hay que tener en cuenta que:

- Las variables se escriben a continuación del read separadas por blancos.

- La ejecución del programa se detiene cuando se encuentra un read y no continua hasta que se pulsa Enter.

Los datos son introducidos como palabras separadas por blancos o tabuladores.

- Las variables se asignan de izquierda a derecha.

- Si el n º de palabras introducido es mayor que el nº de variables especificadas en read, se asigna a la última variable las palabras sobrantes.

- Si el nº de palabras introducido es menor que el nº de variables especificadas en read, estas quedan con valor nulo.

**EJECUCIÓN DE ÓRDENES EN LOS SHELL.**

- Se pueden agrupar varias órdenes en una línea separadas por punto y coma, estas se ejecutaran de izquierda a derecha en un nuevo subshell.

**(orden1; orden2; orden3;.....)**

- Si deseamos que una orden o más ordenes se ejecute en el shell actual del script deberemos encerrarla entre llaves:

**{ orden; }**
**{ orden1; orden2; orden3;..... }**

- Ejecución condicional de ordenes:

**ord1 && ord2 →** *Solo se Ejecuta ord2 si la ejecución de ord1 ha tenido éxito.*

**ord1 || ord2 → Solo se Ejecuta ord2 si la ejecución de ord1 no ha tenido éxito.**

**LA ORDEN EXPR (CÁLCULO NUMÉRICO Y COMPARACIONES):**

Esta orden sirve para evaluar expresiones, su uso más simple es para hacer operaciones aritméticas sencillas y en menor medida para manipular cadenas.

- **expr** argu1 operador argu2 [argu3 operador argu4 ]....

Los Operadores aritméticos son:

- + → Suma argu1 y argu2
- - → Resta argu1 y argu2
- * → Multiplica argu1 y argu2
- / → División entera.
- % → Resto de la división entera.

Si hay varios operadores, la suma y la resta se evalúan en último lugar, a no ser que vayan entre paréntesis.

Hay que recordar que los símbolos paréntesis y asterisco son especiales del lenguaje, por lo que para poderlos usar hay usarlos escapados (con el símbolo \ delante) o encerrados entre comillas simples.

Los Operadores Relacionales son:

- = → ¿Son iguales los argumentos?
- != → ¿Son distintos los argumentos?
- > → ¿Es argu1 mayor que argu2?
- < → ¿Es argu1 menor que argu2?
- >= → ¿Es argu1 mayor o igual que argu2?
- <= → ¿Es argu1 menor o igual que argu2?

Estos operadores devuelven 0 (true) si el resultado de la comparación es cierto y 1 (false) si el resultado es falso.

Los símbolos > y < tienen significado especial para el Sistema (redireccionamientos) así que hay que escaparlos o entrecomillarlos.

Los operadores relacionales se usan para comparar operandos y tomar decisiones en función de los resultados.

Se pueden usar ordenes cálculos más complejos que usen varios comandos expr.

*Otros operadores que se pueden usar con expr:*

- \| → *Devuelve el primer argumento si ninguno de los dos, es nulo o 0, en caso contrario devuelve el segundo argumento.*
- \& → *Devuelve el primer argumento si ninguno de los dos, es nulo o 0, en caso devuelve 0.*

La orden expr se puede usar interactivamante en la línea de comandos del shell como una pequeña calculadora para pequeños cálculos:

```
$ expr 5 + 8
  13
$ expr 10 \* 2      ←   Hay que usar \* para que el asterisco pierda su significado.
  20
$ expr 10 '*' 2     ←   Funciona igual
 20
$ expr 10 * 2       ←   Error
```

La orden expr se usa normalmente en programación para asignar variables, para que la variable reciba el valor de la expresión, esta ha de estar entre acentos graves para que la orden se ejecute y devuelva el valor hacia la variable:

```
$ valor =  ` expr 500 / 25 `
```

```
$ precio_iva= `expr 1200 \*   \( 1 + 16 / 100 \)`
```

El ejemplo anterior no produce el resultado deseado, ¿a qué es debido?.

Si necesitamos cálculos complejos hay que usar la orden bc que en realidad es en sí mismo un lenguaje de programación que admite muchas más posibilidades de cálculo numérico.

## ESTRUCTURAS DE CONTROL DEL SHELL:

### Alternativa if:

```
if    condicion
      then
            orden_1
            orden_2
      [elif condicion2
      then
            orden_3
            orden_4
            ......]
      [else
            orden_5
            orden_6
             .....]
fi
```

- Se pueden anidar sentencias if.
- La construcción elif es una forma abreviada de anidar if cerrándolos con un solo fi. La estructura resultante es una alternativa múltiple similar a un case.
- Se puede usar el *then* en la misma línea que el *if* o *elif* de la siguiente forma:

```
if    condicion; then

            orden_1
            orden_2
            .......

[elif condicion2; then

            orden_3
            orden_4
            ......]
      [else
            orden_5
            orden_6
             .....]
fi
```

- Los if evalúan la condición (generalmente expresiones con la sentencia test, aunque puede ser cualquier comando) y ejecuta las ordenes correspondientes si son ciertas (valor 0).

### Bucles while y until:

```
while condición
do
      orden_1
      orden_2
      .......
done
```

```
while condición; do
      orden_1
      orden_2
      .......
done
```

- El bucle se ejecuta mientras que el valor de retorno de la condición sea True (valor 0).

```
until condición
do
      orden_1
      orden_2
      .......
done
```

```
until condición; do
      orden_1
      orden_2
      .......
done
```

- El bucle se ejecuta hasta que el valor de retorno de la condición sea True (valor 0).

