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

El Shell Scripting es una forma de programación que se realiza utilizando comandos de Shell en sistemas operativos Unix-like o en el símbolo de sistema en Windows.

Un script de Shell es un archivo de texto que contiene una serie de comandos que se ejecutan en secuencia de manera automática, lo que permite automatizar tareas y procesos en el SO.

Se orientan a:

- **Automatización de tareas repetitivas**: Los scripts pueden utilizarse para automatizar tareas que se realizan con frecuencia, como la copia de archivos, generación de informes, administración de usuarios, descarga de archivos, etc.
- **Administración del sistema**: Los administradores de sistemas utilizan scripts para configurar, mantener y monitorear sistemas, redes y servidores. Por ejemplo, se pueden escribir scripts para realizar copias de seguridad, configurar Firewalls, administrar registros de registro y realizar actualizaciones del sistema.
- **Procesamiento de datos**: Los scripts pueden utilizarse para procesar y manipular datos en archivos. Esto es útil en aplicaciones como el análisis de registros, la conversión de formatos de archivos y la extracción de información relevante de conjuntos de datos.
- **Automatización de tareas de desarrollo**: Los desarrolladores pueden utilizar scripts para automatizar tareas relacionadas con el desarrollo de Software, como la compilación de código, la ejecución de pruebas, la implementación y la gestión de repositorios de código fuente.
- **Interacción con aplicaciones y servicios**: Los scripts pueden interactuar con aplicaciones y servicios a través de la línea de comandos, lo que permite automatizar flujos de trabajo que involucran múltiples programas o servicios.

En cuanto a la compilación de scripts, en el contexto de Shell Scripting, los scripts no necesitan ser compiladores como lo haría un lenguaje de programación compilado. En cambio, los scripts de Shell son interpretados por el intérprete de Shell, lo que significa que el código se ejecuta directamente línea por línea sin necesidad de una compilación previa.

La razón principal por la que los scripts de Shell no se compilan es la flexibilidad y la simplicidad que ofrecen. Esto permite a los usuarios y administradores de sistemas crear, modificar y ejecutar rápidamente scripts sin la necesidad de herramientas de desarrollo adicionales. Además, dado que los scripts suelen consistir en una secuencia de comandos que ya están disponibles en el sistema operativo, no es necesario un proceso de compilación para generar un ejecutable separado. Esto hace que los scripts de Shell sean accesibles y fáciles de usar para una amplia gama de usuarios.

2) Investigar la funcionalidad de los comandos echo y read.

Echo: Imprime en pantalla lo que se le adjunte.
Read: Lee una línea mediante la entrada estándar y almacenarla en una variable.

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

Los comentarios se indican con **#**.

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

Solo existen los Strings y Arrays.
Para declarar una variable se utiliza **nombre = "valor"**
Para hacer referencia a una variable se utiliza **$nombre**
Se puede utilizar **{}** para evitar ambigüedades.

nombre = "Leo"
echo ${nombre}

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:"
```

a) Asignar al archivo creado los permisos necesarios de manera que pueda ser ejecutado.

chmod 555 mostrar.sh

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

```
root@6c8b359891ce:/ISO/Practica 3/practica-shell-script# ./mostrar.sh
Introduzca su nombre y apellido:
Leo Luna
Fecha y hora actual:
Mon Sep 23 21:40:13 UTC 2024
Su apellido y nombre es:
Luna Leo
Su usuario es: root
Su directorio actual es:
```

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

Permiten adjuntar la salida del comando al string donde se utiliza.

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.

