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

\*\*Shell Scripting\*\* es una técnica de programación que utiliza comandos de shell (intérprete de comandos) para automatizar tareas en sistemas operativos, principalmente en entornos Unix/Linux. Consiste en escribir una secuencia de comandos en un archivo de texto que puede ser ejecutado por el shell.

\*\*Tipos de tareas orientadas a scripts:\*\*

- Automatización de tareas administrativas del sistema

- Procesamiento por lotes (batch processing)

- Manipulación de archivos y directorios

- Configuración de servicios y entornos

- Monitoreo del sistema

- Instalación y despliegue de aplicaciones

- Procesamiento de textos y datos

- Tareas de backup y mantenimiento

\*\*¿Los scripts deben compilarse?\*\*

No, los scripts de shell \*\*no necesitan compilación\*\*. Esto se debe a que son \*\*interpretados\*\*, no compilados. El shell lee y ejecuta los comandos línea por línea en tiempo de ejecución, a diferencia de los lenguajes compilados como C o Java que requieren ser traducidos previamente a código máquina.

\*\*Razones por las que no se compilan:\*\*

- Son lenguajes interpretados que se ejecutan directamente por el shell

- Permiten una ejecución inmediata sin pasos intermedios

- Facilitan la portabilidad entre diferentes sistemas

- Permiten modificaciones rápidas sin necesidad de recompilar

Claro, aquí tienes la explicación del \*\*punto 2\*\* en formato texto:

---

\*\*2. Investigar la funcionalidad de los comandos echo y read.\*\*

\*\*Comando ECHO:\*\*

El comando `echo` se utiliza para mostrar mensajes o contenido de variables en la salida estándar (pantalla). Es fundamental para la interacción con el usuario en scripts bash.

\*\*Ejemplos:\*\*

- `echo "Hola Mundo"` - Muestra "Hola Mundo"

- `echo $VARIABLE` - Muestra el contenido de una variable

- `echo -e "Texto\ncon salto de línea"` - Habilita interpretación de caracteres especiales

\*\*Comando READ:\*\*

El comando `read` se utiliza para leer entrada del usuario desde la terminal y almacenarla en variables. Permite la interactividad en los scripts.

\*\*Ejemplos:\*\*

- `read nombre` - Lee una entrada y la guarda en la variable "nombre"

- `read -p "Ingrese su edad: " edad` - Muestra un mensaje y lee la entrada

- `read nombre apellido` - Lee múltiples valores separados por espacios

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

Los comentarios se indican con el símbolo `#` al inicio de la línea. Todo lo que sigue después de `#` en la misma línea es ignorado por el shell.

\*\*Ejemplos:\*\*

```bash

# Este es un comentario de una línea

echo "Hola" # Este es un comentario al final de una línea

```

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

\*\*Declaración de variables:\*\*

- Se declaran sin el símbolo `$`: `nombre="Juan"`

- No spaces alrededor del signo igual: `variable=valor`

- Pueden contener letras, números y guiones bajos

\*\*Referencia a variables:\*\*

- Se referencian con `$` antes del nombre: `echo $nombre`

- O usando `${}`: `echo ${nombre}` (útil para concatenar)

\*\*Ejemplos:\*\*

```bash

nombre="Maria" # Declaración

edad=25 # Declaración de número

echo $nombre # Referencia - muestra "Maria"

echo "Hola $nombre" # Referencia en texto - "Hola Maria"

echo "Edad: ${edad}años" # Referencia con {} - "Edad: 25años"

```

Claro, aquí tienes la explicación del \*\*punto 4\*\* en formato texto:

---

\*\*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?\*\*

\*\*Acceso a parámetros:\*\*

Los parámetros enviados al script al momento de su invocación se acceden mediante variables especiales:

- `$0` - Nombre del script

- `$1` - Primer parámetro

- `$2` - Segundo parámetro

- `$3` - Tercer parámetro

- ... y así sucesivamente hasta `$9`

- `${10}` - Décimo parámetro (para parámetros beyond 9)

\*\*Ejemplo:\*\*

Si ejecutamos: `./mi\_script.sh hola mundo 123`

- `$0` = "./mi\_script.sh"

- `$1` = "hola"

- `$2` = "mundo"

- `$3` = "123"

\*\*Variables especiales y su información:\*\*

\*\*$#\*\* - Contiene el \*\*número de parámetros\*\* pasados al script

```bash

# Si: ./script.sh param1 param2 param3

echo $# # Muestra: 3

```

\*\*$\*\*\* - Contiene \*\*todos los parámetros\*\* como una sola cadena

```bash

# Si: ./script.sh hola mundo bash

echo $\* # Muestra: "hola mundo bash"

```

\*\*$?\*\* - Contiene el \*\*código de salida\*\* del último comando ejecutado

```bash

ls /directorio\_inexistente

echo $? # Muestra un código de error (generalmente 1 o 2)

echo "Hola"

echo $? # Muestra 0 (éxito)

```

\*\*$HOME\*\* - Contiene la \*\*ruta del directorio personal\*\* del usuario actual

```bash

echo $HOME # Muestra: /home/usuario (o /Users/usuario en macOS)

```

\*\*Otras variables útiles:\*\*

- `$@` - Similar a `$\*` pero preserva los espacios en los parámetros

- `$$` - PID (Process ID) del script actual

- `$!` - PID del último proceso ejecutado en background

Claro, aquí tienes la explicación del \*\*punto 5\*\* en formato texto:

---

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

\*\*Funcionalidad del comando EXIT:\*\*

El comando `exit` se utiliza para \*\*terminar la ejecución de un script\*\* o sesión de shell y devolver un \*\*código de salida\*\* al proceso padre. Este código indica si el script finalizó correctamente o si ocurrió algún error.