NOTAS:

---

- Los bucles *while* y *until* evalúan la condición al principio (primero evalúan y después procesan el contenido) por lo tanto se ejecutaran de 0 a N veces.
- Se pueden crear bucles infinitos utilizando los valores **true** o **false** (valores 0 y 1, al contrario del lenguaje C).

```
while true
do
…..
done
```

---

**Bucle for:**

```
for variable in lista_de_valores
do
      orden_1
      orden_2
      .......
done
```

- La variable puede ser cualquier variable del Shell.
- Los posibles valores de lista_de_valores se van asignando una por una a la variable que hayamos definido.
- Por cada valor asignado, se ejecutan las ordenes comprendidas entre do... y...done.

```
for variable
do
      orden_1
      orden_2
      .......
done
```

- Esta construcción del for toma los valores para la variable de los argumentos de la línea de órdenes.
- Otra sintaxis de de bucle (este tipo de bucle for comparten una herencia común con el lenguaje de programación C):

```
For  ((expresión1; expresion2; expresion3))
do
      orden1
      orden2
      orden3
done
```

- La sintaxis anterior se caracteriza por una expresión de control de bucle de tres parámetros; consisten en un inicializador (expresión1), una prueba de bucle o condición (expresión2) y una expresión de conteo (expresion3).

**Alternativa case:**

```
case cadena_a_evaluar in
      patron_1)
            Orden_1
            Orden_2
            ..... ;;
      patron_2)
            Orden_3
            Orden_4
            ..... ;;
      patron_n)
            Orden_n1
            Orden_n2
            .....  ;;
esac
```

- La sentencia case proporciona un mecanismo de bifurcación múltiple. Esta sentencia compara la cadena a evaluar con los sucesivos patrones, uno por uno, hasta que encuentre un patrón que coincida o hasta que ya no queden más patrones que comparar. Si se halla una coincidencia se ejecutan las ordenes correspondientes al patrón y la sentencia case finaliza.
- Es típico en los programas Shell si se necesita, poner como último patrón un comodín que se corresponde con cualquier posible valor de la cadena a evaluar.
- Se pueden usar metacaracteres ( *** . ¿ , [...]** ) dentro de los patrones. Además si se usa el símbolo | para separar 2 patrones equivale a un OR lógico.

**Sentencias break y continue:**

- **break** (Salir de un bucle) Hace que cualquier bucle *while*, *until* o *for* termine su ejecución y ceda el control del programa a la instrucción que se encuentre después del bucle.

- **continue** (Nueva iteración) Detiene la iteración actual de un bucle *while*, *until* o *for* *en el punto donde se encuentre situada* y da lugar a una nueva iteración..

- En ambos casos *break* y *continue*, las instrucciones que hay detrás hasta *done* no se ejecuten.

## Uso select

Select está a medio camino entre una orden lectura de entrada y una de control de flujo. Esta orden select nos mostrara por la salida de errores estándar una lista numera que contendrá todas las palabras después de in (en este caso uno dos tres cuatro). Luego se quedara esperando nuestra elección. Si tecleamos uno de los números correspondientes a alguna de las palabras en la lista, se le asignara dicho valor a la variable y se ejecutaran los comandos.

Si no se especifica nada el prompt del sistema que aparece es #?. Se puede variar con cambiando el valor a la variable del sistema llamada PS3.

La sintaxis del comando es:

```
select nombre [in lista]
do
  comandos que pueden utilizar $nombre
done
```

Un ejemplo simple:

```
#!/bin/bash

select OPCION in opcion_1 opcion_2 opcion_3
  do
  if [ $OPCION ]; then
      echo "Opcion elegida: $OPCION"
      break
  else
      echo "Opcion no valida"
  fi
done
```

Otro ejemplo:

```
#!/bin/bash
select ACCION in Empezar Repetir Acabar
do
  case $ACCION in
"Empezar")
   echo "El usuario quiere empezar"
   ;;
"Repetir")
   echo "El usuario quiere repetir"
   ;;
"Acabar")
   echo "El usuario quiere salir"
break
   ;;
*)
   echo "No sé qué quiere el usuario"
   ;;
esac
done
```

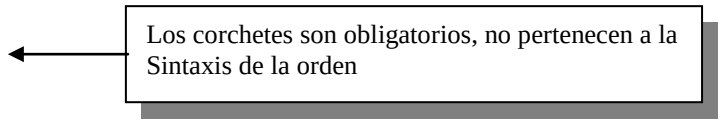
## LA ORDEN TEST (EVALUACIÓN DE ARCHIVOS, CADENAS Y NÚMEROS):

La orden test se usa para evaluar expresiones y generar un valor de retorno, este valor no se devuelve a la salida estándar (pantalla), lo que hace es devolver 0 como código de retorno si la expresión se evalúa como ***true***, y le asigna 1 si la expresión se evalúa como ***false***.

La orden test es de uso típico como condición para estructuras de control del tipo if, while y until.

