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.*

```
#!/bin/bash
# Script: Ejercicio12a.sh
# Uso: ./Ejercicio12a.sh

# Solicitud de números al usuario
echo -n "Introducir primer número: "
read num1
echo -n "Introducir segundo número: "
read num2

# Operaciones aritméticas
suma=$((num1 + num2))
resta=$((num1 - num2))
multiplicacion=$((num1 * num2))

# ¿Cuál es el mayor de los números leídos?
if [ $num1 -gt $num2 ]; then
   mayor=$num1
elif [ $num1 -lt $num2 ]; then
   mayor=$num2
else
   mayor="Son iguales"
fi

# Resultados
echo "------------------------------"
echo "RESULTADOS:"
echo " - La suma es: $suma"
echo " - La resta es: $resta"
echo " - La multiplicación es: $multiplicacion"
echo " - El mayor es: $mayor"
echo "------------------------------"
```

(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
# Script: Ejercicio12b.sh
# Uso: ./Ejercicio12b.sh <num1> <num2>

# Control de cantidad de parámetros
if [ $# -ne 2 ]; then
```

```
    echo "Error: Se deben ingresar, exactamente, 2 parámetros"
    echo "Uso: $0 <num1> <num2>"
    exit 1
fi

# Asignación de parámetros
num1=$1
num2=$2

# Operaciones aritméticas
suma=$((num1 + num2))
resta=$((num1 - num2))
multiplicacion=$((num1 * num2))

# ¿Cuál es el mayor de los números leídos?
if [ $num1 -gt $num2 ]; then
    mayor=$num1
elif [ $num1 -lt $num2 ]; then
    mayor=$num2
else
    mayor="Son iguales"
fi

# Resultados
echo "------------------------------"
echo "RESULTADOS:"
echo " - La suma es: $suma"
echo " - La resta es: $resta"
echo " - La multiplicación es: $multiplicacion"
echo " - El mayor es: $mayor"
echo "------------------------------"
```

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

```
#!/bin/bash
# Script: Ejercicio12c.sh
# Uso: ./Ejercicio12c.sh <operación> <num1> <num2>

# Control de cantidad de parámetros
if [ $# -ne 3 ]; then
    echo "Error: Se deben ingresar, exactamente, 3 parámetros"
    echo "Uso: $0 <operación> <num1> <num2>"
    echo "Operaciones válidas: + - * %"
    exit 1
fi

# Asignación de parámetros
```

```
operacion=$1
num1=$2
num2=$3

# Validación de números
if ! [[ $num1 =~ ^-?[0-9]+$ && $num2 =~ ^-?[0-9]+$ ]]; then
    echo "Error: Los operandos deben ser números enteros"
    exit 1
fi

# Calculadora
case $operacion in
    "+")
        resultado=$((num1 + num2))
        ;;
    "-")
        resultado=$((num1 - num2))
        ;;
    "*")
        resultado=$((num1 * num2))
        ;;
    "%")
        if [ $num2 -eq 0 ]; then
            echo "Error: No se puede dividir por cero"
            exit 1
        fi
        resultado=$((num1 % num2))
        ;;
    *)
        echo "Operación inválida. Se debe usar: + - * %"
        exit 1
        ;;
esac
echo "Resultado: $resultado"
```

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.*

```
#!/bin/bash
# Script: Ejercicio13a.sh
# Uso: ./Ejercicio13a.sh

# Algoritmo
for ((i = 1; i <= 100; i++)); do
   echo "Número: $i - Cuadrado: $((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 directorio actual.*
- *DondeEstoy: Muestra la ruta del directorio donde me encuentro ubicado.*
- *QuienEsta: Muestra los usuarios conectados al sistema.*

```
#!/bin/bash
# Script: Ejercicio13b.sh
# Uso: ./Ejercicio13b.sh

# Selección de opción por parte del usuario
echo "Seleccionar una opción:"
echo "1) Listar"
echo "2) DondeEstoy"
echo "3) QuienEsta"
read -p "Opción: " opcion

# Algoritmo
case $opcion in
   1|Listar|listar)
      echo "Contenido del directorio actual:"
      ls
      ;;
   2|DondeEstoy|dondeestoy)
      echo "Ruta actual:"
      pwd
      ;;
   3|QuienEsta|quienesta)
      echo "Usuarios conectados:"
      who
      ;;
   *)
```

```
        echo "Opción inválida"
        ;;
esac
```

