U4 Administración de Sistemas Operativos Linux

Parte IV. Scripts

Implantación de Sistemas Operativos

Funciones

- Las funciones permiten agrupar varias líneas de código y reutilizarlas, por lo que su uso es conveniente si empleamos scripts con muchas líneas de código o con elementos repetitivos
- La sintaxis que empleamos es la siguiente:

```
nombre_de_la_funcion(){
    instrucciones #echo “soy una función”
}
```

- Invocamos a la función (debe haberse definido antes) en el script con: nombre_de_la_función (ej. 34)

```
#!/bin/bash
# Aquí sólo estamos definiendo la función,
# no se ejecutará hasta que se la llame
funcion () {
    echo "Soy una función"
}
 echo "Vamos a llamar a la función..."
funcion
```

Funciones

- El código de las funciones se pone al principio, antes de los comandos del script.
- Normalmente se le da un nombre representativo de lo que hace.
- Igual que un script puede recibir parámetros y se accede a ellos de forma idéntica $1, $2, $3, $* para la lista de parámetros, $# para el número de parámetros, etc.
- La función tendrá acceso a los parámetros que le pasemos a ella directamente y no tendrá acceso a los parámetros que se le hayan pasado al programa principal. (ej. 35)

```
#!/bin/bash
# Aquí sólo estamos definiendo la función,
# no se ejecutará hasta que se la llame
funcion () {
        echo "He recibido $# parámetros"
        echo "Parametro 1: $1"
        echo "Parametro 2: $2"
}
funcion “Hola" “Adios"
```

Funciones

- Las funciones tienen una instrucción **return** similar al **exit** en el script, sale de la función devolviendo un valor entre 0 y 255. Normalmente con valor 0 si la ejecución ha sido correcta y entre 1 y 255 si ha sido incorrecta.

- Cuando la función termina se accede al valor que ha devuelto mediante la variable **$?** (ver ej. 36)

- Si lo último que ejecuta la función es un comando, y la salida de la función va a depender de si el comando falla o no, podríamos ahorrarnos la instrucción return, ya que la salida de ese comando se guardará en la variable global **$?** igualmente. (ver ej. 37 y 38)

Funciones

Ej. 36

```
#!/bin/bash
# Esta función comprueba si un archivo existe.
# Si existe devuelve 0->Verdadero, y si no 1->Falso
existe () {
        if [[ -e $1 ]]
        then
                return $CORRECTO
        else
                return $ERROR
        fi
}
existe "archivo1.txt"
 # Comprobamos el valor devuelto por la función
if [[ $? -eq $CORRECTO ]]
then
        echo "El archivo existe."
else
        echo "El archivo NO existe."
fi
```

Funciones

Ej. 37 bien (salida 0)

```
#!/bin/bash
# Función que crea un fichero
crear_fichero () {
        touch $fichero
}

read -p "Dime nombre de fichero: "
fichero
if  crear_fichero
then
            echo "Fichero $fichero creado";
exit 0;
else
        echo "Error en función
crear_fichero"; exit 1;
fi
```

Ej. 38 mal (salida distinta de 0)

```
#!/bin/bash
# Función que crea un fichero
crear_fichero () {
        touchs $fichero
}

read -p "Dime nombre de fichero: "
fichero
if  crear_fichero
then
            echo "Fichero $fichero creado";
exit 0;
else
        echo "Error en función
crear_fichero"; exit 1;
fi
```

Funciones

- Por defecto todas las variables que usemos en una función son **globales**. Eso quiere decir que cuando la función termine seguirán existiendo en el script.
- Para indicar que una variable es **local** a la función, pondremos la palabra **local** o **declare** delante de la misma la primera vez que le demos valor. De esta forma la variable dejará de existir cuando acabe la función. (Ej. 39)

```
#!/bin/bash
suma () {
        local num1=$1
        declare num2=$2
        let "resultado = $num1+$num2"
}
suma 4 6
# num1 y num2 no existen fuera de la función al ser locales
# así que no los mostrará por pantalla. Resultado sin embargo
# no ha sido definida como local, así que estará accesible desde fuera.
echo "$num1 + $num2 = $resultado"
```

Arrays o vectores

- Un vector a array es una variable que en lugar de contener un solo valor, contiene varios. Se define igual que una variable normal, pero encerrando los valores entre paréntesis y separados por espacios.