### Formato de test para archivos:

- test –opcion archivo
- [ -opcion archivo ]




**IMPORTANTE:**

**Si se usa el 2º formato con corchetes, a ambos lados de los mismos debe existir un espacio en blanco**.

Opciones importantes son:

- -f Devuelve verdadero (0) si el archivo existe y es ordinario.
- -s Devuelve verdadero (0) si el archivo existe y tiene un tamaño mayor que 0.
- -r Devuelve verdadero (0) si el archivo existe y tiene permiso de lectura.
- -w Devuelve verdadero (0) si el archivo existe y tiene permiso de escritura.
- -x Devuelve verdadero (0) si el archivo existe y tiene permiso de ejecución.
- -d Devuelve verdadero (0) si el archivo existe y es un directorio.
- -b Devuelve verdadero (0) si el archivo existe y es especial de bloques.
- -c Devuelve verdadero (0) si el archivo existe y es especial de bloques.
- -h Devuelve verdadero (0) si el archivo existe y es un enlace simbólico.  *-e Devuelve verdadero (0) si el archivo existe independientemente del tipo que sea*

### Formato de test para cadenas

- test cadena1 relación cadena2
- [ cadena1 relación cadena2 ]

- test opcion cadena

- [ opcion cadena ]

Las relaciones son:

- = Devuelve verdadero (0) si ambas cadenas son iguales
- != Devuelve verdadero (0) si las cadenas son distintas

Las opciones son:

- -n Devuelve verdadero (0) si cadena tiene un valor (no nulo)
- -z Devuelve verdadero (0) si cadena no tiene valor (es nula).

**Formato de test para cadenas que representen valores numéricos enteros:**

- test valor1 relación valor2
- [ valor1 relación valor2 ]

Las relaciones son:

- -lt Devuelve verdadero (0) si valor es menor que valor2.
- -le Devuelve verdadero (0) si valor es menor o igual que valor2.
- -gt Devuelve verdadero (0) si valor es mayor que valor2.
- -ge Devuelve verdadero (0) si valor es mayor o igual que valor2.
- -eq Devuelve verdadero (0) si valor es igual que valor2.
- -ne Devuelve verdadero (0) si valor es distinto que valor2.

**Operadores And y OR y NOT con cualquier tipo de test:**

-o → OR lógico.

Ejemplo:

*[ -x /usr/miorg -o "$1" -eq 0 ]*

-a → AND lógico

Ejemplo:

*[ -x /usr/miorg -a "$1 -ne 0 ]*

! → NOT lógico.

Ejemplo:

*[ ! -d /etc/usrmail ]*

**Uso de paréntesis para agrupar expresiones:**

Se pueden usar paréntesis para agrupar y dar prioridad a expresiones, al escribirlos hay que poner espacios en blanco antes y detrás de los paréntesis.

Los paréntesis tienen un significado especial para el lenguaje, para poder usarlos hay que escaparlos: \ ( ..... \) o encerrados entre comillas simples.

**TRATAMIENTO AVANZADO DE CADENAS CON EXPR:**

- **expr substr cadena posi numc**

Esta expresión crea una subcadena a partir de cadena referenciada que contiene *num* caracteres a partir de la posición *posi*.

Ejemplos:

*$ palabra=”calamares”*

*$ expr substr $palabra 1 4*
*cala*
*$ expr substr $palabra 5 5*
*mares*
*$ expr substr $palabra 3 4*
*lama*
*$ expr substr $palabra 4 4*
*amare*
*$*

- **expr length cadena**

Esta expresión devuelve el nº de caracteres que tiene la cadena.

Ejemplos:

*$ palabra=”calamares”*

*$ expr length $palabra*
*8*
*$*

- **expr index cadena carácter**

Esta expresión devuelve la posición de la primera aparición del carácter especificado dentro de la cadena.

*$ palabra=”calamares”*

*$ expr index $palabra l*
*3*
*$*

**EJEMPLOS DE USO DE ESTRUCTURAS, TEST y EXPR:**

**echo “Introduce el código del producto \c”**
**read codi**

- El código anterior muestra como entrar datos en la misma línea que la pregunta (El read del Shell Korn dispone de la opción de incluir mensaje). Se pueden usar también los \n y \t.

**echo “Introduce el código y el precio del producto \c”**
**read codi precio**

- El código anterior es un ejemplo de introducción de más de una variable en un read

**conta=0**
**....**
**conta = ` expr $conta + 1 `**

- El código anterior muestra cómo usar un acumulador.

**FECHA=`date`**

```
echo El contenido de la variable \$FECHA  es $FECHA
```

o El código anterior muestra una sustitución del contenido de una orden correctamente.

```
FECHA=date
echo la fecha actual es $FECHA
```

o El código anterior muestra el intento erróneo de sustitución del contenido de una orden.

```
FECHA=date
eval $FECHA
```

- El código anterior muestra un ejemplo de la **orden eval** que sirve para ejecutar el contenido de una variable (no está en los apuntes).

```
echo "Dime tu nombre de usuario  \c"
read USUARIO
PROGRAMAS=" `ps –ef | grep $USUARIO` "
echo "El usuario $USUARIO está corriendo los siguientes procesos:  \n\n
$PROGRAMAS"
```

- El código anterior es un ejemplo de sustitución del contenido de una orden, informa de los programas en ejecución de un usuario introducido por teclado.

```
if  test  $#  -ne  3
then
        echo "No has introducido los 3 argumentos indicados"
        exit 1
else
        total=$1 + $2 + $3
        echo "La suma de $1 $2 y $3 es $total"
fi
```

- El código anterior es un ejemplo de cómo controlar que el nº de argumentos de una orden sea el correcto, el programa pretende sumar 3 valores numéricos pasados como argumentos. El uso de la palabra test se considera anticuado siendo sustituido por los corchetes.