(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
# Script: Ejercicio13c.sh
# Uso: ./Ejercicio13c.sh

# Control de cantidad de parámetros
if [ $# -ne 1 ]; then
   echo "Error: Se debe ingresar, exactamente, 1 parámetro"
   echo "Uso: $0 <nombre_archivo>"
   exit 1
fi

# Asignación de parámetros
nombre=$1

# Algoritmo
if [ -e "$nombre" ]; then
   if [ -d "$nombre" ]; then
      echo "'$nombre' existe y es un directorio"
   elif [ -f "$nombre" ]; then
      echo "'$nombre' existe y es un archivo regular"
   else
      echo "'$nombre' existe, pero no es ni archivo ni directorio"
   fi
else
   echo "'$nombre' no existe. Se creará un directorio con ese nombre..."
   mkdir "$nombre"
   echo "Directorio '$nombre' creado correctamente"
fi
```

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 y /tmp/bEJ. Y, si se ejecuta ./renombra /tmp/ -b EJ, el resultado será /tmp/EJa /tmp/EJb.

```
#!/bin/bash
# Script: Ejercicio14.sh
# Uso: ./Ejercicio14.sh

# Control de cantidad de parámetros
if [ $# -ne 3 ]; then
   echo "Error: Se deben ingresar, exactamente, 3 parámetros"
   echo "Uso: $0 <directorio> -a|-b <CADENA>"
   exit 1
fi

# Asignación de parámetros
directorio=$1
opcion=$2
cadena=$3

# Verificación de que el directorio exista
if [ ! -d "$directorio" ]; then
   echo "Error: '$directorio' no es un directorio válido"
   exit 1
fi

# Algoritmo
case $opcion in
   -a)
      for archivo in "$directorio"/*; do
         if [ -f "$archivo" ]; then
            nombre=$(basename "$archivo")
            mv "$archivo" "$directorio/${nombre}${cadena}"
         fi
      done
      echo "Archivos renombrados agregando '$cadena' al final"
      ;;
   -b)
      for archivo in "$directorio"/*; do
```

```
        if [ -f "$archivo" ]; then
            nombre=$(basename "$archivo")
            mv "$archivo" "$directorio/${cadena}${nombre}"
        fi
    done
    echo "Archivos renombrados agregando '$cadena' al comienzo"
    ;;
  *)
    echo "Opción inválida. Se debe usar: -a o -b"
    exit 1
    ;;
esac
```

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.

```
#!/bin/bash
# Script: Ejercicio16.sh
# Uso: ./Ejercicio16.sh

# Control de cantidad de parámetros
if [ $# -ne 1 ]; then
   echo "Error: Se debe ingresar, exactamente, 1 parámetro"
   echo "Uso: $0 <extension>"
   exit 1
fi

# Asignación de parámetros
extension=$1
salida="reporte.txt"

# Limpieza del archivo de salida
> "$salida"

# Algoritmo
echo "Nombre de usuario | Cantidad de archivos con extensión .$extension" >> "$salida"
echo "------------------------------------------------------------" >> "$salida"
for dir in /home/*; do
   if [ -d "$dir" ]; then
      usuario=$(basename "$dir")
      cantidad=$(find "$dir" -type f -name "*.$extension" 2>/dev/null | wc -l)
      echo "$usuario | $cantidad" >> "$salida"
   fi
done
echo "Resultado guardado en $salida"
```

Ejercicio 17.

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

Por ejemplo, si, en el directorio actual, están los siguientes archivos:

- *IsO.*
- *pepE.*
- *Maria.*

y se ejecutó ./ejercicio17 , se obtendrá, como resultado:

- *iSo.*
- *PEPe.*
- *mRI.*

Ayuda: Investigar el comando tr.

