Trabajo Práctico 3: "Introducción a los sistemas operativos"

Facultad de Informática, UNLP.
Alumno: Gonzalez, Joaquín Manuel.
Fecha de realización: 18/09/2023.

Contenido

1. ¿Qué es el Shell Scripting? ¿A qué tipos de tareas están orientados los script? ¿Los scripts deben compilarse? ¿Por qué?

- **Shell Scripting:** Es la práctica de escribir secuencias de comandos o scripts que están destinados a ser ejecutados por un intérprete de comandos/Shell. Los scripts de Shell están compuestos de comandos que normalmente se ejecutarían desde una Shell de forma interactiva, pero se agrupan en un archivo para **automatizar tareas o secuencias de tareas más complejas.**

  Tipos de tareas a las que están orientados los Scripts:

  - **Automatización de Tareas del Sistema, de Desarrollo, Tareas Repetitivas, de Seguridad.**
  - **Procesamiento de Archivos y Datos.**
  - **Gestión de Usuarios y Permisos.**
  - **Interacción con Redes y Servidores, Bases de Datos, APIs y Servicios Web.**
  - **Generación de Informes y Logs.**
  - **Configuración de Ambientes de Desarrollo.**

  Los scripts no deben compilarse, estos son interpretados por la Shell en tiempo de ejecución, es decir, el código es leído línea por línea y ejecutado de inmediato por la Shell, las razones por las cuales no necesitan ser compilados son:

  - **Interpretación en tiempo real:** El Shell los interpreta en tiempo real permitiendo una rápida iteración y prueba de código.
  - **Portabilidad:** Los sripts suelen ser independientes del hardware y el sistema operativo ya que el Shell se encarga de adaptarlos a cada entorno.
  - **Facilidad de Edición:** Se puede modificar un script directamente en un editor de texto y ejecutarlo sin pasar por un proceso de compilación.
  - **Menor sobrecarga de Desarrollo:** Al no ser necesaria la compilación y enlazamiento del código antes de ejecutarlo, el proceso de desarrollo y prueba se vuelve más ágil.

2. Investigar la funcionalidad de los comandos echo y read

- **Comandos:**
  - ❖ **"echo":** Su función principal es imprimir texto o variables en la salida estándar del sistema.

    Parámetros:
    - ➢ **"-n":** Evita que se añada un salto de línea automático al final del texto.
    - ➢ **"-e":** Habilita la interpretación de escape o saltos de línea usando **"\n (salto de línea) ó \t (tabulación)".**
    - ➢ **"-E":** Deshabilita la interpretación de escape o saltos de línea.
    - ➢ Es común usar este comando con **Redirecciones de Salida (> ó >>).**
  - ❖ **"read":** Se utiliza para leer entradas del usuario desde la línea de comandos directamente desde el teclado mientras se ejecuta un script, **la entrada se almacena en una variable. "read [opciones] variable".**

    Parámetros:
    - ➢ **"-p [prompt]":** Muestra un mensaje antes de solicitar la entrada.
    - ➢ **"-s":** Habilita el modo silencioso, es decir, la entrada del usuario no se va a mostrar en la pantalla.
    - ➢ **"-[n]":** Especifica el número máximo de caracteres, ese número se tiene que especificar en **n.**
    - ➢ **"-t [n]":** Especifica un tiempo límite de espera en segundos para ingresar la entrada. Se debe especificar la cantidad de segundos en **n.**
    - ➢ **"-r":** Desactiva la interpretación de barras invertidas **\.**

a. ¿Cómo se indican los comentarios dentro de un script?

- **Se pueden indicar de 2 formas distintas:**
  - ❖ **Comentario de una línea:**

```
joacogoonz@Debian:~$ #Esto es un comentario
```

- **Comentario de muchas líneas:**

```
joacogoonz@Debian:~$ <<TITULO_COMENTARIO
> Esto es de muchas
> Líneas
> TITULO_COMENTARIO
```

b. ¿Cómo se declaran y se hace referencia a variables dentro de un script?

- **Formas de nombrar Variables:**
  - Deben comenzar con una letra o un guión bajo.
  - Pueden contener letras, números y guiones bajos.
  - Los nombres de las variables son case sensitive.

  Declaración:
  - Para la declaración se utilizan las normas de nombramiento anteriores y la forma de darles valor es la siguiente: **mi_variable=valor.** No se deben dejar espacios entre el signo **"=".**

  Referencia:
  - Para hacer referencia a una variables y obtener su valor se debe utilizar el símbolo **"$"** seguido del nombre de la variable, por ejemplo, **$mi_variable.**

3. Crear dentro del directorio personal del usuario logueado un directorio llamado practica-shell-script y dentro de él un archivo llamado mostrar.sh cuyo contenido sea el siguiente:

```
#!/bin/bash
# Comentarios acerca de lo que hace el script
# Siempre comento mis scripts, si no hoy lo hago
# y mañana ya no me acuerdo de lo que quise hacer
echo "Introduzca su nombre y apellido:"
read nombre apellido
echo "Fecha y hora actual:"
date
echo "Su apellido y nombre es:
echo "$apellido $nombre"
echo "Su usuario es: `whoami`"
echo "Su directorio actual es:"
```

- **Comandos:**
  - ❖ “mkdir practica-shell-script”
  - ❖ “cd /practica-shell-script”
  - ❖ “touch mostrar.sh”
  - ❖ “vi mostrar.sh” (Agrego todo el texto indicado).

a. Asignar al archivo creado los permisos necesarios de manera que pueda ejecutarlo

```
joacogoonz@Debian:~/practica-shell-script$ chmod 755 mostrar.sh
joacogoonz@Debian:~/practica-shell-script$ ls -l mostrar.sh
-rwxr-xr-x 1 joacogoonz joacogoonz 371 Sep 13 21:32 mostrar.sh
```

b. Ejecutar el archivo creado de la siguiente manera: ./mostrar. ¿Qué resultado visualiza?

```
joacogoonz@Debian:~/practica-shell-script$ ./mostrar.sh
Introduzca su nombre y apellido:
Joaquin Manuel Gonzalez
Fecha y hora actual:
Thu Sep 14 08:25:41 AM -03 2023
Su apellido y nombre es:
Manuel Gonzalez Joaquin
Su usuario es: joacogoonz
Su directorio actual es: /home/joacogoonz/practica-shell-script
```

d. Las backquotes (`) entre el comando whoami ilustran el uso de la sustitución de comandos. ¿Qué significa esto?