- ***miarray=(a b Cambiado d)*** *(sin espacios en el igual)*

- Para acceder a los elementos del array:
  - *miarray[0] → primer valor "a"*
  - *miarray[2] → tercer valor "Cambiado"*

  - *${miarray[*]} → todos los valores "a b Cambiado d"*
  - *${miarray[@]} → todos los valores "a b Cambiado d"*

  - *${#miarray[*]} → numero de elementos del array 4*
  - *${#miarray[@]} → numero de elementos del array 4*

Arrays o vectores

- Los índices del array van desde **0** hasta **numero de valores-1**. En este caso de 0 a 3
- Para ver los índices de un array: echo ${!miarray[@]}

Ejemplos:

```
#!/bin/bash
# Vamos a mostrar los valores del vector por pantalla
echo "${miarray[@]}“
echo "${miarray[*]}“

#!/bin/bash
vector=( uno dos tres cuatro cinco seis )
# Vamos a mostrar el número de elementos del vector por pantalla
echo "Numero de elementos: ${#vector[@]}“
echo "Numero de elementos: ${#vector[*]}"
```

Arrays o vectores

- Eliminar un elemento del array: **unset miarry[1]**
- Cuando eliminamos un elemento del array su posición queda vacía.

INDICE	0	1	2	3
VALOR	a		Cambiado	d

- Podemos añadir elementos al vector o modificar elementos existentes.

  miarray[1]=Hola

  miarray[4]=f

INDICE	0	1	2	3	4
VALOR	a	Hola	Cambiado	d	f

- Podemos asignar un valor al array dejando espacios en blanco, aunque no se aconseja.

  miarray[7]=g

INDICE	0	1	2	3	4	5	6	7
VALOR	a	Hola	Cambiado	d	f			g

Arrays o vectores

Ej. 40

```
#!/bin/bash
vector=( a b Cambiado d )
echo "Primer elemento: ${vector[0]}"
echo "Segundo elemento: ${vector[1]}"
echo "Tercer elemento: ${vector[2]}"
echo "Ultimo elemento: ${vector[3]}"
echo "Elemento no existente: ${vector[5]}“
```

Ej. 41

```
#!/bin/bash
ficheros=( `ls` )
echo ${ficheros[*]}
echo “Hay ${#ficheros[*]} ficheros
```

Arrays o vectores

Como recorrer un array **FOR_IN**:

Ej 42

```
#!/bin/bash
vector=( uno dos tres cuatro cinco seis )

for valor in ${vector[@]}
do
        echo "$valor "
done

echo

for i in ${!vector[@]}
do
        echo "${vector[$i]}"
done
```

Arrays o vectores

Como recorrer un array **FOR**:

Ej 43 ¿por qué ponemos i < ${#vector[*]}, en vez de <=?

```
#!/bin/bash

vector=( uno dos tres cuatro cinco seis )

# De esta manera debemos saber que hay 6 elementos en el vector
for (( i = 0; i < 6; i++ ))
do
	echo "Elemento ${i}: ${vector[$i]} "
done

# Aquí no hace falta saber cuantos elementos tenemos
for (( i = 0; i < ${#vector[*]}; i++ ))
do
	echo "Elemento ${i}: ${vector[$i]} "
done
```

Arrays o vectores

- Podemos llenar el array con la salida de un comando, los espacios y saltos de línea actuaran como separadores de elementos:
  - array=($(cat /etc/passwd | cut -d ':' -f 1))
  - size=(`ls -l | tail +2 | tr -s ' ' | cut -d ' ' -f5`)

- Puede que necesitemos no utilizar el espacio como separador de elementos, sino solamente el salto de línea. Esto es útil si queremos guardar valores de un archivo.
- Por ejemplo (ej. 44):

```
#!/bin/bash
IFS=$'\n'
a=($(cat /etc/passwd | cut -d ':' -f1))
echo "${a[*]}"
```

El salto de línea solo es efectivo para la opción con * y “”

- IFS=$'\n' → Si no ponemos dólar delante no lo interpreta como salto de línea, sino de forma literal.
- Ejecutad el script con salto y sin salto de línea, para ver la diferencia.

Expresiones Regulares

- Las expresiones regulares muchas veces hacen referencia al carácter anterior, ya sea letra o número.
- Sirven para identificar un patrón dentro de un texto, por ejemplo si queremos buscar o identificar un número de teléfono, un correo electrónico, un DNI, etc.