```
#!/bin/bash
# Script: Ejercicio17.sh
# Uso: ./Ejercicio17.sh

# Algoritmo
for archivo in *; do
    # Transformación del nombre:
    # - Intercambia minúsculas y mayúsculas: 'tr a-zA-Z A-Za-z'
    # - Elimina todas las 'a' o 'A': 'tr -d'
    if [ -f "$archivo" ]; then
        nuevo_nombre=$(echo "$archivo" | tr 'a-zA-Z' 'A-Za-z' | tr -d 'aA')
        echo "$nuevo_nombre"
    fi
done
```

Ejercicio 18.

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

```
#!/bin/bash
# Script: Ejercicio18.sh
# Uso: ./Ejercicio18.sh

# Control de cantidad de parámetros
if [ $# -ne 1 ]; then
   echo "Error: Se debe ingresar, exactamente, 1 parámetro"
   echo "Uso: $0 <nombre_usuario>"
   exit 1
fi

# Asignación de parámetros
usuario=$1

# Algoritmo
while true; do
   if who | grep -wq "$usuario"; then
      echo "Usuario $usuario logueado en el sistema"
      exit 0
   else
      echo "Esperando que $usuario se loguee..."
      sleep 10
   fi
done
```

Ejercicio 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". Por ejemplo:

MENU DE COMANDOS
03. Ejercicio 3
12. Evaluar Expresiones
13. Probar estructuras de control
...
Ingrese la opción a ejecutar: 03

```
#!/bin/bash
# Script: Ejercicio19.sh
# Uso: ./Ejercicio19.sh

# Ruta donde están los scripts creados en esta práctica
SCRIPTS_DIR="/home/jmenduina/practica-shell-script"

# Función "mostrar_menu"
mostrar_menu() {
   clear
   echo "================================"
   echo       "MENÚ DE COMANDOS"
   echo "================================"
   echo "3. Ejercicio 3"
   echo "12a. Ejercicio 12a"
   echo "12b. Ejercicio 12b"
   echo "12c. Ejercicio 12c"
   echo "13a. Ejercicio 13a"
   echo "13b. Ejercicio 13b"
   echo "13c. Ejercicio 13c"
   echo "17. Ejercicio 17"
   echo "19. Ejercicio 19"
   echo "20. Ejercicio 20"
   echo "21. Ejercicio 21"
   echo "22. Ejercicio 22"
   echo "23. Ejercicio 23"
   echo "24. Ejercicio 24"
   echo "25. Ejercicio 25"
   echo "26. Ejercicio 26"
   echo "27. Ejercicio 27"
   echo "28. Ejercicio 28"
   echo "29. Ejercicio 29"
   echo "30. Ejercicio 30"
   echo "31. Ejercicio 31"
```

```
    echo "99. Salir"
    echo "------------------------------"
    echo
    echo -n "Introducir número de script a ejecutar: "
}

# Algoritmo
while true; do
    mostrar_menu
    read opcion
    echo
    case "$opcion" in
        3)
            echo "Ejecutando Ejercicio 3..."
            echo
            bash "$SCRIPTS_DIR/ejercicio3.sh"
            ;;
        12a)
            echo "Ejecutando Ejercicio 12a..."
            echo
            bash "$SCRIPTS_DIR/ejercicio12a.sh"
            ;;
        12b)
            echo "Ejecutando Ejercicio 12b..."
            echo
            bash "$SCRIPTS_DIR/ejercicio12b.sh"
            ;;
        12c)
            echo "Ejecutando Ejercicio 12c..."
            echo
            bash "$SCRIPTS_DIR/ejercicio12c.sh"
            ;;
        13a)
            echo "Ejecutando Ejercicio 13a..."
            echo
            bash "$SCRIPTS_DIR/ejercicio13a.sh"
            ;;
        13b)
            echo "Ejecutando Ejercicio 13b..."
            echo
            bash "$SCRIPTS_DIR/ejercicio13b.sh"
            ;;
        13c)
            echo "Ejecutando Ejercicio 13c..."
            echo
            bash "$SCRIPTS_DIR/ejercicio13c.sh"
            ;;
        17)
            echo "Ejecutando Ejercicio 17..."
            echo
```