- En el contexto del comando **"whoami"** las backquotes se utilizan para ejecutar un comando dentro de otro comando y luego sustituir la salida del comando interno en el comando externo aunque ahora se recomienda utilizar **"$(comando)".**

e. Realizar modificaciones al script anteriormente creado de manera de poder mostrar distintos resultados (cuál es su directorio personal, el contenido de un directorio en particular, el espacio libre en disco, etc.). Pida que se introduzcan por teclado (entrada estándar) otros datos

```
joacogoonz@Debian:~/practica-shell-script$ ./mostrar.sh
Introduzca su nombre y apellido:
Joaquin Gonzalez
Fecha y hora actual:
Thu Sep 14 08:45:58 AM -03 2023
Su apellido y nombre es:
Gonzalez Joaquin
Su usuario es: joacogoonz
Su directorio actual es: /home/joacogoonz/practica-shell-script
Su directorio personal es: /home/joacogoonz
El contenido de su directorio personal es: mostrar.sh
El almacenamiento libre en disco es: Filesystem     1K-blocks     Used Available Use% Mo
unted on
udev             2504468        0   2504468   0% /dev
tmpfs             507264     1116    506148   1% /run
/dev/sda1       14389128 10498780   3137608  77% /
tmpfs            2536308    27384   2508924   2% /dev/shm
tmpfs               5120        8      5112   1% /run/lock
tmpfs             507260      108    507152   1% /run/user/1000
Ingrese su número de legajo 21247/1
21247/1
```

4. Parametrización: ¿Cómo se acceden a los parámetros enviados al script al momento de su invocación? ¿Qué información contienen las variables $#, $, $? Y $HOME dentro de un script?*

- Cuando se acceden a los parámetros enviados al script al momento de su invocación se hace a través de variables especiales, estas suelen estar predefinidas y contienen la información de los argumentos proporcionados al script, estos parámetros se almacenan en las variables **"$1, $2, $3, …"** al ejecutar el script con argumentos se debe hacer de la siguiente manera **"./script.sh parámetro_1 parámetro_2"**.

  Variables:

  - **"$#":** Contiene el número de argumentos que se pasaron al script.
  - **"$*":** Representa todos los argumentos en una sola cadena de texto, esta variable **no preserva los espacios en blanco entre los argumentos.**
  - **"$?":** Contiene el código de salida del último comando ejecutado. Si el comando se ejecuta con éxito, **"$? = 0"**. Si hay algún error, será un valor diferente de 0. **Esto es útil para verificar si un comando fue exitoso y tomar decisiones basadas en eso.**
  - **"$HOME":** Contiene la ruta al directorio raíz del usuario actual.

5. ¿Cuál es la funcionalidad de comando exit? ¿Qué valores recibe como parámetro y cuál es su significado?

- **"exit":** Se utiliza para finalizar la ejecución de un script o programa en una Shell. Puede recibir un valor como argumento que indica el código de salida que el script o programa devolverá al sistema operativo.

Sin argumentos: Sin argumento alguno, el comando **“exit”** devuelve el código de salida predeterminado, que es **0**. Un código de salida igual a 0 generalmente indica que el programa se ejecutó con éxito y sin errores.

Con argumentos: Si se proporciona un argumento, este será el código de salida que se devolverá al sistema operativo, estos códigos **no deben ser negativos** y generalmente se utilizan para indicar **si el programa se ejecutó con éxito o si ocurrió algún tipo de error.** Es común usar códigos de salida diferentes a cero para indicar diferentes **tipos de errores.**

6. El comando expr permite la evaluación de expresiones. Su sintaxis es: “expr arg1 op arg2”, donde “arg1 y arg2” representan argumentos y “op” la operación de la expresión. Investigar qué tipo de operaciones se pueden utilizar.

- **“expr”:** Herramienta de Shell que permite la avaluación de expresiones en scripts de Shell. Las expresiones pueden incluir:

  Operadores Aritméticos:
  - **Suma (+):** Suma dos números **“expr 5 + 3”.**
  - **Resta (-):** Resta dos números **“expr 8 – 2”.**
  - **Multiplicación (*):** Multiplica dos números **“expr 4 * 3”**
  - **División (/):** Divide un número por otro **“expr 10 / 2”.**
  - **Módulo (%):** Devuelve el resto de la división de dos números **“expr 7 % 3”.**

  Operadores de Comparación:
  - **Igual (=):** Compara si dos cadenas son iguales **“expr “hola” = “hola””.**
  - **Diferente (!=):** Compara si dos cadenas son diferentes **“expr “hola” != “chau””.**

  Operadores Lógicos:
  - **AND (&&):** Operación lógica **“y”; “expr 1 && 0”.**
  - **OR (||):** Operación lógica **“o”; “expr 1 || 0”.**

  Operadores Relacionales:

- **Menor que (<):** Compara si el primer argumento es menor que el segundo **“expr 5 < 10”.**
- **Mayor que (>):** Compara si el primer argumento es mayor que el segundo **“expr 8 > 3”.**
- **Menor o igual que (<=):** Compara si el primer argumento es menor o igual que el segundo **“expr 5 <= 5”.**
- **Mayor o igual que (>=):** Compara si el primer argumento es mayor o igual al segundo **“expr 10 >= 8”.**

Otros operadores:

- **Longitud de Cadena (length):** Devuelve la longitud de una cadena **“expr length hola”.**

7. El comando “test expresión” permite evaluar expresiones y generar un valor de retorno, true o false. Este comando puede ser reemplazado por el uso de corchetes de la siguiente manera [expresión]. Investigar qué tipo de expresiones pueden ser usadas con el comando test. Tenga en cuenta operaciones para: evaluación de archivos, evaluación de cadenas de caracteres y evaluaciones numéricas.