Símbolo	Significado
*	0 o más repeticiones del valor anterior
.	Un carácter cualquiera, pero únicamente 1
?	Una o ninguna repetición del carácter anterior.
+	El carácter anterior se repetir una o más veces.
[]	Indica una lista de caracteres que se pueden dar.
{}	Nos permite especificar el número de repeticiones.
^	La línea comienza por ...
[^...]	Todo lo que NO sea ...
$	La línea termina por ...

Expresiones Regulares

- Veremos su uso con **grep –E** o **egrep** (son equivalentes) y con el comando **sed.**
- En muchos casos hay que entrecomillar la expresión buscada ‘ ‘.

- **. (punto):** da igual lo que ponga en su lugar siempre y cuando sea un solo carácter. Incluye el salto de línea.
  Por ejemplo rXXot no es seleccionado, pero si lo es rXot.

```
ad@SALA_PROF_1:~$ cat pruebas
root
Xroot
rootX
rXot
rXXot
rXXXot
XrXot
XrXotX
```

```
ad@SALA_PROF_1:~$ grep -E -w r.ot pruebas
root
rXot
```

Expresiones Regulares

- *** (asterisco):** la letra precedente puede estar o no, y si está, puede estar las veces que quiera.

```
root
rroot
Xrroot
rrrrroot
Xrrrrroot
rrootX
rrrrrootX
oot
```

```
ad@SALA_PROF_1:~$ grep -E -w r*oot pruebas
root
rroot
rrrrroot
oot
```

En este caso se seleccionará todo lo que empiece por una o más “r”, o por ninguna “oot”

Expresiones Regulares

- **? (interrogante):** la letra precedente se repetirá una o ninguna vez.

```
root
rroot
Xrroot
rrrrroot
Xrrrrroot
rrootX
rrrrrootX
oot
```

```
ad@SALA_PROF_1:~$ grep -E -w r?oot pruebas
root
oot
```

En este ejemplo NO se selecciona “rroot”, por ejemplo, pero si “root” y “ott”.

- **+ (mas):** la letra precedente se repetirá una o más veces.

```
root
rroot
Xrroot
rrrrroot
Xrrrrroot
rrootX
rrrrrootX
oot
```

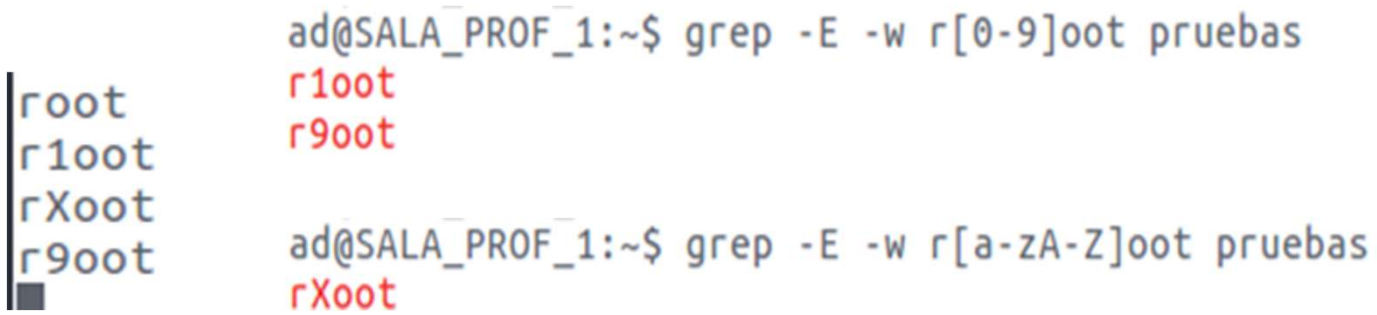
```
ad@SALA_PROF_1:~$ grep -E -w r+oot pruebas
root
rroot
rrrrroot
```

En este ejemplo NO se selecciona “oot”, porque la “r” tiene que estar al menos una vez, pero si selecciona “root”, “rrott”, “rrrrroot”, etc.

Expresiones Regulares

- **[] (corchetes):** lista de valores que se pueden dar.

  Números [0-9]. Letras [a-zA-Z]



- **{} (llaves):** normalmente tiene 3 utilidades:

  - {N} Especificar exactamente el número de repeticiones
  - {N,} Se repite N o mas veces (fijaos en la coma)
  - {N,M} se repite entre N (mínimo) y M (máximo) veces, ambos incluidos.