```
    bash "$SCRIPTS_DIR/ejercicio17.sh"
    ;;
19)
    echo "Ejecutando Ejercicio 19..."
    echo
    bash "$SCRIPTS_DIR/ejercicio19.sh"
    ;;
20)
    echo "Ejecutando Ejercicio 20..."
    echo
    bash "$SCRIPTS_DIR/ejercicio20.sh"
    ;;
21)
    echo "Ejecutando Ejercicio 21..."
    echo
    bash "$SCRIPTS_DIR/ejercicio21.sh"
    ;;
22)
    echo "Ejecutando Ejercicio 22..."
    echo
    bash "$SCRIPTS_DIR/ejercicio22.sh"
    ;;
23)
    echo "Ejecutando Ejercicio 23..."
    echo
    bash "$SCRIPTS_DIR/ejercicio23.sh"
    ;;
24)
    echo "Ejecutando Ejercicio 24..."
    echo
    bash "$SCRIPTS_DIR/ejercicio24.sh"
    ;;
25)
    echo "Ejecutando Ejercicio 25..."
    echo
    bash "$SCRIPTS_DIR/ejercicio25.sh"
    ;;
26)
    echo "Ejecutando Ejercicio 26..."
    echo
    bash "$SCRIPTS_DIR/ejercicio26.sh"
    ;;
27)
    echo "Ejecutando Ejercicio 27..."
    echo
    bash "$SCRIPTS_DIR/ejercicio27.sh"
    ;;
28)
    echo "Ejecutando Ejercicio 28..."
    echo
```

```
        bash "$SCRIPTS_DIR/ejercicio28.sh"
        ;;
    29)
        echo "Ejecutando Ejercicio 29..."
        echo
        bash "$SCRIPTS_DIR/ejercicio29.sh"
        ;;
    30)
        echo "Ejecutando Ejercicio 30..."
        echo
        bash "$SCRIPTS_DIR/ejercicio30.sh"
        ;;
    31)
        echo "Ejecutando Ejercicio 31..."
        echo
        bash "$SCRIPTS_DIR/ejercicio31.sh"
        ;;
    99)
        echo "Saliendo del menú. ¡Hasta luego!"
        echo
        break
        ;;
    *)
        echo "Opción inválida. Intentar nuevamente"
        ;;
  esac
  echo
  echo -n "Presionar ENTER para continuar..."
  read
done
```

Ejercicio 20.

Realizar un script que simule el comportamiento de una estructura de PILA e implementar 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.*

Dentro del mismo script y, utilizando las funciones implementadas:

- *Agregar 10 elementos a la pila*
- *Sacar 3 de ellos.*
- *Imprimir la longitud de la pila.*
- *Luego, imprimir la totalidad de los elementos que en ella se encuentran.*

```
#!/bin/bash
# Script: Ejercicio20.sh
# Uso: ./Ejercicio20.sh

# ESTRUCTURA GLOBAL
pila=()

# FUNCIONES

# Push: Recibe un parámetro y lo agrega a la pila
push() {
   pila+=("$1")
}

# Pop: Saca un elemento de la pila
pop() {
   if [ ${#pila[@]} -eq 0 ]; then
      echo "La pila está vacía. No se puede hacer pop"
   else
      unset 'pila[-1]'
   fi
}