- **“test”:** Herramienta utilizada para evaluar expresiones y devolver un valor de éxito o fracaso **(true o false).** Normalmente se utiliza como **[expresión].**

  Evaluación de Archivos:

  - **“-e archivo”:** Comprueba si el archivo existe.
  - **“-f archivo”:** Comprueba si es un archivo regular y no un directorio o dispositivo.
  - **“-d directorio”:** Comprueba si el directorio existe.
  - **“-s archivo”:** Comprueba si el archivo tiene un tamaño mayor que cero.
  - **“-r archivo”:** Comprueba si el archivo tiene permisos de lectura.

- ❖ **“-w archivo”:** Comprueba si el archivo tiene permisos de escritura.
- ❖ **“-x archivo”:** Comprueba si el archivo tiene permisos de ejecución.

Evaluación de Cadenas de Caracteres:

- ❖ **“-z cadena”:** Comprueba si la cadena es vacía **(length = 0).**
- ❖ **“-n cadena”:** Comprueba si la cadena no es vacía **(length > 0).**
- ❖ **“cadena1 = cadena2”:** Comprueba si las cadenas son iguales.
- ❖ **“cadena1 != cadena2”:** Comprueba si las cadenas son distintas.

Evaluaciones Numéricas:

- ❖ **“num1 –eq num2”:** Comprueba si los números son iguales.
- ❖ **“num1 –ne num2”:** Comprueba si los números son distintos.
- ❖ **“num1 –lt num2”:** Comprueba si el primer número es menor que el segundo.
- ❖ **“num1 –le num2”:** Comprueba si el primer número es menor o igual que el segundo.
- ❖ **“num1 –gt num2”:** Comprueba si el primer número es mayor que el segundo.
- ❖ **“num1 –ge num2”:** Comprueba si el primer número es mayor o igual que el segundo.

Otros operadores:

- ❖ **“! [Expresión]”:** Negación.
- ❖ **“expresión1 –a expresión2”:** AND lógico.
- ❖ **“expresión1 -0 expresión2”:** OR lógico.

8. Estructuras de control. Investigue la sintaxis de las siguientes estructuras de control incluidas en shell scripting: (If; Case; While; For; Select).

- **Sintaxis de CASE:**

```
case $variable in
"valor 1")
block
;;
"valor 2")
block
;;
*)
block
;;
esac
```

- **Sintaxis de IF:**

```
if [ condition ]
then
block
fi
```

- **Sintaxis WHILE:**

while

```
while [ condition ] #Mientras se cumpla la condición
do
block
done
```

until

```
until [ condition ] #Mientras NO se cumpla la condición
do
block
done
```

- **Sintaxis FOR:**

- *C-style*:

```
for ((i=0; i < 10; i++))
do
block
done
```

- Con lista de valores (`foreach`):

```
for i in value1 value2 value3 valueN;
do
block
done
```

- **Sintaxis Select:**

Menú de opciones:

```
select variable in opcion1 opcion2 opcion3
do
# en $variable está el valor elegido
 block
done
```

- **Uso del Select con el Case:**

Menú de opciones:

Ejemplo:

```
select action in New Exit
do
case $action in
"New")
echo "Selected option is NEW
    "
;;
"Exit")
exit 0
;;
esac
done
```

Imprime:

```
1) new
2) exit
#?
```

y espera el número de opción por teclado

9. ¿Qué acciones realizan las sentencias "break y continue" dentro de un bucle? ¿Qué parámetros reciben?

- **Sentencias:**
  - **Break:** Se utiliza para salir inmediatamente de un bucle **for, while, o until** antes de que se complete su iteración normal.
    Parámetros:
    - **"break [n]": "n"** Es un número entero opcional que indica cuántos niveles de bucles se deben salir/saltear. Si no se proporciona valor alguno, **sale del bucle actual.**
  - **Continue:** Se utiliza para pasar a la siguiente iteración del bucle sin ejecutar el código que sigue a continuación en el bucle, es decir, salta la parte restante del bucle para la iteración actual y pasa a la siguiente iteración.
    Parámetros:
    - **"continue [n]": "n"** Es un número entero opcional que indica cuántos niveles de bucles se deben saltear. Si no se proporciona valor alguno, **saltea el bucle actual.**

10. ¿Qué tipo de variables existen? ¿Es shell script fuertemente tipado? ¿Se pueden definir arreglos? ¿Cómo?

- En Shell scripting las variables solo pueden ser **strings y arrays.**
  En cuanto a si Shell script es fuertemente tipado, esto no es cierto, **no es necesario especificar el tipo de datos al declarar una variable** y las mismas pueden almacenar diferentes tipos de datos en diferentes momentos.
  Arrays en Shell Scripting:

- Creación:

```
arreglo_a=() # Se crea vacío
arreglo_b=(1 2 3 5 8 13 21) # Inicializado
```

- Asignación de un valor en una posición concreta:

```
arreglo_b[2]=spam
```

- Acceso a un valor del arreglo (En este caso las llaves no son opcionales):

```
echo ${arreglo_b[2]}
copia=${arreglo_b[2]}
```

- Acceso a todos los valores del arreglo:

```
echo ${arreglo[@]} # o bien ${arreglo[*]}
```

- Tamaño del arreglo:

```
${#arreglo[@]} # o bien ${#arreglo[*]}
```

- Borrado de un elemento (reduce el tamaño del arreglo pero no elimina la posición, solamente la deja vacía):

```
unset arreglo[2]
```

- Los índices en los arreglos comienzan en 0

11. ¿Pueden definirse funciones dentro de un script? ¿Cómo? ¿Cómo se maneja el pasaje de parámetros de una función a la otra?