```
#!/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 $nombre"
usuario=$(whoami)
echo "Su usuario es: ${usuario}"
echo "Su directorio actual es: `pwd`"
echo "Su espacio libre en disco es: `df --total | grep total`"
echo "Su directorio personal es `cat /etc/passwd | grep ${usuario} | cut -f6 -d:`"
echo "Ingrese la ruta absoluta de un directorio para ver su contenido:"
read ruta
echo "`ls ${ruta}`"
```

Código completo bajo la carpeta /Prácticas/Docker/Práctica 3

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?

Variable	Descripción
`$0`	Contiene el nombre del script en ejecución.
`$1`, `$2`, ..., `$N`	Contiene los argumentos pasados al script. `$1` es el primer argumento, `$2` es el segundo, y así sucesivamente. Por ejemplo, si ejecutas `script.sh arg1 arg2`, `$1` será `arg1` y `$2` será `arg2`.
`$#`	Contiene el número de argumentos pasados al script.
`$*`	Contiene todos los argumentos pasados al script como una sola cadena. Si se utilizan con comillas dobles (`"$*"`), se consideran como un solo argumento.
`$@`	Similar a `$*`, pero cuando se usa con comillas dobles (`"$@"`), cada argumento se mantiene como una palabra separada.
`$?`	Contiene el código de salida del último comando ejecutado. Un valor de `0` generalmente indica que el comando se ejecutó correctamente, mientras que cualquier otro valor indica un error.
`$HOME`	Contiene la ruta del directorio personal del usuario actual. Este es el directorio que se asigna a un usuario cuando inicia sesión.

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

Exit se utiliza para terminar un script. Al ejecutarse, el script finaliza y devuelve un valor entre 0 y 255. El valor 0 indica que el script se ejecutó de forma exitosa, el resto de los valores indican un error.

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 expresión. Investigar qué tipo de operaciones se pueden utilizar.

exp EXPRESION imprime el valor de **EXPRESION** en la salida estándar. Una línea en blanco debajo separa grupos de precedencia creciente.

Tenga en cuenta que muchos operadores deben escaparse o “encomillarse” para las Shells. Las comparaciones son aritméticas si ambos argumentos son numéros, de lo contrario, son lexicográficas. Las coincidencias de patrones devuelven la cadena coincidente entre \(y\) o null; si no se utilizan \(y\), devuelven el número de caracteres coincidentes o 0.

El estado de salida es 0 si **EXPRESION** no es nula ni 0, 1 si **EXPRESION** es nula o 0, 2 si **EXPRESION** es sintácticamente inválida y 3 si se ha producido un error.

Expresión	Descripción
`ARG1 | ARG2`	**ARG1** si no es nulo ni 0, de lo contrario **ARG2**.
`ARG1 & ARG2`	**ARG1** si ninguno de los argumentos es nulo o 0, de lo contrario, 0.
`ARG1 < ARG2`	**ARG1** es menor que **ARG2**.
`ARG1 <= ARG2`	**ARG1** es menor o igual a **ARG2**.
`ARG1 = ARG2`	**ARG1** es igual a **ARG2**.
`ARG1 != ARG2`	**ARG1** es diferente de **ARG2**.
`ARG1 >= ARG2`	**ARG1** es mayor o igual a **ARG2**.
`ARG1 > ARG2`	**ARG1** es mayor que **ARG2**.
`ARG1 + ARG2`	Suma aritmética de **ARG1** y **ARG2**.
`ARG1 - ARG2`	Resta aritmética de **ARG1** y **ARG2**.
`ARG1 * ARG2`	Producto aritmético de **ARG1** y **ARG2**.
`ARG1 / ARG2`	Cociente aritmético de **ARG1** dividido por **ARG2**.
`ARG1 % ARG2`	Resto aritmético de **ARG1** dividido por **ARG2**.
`STRING : REGEXP`	Coincidencia de patrón anclada de **REGEXP** en **STRING**.
`match STRING REGEXP`	Igual que **STRING : REGEXP**.
`substr STRING POS LENGTH`	Subcadena de **STRING**, comenzando en la posición **POS** contada desde 1, con longitud **LENGTH**.
`index STRING CHARS`	Índice en **STRING** donde se encuentra cualquier carácter de **CHARS**, o 0 si no se encuentra.
`length STRING`	Longitud de **STRING**.
`+ TOKEN`	Interpretar **TOKEN** como una cadena, incluso si es una palabra clave como 'match' o un operador como '/'.
`( EXPRESSION )`	Valor de la **EXPRESSION**.

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.

El comando **test** (equivale a [ ]) permite evaluar expresiones y devolver un valor de retorno, ya sea verdadero o falso, que puede utilizarse en scripts de Shell para controlar el flujo del programa.

Las comparaciones con **>** y **<** no son soportadas por los corchetes simples, en su lugar, deben utilizarse los corchetes dobles (**[[ ]]**). Se pueden escapar los operadores utilizando \ para que Bash evite tomarlos como redirecciones: [“$cadena1” \< “$cadena2”].

Expresión		Descripción
Expresiones		
`( EXPR )`		**EXPR** es verdadera
`! EXPR`		**EXPR** es falsa
Strings		
`-z STR`		La longitud de **STR** es cero
`-n STR`	`STR`	La longitud de **STR** es distinta de cero
`STR1 =, !=, <, > STR2`		Comparar strings
Enteros		
`INT1 -eq INT2`		**INT1** es igual a **INT2**
`INT1 -ne INT2`		**INT1** no es igual a **INT2**
`INT1 -lt INT2`		**INT1** es menor estricto que **INT2**
`INT1 -le INT2`		**INT1** es menor o igual que **INT2**
`INT1 -gt INT2`		**INT1** es mayor estricto que **INT2**
`INT1 -ge INT2`		**INT1** es mayor o igual que **INT2**
Archivos		
`-e FILE`		**FILE** existe
`-d FILE`		**FILE** existe y es un directorio
`-f FILE`		**FILE** existe y es un archivo regular
`-s FILE`		**FILE** existe y tiene un tamaño mayor que 0.

8) Estructuras de control. Investigue la sintaxis de las siguientes estructuras de control incluidas en Shell Scripting:

- If
- Case
- While
- For
- Select

IF - ELIF - ELSE