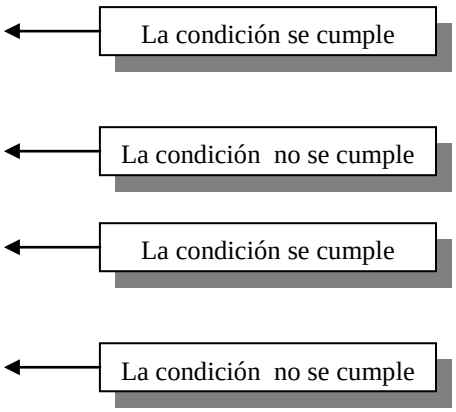
```
if   [ "$#"  -ne  3 ]
then
        echo "No has introducido los 3 argumentos indicados"
        exit 1
else
        total=$1 + $2 + $3
        echo "La suma de  $1 $2 y $3 es $total"
fi
```

- El código anterior es una versión con corchetes del test. Es buena costumbre encerrar todas las variables entre comillas.

```
if   [ "$#"  !=  3 ]
then
        echo "No has introducido los 3 argumentos indicados"
        exit 1
else
        total=$1 + $2 + $3
        echo "La suma de  $1 $2 y $3 es $total"
fi
```

- El código anterior no es muy recomendable, ya que usa la orden test para comparar cadenas, en este caso funciona correctamente.  Observar el caso siguiente:

```
dato1="6 "        ←   Hay un blanco detrás de la cifra 6
dato2=0005
if [ "$dato1" –eq 6 ]
then…….        else…..
fi
if [ "$dato1" = 6 ]
then…….        else…..
fi
if [ "$dato2" -eq 5 ]
then…….        else…..
fi
if [ "$dato2" = 5 ]
then…….        else…..
fi
```




- En el ejemplo anterior se ejecuta el then del if 1º y 3º, pero en los if 2º y 4º se ejecuta el else.

```
echo "Dime un nombre para el directorio \c"
read DIRE
if `mkdir $DIRE`
then
        echo "El directorio $DIRE se ha creado correctamente"
else
        echo "No se ha podido crear el directorio $DIRE "
fi
```

- El código anterior sirve para comprobar si se ha creado un directorio, mediante una sustitución de orden en la condición del if.

  Si la instrucción se ejecuta devuelve el codigo de estado 0 que es true, por lo que se ejecuta la condición then del if.

  El problema de esta construcción es que la orden mkdir muestra la salida estándar de error por pantalla cuando no se puede ejecutar.

```
echo "Dime un nombre para el directorio \c"
read DIRE
if [ -d "$DIRE" ]
then
        echo "El directorio $DIRE ya existe
else
mkdir $DIRE
        echo "Se ha creado el directorio $DIRE "
fi
```

- En el ejemplo anterior se hace uso de la opción –d del test para saber si el nombre introducido ya existe y es un directorio. Es mucho mejor que el otro.

  Además se ve la importancia de usar comillas en las variables:

  ```
  if [ -d $DIRE ]
  ……..
  ```
  En el caso de que $DIRE no existiese, el if quedaría como sigue:

  ```
  if [ -d ]
  .........
  ```
  Algo que seguramente al intérprete del shell no él va a gustar nada.

```
echo "Dime un nombre para el directorio \c"
```

```
read DIRE
mkdir $DIRE  >2  /dev/null
if [ $?  -eq  0 ]
then
        echo “El directorio $DIRE se ha creado correctamente”
else
        echo “No se ha podido crear el directorio $DIRE “
fi
```

- Este último código es como...más fino, usa la variable del código de estado $? Para comprobar si la orden se ha ejecutado correctamente.

```
echo “Dime un  nombre para el directorio  \c”
read DIRE
mkdir $DIRE  >2  /dev/null
ESTADO=$?
if [ $ESTADO  -eq  0 ]
then
        echo “El directorio $DIRE se ha creado correctamente”
else
        echo “No se ha podido crear el directorio $DIRE “
fi
```

- El ejemplo es similar capturando la variable de estado previamente en una variable. Hay que hacerlo inmediatamente después de la orden que se quiere comprobar ya que si no obtendremos resultados inesperados:

```
echo “Dime un  nombre para el directorio  \c”
read DIRE
mkdir $DIRE  >2  /dev/null
echo “Creando el directorio $DIRE”
if [ $?  -eq  0 ]
then
.....
fi
```

Este ejemplo no hace lo que pretendemos ya que el código de estado va a devolver siempre 0 ya que corresponde a la orden echo anterior que es correcta.

```
nombre=”juan”
apellido=”nadie”
[ “$nombre” = “$apellido”
] echo $? [ “$nombre” =
“juan” ] echo $?
```

- En el ejemplo anterior observamos que no es necesario usar los test en un if. Las dos líneas del echo con el código de estado imprimirán 1 y 0 respectivamente.

```
[ !  -w “$MIFICHERO”  ]  && continue
```

```
[  -r  “$MIFICHERO”  ]  &&  cp  “$MIFICHERO”  /tmp
```

- Los operadores && y || resultan muy útiles para sustituir if de forma que el código queda más compacto (pero menos evidente para programadores inexpertos).

  El primer ejemplo supone que está dentro de un bucle y hace que se produzca una nueva iteración cuando el fichero referenciado por la variable exista y no tenga permiso de escritura.