- Si, se pueden definir funciones dentro de un script, esto nos ayuda a modularizar el comportamiento de los scripts. **Se pueden declarar de 2 formas:**

```
function nombre { block }
nombre() { block }
```

Return:

- Utilizando la sentencia **"return"** se retorna un valor entre **0 y 255.** Este valor se puede evaluar mediante la variable **"$?"** y las funciones reciben argumentos en las variables **"$1, $2, etc".**

Variables:

- Las variables que se usarán solo dentro de la función se definen con la sentencia **"local variable"** ya que estas por defecto son **globales**. Las variables que no son inicializadas se reemplazan por un valor **nulo o 0.** Todas las variables de **entorno** son heredadas por los procesos hijos y para exponer una variable **global** a los procesos hijos se usa el comando **"export VARIABLE_GLOBAL".**

Invocación:

- Para la invocación de una función en el código se debe hacer **"nombre_de_la_función argumento1 argumento2 argumentoN".**

12. Evaluación de expresiones:

a. Realizar un script que le solicite al usuario 2 números, los lea de la entrada Standard e imprima la multiplicación, suma, resta y cuál es el mayor de los números leídos.

```
#!/bin/bash
read -p "Ingrese el primer número: " num_1
read -p "Ingrese el segundo número: " num_2
echo "La suma de los 2 número es $(($num_1 + $num_2))"
echo "La multiplicación de los números es $(($num_1 * num_2))"
echo "La división entre los 2 números es $(($num_1 / $num_2))"
echo "La resta entre los 2 números es $(($num_1 - $num_2))"
if [ $num_1 -gt $num_2 ]
then
        echo "El número más grande de los 2 es $num_1"
else
        echo "El número más grande de los 2 es $num_2"
fi
```

b. Modificar el script creado en el inciso anterior para que los números sean recibidos como parámetros. El script debe controlar que los dos parámetros sean enviados.

```
#!/bin/bash
if (($# != 2))
then
        exit 1
else
        echo "La suma de los 2 números es: $(($1 + $2))"
        echo "La multiplicación de los números es: $(($1 * $2))"
        echo "La división entre los 2 números es: $(($1 / $2))"
        echo "La resta entre los 2 números es: $(($1 - $2))"
        if [ $1 -gt $2 ]
        then
                echo "El número más grande de los 2 es: $1"
        else
                echo "El número más grande de los 2 es: $2"
        fi
fi
```

c. Realizar una calculadora que ejecute las 4 operaciones básicas: +, - ,*, %. Esta calculadora debe funcionar recibiendo la operación y los números como parámetros

```
#!/bin/bash
if (($# != 3))
then
        exit 1
else
        case $2 in
                "+") echo "$(($1 + $3))" ;;
                "-") echo "$(($1 - $3))" ;;
                "*") echo "$(($1 * $3))" ;;
                "%") echo "$(($1 % $3))" ;;
                *) echo "OPERACIÓN NO VÁLIDA" ;;
        esac
fi
```

13. Uso de las estructuras de control:

a. Realizar un script que visualice por pantalla los números del 1 al 100 así como sus cuadrados.

```
#!/bin/bash

for i in {1..100} ; do
        echo "$i y su cuadrado es: $(($i * $i))"
done
```

b. Crear un script que muestre 3 opciones al usuario: Listar, DondeEstoy y QuienEsta. Según la opción elegida se le debe mostrar: (Listar: lista el contenido del directoria actual). (DondeEstoy: muestra el directorio donde me encuentro ubicado). (QuienEsta: muestra los usuarios conectados al sistema).

```
#!/bin/bash

select action in Listar DondeEstoy QuienEsta Exit
do
        case $action in
                "Listar") echo "Listando contenido: $(ls)" ;;
                "DondeEstoy") echo "Estoy en: $(pwd)" ;;
                "QuienEsta") echo "Están conectados: $(who)" ;;
                "Exit") exit 0 ;;
                *) exit 1 ;;
        esac
done
```

c. Crear un script que reciba como parámetro el nombre de un archivo e informe si el mismo existe o no, y en caso afirmativo indique si es un directorio o un archivo. En caso de que no exista el archivo/directorio cree un directorio con el nombre recibido como parámetro.

```
#!/bin/bash

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

if ! [ -e $1 ]
then
        $(mkdir $1)
elif [ -f $1 ]
then
        echo "El parámetro proporcionado es un archivo"
else
        echo "El parámetro proporcionado es un directorio"
fi
```

14. Renombrando Archivos: haga un script que renombre solo archivos de un directorio pasado como parámetro agregándole una CADENA, contemplando las opciones: ("-a CADENA": renombra el fichero concatenando CADENA al final del nombre del archivo) y ("-b CADENA": renombra el fichero concatenando CADENA al principio del nombre del archivo).