Expresiones Regulares

- {} (llaves): **{N}** Especificar exactamente el número de repeticiones

```
648293923
965482938
728394022
+34648920103
HOLA
1234
43219
A123456789A
```

```
ad@SALA_PROF_1:~$ grep -E -w [0-9]{9} pruebas
648293923
965482938
728394022
```

En este ejemplo localizaremos los patrones de 9 dígitos

- {} (llaves): **{N,}** Se repite N o mas veces (fijaos en la coma)

```
contrasenya1234
1234contrasenya
1234
password_1234
miPassword1234
4321
12345678
pass
```

```
salvu@mint:~$ grep -E -w '[0-9a-zA-Z]{8,}' pruebas
contrasenya1234
1234contrasenya
miPassword1234
12345678
```

El siguiente ejemplo localiza una contraseña que contiene letras o número (no signos) y además debe tener como mínimo 8 caracteres. Fíjate en que se debe **entrecomillar** en esta ocasión, puedes entrecomillar siempre ante la duda.

Expresiones Regulares

- {} (llaves): **{N, M}** se repite entre N (mínimo) y M (máximo) veces, ambos incluidos.

```
contrasenya1234
1234contrasenya
1234
password_1234
miPassword1234
4321
12345678
pass
```

```
salvu@mint:~$ grep -E -w '[0-9a-zA-Z]{8,14}' pruebas
miPassword1234
12345678
```

Es el mismo ejemplo anterior, pero en este caso la contraseña debe tener entre 8 y 14 caracteres.

- **^ (acento circunflejo):** la línea empieza por la expresión regular indicada.

```
contrasenya1234
1234contrasenya
1234
password_1234
miPassword1234
4321
12345678
pass
```

```
salvu@mint:~$ grep -E -w '^[0-9]+.*' pruebas
1234contrasenya
1234
4321
12345678
```

Buscamos una contraseña que empiece por número (^), los números se pueden repetir una o más veces (+), seguida de cualquier cosa (.*).

Expresiones Regulares

- **^ (acento circunflejo):** la línea empieza por la expresión regular indicada.

```
contrasenya1234
1234contrasenya
1234
password_1234
miPassword1234
4321
12345678
pass
```

```
salvu@mint:~$ grep -E -w '[^0-9]*' pruebas
pass
```

Cuando ponemos **"^" dentro de los corchetes** actúa como una negación. En el ejemplo se selecciona las cadenas de texto que NO tengan en ningún momento números (*).

- **$ (dólar):** la línea termina por la expresión regular indicada.

```
contrasenya1234
1234contrasenya
1234
password_1234
miPassword1234
4321
12345678
pass
```

```
salva@profesor:~$ grep -E '1234$' pruebas.txt
contrasenya1234
1234
password_1234
miPassword1234
```

Indicamos que el texto termine con 1234

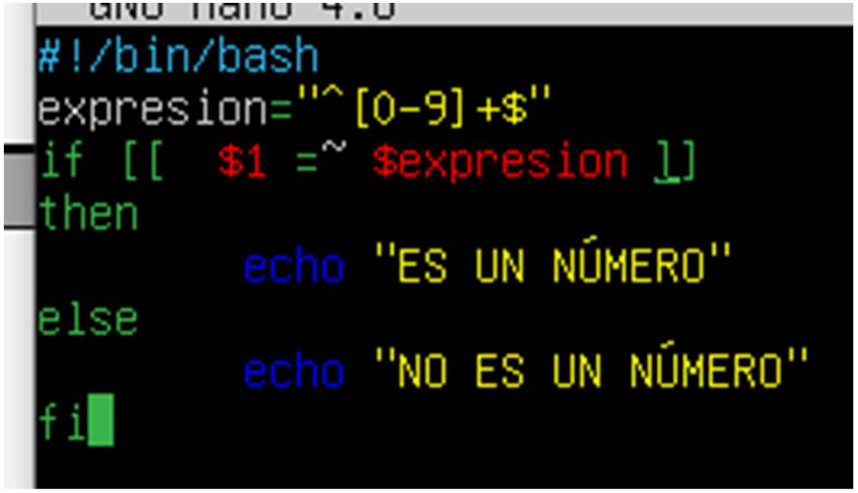
Expresiones Regulares

- Hay ciertas expresiones regulares que se suelen combinar con otras. Ejemplos habituales:
  - **.* (punto asterisco)** → **Cualquier cosa**. El punto se puede sustituir por cualquier carácter y el asterisco que se repite ese carácter 0 o más veces.
  - **^$ (acento circunflejo dólar)** → **Línea en blanco**. Estamos indicando que empiece y acabe sin nada en medio.
  - **^.*$** → **Línea con lo que quieras en medio.**
  - **^xxxxx$** → **Línea con exactamente xxxxx**

Expresiones Regulares

- Uso de expresiones regulares dentro de comparaciones.