El 2º ejemplo copia el fichero referenciado en la variable en el directorio /tmp, siempre que exista y tenga permiso de lectura.

```
echo “Introduce un nombre de fichero”
read dato
if [ -f $dato ] then
 echo “ $dato es un fichero ordinario” elif [ -
d $dato ] then echo “ $dato es un directorio”
elif [ -l $dato ] then
        echo “ $dato es un enlace blando”
elif [ -c $dato -o -b $dato ] then
        echo “ $dato es un fichero especial de dispositivo”
else
        echo “No se que demonios es $dato” fi
```

- Construcción if con alternativas múltiples, se evalúan de arriba abajo hasta que se encuentre una que sea cierta y se finaliza el if. En el caso de que no se cumpla ninguna condición se ejecutara las sentencias del else si es que existe.

```
echo “Introduce un nombre de fichero”
read dato
if [ -f $dato ]; then
        echo “ $dato es un fichero ordinario”
elif [ -d $dato ]; then
        echo “ $dato es un directorio”
elif [ -l $dato ]; then
        echo “ $dato es un enlace blando”
elif [ -c $dato -o -b $dato ]; then
        echo “ $dato es un fichero especial de dispositivo”
else
        echo “No se que demonios es $dato” fi
```

- Ejemplo idéntico al previo, pero ahorrándonos talar algún árbol (ocupa menos papel).

```
clear clave=”agente007” echo
“Introduzca la Clave \c” read
respuesta
while [ “$respuesta” != “$clave” ] do
  echo “Error, vuelve a intentarlo”
echo “Introduzca la Clave \c”
read respuesta done
echo “ Puede usted pasar”
```

- Ejemplo de bucle while con test de cadenas.

```
clear clave=”agente007” echo
“Introduzca la Clave \c” read
respuesta
until [ “$respuesta” = “$clave” ] do
   echo “Error, vuelve a intentarlo”
echo “Introduzca la Clave \c”
read respuesta done
echo “ Puede usted pasar”
```

- El mismo ejemplo pero con un until. Recordar que lleva evaluación al principio.

```
clear respu=s
while [ “$respu”=s -o “$respu”=S ] do      echo “Que
cadena quieres buscar \c”; read cadena      echo “En que
```

```
fichero quieres buscarla \c"; read fichero if [ -f
"$fichero" ] then grep "$cadena" "$fichero" >
resultado if [ -s resultado ] then
        echo "La cadena $cadena buscada existe en el fichero $fichero"
else
        echo "La cadena $cadena buscada no existe en el fichero $fichero"
fi
    rm resultado
else
    echo "El fichero $fichero no es un fichero ordinario"
fi
  echo "quieres hacer mas búsquedas (s-n) \c"; read respu
done
```

- El ejemplo anterior es un script que realiza una búsqueda de una cadena en un fichero usando la orden grep. Como esta orden muestra su salida estándar por pantalla, se redirecciona a un fichero llamado respuesta.

```
while true
do
.....
if…… break
done
```

  - Si creamos un bucle infinito hemos de salir de alguna forma, mediante break para saltar a la sentencia posterior al mismo o con exit para finalizar el script.

**EJEMPLOS DE USO DE ESTRUCTURAS, TEST y EXPR (Continuación):**

**Bucles For:**

```
$ cat > bucle for
contador in 1 2 3 do
  echo "Este bucle se esta ejecutando $contador veces"
done
```

  - El código anterior muestra un uso de la estructura for, la variable contador toma los valores de la lista sucesivamente.

```
$ cat > mismensajes
 for puesto in kvsdai01 kvdai02 kvdai03 kvdai04
do
  mail $puesto < micarta
done
```

  - Similar al anterior, envía un mail preescrito en el fichero micarta a 4 usuarios.

```
$ cat > mira
for fichero in *
do
  echo $fichero
done
```

  - En este caso se muestran todos los ficheros del directorio actual, ya que * representa a todos los ficheros.

```
$ cat > cuenta
cuenta=0 for
fichero in *
 do
```

```
    cuenta=`expr $cuenta + 1 `
done
  echo "El nº de ficheros del directorio actual es $cuenta"
```

- En este caso se cuentan todos los ficheros del directorio actual.

```
$ cat > lista    for
listado in ` ls `    do
    echo  $listado
done
```

- El ejemplo anterior no es demasiado útil, pero ilustra como for toma la lista de valores mediante la sustitución de la salida del a orden ls.

```
$ cat > informa
for  info       do
echo        $info
done
```

- Este caso muestra el uso de la estructura for sin poner la lista de valores, entonces toma como valores los argumentos de línea de órdenes.
  El ejemplo lo único que hace es mostrarnos los argumentos que le pasamos al programa en la llamada

```
$ cat > voltea    lista=""
for datos    do      lista =
"$datos $lista"    done
echo $ lista
```

- El ejemplo anterior ¿Qué hace?, si no estaís seguros probarlo de la siguiente manera: *$ voltea ana quiere juan*

```
$ cat > impre
  for fichero in ` find  /usr/kvdai12  –type f   -print  `
do
    lp $fichero   &&   rm $fichero
done
```