- **Comandos:**
  - **"pushd [ruta]":** Cambia el directorio actual y guarda el directorio anterior en la pila de directorios, si no se especifica una **ruta** intercambia el directorio actual con el último guardado en la pila.
  - **"popd":** Cambia el directorio actual con el último guardado en la pila y lo retira.

```
#!/bin/bash
if [ $# -ne 3 ] ; then
        echo "La cantidad de parámetros no es la esperada"
        exit 1
fi

if [ ! -d $1 ] ; then
        echo "No se proporcionó un DIRECTORIO"
        exit 2
fi

if [ "$2" != "-a" ] && [ "$2" != "-b" ] ; then
        echo -e "No se proporcionó un flag válido\nLas opciones válidas son -a o -b"
        exit 3
fi

pushd $1
if [ "$2" = "-a" ] ; then
        for archivo in $(ls) ; do
                if [ -f $archivo ] ; then
                        mv $archivo $archivo$3
                else
                        continue
                fi
        done
else
        for archivo in $(ls) ; do
                if [ -f $archivo ] ; then
                        mv $archivo $3$archivo
                else
                        continue
                fi
        done
fi
popd
```

15. Comando cut. El comando cut nos permite procesar la líneas de la entrada que reciba (archivo, entrada estándar, resultado de otro comando, etc) y cortar columnas o campos, siendo posible indicar cuál es el delimitador de las mismas. Investigue los parámetros que puede recibir este comando y cite ejemplos de uso.

- **“cut”:** Comando que nos permite extraer secciones de líneas de texto de archivos o de la entrada estándar para luego imprimir el resultado en la salida estándar, normalmente es usado para extraer columnas específicas de un archivo de texto delimitado por caracteres, además permite especificar un determinador personalizado en lugar del **tabulador “\t”** que es el predeterminado.
  Parámetros:

- ❖ **“-d [delimitador]”:** Especifica el delimitador, por defecto, el tabulador.
- ❖ **“-f [num_campo] [archivo]”:** Especifica los campos que se quieren extraer.
- ❖ **“-c [rango] [archivo]”:** Extraer un rango de caracteres específicos de una línea de texto en lugar de columnas basadas en un delimitador.
- ❖ **“-b [rango] [archivo]”:** Trabaja igual que el parámetro **“-c”** pero en vez de caracteres usa bytes.

16. Realizar un script que reciba como parámetro una extensión y haga un reporte con 2 columnas, el nombre de usuario y la cantidad de archivos que posee con esa extensión. Se debe guardar el resultado en un archivo llamado “reporte.txt”

```
#!/bin/bash

if [ $# != 1 ] ; then
        echo "La cantidad de parámetros no es la esperada"
        exit 1
fi

echo -n > $HOME/reporte.txt

for usuario in $(cat /etc/passwd | cut -d: -f 1,6) ; do
        nombre=$(echo $usuario | cut -d: -f1)
        directorio=$(echo $usuario | cut -d: -f2)
        if [ -d $directorio ] ; then
                cantidad=$(sudo find $directorio -type f -name "*.$1" -user $nombre | wc -l)
                echo "$nombre $cantidad" >> $HOME/reporte.txt
        fi
done
```

17. Escribir un script que al ejecutarse imprima en pantalla los nombre de los archivos que se encuentran en el directorio actual, intercambiando minúsculas por mayúsculas, además de eliminar la letra a (mayúscula o minúscula).

- **Comando "tr":** Se utiliza para transformar o traducir caracteres en un flujo de entrada y producir un flujo de salida modificado. Permite reemplazar o eliminar caracteres de una cadena de texto, reducir múltiples ocurrencias consecutivas de un mismo carácter o complementar un conjunto de caracteres, y puede ser utilizado para eliminar caracteres específicos como también para cambiar minúsculas y mayúsculas por ejemplo.

  Parámetros:

  - **"-d ["Caracteres"]":** Elimina todos los caracteres que formen parte del conjunto especificado.
  - **"-s ["Caracter"]":** Reemplaza múltiples ocurrencias consecutivas de un carácter por una única ocurrencia.
  - **"-c ["Caracteres"] ["caracter"]":** Reemplaza todos los caracteres que no están en el conjunto especificado por el que se proporcione.
  - **"[:upper:] y [:lower:]":** Conversión de Mayúsculas y Minúsculas.

```
#!/bin/bash

if [ $# != 0 ] ; then
        echo "Este script no necesita parámetros"
        exit 1
fi

for linea in $(ls) ; do
        echo "$(echo $linea | tr -d 'aA' | tr '[:upper:][:lower:]' '[:lower:][:upper:]')"
done
```

18. Crear un script que verifique cada 10 segundos si un usuario se ha loqueado en el sistema (el nombre del usuario será pasado por parámetro). Cuando el usuario finalmente se loguee, el programa deberá mostrar el mensaje "Usuario XXX logueado en el sistema" y salir.

```
#!/bin/bash

if [ $# != 1 ] ; then
        echo "Cantidad de parámetros incorrecta"
        exit 1
fi

existe=$(cat /etc/passwd | cut -d: -f1 | grep $1 | wc -l)
if [ $existe -eq 0 ] ; then
        echo "El usuario no existe en el sistema"
        exit 2
fi

while true ; do
        conectado=$(who | cut -d' ' -f1 | grep $1 | wc -l)
        if [ $conectado -gt 0 ] ; then
                echo "Usuario $1 logueado en el sistema"
                exit 0
        fi
        sleep 10
done
```

19. Escribir un Programa de "Menú de Comandos Amigable con el Usuario" llamado menú, el cual, al ser invocado, mostrará un menú con la selección para cada uno de los scripts creados en esta práctica. Las instrucciones de cómo proceder deben mostrarse junto con el menú. El menú deberá iniciarse y permanecer activo hasta que se seleccione Salir.

```
#!/bin/bash

if [ $# -ne 0 ] ; then
        echo "Este script no requiere de parámetros"
        exit 1
fi

select action in $(ls | grep ".sh") Salir ; do
        case $action in
                "Salir")
                        echo "Saliendo del menú"
                        exit 0 ;;
                *) bash $action ;;
        esac
done
```