# Length: Devuelve la longitud de la pila
length() {
   echo "${#pila[@]}"
}

# Print: Imprime todos los elementos de la pila
print() {
   for elem in "${pila[@]}"; do
```

```
        echo " - $elem"
    done
}

# PRUEBA

echo "Agregando 10 elementos a la pila..."
for i in {1..10}; do
    push "Elemento_$i"
done

echo "Sacando 3 elementos de la pila..."
for i in {1..3}; do
    pop
done

echo "Longitud de la pila: $(length)"

echo "Impresión de la totalidad de los elementos en la pila:"
print
```

Ejercicio 21.

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).

Implementar la función productoria dentro de este script, cuya tarea sea multiplicar todos los números que el arreglo contiene.

```
#!/bin/bash
# Script: Ejercicio21.sh
# Uso: ./Ejercicio21.sh

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

# Algoritmo
productoria() {
   local prod=1
   for n in "${num[@]}"; do
      prod=$((prod * n))
   done
   echo "$prod"
}

# Resultado
echo "La productoria del vector (${num[@]}) es: $(productoria)"
```

Ejercicio 22.

Implementar 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
# Script: Ejercicio22.sh
# Uso: ./Ejercicio22.sh

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

# Algoritmo
impares=0
for n in "${num[@]}"; do
   if (( n % 2 == 0 )); then
      echo "Número par: $n"
   else
      ((impares++))
   fi
done

# Resultado
echo "Cantidad de números impares: $impares"
```

Ejercicio 23.

Dada la definición de 2 vectores del mismo tamaño y cuyas longitudes no se conocen.

vector1=(1 .. N).
vector2=(1.. N).

Por ejemplo:

vector1=(1 80 65 35 2) y vector2=(5 98 3 41 8).

Completar 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:

- *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
# Script: Ejercicio23.sh
# Uso: ./Ejercicio23.sh

# Vectores
vector1=(1 80 65 35 2)
vector2=(5 98 3 41 8)

# Algoritmo
for ((i=0; i<${#vector1[@]}; i++)); do
   suma=$(( ${vector1[i]} + ${vector2[i]} ))
   echo "La suma de los elementos de la posición $i de los vectores es $suma"
done
```

Ejercicio 24.

Realizar 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
# Script: Ejercicio24.sh
# Uso: ./Ejercicio24.sh

# Grupo
group="users"

# Obtención de todos los nombres de los usuarios del sistema pertenecientes al grupo "$group"
usuarios=($(getent group "$group" | cut -d: -f4 | tr ',' ' '))

# Verificación de usuarios encontrados
if [ ${#usuarios[@]} -eq 0 ]; then
   echo "No se encontraron usuarios del sistema pertenecientes al grupo '$group'"
   exit 1
fi

# Algoritmo
case "$1" in
   -b)
      n=$2
      if [ -z "$n" ]; then
         echo "Error: Se debe especificar un número después de '-b'"
         exit 1
      elif [ "$n" -ge 0 ] && [ "$n" -lt "${#usuarios[@]}" ]; then
         echo "Elemento en la posición $n: ${usuarios[$n]}"
      else
         echo "Error: No existe la posición $n en el arreglo"
      fi
      ;;
   -l)
      echo "La longitud del arreglo es: ${#usuarios[@]}"
      ;;
   -i)
      echo "Usuarios en el grupo '$group':"
      for u in "${usuarios[@]}"; do
         echo " - $u"
      done
```

```
        ;;
    *)
        echo "Uso: $0 [-b n | -l | -i]"
        echo " -b n : Retorna el elemento de la posición n del arreglo"
        echo " -l   : Retorna la longitud del arreglo"
        echo " -i   : Imprime todos los elementos del arreglo"
        ;;
esac
```

Ejercicio 25.