- En el código anterior, la lasta de valores la devuelve la orden find que devuelve los archivos de tipo ordinario que hay en /usr/kvdai12
  El programa imprimirá y eliminara todos los archivos referenciados, fijarse la construcción con && para que elimine el fichero solo en el caso de que la orden lp hay actuado correctamente. Este script imprime y elimina todos los ficheros dentro del directorio indicado y subdirectorios.

```
$ cat  > comprime
   if [ "$#"  -eq  0 ]; then
 echo "Se necesita un parámetro que sea el nombre de un directorio"
exit 1
fi
  for midir
  do
     if [ ! –d "$midir"  ]; then
        echo " El argumento $midir no es un directorio"
continue
     fi
     if [ ! –r  "$midir"  -o  -x  "$midir"  ]; then
        echo " El directorio $midir no tiene permiso de lectura o de entrada"
continue
     fi
     for fichero in ` ls $midir `
do
```

```
        if [ –f "$midir/$fichero" ]; then echo
    "Comprimiendo el archivo $fichero.....
    compress "$midir/$fichero" fi
      done
    done
```

o El ejemplo anterior, sirve para comprimir los archivos de los directorios que le pasemos al script como argumentos en la línea de ordenes:

*$ comprime /usr/kvdai11 /usr/ejemplos /usr/ejerc*

La orden para comprimir archivos es:

- **compress nombre_fichero**

Esta orden comprime le fichero referenciado creando un nuevo archivo terminado en **.z** y destruyendo el original. La orden para descomprimir un archivo comprimido con compres es:

- **uncompress nombre_fichero.z**

Esta orden restaura el archivo a su formato original, creando un nuevo archivo sin el sufijo **.z** haciendo desaparecer el comprimido.

**Estructura case:**

```
$ cat > mimenu
 # Estructura básica de un menú echo "
1: mostar fecha y Hora" echo " 2: Listar
el directorio de conexión" echo " 3:
Visualizar el calendario" echo " 0: Salir"
echo "Introduzca opción: \c" read opcion
case $opcion in
0) echo adios ;;
1) date ;;
2) ls $HOME ;;
3) cal ;;
    *) echo "Opcion incorrecta"
esac
```

o En el código anterior vemos el esqueleto básico de un menú mediante el uso de la estructura case.

Observar que el valor para cuando no se cumple ninguna de las opciones es * y se pone como última opción de la estructura case—esac.

```
$ cat > mimenu
 # Estructura básica de un menú echo "
a: mostar fecha y Hora" echo " b: Listar
el directorio de conexión" echo " c:
Visualizar el calendario" echo " d: Salir"
echo "Introduzca opción: \c"
 read opcion case
$opcion in a | A)
date ;; b | B) ls
$HOME ;; c | C)
cal ;; d| D) echo
adios ;;
    *) echo "Opcion incorrecta"
esac
```

o Similar al anterior pero ahora usando letras para el menú y controlando mayúsculas y minúsculas.

El símbolo | equivale a un OR.

```
$ cat > saludo
hora=`date + %H `
case $hora in
   0? |  1[0-1] ) echo “Buenos dias” ;;
         1[2-7] ) echo “Buenas Tardes” ;;
*) echo “Buenas noches”;;    exac
  exit 0
```

## CREACIÓN DE FUNCIONES EN LOS SCRIPTS

El shell Bourne permite definir funciones, estas constan de una serie de órdenes llamadas **cuerpo de la función**, a las cuales se les da un nombre. Las órdenes contenidas en el cuerpo de la función se invocan empleando el nombre de la misma.

Normalmente, se usan funciones cuando hay una parte del código que se repite en distintos lugares del guión. Al convertir este código en guión, se ahorra tiempo de escritura. La desventaja es que el mecanismo de paso de control al código de la función y de devolución del control a código que hace la llamada requiere tiempo, lo cual hace que el tiempo necesario para ejecutar el guión sea aumenta ligeramente.

Otra forma de ahorrar tiempo de escritura consiste en crear otro archivo de guión, y llamar ese código script como si fuese una orden. Con este método se consigue una programación modular, pero es más lento que el del uso de funciones ya que los guiones residen en disco y tienen que pasar a memoria para ejecutarse, mientras las funciones residen permanentemente en la memoria del ordenador.

Para definir una función se nos presentan dos posibilidades:

- Si son de uso general, las podemos colocar en el archivo *.profile* del directorio del usuario, exportándolas para que estén a disposición de todos los procesos hijos del shell actual.

- Si son específicas de un determinado guión, suelen colocarse en el mismo archivo que contiene el guión.

Una función se declara de la siguiente manera:

```
Nombre_funcion()
{
  lista_ordenes
}
```

Las órdenes que hay en el cuerpo de la función no se ejecuten mientras no se invoque a esa función. Las funciones se llaman usando el nombre de la función como si fuese una orden más.

Cuando se llama a una función, se ejecuta el cuerpo de la misma, y el control vuelve a la siguiente orden a la de la llamada a la función.

Para poder llamar a una función esta debe estar previamente definida en el código, así que si se trata de funciones específicas de un guión, deben estar declaradas en la parte superior del mismo (al contrario de otros lenguajes) o en otro lugar pero previamente una posible llamada a las mismas.

La orden set visualiza todas las definiciones de funciones que haya en nuestro entorno actual.

Evidentemente las funciones permanecen en memoria mientras dura la sesión, ya se comentó que existe la posibilidad de guardarlas en el archivo .profile y exportarlas. Otra posibilidad es crear un fichero que contenga declaración de funciones y ejecutarlo como una orden más. Hay que tener cuidado ya que las funciones no se exportan automáticamente y no están disponibles para un subshell de nueva creación.

Es posible pasar argumentos a las funciones definidas en el Shell de la forma que ya conocemos. Dentro de la función se puede acceder a los valores empleando $1, $2, $3, etc..... de la misma forma que se accede a los argumentos de un fichero de comandos del shell. Hay que tener en cuenta que dentro de una función, los argumentos del fichero de comandos del propio shell no están disponibles.

