Juan Menduiña

Trabajo Práctico N° 3:
Bash. *Script*. Sintaxis. GNU/Linux.

Ejercicio 1.

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

El *Shell Scripting* es el proceso de escribir programas o secuencias de comandos (*scripts*) que se ejecutan en un intérprete de comandos del sistema operativo, como Bash, Zsh, Sh, Ksh, etc. Un *shell script* es, básicamente, un archivo de texto con instrucciones que el sistema interpreta línea por línea, igual que si se escribieran, manualmente, en la terminal. Generalmente, su extensión es .sh.

Los *scripts* de *shell* están orientados, principalmente, a tareas repetitivas o administrativas en sistemas GNU/Linux (aunque también en Windows con PowerShell).

Los *scripts* no deben compilarse, se interpretan. Esto significa que no se traducen a código máquina antes de ejecutarse, como ocurre con los programas en C o Java. En cambio, el intérprete del *shell* lee y ejecuta cada línea del *script*, directamente, en tiempo de ejecución.

No se compilan porque el *shell* es un intérprete, no un compilador. Su diseño busca flexibilidad y rapidez en la escritura y la ejecución, no velocidad de ejecución máxima. La idea es poder modificar y probar *scripts* fácilmente, sin necesidad de un proceso de compilación intermedio.

Ejercicio 2.

(a) *Investigar la funcionalidad de los comandos echo y read.*

El comando *echo* muestra texto o el valor de variables por pantalla (en la terminal). Es útil para dar mensajes al usuario o mostrar resultados. Ejemplo:

```
echo "Hola mundo"
echo "Tu nombre es $nombre"
```

El comando *read* permite leer datos ingresados por el usuario desde el teclado y guardarlos en una variable. Se usa para hacer *scripts* interactivos. Ejemplo:

```
echo "Ingresar nombre:"
read nombre
echo "Hola $nombre, ¡bienvenido!"
```

(b) *¿Cómo se indican los comentarios dentro de un script?*

Los comentarios en *shell script* se indican con el símbolo #. Todo lo que sigue después del # en esa línea no se ejecuta. Ejemplo:

```
# Éste es un comentario
echo "Hola mundo" # Esto también es un comentario al final de una línea
```

(c) *¿Cómo se declaran y se hace referencia a variables dentro de un script?*

Las variables se declaran sin espacios entre el nombre, el signo = y el valor. Ejemplo:

```
nombre="Juan"
edad=25
```

Para usar o mostrar el valor de una variable, se antepone el signo $ al nombre de la variable. Ejemplo:

```
echo "Mi nombre es $nombre y tengo $edad años."
```

Ejercicio 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 lo hago hoy,
# 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 se pueda ejecutar.*

```
mkdir /home/jmenduina/practica-shell-script
cd /home/jmenduina/practica-shell-script
touch /home/jmenduina/practica-shell-script/mostrar.sh
ls -l /home/jmenduina/practica-shell-script

nano /home/jmenduina/practica-shell-script/mostrar.sh

chmod +x /home/jmenduina/practica-shell-script/mostrar.sh
ls -l /home/jmenduina/practica-shell-script
```

(b) *Ejecutar el archivo creado de la siguiente manera: ./mostrar.sh.*

```
./mostrar.sh
```

(c) *¿Qué resultado visualiza?*

El resultado que se visualiza es:

```
Introduzca su nombre y apellido:
Juan Menduiña
Fecha y hora actual:
Sun Nov 9 13:32:34 -03 2025
Su apellido y nombre es:
Menduiña Juan
```

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?*

La sustitución de comandos en *shell* significa que el resultado (salida) de un comando se reemplaza, directamente, en la línea donde aparece. El *shell* ejecuta el comando que está entre las comillas invertidas o dentro de *$( )*, toma su salida y la sustituye en el lugar donde estaba el 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.). Pedir que se introduzcan por teclado (entrada estándar) otros datos.*

Sólo agrego lo siguiente:

```
echo "Su directorio actual es:"
pwd
```

Ahora, el resultado que se visualiza es:

```
Introduzca su nombre y apellido:
Juan Menduiña
Fecha y hora actual:
Sun Nov 9 13:32:34 -03 2025
Su apellido y nombre es:
Menduiña Juan
Su usuario es: root
Su directorio actual es:
/home/jmenduina/practica-shell-script
```

Ejercicio 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 ejecuta un *script* en Bash, se le pueden pasar parámetros o argumentos al invocarlo desde la línea de comandos, separados por espacios: *./mi_script.sh juan menduiña 30*. A los parámetros enviados al *script* al momento de su invocación se accede mediante variables especiales numeradas: *$0* (nombre del *script*), *$1* (primer parámetro), *$2* (segundo parámetro), …

Existen las siguientes variables especiales en Bash:

- *$#*: Cantidad de parámetros pasados al *script*.
- *$**: Todos los parámetros en una sola cadena (separados por espacios).
- *$?*: Código de salida del último comando ejecutado (0 = éxito, distinto de 0 = error).
- *$HOME*: Directorio personal del usuario actual.