Escribir 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, sólo para aquellos parámetros que se encuentren en posiciones impares (el primero, el tercero, etc.), verificar si el archivo o directorio existen en el sistema, imprimiendo en pantalla qué tipo de objeto es (archivo o directorio). Además, se deberá informar la cantidad de archivos o directorios inexistentes en el sistema.

```
#!/bin/bash
# Script: Ejercicio25.sh
# Uso: ./Ejercicio25.sh

# Control de cantidad de parámetros
if [ $# -lt 1 ]; then
   echo "Error: Se debe ingresar, al menos, 1 parámetro"
   echo "Uso: $0 <ruta1> <ruta2> ..."
   exit 1
fi

# Algoritmo
inexistentes=0
posicion=1
for ruta in "$@"; do
   if (( posicion % 2 != 0 )); then
      if [ -e "$ruta" ]; then
         if [ -f "$ruta" ]; then
            echo "Posición $posicion: '$ruta' existe y es un archivo"
         elif [ -d "$ruta" ]; then
            echo "Posición $posicion: '$ruta' existe y es un directorio"
         else
            echo "Posición $posicion: '$ruta' existe, pero no es ni archivo ni directorio"
         fi
      else
         echo "Posición $posicion: '$ruta' no existe"
         ((inexistentes++))
      fi
   fi
   ((posicion++))
done

# Resultado
echo "Cantidad de archivos o directorios inexistentes: $inexistentes"
```

Ejercicio 26.

Realizar 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 <parametro1>: Agrega al final del arreglo el parámetro recibido.*
- *eliminar_elem <parametro1>: 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 <parametro1><parametro2>: Crea un arreglo con longitud <parametro1> y, en todas las posiciones, asigna el valor <parametro2>.*

```
#!/bin/bash
# Script: Ejercicio26.sh
# Uso: ./Ejercicio26.sh

# FUNCIONES

# Inicializar: Crea un arreglo llamado array vacío
inicializar() {
   array=()
   echo "Arreglo llamado 'array' creado vacío"
}

# Agregar_elem: Agrega al final del arreglo el parámetro recibido
agregar_elem() {
   if [ $# -ne 1 ]; then
      echo "Error: Se debe indicar el valor para agregar"
      return 1
   fi
   array+=("$1")
   echo "Elemento '$1' agregado al final del arreglo"
}

# Eliminar_elem: Elimina del arreglo el elemento que se encuentra en la posición recibida como parámetro
eliminar_elem() {
   if [ $# -ne 1 ]; then
      echo "Error: Se debe indicar la posición a eliminar"
      return 1
   fi
   local pos=$1
   if [ "$pos" -lt 0 ] || [ "$pos" -ge "${#array[@]}" ]; then
      echo "Error: Posición fuera de rango (0 a $((${#array[@]} - 1)))"
      return 1
```

```
    fi
    unset 'array[pos]'
    array=("${array[@]}")
    echo "Elemento en posición $pos eliminado"
}

# Longitud: Imprime la longitud del arreglo
longitud() {
    echo "Longitud del arreglo: ${#array[@]}"
}

# Imprimir: Imprime todos los elementos del arreglo
imprimir() {
    if [ ${#array[@]} -eq 0 ]; then
        echo "El arreglo está vacío"
    else
        echo "Elementos del arreglo: ${array[@]}"
    fi
}

# Inicializar_con_valores: Crea un arreglo con longitud <parametro1> y, en todas las
posiciones, asigna el valor <parametro2>
inicializar_con_valores() {
    if [ $# -ne 2 ]; then
        echo "Error: Se debe indicar longitud y valor"
        return 1
    fi
    local longitud=$1
    local valor=$2
    array=()
    for ((i = 0; i < longitud; i++)); do
        array+=("$valor")
    done
    echo "Arreglo llamado 'array' creado con longitud $longitud y valor '$valor' en todas
las posiciones"
}

# PRUEBA

inicializar
agregar_elem "A"
agregar_elem "B"
agregar_elem "C"
imprimir
eliminar_elem 1
imprimir
longitud
inicializar_con_valores 3 X
imprimir
```

Ejercicio 27.