Dentro de una función se puede acceder a las variables del script donde está situada, y el por el contrario las órdenes del script pueden acceder a las variables definidas en la función.

Se pueden devolver un estado de salida ($?) de una función con la palabra reservada **return** y un valor (el 0 significará éxito).

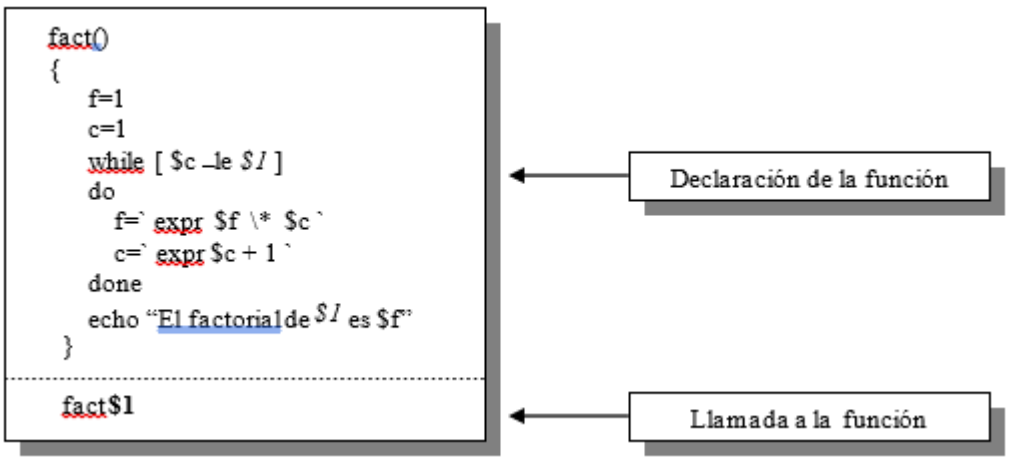
Otra forma de definir el estado de la salida ($?) es usando exit y un valor numérico como como ya conocemos.

Si lo que se desea es usar otra salida, lo normal es hacerlo con echo, y la llamada a la función debe ser de la forma:

$A = `mifunción`

Ejemplo de un factorial (usar una función en este caso es complicar el código, pero es didáctico), el scritp factorial lleva dentro la función fact. Se supone que el programa se ejecuta con un argumento:

$ cat > factorial




Observar que:

- La declaración de la función está en el código antes de la llamada a la misma.
- La llamada a la función incluye la variable posicional $1 que es el argumento de la línea de órdenes del script:

  $ factorial 5
  $ A=` factorial 5 `

- La variable posicional $1 dentro de la función es la correspondiente al argumento de la llamada a la función, no tiene nada que ver con la otra $1 (aunque en este caso toma el mismo valor).

Una forma de comprenderlo es variar el código:

```
fact()
        {
            f=1

        c=1
        while [ $c –le $1 ]
            do

        f=` expr  $f  \*  $c `
        c=` expr $c + 1
            done

            echo “El factorial de $n es $f”
          }
        fact  5
```

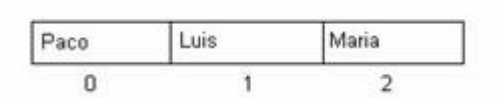
*Cuando se ejecute el Script, el argumento de llamada del mismo, no se ha recogido en ingún lugar del código. Desde la función no se pueden ver directamente las variables posicionales de los argumentos de a línea de ordenes.*

*En este caso el $1 de la función tiene valor nulo ya que no ha recibido ningún valor desde la llamada a la función en el código.*

- En este caso la llamada a la función dentro del código no incluye ningún argumento, alguien podría pensar que se va a pasar los argumentos de la línea de órdenes a la variable posicional de la función, pero no es así y se produce un error.

## ARRAYS

Los **arrays** de los script funcionan de la misma forma que los arrays de cualquier lenguaje de programación. Un array es un conjunto o agrupaciones de valores cuyo acceso se realiza por los índices, en una script se puede almacenar en un mismo array todo tipo de cosas: números, cadena, caracteres, etc.




En los **arrays** el primer elemento que se almacena lo hace en la posición 0 (en el ejemplo sería Paco). En los script no hace falta declarar el tamaño del array, puedes insertar tanto valores como desees. Para declarar un array es:

```
declare –a nombre_array
declare –a nombres
```

- La opción –a sirve para decir que lo que va a declarar es un **array**.
- Para darle valores se puede hacer de los formas:

1. Darle valores posición por posición.

```
nombre_aray[posición]=valor

nombres[3]=manolo
```

2. Darle todos los valores de golpe (aunque también se puede decir la posición deseada en la que quieres guardar uno de los valores)

```
nombre_array=( valor1 valor2 valor3 [posición]=valor4 ….valor n )
```

nombres=(Maria Alberto Rodrigo [7]=Javier )

- Para ver el contenido de un **array** en una posición:

```
${nombre_array[posición]}

${nombres[0]}
```

- Para saber cuántos elementos contiene un **array**:

```
${#nombre_array[*]}

${#nombres[*]}
```

- Para recuperar todos los elementos de un **array**:

```
${nombre_array[*]}

${nombres[*]}
```

A continuación un ejemplo de **arrays**:

arrays.sh

```
#! /bin/bash
clear
contador=0
declare -a usuario=( Alberto John Roberto Laura Sergio Cristian
Dani )
for valor in ${usuario[*]}
do
echo "El usuario $contador vale $valor"
contador=`expr $contador + 1`
done
```

## LEER UN FICHERO

Para leer un fichero es necesario un bucle en la primera línea más un EOF (End Of File) y redireccionar la entrada para ese bucle: while condicion do read linea

comandos…

done < /ruta/fichero

Esto se verá mejor con un ejemplo: Tenemos el siguiente fichero en */tmp/ejemplo_texto* y contiene: En un Lugar de la Mancha de cuyo nombre no quiero acordarme y mucho más….