```
if [ 1 = 1 ]
then
        echo “1 es igual a 1”
fi
```

```
if [ 1 = 1 ]; then
        echo “1 es igual a 1”
fi
```

```
if [ 1 = 1 ]
then
        echo “1 es igual a 1”
else
        echo “1 no es igual a 1”
fi
```

```
if [ 1 = 1 ]; then
        echo “1 es igual a 1”
else
        echo “1 no es igual a 1”
fi
```

```
if [ 1 = 1 ]
then
        echo “1 es igual a 1”
elif [ 2 -gt 1 ]
then
        echo “2 es mayor que 1”
elif [ 1 -lt 5 ]
then
        echo “1 es menor que 5”
else
        echo “Cuerpo del else”
fi
```

```
if [ 1 = 1 ]; then
        echo “1 es igual a 1”
elif [ 2 -gt 1 ]; then
        echo “2 es mayor que 1”
elif [ 1 -lt 5 ]; then
        echo “1 es menor que 5”
else
        echo “Cuerpo del else”
fi
```

CASE

```
echo “Ingrese un sabor de helado: “
read sabor
case $sabor in
        chocolate)
                echo “Pasable”
                ;;
        vainilla)
                echo “Rico”
                ;;
        “dulce de leche”)
                echo “Lo más”
                ;;
        *)
                echo “Opción default”
```

SELECT