20. Realice un script que simule el comportamiento de una estructura de PILA e implemente las siguientes funciones aplicables sobre una estructura global definida en el script: (push: Recibe un parámetro y lo agrega en la pila) (pop: Saca un elemento de la pila) (length: Devuelve la longitud de la pila) (print: Imprime todos elementos de la pila)

```bash
#!/bin/bash

if [ $# -ne 0 ] ; then
        echo "Este script no requiere de parámetros"
        exit 1
fi

function operaciones {
        local pila=()

        function push {
                pila=(${pila[*]} $1)
                echo "Elemento agregado"
                return 0
        }

        function lenght {
                return ${#pila[*]}
        }

        function pop {
                if [ ${#pila[*]} -ne 0 ] ; then
                        elemento="${pila[${#pila[*]}-1]}"
                        echo "Elemento popeado: $elemento"
                else
                        echo "La pila está vacía"
                fi
                return 0
        }

        function print {
                echo "Elementos: ${pila[*]}"
                return 0
        }

       select action in push lenght pop print exit ; do
              case $action in
                     "push")
                            read -p "Ingrese el elemento a agregar: " elemento
                            push $elemento ;;
                     "lenght")
                            lenght
                            echo "La cantidad de elementos de la pila es: $?" ;;
                     "pop") pop ;;
                     "print") print ;;
                     "exit") exit 0 ;;
                     *)
                            echo "Opción inválida"
                            exit 2 ;;
              esac
       done
}

operaciones
```

21. Dentro del mismo script y utilizando las funciones implementadas: Agregue 10 elementos a la pila. Saque 3 de ellos. Imprima la longitud de la cola. Luego imprima la totalidad de los elementos que en ella se encuentran.

- **Realizar el código e ir probándolo, en mi caso funcionó correctamente.**

22. Dada la siguiente declaración al comienzo de un script: "num=(10 3 5 7 9 3 5 4)" (la cantidad de elementos del arreglo puede variar). Implemente la función "productoria" dentro de este script, cuya tarea sea multiplicar todos los números del arreglo

```
#!/bin/bash

if [ $# -ne 0 ] ; then
        echo "Este script no necesita parámetros"
        exit 1
fi

num=(10 3 5 7 9 3 5 4)

function productoria {
        local resultado="1"

        for valor in ${num[*]} ; do
                resultado=$(($valor * $resultado))
        done

        echo "EL resultado de multiplicar los elementos del array es: $resultado"
}

productoria
exit 0
```

23. Implemente un script que recorra un arreglo compuesto por números e imprima en pantalla sólo los números pares y que cuente sólo los números impares y los informe en pantalla al finalizar el recorrido

```
#!/bin/bash

if [ $# -ne 0 ] ; then
        echo "Este script no requiere parámetros"
        exit 1
fi

num=(1 2 3 4 5 6 7 8 9 10 11 12 13 14 15)

function recorrido {
        local total_imp=0

        for valor in ${num[*]} ; do
                if (( $valor % 2 == 0 )) ; then
                        echo $valor
                else
                        ((total_imp++))
                fi
        done

        echo "La cantidad total de elementos impares es: $total_imp"

        return 0
}

recorrido
exit 0
```

24. Dada la definición de 2 vectores del mismo tamaño y cuyas longitudes no se conocen. Complete este script de manera tal de implementar la suma elemento a elemento entre ambos vectores y que la misma sea impresa en pantalla de la siguiente manera: (EJEMPLO vector1=(1 80 65 35 2) vector2=(5 98 3 41 8).

```
La suma de los elementos de la posición 0 de los vectores es 6
La suma de los elementos de la posición 1 de los vectores es 178
...
La suma de los elementos de la posición 4 de los vectores es 10
```

```
#!/bin/bash

if [ $# -ne 0 ] ; then
        echo "Este script no requiere parámetros"
        exit 1
fi

array1=(1 80 65 35 2)
array2=(5 98 3 41 8)

for ((i=0; i < ${#array1[*]}; i++)) ; do
        echo "La suma de los elementos de la posición $i de los vectores es $((${array1[$i]} + ${array2[$i]}))"
done

exit 0
```

25. Realice un script que agregue en un arreglo todos los nombres de los usuarios del sistema pertenecientes al grupo “users”. Adicionalmente el script puede recibir como parámetro: (“-b n”: Retorna el elemento de la posición n del arreglo si el mismo existe. Caso contrario, un mensaje de error). (“-l”: Devuelve la longitud del arreglo) (“-i”: Imprime todos los elementos del arreglo en pantalla)

```
#!/bin/bash

if [ $# -gt 2 ] ; then
        echo "La cantidad de parámetros no es la esperada"
        exit 1
fi

usuarios=($(cat /etc/group | grep -w "users" | cut -d: -f4 | tr "," " "))

case $1 in
        "-b")
                if [ $# -eq 2  ] && [ $2 -lt ${#usuarios[*]} ] ; then
                        echo "El elemento de la posición $2 es: ${usuarios[$2]}"
                else
                        echo "Posición inválida"
                        exit 2
                fi ;;
        "-l") echo "Longitud del arreglo: ${#usuarios[*]}" ;;
        "-i") echo "Elementos: ${usuarios[*]}" ;;
        *)
                echo "Parámetro inválido"
                exit 3
esac
exit 0
```

26. Escriba un script que reciba una cantidad desconocida de parámetros al momento de su invocación (debe validar que al menos se reciba uno). Cada parámetro representa la ruta absoluta de un archivo o directorio en el sistema. El script deberá iterar por todos los parámetros recibidos, y solo para aquellos parámetros que se encuentren en posiciones impares (el primero, el tercero, el quinto, etc.), verificar si el archivo o directorio existen en el sistema, imprimiendo en pantalla que tipo de objeto es (archivo o directorio). Además, deberá informar la cantidad de archivos o directorios inexistentes en el sistema.

```
#!/bin/bash

if [ $# -eq 0 ] ; then
        echo "No se puede ejecutar el script sin parámetros"
        exit 1
fi

rutas=($@)
cantidad_inexistente=0

for ((i=0; i < ${#rutas[*]}; i++)) ; do
        if ! [ -e ${rutas[$i]} ] ; then
                ((cantidad_inexistente++))
        else
                if (( $i % 2 != 0 )) ; then
                        if [ -f ${rutas[$i]} ] ; then
                                echo "El tipo del objeto $i es: archivo"
                        elif [ -d ${rutas[$i]} ] ; then
                                echo "El tipo del objeto $i es: directorio"
                        else
                                echo "El tipo del objeto $i es: other"
                        fi
                fi
        fi
done

echo "La cantidad de archivos o directorios inexistentes en el sistema es: $cantidad_inexistente"

exit 0
```