Muchas veces en scripts tenemos que ver si cierta variable cumple ciertas expresiones regulares. Para ello tenemos que utilizar **"=~" (igual virgulilla)**



En este ejemplo estamos comprobando si el argumento pasado al script es un número o no.

Pasos en la comprobación:

1. Asignar la expresión regular que vamos a usar a una variable.
2. Hacer uso de los dobles corchetes [[…]].
3. Hacer uso del comparador especial =~ para expresiones regulares.

Expresiones Regulares

- Algunos caracteres especiales (punto, asterisco, etc.) deben ir “escapados”, precedidos de la barra '\' si queremos representarlos como caracteres normales y no como caracteres especiales que definen la expresión regular.
  - Caracteres normales → '\/\(\)\{\}\+*\.\[\]’ .
- Por ejemplo, si quiero buscar un número con decimales que vayan separados por punto, tendré que escapar el punto, porque no quiero que tenga la función especial de sustituir cualquier carácter una vez:
  - ‘[0-9]+\.?[0-9]*’

Expresiones Regulares

- Comprobar si un parámetro pasado al script es un DNI

```
#!/bin/bash
expresión="[0-9]{8,9}[A-Z]"

if [[ $1 =~ $expresión ]]
then
        echo "ES UN DNI"
else
        echo "NO ES UN DNI"
fi
```

¿qué pasa si ejecutamos con AAAAA012345678Tksdgkdjagn14574259?
¿cómo se soluciona?

Expresiones Regulares

- Tenemos que poner ^ y $ para que no haya nada por delante ni detrás del patrón que estamos buscando, para que la búsqueda de ese patrón sea exacta. Si no lo ponemos lo que estamos haciendo es buscar ese patrón dentro de una cadena de texto que puede ser más larga. Ej. 45

```
#!/bin/bash
expresión="^[0-9]{8,9}[A-Z]$"

if [[ $1 =~ $expresión ]]
then
        echo "ES UN DNI"
else
        echo "NO ES UN DNI"
fi
```

Expresiones Regulares

- Quedarnos solo con las líneas de un archivo de configuración que no sean líneas en blanco ni comentarios /etc/ssh/sshd_config

```
#!/bin/bash

grep -E '[^^$]' $1 | grep -E '^[^#]'
```

Manipulación de cadenas de texto

Extraer subcadena

- Mediante **${cadena:posicion:longitud}** podemos extraer una subcadena de otra.
- Por ejemplo en la cadena **string=abcABC123ABCabc:**
  - **echo ${string:0}** : abcABC123ABCabc (sin longitud, extrae la cadena entera)
  - **echo ${string:0:1}** : a (desde la posición 0, extrae el primer carácter)
  - **echo ${string:7}** : 23ABCabc (desde la posición 7, extrae la cadena entera)
  - **echo ${string:7:3}** : 23A (desde la posición 7, extrae 3 caracteres)
  - **echo ${string: -4}** : Cabc (atención al espacio antes del menos)
  - **echo ${string: -4:2}** : Ca (atención al espacio antes del menos)

Reemplazar cadena

- Mediante **${cadena/buscar/reemplazar}** podemos reemplazar la primera coincidencia *buscar* por *reemplazar* y con **${cadena//buscar/reemplazar}** reemplazamos todas.
- Por ejemplo en la cadena **string=abcABC123ABCabc**
  - **echo ${string/abc/xyz}** : xyzABC123ABCabc
  - **echo ${string//abc/xyz}** : xyzABC123ABCxyz

Manipulación de cadenas de texto

Borrar prefijo

- Mediante **${cadena#subcadena}** podemos borrar la coincidencia más corta de subcadena desde el principio de la cadena y con **${cadena##subcadena}** la más larga.
- Por ejemplo en la cadena **string=abcABC123ABCabc**
  - **echo ${string#a*C}** : 123ABCabc
  - **echo ${string##a*C}** : abc