```
select sabor in chocolate vainilla “dulce de leche” pistacho salir
do
        case $sabor in
                chocolate)
                        echo “Pasable”
                        ;;
                vainilla)
                        echo “Rico”
                        ;;
                “dulce de leche”)
                        echo “Lo más”
                        ;;
                pistacho)
                        echo “???”
                        ;;
                salir)
                        echo “Saliendo del menú de selección”
                        break
                        ;;
        esac
done
```

FOR

for valor in uno dos 3 cuatro cinco seis
do
  echo $valor
done
	arreglo=(uno tres 4 seis seventeen)
	for valor in ${arreglo[*]}
	do
	  echo $valor
	done

WHILE - UNTIL

i=1
while [$i -lt 10]
do
  echo “Pasada $i”
  let i++
done
echo “Fin del while: i => $i”
	i=1
	until [$i -lt 10]
	do
	  echo “Pasada $i”
	  let i++
	done
	echo “Fin del until: i => $i”

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

break termina el bucle actual y pasa el control del programa al comando que sigue al bucle terminado. Se utiliza para salir de un bucle for, while, until o select.

Puede recibir un argumento **n**, que debe ser mayor o igual a 1. **break 1** equivale a **break**.

continue omite los comandos restantes dentro del cuerpo del bucle que lo encierra para la iteración actual y pasa el control del programa a la siguiente iteración del bucle.

Puede recibir un argumento **n**, que debe ser mayor o igual a 1. **continue 1** equivale a **continue**.

10) ¿Qué tipo de variables existen? ¿Es Shell Script fuertemente tipado? ¿Se pueden definir arreglos? ¿Cómo?

Bash soporta Strings y Arrays.

Bash es un lenguaje de tipado débil y dinámico. No es necesario declarar explícitamente el tipo de una variable y las variables pueden cambiar de tipo en tiempo de ejecución.

Los nombres de las variables son case-sensitive y pueden contener mayúsculas, minúsculas, números y el símbolo “_”, no pueden empezar con un número.

nombre=Leo	nombre=”Leo”

echo $nombre	echo ${nombre}**no_es_de_la_variable**

No es necesario utilizar comillas (“ “) a menos que se quiera utilizar espacios en el String.

Las variables, por defecto, son de alcance global. Se pueden definir variables locales a funciones utilizando la palabra clave “local” antes del nombre.

cargaManual[0]=”Leo” cargaManual[1]=”Luna” cargaManual[2]=”Practica 3 ISO”	arregloCargado=(1 2 3 4 5)

Trabajando con arreglos

```
echo "${Fruits[0]}"           # Element #0
echo "${Fruits[-1]}"          # Last element
echo "${Fruits[@]}"           # All elements, space-separated
echo "${#Fruits[@]}"          # Number of elements
echo "${#Fruits}"             # String length of the 1st element
echo "${#Fruits[3]}"          # String length of the Nth element
echo "${Fruits[@]:3:2}"       # Range (from position 3, length 2)
echo "${!Fruits[@]}"          # Keys of all elements, space-separated
```

Operaciones

```
Fruits=("${Fruits[@]}" "Watermelon")    # Push
Fruits+=('Watermelon')                  # Also Push
Fruits=( "${Fruits[@]/Ap*/}" )          # Remove by regex match
unset Fruits[2]                         # Remove one item
Fruits=("${Fruits[@]}")                 # Duplicate
Fruits=("${Fruits[@]}" "${Veggies[@]}") # Concatenate
lines=(`cat "logfile"`)                 # Read from file
```

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

#Forma 1
unaFuncion() {
echo “Hola, $1!”
}
	#Invocación
	unaFuncion “Leo”
	#Salida
	“Hola, Leo!”

#Forma 2
function unaAlternativa() {
echo “Hola, $1!”
}
	#Invocación
	unaAlternativa “Gonza”
	#Salida
	“Hola, Gonza!”

#Forma 3
function otraAlternativa {
echo “Hola, $1!”
}
	#Invocación
	otraAlternativa “Eze”
	#Salida
	“Hola, Eze!”

return es una palabra clave que finaliza la ejecución de una función y permite retornar un valor entre 0 y 255 (default 0). El valor 0 se considera como un valor de retorno exitoso.

myFunc() {
return 1
}
	if myFunc; then
	echo “éxito”
	else
	echo “fallo”
	fi

myFunc() {
local resultado=”un valor”
echo “$resultado”
}
	result=$(myFunc)
	
	#Imprime “un valor”

<u>Los siguientes ejercicios se encuentran en la carpeta “Docker”, dentro de “Prácticas”</u>