27. Realice un script que implemente a través de la utilización de funciones las operaciones básicas sobre arreglos: (inicializar: Crea un arreglo llamado "array" vacío) (agregar_elem: Agrega al final del arreglo el parámetro recibido) (eliminar_elem : Elimina del arreglo el elemento que se encuentra en la posición recibida como parámetro. Debe validar que se reciba una posición válida) (longitud: Imprime la longitud del arreglo en pantalla) (imprimir: Imprime todos los elementos del arreglo en pantalla) (inicializar_Con_Valores: Crea un arreglo con longitud y en todas las posiciones asigna el valor)

```
#!/bin/bash

if [ $# -ne 0 ] ; then
        echo "Este script no requiere parámetros"
        exit 1
fi

function menu {

        local options=(inicializar agregar_elem eliminar_elem longitud imprimir inicializar_con_valores Exit)

        function inicializar {
                array=()
                echo "Vector vacío inicializado"
                return 0
        }

        function agregar_elem {
                array=(${array[*]} $1)
                echo "Elemento agregado"
                return 0
        }

        function eliminar_elem {
                if (($1 < ${#array[*]} )) ; then
                        unset array[$1]
                        echo "Elemento eliminado"
                else
                        echo "Posición inválida"
                        return 1
                fi
                return 0
        }
```

```
        function longitud {
                echo "Longitud del array: ${#array[*]}"
                return 0
        }

        function imprimir {
                echo "Elementos: ${array[*]}"
                return 0
        }

        function inicializar_con_valores {
                local array_con_valores=()
                for ((i=0; i < $1; i++)) ; do
                        array_con_valores[$i]=$2
                done
                echo "Nuevo Arreglo: ${array_con_valores[*]}"
                return 0
        }

    select action in ${options[*]} ; do
           case $action in
                   "inicializar") inicializar ;;
                   "agregar_elem")
                           read -p "Ingrese el elemento a agregar: " elemento
                           agregar_elem $elemento ;;
                   "eliminar_elem")
                           read -p "Ingrese la posición de elemento a eliminar: " pos
                           eliminar_elem $pos ;;
                   "longitud") longitud ;;
                   "imprimir") imprimir ;;
                   "inicializar_con_valores")
                           read -p "Ingrese la cantidad de elementos: " cant
                           read -p "Ingrese el valor a asignar a cada posición: " valor
                           inicializar_con_valores $cant $valor ;;
                   "Exit")
                           echo "Saliendo del menú"
                           exit 0 ;;
                   *)
                           echo "Opción inválida"
                           exit 2 ;;
           esac
    done
}

menu
exit 0
```

28. Realice un script que reciba como parámetro el nombre de un directorio. Deberá validar que el mismo exista y de no existir causar la terminación del script con código de error 4. Si el directorio existe deberá contar por separado la cantidad de archivos que en él se encuentran para los cuales el usuario que ejecuta el script tiene permiso de lectura y escritura, e informar dichos valores en pantalla. En caso de encontrar subdirectorios, no deberán procesarse, y tampoco deberán ser tenidos en cuenta para la suma a informar.

```
#!/bin/bash

if [ $# -ne 1 ] ; then
        echo "La cantidad de parámetros no es la esperada"
        exit 1
fi

if ! [ -e $1 ] && ! [ -d $1 ] ; then
       echo "La ruta especificada no existe/no es un directorio"
       exit 4
fi

pushd $1
contador=0
for archivo in $(ls) ; do
        if [ -f $archivo ] ; then
                if [ -r $archivo ] && [ -w $archivo ] ; then
                        ((contador++))
                fi
        fi
done
popd

echo "La cantidad de archivos con permisos de escritura y lectura son: $contador"
exit 0
```

29. Implemente un script que agregue a un arreglo todos los archivos del directorio /home cuya terminación sea .doc. Adicionalmente, implemente las siguientes funciones que le permitan acceder a la estructura creada:

- verArchivo <nombre_de_archivo>: Imprime el archivo en pantalla si el mismo se encuentra en el arreglo. Caso contrario imprime el mensaje de error "Archivo no encontrado" y devuelve como valor de retorno 5
- cantidadArchivos: Imprime la cantidad de archivos del /home con terminación .doc
- borrarArchivo <nombre_de_archivo>: Consulta al usuario si quiere eliminar el archivo lógicamente. Si el usuario responde Si, elimina el elemento solo del arreglo. Si el usuario responde No, elimina el archivo del arreglo y también del FileSystem. Debe validar que el archivo exista en el arreglo. En caso de no existir, imprime el mensaje de error "Archivo no encontrado" y devuelve como valor de retorno 10

```
#!/bin/bash

if [ $# -ne 0 ] ; then
        echo "Este script no requiere parámetros"
        exit 1
fi

function crear_array {
        archivos=($(ls $HOME | grep ".doc"))
}

function verArchivo {
        for archivo in ${archivos[*]} ; do
                if [ $archivo = $1 ] ; then
                        echo $(cat "$HOME/$archivo")
                        return 0
                fi
        done
        echo "Archivo no encontrado"
        return 5
}

function cantidadArchivos {
        echo "La cantidad de archivos del /home con terminación en .doc es de: ${#archivos[*]}"
        return 0
}
```

```
function borrarArchivo {
        for ((i=0; i < ${#archivos[*]}; i++)) ; do
                if [ ${archivos[$i]} = $1 ] ; then
                        read -p "Desea borrar el archivo logicamente (Si/No) " desicion
                        case $desicion in
                                "Si") unset archivos[$i] ;;
                                "No")
                                        unset archivos[$i]
                                        rm "$HOME/$1" ;;
                                *)
                                        echo "Opción inválida"
                                        return 1
                        esac
                        return 0
                fi
        done
        echo "Archivo no encontrado"
        return 10
}

crear_array
cantidadArchivos
echo "Elementos el Array: ${archivos[*]}"
read -p "Ingrese el nombre del archivo que quiere ver: " nombre
verArchivo $nombre
echo "Valor de retorno: $?"
read -p "Ingrese el nombre del archivo que quiere eliminar: " eliminar
borrarArchivo $eliminar
echo "Valor de retorno: $?"
echo "Elementos luego de eliminar: ${archivos[*]}"
```