Ejercicio 5.

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

La funcionalidad del comando *exit* es detener, inmediatamente, la ejecución del *script* y devuelve un valor numérico (el código de salida o *exit status*) al sistema.

El valor que recibe como parámetro *n* opcional numérico puede ser entre 0 y 255, que representa el estado de finalización del *script* (0 = éxito, distinto de 0 = error). Si no se especifica ningún número, Bash usa el código de salida del último comando ejecutado.

Ejercicio 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.

El comando *expr* (abreviatura de *expression*) sirve para evaluar expresiones y mostrar su resultado por pantalla. Se usa, principalmente, en *scripts* antiguos o cuando no se dispone de la expansión aritmética *$(( ))*.

El tipo de operaciones que se pueden utilizar son:

1. Operaciones aritméticas: + (suma), - (resta), * (multiplicación), / (división entera), % (módulo).
2. Operaciones relacionales o de comparación: = (igualdad de cadenas), *!=* (desigualdad de cadenas), > (mayor que), < (menor que), >= (mayor o igual que), <= (menor o igual que).
3. Operaciones lógicas: | (OR, o lógico), *&* (AND, y lógico).

Ejercicio 7.

El comando test expresion 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. Tener en cuenta operaciones para: evaluación de archivos, evaluación de cadenas de caracteres y evaluaciones numéricas.

El comando *test* (y su forma equivalente con corchetes *[expresión]*) sirve para evaluar condiciones en un *script*. Su salida es un valor de retorno: 0 (verdadero), 1 (falso). Se usa, normalmente, en estructuras *if*, *while*, etc.

El tipo de expresiones que pueden ser usadas con el comando *test* son:

1. Evaluación de archivos: Sirven para comprobar si un archivo o directorio existe o cumple ciertas condiciones.
   Ejemplos:
   - *-e archivo* (verdadero si existe el archivo).
   - *-f archivo* (verdadero si existe y es un archivo regular).
   - *-d archivo* (verdadero si existe y es un directorio).
   - *-r archivo* (verdadero si el archivo es legible).
   - *-w archivo* (verdadero si el archivo es escribible).
   - *-x archivo* (verdadero si el archivo es ejecutable).
   - *-s archivo* (verdadero si el archivo no está vacío).
   - *archivo1 -nt archivo2* (verdadero si archivo1 es más nuevo que archivo2).
   - *archivo1 -ot archivo2* (verdadero si archivo1 es más viejo que archivo2).
2. Evaluación de cadenas de caracteres: Sirven para comparar o verificar cadenas.
   Ejemplos:
   - *-z cadena* (verdadero si la longitud es cero).
   - *-n cadena* (verdadero si la longitud no es cero).
   - *cadena1 = cadena2* (verdadero si son iguales).
   - *cadena1 != cadena2* (verdadero si son distintas).
3. Evaluaciones numéricas: Sirven para comparar valores enteros (no cadenas).
   Ejemplos:
   - *-eq* (igual).
   - *-ne* (distinto).
   - *-gt* (mayor que).
   - *-lt* (menor que).
   - *-ge* (mayor o igual que).
   - *-le* (menor o igual que).

Ejercicio 8.

Estructuras de control. Investigar la sintaxis de las siguientes estructuras de control incluídas en shell scripting:

(a) *if.*

Se usa para evaluar una condición y ejecutar comandos según sea verdadera o falsa.

```
if [ condición ]; then
   comandos_si_verdadero
elif [ otra_condición ]; then
   comandos_si_se_cumple_elif
else
   comandos_si_falso
fi
```

(b) *case.*

Se usa para evaluar una variable frente a varios posibles valores (como un *switch* en otros lenguajes).

```
case variable in
   valor1)
      comandos
      ;;
   valor2|valor3)
      otros_comandos
      ;;
   *)
      comandos_por_defecto
      ;;
esac
```

(c) *while.*

Ejecuta un bloque de comandos mientras se cumpla una condición.

```
while [ condición ]; do
   comandos
done
```

(d) *for.*

Permite recorrer una lista de elementos o un rango.

```
for variable in lista; do
   comandos
done
```

(e) *select.*

Se usa para crear menús interactivos (útil en *scripts* de usuario).

```
select variable in lista; do
   comandos
done
```

Ejercicio 9.