Vamos a hacer un pequeño script que nos cuente el número de líneas que tiene el fichero, como lo hace wc:

```
LeerFichero.sh
#! /bin/bash
clear
linea=”algo”
while [ ! -z “$linea” ]
do
read linea
num_linea=`expr $num_linea + 1`
if [ ! -z “$linea” ]
then
echo “La linea numero: $num_linea del fichero es… $linea”
fi
done < /tmp/ejemplo_texto
echo “Total lineas: `expr $num_linea - 1`”
```

Y como resultado tendríamos:

```
$ ./LeerFichero.sh
La linea numero: 1 del fichero es… En un Lugar de la Mancha La
linea numero: 2 del fichero es… de cuyo nombre no quiero
acordarme
La linea numero: 3 del fichero es… y mucho más. Total
lineas: 3
```

## COMANDO bc

El comando bc se utiliza como calculadora de la línea de comandos. Es similar a una calculadora básica. Usándolo podemos hacer cálculos matemáticos básicos.

**Sintaxis:**

bc [opciones]

**Opciones:**

| | |
|---|---|
| -c | Sólo compilar. El output son comandos dc que son enviados al salida estándar. |
| -l | Define las funciones matemáticas e inicializa la escala a 20, en vez de al cero por defecto. |
| filename | Nombre del archivo que contiene los comandos básicos de cálculo, éste no es un comando necesario. |

- `bc` se incluye en las distribuciones Linux como estándar, así como en Unix.
- Los resultados de cálculos en algunos formatos propietarios de `bc` tienen hasta 99 dígitos decimales antes y después del punto decimal. Este límite se ha superado mucho en GNU.
- Lo útil sobre `bc` es que acepta como entrada desde ficheros y desde entrada estándar.

La mayoría de estos ejemplos les sigue una simple fórmula.

## Suma

```
$ echo '57+43' | bc
100 Resta

$ echo '57-43' | bc
14
```

## Multiplicación

```
$ echo '57*43' | bc
2451
```

## scale

La variable `scale` determina el número de dígitos que siguen al punto decimal en tu resultado. Por defecto, el valor de la variable `scale` es cero. (Salvo que usemos la opción `-l` en cuyo caso por defecto vale 20 posiciones decimales.) Esto se puede configurar declarando `scale` antes de hacer los cálculos, como en el siguiente ejemplo de división:

```
$ echo 'scale=25;57/43' | bc
1.3255813953488372093023255
```

## Raíz cuadrada

```
$ echo 'scale=30;sqrt(2)' | bc
1.414213562373095048801688724209
 Pote ia

$  echo
```

## '6nc

```
^6' | bc
46656
```

## Paréntesis

Uso paréntesis (brackets) para separar los diferentes componentes de mis sumas siempre que es posible, elimina cualquier posible duda de que obtenga una respuesta errónea. Considera los siguientes cálculos:

```
$ echo '7+(6*5)' | bc
$ echo '7+6*5' | bc
$ echo '6*5+7' | bc
```

Todos dan la misma respuesta, 37, pero habría escrito el primer cálculo, salvo por supuesto, si quisiera decir:

```
$ echo '(7+6)*5' | bc
```

O ponerlo de otra forma:

```
$ echo '13*5' | bc  lo que da 65.
```

## obase e ibase

`obase` e `ibase` son variables especiales que definen la base de entrada y de salida. Justifica `obase` valores entre 2 y 999, a pesar de que nada mayor que 16 vale la pena para mí! Justifica `ibase` valores entre 2 y 16. Algunos ejemplos lo explicarán mejor.

## Convierte de decimal a hexadecimal

Aquí estamos convirtiendo 255 de base 10 a base 16:

```
$ echo 'obase=16;255' | bc
FF
```

## Convierte de decimal a binario

Y aquí estamos convirtiendo el número 12 de base 10 a base 2:

```
$ echo 'obase=2;12' | bc
1100
```

## Convierte de binario a decimal

Aquí estamos convirtiendo el número binario 10 a base 10 (decimal).

```
$ echo 'ibase=2;obase=A;10' | bc
2
```

Nótese que la `obase` es "`A`" y no "`10`". Lo siento, tenías que aprender algo de hexadecimal. La razón es que si hemos configurado la `ibase` a "`2`", y ahora intentamos usar "`10`" como valor para la `obase`, nos quedará en "`2`", porque "`10`" en base 2 es "`2`". Por lo que necesitamos usar hexadecimal para "salir" del modo binario.

```
$ echo 'ibase=2;obase=A;10000001' | bc
129
```

## Convierte de hexadecimal a decimal

```
$ echo 'ibase=16;obase=A;FF' | bc
255
```

## Uso de bc con scripts del shell

Podemos usar variables con `bc`, lo que es muy útil en los scripts:

```
$ FIVE=5 echo
"$FIVE^2" | bc
25
```