30. Realice un script que mueva todos los programas del directorio actual (archivos ejecutables) hacia el subdirectorio "bin" del directorio HOME del usuario actualmente logueado. El script debe imprimir en pantalla los nombres de los que mueve, e indicar cuántos ha movido, o que no ha movido ninguno. Si el directorio "bin" no existe, deberá ser creado.

- **"ls –F":** Muestra una lista de archivos pero al final del nombre de cada uno, agrega una **marca** para indicar el tipo de los mismos (**"*" archivo ejecutable; "/" directorio).**

```
#!/bin/bash

if [ $# -ne 0 ] ; then
        echo "Este script no requiere parámetros"
        exit 1
fi

archivos=($(ls -F | grep "*"))

if [ ${#archivos[*]} -ne 0 ] ; then
        bin=$HOME/bin
        if ! [ -e $bin ] ; then
                mkdir $bin
        fi
                mv ${archivos[*]} $bin
        echo "Elementos movidos: ${archivos[*]}"
        echo "La cantidad de archivos movidos fue de: ${#archivos[*]}"
else
        echo "No hay archivos para mover"
fi
exit 0
```

Comandos Adicionales:

- **"sed":** Comando que se utiliza para manipular y transformar texto. Funciona leyendo líneas de texto como entrada para aplicar un conjunto de reglas edición especificadas por el usuario.
  Sintaxis básica: sed [opciones] "comando" archivo(s)
  Opciones comunes:
  - **"-e":** Permite especificar múltiples comandos de edición.
  - **"-i":** Realiza la edición en el propio archivo.
  - **"-n":** Suprime la salida automática de las líneas.
  - **"-r ó –E":** Habilita el uso de expresiones regulares extendidas.

  Comandos de edición:
  - **"s":** Sustituye texto.
  - **"p":** Imprime líneas.
  - **"d":** Borra líneas.
  - **"i":** Inserta texto antes de una línea.
  - **"a":** Añade texto después de una línea.
  - **"c":** Cambia líneas.
  - **"r":** Lee un archivo y lo inserta en el texto.

- **"w":** Escribe el resultado en un archivo.
- **"q":** Sale del script "sed".

Expresiones regulares:

- **".":** Representa cualquier carácter.
- **"^":** Representa el inicio de una línea.
- **"$":** Representa el final de una línea.
- **"*":** Representa cero o más repeticiones del elemento anterior.
- **"[]":** Representa un conjunto de caracteres.
- **"\":** Escapa un carácter especial.

Direcciones:

- **"n":** Número de línea específico.
- **"/patrón/":** Líneas que coinciden con el patrón.
- **"inicio,fin":** Rango de líneas.

Ejemplos:

- Reemplazar "hola" con "aiós" en todas las instancias de un archivo: **$(sed 's/hola/adiós/g' archivo.txt).**
- Imprimir líneas que coinciden con un patrón: **$(sed –n '/patrón/p' archivo.txt).**
- Borrar líneas que coinciden con un patrón: **$(sed '/patrón/d' archivo.txt).**

- **"awk":** Permite analizar y manipular archivos de texto en función de patrones y acciones definidas por el usuario.

  Sintaxis básica: awk 'patrón {acción}' archivo

  - **"patrón":** Expresión que define qué líneas o bloques de texto se van a procesar.
  - **"acción":** Es la instrucción que se ejecutará cuando el patrón coincida.

  Funcionalidades importantes:

  - **Campos y Delimitadores:**
    - **"FS":** Es el separador de campos. Por defecto, es un espacio en blanco.
    - **"$n":** Representa el campo número "n".
    - **"NF":** Número total de campos en una línea.
  - **Condiciones y Expresiones:**

- **“if, else, else if”:** Permite iterar sobre líneas o realizar operaciones repetidas.

❖ **Estructuras de Control:**

- **“for, while, do-while”:** Permiten iterar sobre líneas o realizar operaciones repetidas.

❖ **Funciones Integradas:**

- **“length()”:** Devuelve la longitud de una cadena.
- **“tolower(), toupper()”:** Covierten texto a minúsculas o mayúsculas.

❖ **Arreglos:**

- Permite almacenar y manipular datos usando índices.

❖ **Patrones Avanzados:**

- Expresiones regulares para patrones más complejos.
- Operadores de comparación y lógicos.

Parámetros Importantes:

❖ **“-F”:** Especifica un separador de campos personalizado.

❖ **“-v”:** Permite asignar valores a variables “awk” desde el Shell.

❖ **“-f”:** Permite especificar un archivo de script de “awk” separado.

❖ **“-E”:** Habilita características de extensión de “awk”.

❖ **“-F”:** Indica el separador de salida (por defecto es un espacio).