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

Las acciones que realizan las sentencias *break* y *continue* dentro de un bucle y los parámetros que reciben son:

- *break*: El programa sale del ciclo y continúa ejecutando las instrucciones que siguen después del bucle.
- *continue*: Salta a la siguiente iteración del bucle, sin ejecutar las instrucciones restantes de la iteración actual.

Ambas sentencias pueden recibir un parámetro *n* opcional numérico, que indica cuántos niveles de bucles anidados deben afectarse:

- *break*: Sale del bucle actual (por defecto).
- *break n*: Sale de *n* niveles de bucles anidados.
- *continue*: Salta a la siguiente iteración del bucle actual (por defecto).
- *continue n*: Salta a la siguiente iteración del n-ésimo bucle exterior.

Ejercicio 10.

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

Existen los siguientes tipo de variables:

1. Variables del usuario (o locales): Son las que se definen dentro del *script*. Sólo existen mientras el *script* se está ejecutando.
2. Variables de entorno: Son variables del sistema o del entorno del usuario. Están disponibles para todos los procesos y los programas.

Shell Script no es fuertemente tipado, lo cual significa que todas las variables se manejan como cadenas de texto, aunque contengan números.

Se pueden definir arreglos en Bash (no en todos los *shell* antiguos). Ejemplo: *numeros=(10 20 30 40)*.

Ejercicio 11.

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

Sí, dentro de un *script*, pueden definirse funciones. Hay dos formas válidas de definir funciones:

```
nombre_funcion() {
   comandos
}

function nombre_funcion {
   comandos
}
```

El pasaje de parámetros de una función a la otra se maneja igual que en un *script*, se pasan al invocar la función, separados por espacios. Dentro de la función, se accede a ellos con las variables posicionales: *$1* (primer parámetro), *$2* (segundo parámetro), …

Bash no devuelve valores "numéricos" directamente, como otros lenguajes. Hay dos formas de devolver resultados:

1. Usando *echo*: Permite capturar el valor en una variable.
2. Usando *return*: Sólo para códigos de estado 0-255.

Ejercicio 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.*

(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.*

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

Ejercicio 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.*

(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 directorio actual.*
- *DondeEstoy: Muestra la ruta del directorio donde me encuentro ubicado.*
- *QuienEsta: Muestra los usuarios conectados al sistema.*

(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.*

Ejercicio 14: Renombrando Archivos.

Hacer un script que renombre sólo 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.*
- *"-b CADENA": Renombra el fichero concatenando CADENA al comienzo del nombre del archivo.*

Ejemplos: Si se tienen los siguientes archivos /tmp/a y /tmp/b, al ejecutar ./renombra /tmp/ -a EJ, se obtendrá, como resultado, /tmp/aEJ /tmp/bEJ. Y, si se ejecuta ./renombra /tmp/ -b EJ, el resultado será /tmp/EJa /tmp/EJb.

Ejercicio 15.

El comando cut nos permite procesar las 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. Investigar los parámetros que puede recibir este comando y citar ejemplos de uso.

El comando *cut* en *Shell* sirve para extraer secciones específicas de cada línea de texto en un archivo o entrada estándar (*stdin*). Se puede pensar como una forma de "recortar columnas" o "campos" de texto según su posición o delimitador.

La sintaxis general es: *cut [opciones] [archivo]*. Si no se indica archivo, *cut* lee de la entrada estándar (por ejemplo, se puede usar con *pipe*).

Los parámetros que puede recibir este comando son:

- *-b LISTA*: Selecciona bytes específicos de cada línea (por posición).
- *-c LISTA*: Selecciona caracteres específicos de cada línea.
- *-f LISTA*: Selecciona campos (*fields*), separados por un delimitador.
- *-d 'DELIM'*: Define el delimitador de campos (por defecto es tabulación).
- *--complement*: Muestra todo menos los bytes/campos/caracteres indicados.
- *-s*: Suprime líneas que no contienen el delimitador (útil con *-f*).

Ejemplos de uso:

- *cut -b 1-3 archivo.txt*: Muestra los primeros 3 bytes (útil para datos binarios o sin acentos).
- *cut -c 1-5 archivo.txt*: Muestra los primeros 5 caracteres de cada línea del archivo.
- *cut -d ',' -f 1 datos.csv*: Muestra sólo la primera columna (nombre).
- *cut -d ',' -f 1 --complement datos.csv*: Muestra todos los campos excepto el primero.

Ejercicio 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.

Ejercicio 17.

Ejercicio 18.

Ejercicio 19.

Ejercicio 20.

Ejercicio 21.

Ejercicio 22.

Ejercicio 23.

Ejercicio 24.

Ejercicio 25.

Ejercicio 26.

Ejercicio 27.

Ejercicio 28.

Ejercicio 29.

Ejercicio 30.

Ejercicio 31.