Realizar un script que reciba como parámetro el nombre de un directorio. Se 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
# Script: Ejercicio27.sh
# Uso: ./Ejercicio27.sh

# Control de cantidad de parámetros
if [ $# -ne 1 ]; then
   echo "Error: Se debe ingresar, exactamente, 1 parámetro"
   echo "Uso: $0 <directorio>"
   exit 1
fi

# Asignación de parámetros
directorio="$1"

# Verificación de que el directorio exista
if [ ! -d "$directorio" ]; then
   echo "Error: '$directorio' no es un directorio válido"
   exit 4
fi

# Algoritmo
lectura=0
escritura=0
for archivo in "$directorio"/*; do
   if [ -f "$archivo" ]; then
      if [ -r "$archivo" ]; then
         ((lectura++))
      fi
      if [ -w "$archivo" ]; then
         ((escritura++))
      fi
   fi
done

# Resultados
echo "Archivos con permiso de lectura: $lectura"
echo "Archivos con permiso de escritura: $escritura"
```

Ejercicio 28.

Implementar un script que agregue a un arreglo todos los archivos del directorio /home cuya terminación sea .doc. Adicionalmente, implementar las siguientes funciones que 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 "Sí", elimina el elemento sólo 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.*

Ejercicio 29.

Realizar 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.

Ejercicio 30.

Implementar la estructura de datos Set (Conjunto de valores) en Bash. Un conjunto se define como una colección de valores únicos, es decir, que sólo almacena una vez cada valor, aun cuando se intente agregar el mismo valor más de una vez. La implementación debe soportar las siguientes operaciones mediante funciones:

- *initialize: Inicializa el set vacío.*
- *initialize_with: Inicializa el set con un conjunto de valores que recibe como argumento (debe validar que se reciba, al menos, uno).*
- *add: Agrega un valor al conjunto, el cual recibe como argumento. No debe agregar elementos repetidos. El resultado de la operación será un éxito sólo si el valor puede ser agregado al conjunto.*
- *remove: Elimina uno o más valores del conjunto, los cuales recibe como argumentos. Si la operación elimina, al menos, un valor, se considera un éxito.*
- *contains: Chequea si el conjunto contiene un valor recibido como argumento. El resultado será éxito si el valor está en el conjunto.*
- *print: Imprime los elementos del conjunto, de a uno por línea.*
- *print_sorted: Imprime los elementos del conjunto, de a uno por línea y ordenados alfabéticamente. Tip: Investigar cómo combinar el comando sort con la función print.*

En un script separado, incorporar y utilizar las funciones implementadas para desarrollar un juego de bingo. El bingo deberá generar números aleatorios dentro de un rango entre 0 y un valor máximo que puede especificarse mediante un argumento del script, de manera opcional. El valor máximo no puede ser 0 ni superior a 32.767, y en caso de no especificarse, se tomará como valor por defecto 99. En cada ronda, se generará un nuevo número que ya no haya sido utilizado y se lo cantará, imprimiendo en la salida estándar. Luego de esto, se esperará entrada del usuario para saber si se debe cantar "BINGO" para finalizar la partida o se debe cantar un nuevo número. Al finalizar, el script deberá imprimir, en orden, los números que se cantaron hasta que se produjo el bingo.

Tip: Investigar la variable de entorno $RANDOM de Bash para obtener valores aleatorios.

Ejercicio 31.

Realizar un script que reciba como argumento una lista de posibles nombres de usuarios del sistema y, para cada uno de los que, efectivamente, existan en el sistema y posean un directorio personal configurado que sea válido, realice las modificaciones necesarias en su directorio personal para que tenga un subdirectorio llamado "directorio_iso" con la siguiente estructura:

Para resolver la creación de los directorios y archivos, utilizar la funcionalidad "Brace Expansion" brindada por bash: https://www.gnu.org/software/bash/manual/html_node/Brace-Expansion.html.