Borrar sufijo

- Funciona igual que borrar prefijo pero borra los últimos caracteres con % para la coincidencia más corta o %% para la más larga
- Por ejemplo en la cadena **string=abcABC123ABCabc**
  - **echo ${string%A*c}** : abcABC123
  - **echo ${string%%A*c}** : abc

Ejemplos de manipulación de cadenas

- A partir de la ruta absoluta de un archivo:
  - *myfile=/home/marina/scripts/0_Teoria_ejemplos/prueba.txt*
- Extrae la ruta del archivo
  - *echo"${myfile%/*}" → /home/marina/scripts/0_Teoria_ejemplos*
  - *borra desde la última barra hasta el final (*)*
- Extrae el nombre del archivo
  - *echo "${myfile##*/}" → prueba.txt*
  - *borra la cadena más larga desde el principio hasta /*
- Extrae el nombre sin la extensión
  - *filename="${myfile##*/}" → prueba.txt*
  - *echo ${filename%.*} → prueba*
- Obtén la extensión del archivo
  - *echo ${myfile##*.}*

shift

- Permite desplazar los argumentos que recibe el script un número de posiciones.

1 2 3 4 5
shift
2 3 4 5
shift
3 4 5

- Este desplazamiento puede simplificar el código en algunos casos. Ej 47:

```
#!/bin/bash
echo "lista de argumentos iniciales:   $*"
shift 2
#desplazamos 2 posiciones: en $@ ($*). $1 y $2 ya no están.
echo "lista de argumentos después de hacer shift 2:  $*
```

Ejemplo con argumentos: vicente 15/1/2019 ¿Para qué sirve shift?

wc (word count)

wc /etc/passwd

36 56 1788 /etc/passwd

- Cuenta el número de líneas (36), palabras (56) y bytes (1788) de un fichero. Las opciones permiten especificar una única de las salidas:

  - wc –l fichero: cuentas las líneas
  - wc –w fichero: cuentas las palabras
  - wc –c fichero: cuentas los bytes
  - wc –m fichero: cuentas los caracteres

- Ejemplo 48:

  #!/bin/bash
  echo "El número de usuarios totales que hay en el sistema es `wc –l /etc/passwd`"
  ¿cómo quitamos el nombre del archivo?

Cut

- Se emplea para extraer segmentos de líneas de texto de un fichero de texto o de la entrada estándar. Los parámetros que admite son:
  - -c rango: obtiene el rango indicado de caracteres sueltos
  - -f rango: obtiene el rango indicado de campos separados por el delimitador -d
  - -d “delimitador“

- Ejemplos:
  - **cut -c1-5 /etc/passwd** → Obtiene los 5 primeros caracteres de cada línea de texto del fichero.
  - **Obtén los usuarios del sistema y sus uid’s:**
    - **cut -d ':' -f1,3 /etc/passwd** → Obtiene los textos de las columnas 1 y 3 de cada línea del texto de entrada, tomando como separador de columna el carácter ':’. La salida serán los usuarios y sus identificadores

tr

- Elimina o reemplaza caracteres de una cadena de entrada con las opciones siguientes:
  - -d cadena1: elimina las ocurrencias de cadena1 en el texto de entrada
  - -s cadena1 cadena2: sustituye las ocurrencias de la cadena1 por la cadena2. Si hubiera varias cadenas 1 seguidas, sólo sustituye 1.

- Ejemplo:
  - **ls –l | tr –d " "** → Elimina los espacios en blanco
  - **ls -l | tr -s "[az]" "[AZ]"** → Muestra por pantalla el resultado del comando ls -l, pero cambiando todas las letras por mayúsculas.

- Ejemplo: obtén el tamaño de los elementos que contiene tu directorio empleando ls
  - **ls –l | tr –s ' ' | cut –d ' ' –f5**

bc (basic calculator)

- Consiste en una calculadora que facilita algunas tareas:
  - Ejemplo 50: script que calcula un promedio de 3 números y devuelve el resultado con 2 decimales:

```
#!/bin/bash
echo “scale=2;  ($1+$2+$3)/3” | bc
```

  - Ejemplo 51: script que calcula la raíz cuadrada de un número con 20 decimales:

```
#!/bin/bash
squareroot=$(echo “scale=20;  sqrt($1)” | bc )
echo $squareroot
```