\*\*Valores que recibe como parámetro:\*\*

El comando `exit` recibe un \*\*número entero\*\* como parámetro, que representa el código de salida (exit status).

\*\*Significado de los valores:\*\*

- \*\*`exit 0`\*\* - \*\*Éxito\*\*: Indica que el script terminó correctamente sin errores

- \*\*`exit 1`\*\* - \*\*Error general\*\*: Indica que ocurrió un error no especificado

- \*\*`exit 2`\*\* - \*\*Uso incorrecto\*\*: Indica que se usó el script de manera incorrecta (parámetros erróneos)

- \*\*`exit 126`\*\* - \*\*No ejecutable\*\*: El archivo no es ejecutable o no se tienen permisos

- \*\*`exit 127`\*\* - \*\*Comando no encontrado\*\*: El comando solicitado no existe

- \*\*`exit 130`\*\* - \*\*Terminado con Ctrl+C\*\*: El script fue interrumpido por el usuario

- \*\*Cualquier otro número\*\* - \*\*Error personalizado\*\*: Puede definirse según las necesidades del script

\*\*Ejemplos de uso:\*\*

```bash

#!/bin/bash

# Verificar si existe un archivo

if [ ! -f "archivo.txt" ]; then

echo "Error: El archivo no existe"

exit 1

fi

# Verificar número de parámetros

if [ $# -lt 2 ]; then

echo "Uso: $0 <param1> <param2>"

exit 2

fi

# Proceso exitoso

echo "Script ejecutado correctamente"

exit 0

```

\*\*Verificación del código de salida:\*\*

Después de ejecutar un script, se puede verificar su código de salida con `$?`:

```bash

./mi\_script.sh

echo $? # Muestra el código de salida del script

```

\*\*Importancia:\*\*

Los códigos de salida son fundamentales para:

- Automatización de tareas

- Control de flujo entre scripts

- Detección y manejo de errores

- Integración con sistemas de monitoreo

Claro, aquí tienes la explicación del \*\*punto 6\*\* en formato texto:

---

\*\*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 que tipo de operaciones se pueden utilizar.\*\*

\*\*Comando EXPR:\*\*

El comando `expr` (expression evaluator) es una utilidad de Unix/Linux que permite \*\*evaluar expresiones\*\* matemáticas y de cadenas directamente desde la línea de comandos o en scripts bash.

\*\*Sintaxis básica:\*\*

```bash

expr arg1 operación arg2

```

\*\*Tipos de operaciones disponibles:\*\*

\*\*Operaciones aritméticas:\*\*

- `+` - \*\*Suma\*\*: `expr 5 + 3` → 8

- `-` - \*\*Resta\*\*: `expr 10 - 4` → 6

- `\*` - \*\*Multiplicación\*\*: `expr 6 \* 3` → 18 (¡usar \\* escapado!)

- `/` - \*\*División entera\*\*: `expr 15 / 4` → 3

- `%` - \*\*Módulo\*\* (resto de división): `expr 15 % 4` → 3

\*\*Operaciones con cadenas:\*\*

- `:` - \*\*Comparación/coincidencia\*\*: `expr "cadena" : ".\*"` → longitud

- `match` - \*\*Coincidencia con expresión regular\*\*: `expr match "abc" ".\*"` → 3

- `substr` - \*\*Subcadena\*\*: `expr substr "holamundo" 4 5` → "mundo"

- `index` - \*\*Posición de carácter\*\*: `expr index "abcdef" "d"` → 4

- `length` - \*\*Longitud de cadena\*\*: `expr length "hola"` → 4

\*\*Operaciones lógicas/comparación:\*\*

- `&` - \*\*AND\*\* lógico

- `|` - \*\*OR\*\* lógico

- `=, >, >=, <, <=, !=` - \*\*Comparaciones\*\* (devuelven 1 si es verdadero, 0 si es falso)

\*\*Ejemplos prácticos:\*\*

```bash

#!/bin/bash

# Operaciones aritméticas

resultado=$(expr 10 + 5)

echo "10 + 5 = $resultado"

resultado=$(expr 20 \\* 2) # ¡Importante: escapar el \*

echo "20 \* 2 = $resultado"

# Operaciones con cadenas

longitud=$(expr length "Hola Mundo")

echo "Longitud de 'Hola Mundo': $longitud"

subcadena=$(expr substr "abcdefgh" 3 4)

echo "Subcadena desde posición 3, 4 caracteres: $subcadena"

# En scripts con variables

a=15

b=4

division=$(expr $a / $b)

echo "15 / 4 = $division"

modulo=$(expr $a % $b)

echo "15 % 4 = $modulo"

```

\*\*Consideraciones importantes:\*\*

- Los \*\*operadores deben estar rodeados de espacios\*\*

- El asterisco `\*` para multiplicación debe \*\*escaparse con \\*\*

- `expr` solo trabaja con \*\*números enteros\*\* (no soporta decimales)

- Es útil en scripts bash antiguos, pero en bash moderno se prefiere `$(( ))`

\*\*Alternativa moderna en bash:\*\*

```bash

# En lugar de expr, se puede usar:

resultado=$((10 + 5))

resultado=$((a \* b))

```

Claro, te explico lo de \*\*escapar el \*\*\*:

---

\*\*¿Por qué hay que escapar el \* en expr?\*\*

El carácter `\*` tiene un \*\*significado especial\*\* en la línea de comandos de Linux - representa \*\*comodín\*\* o \*\*wildcard\*\*, que se expande para coincidir con nombres de archivos.

\*\*Problema:\*\*

Cuando escribes `expr 5 \* 3`, el shell \*\*antes de ejecutar expr\*\* intenta expandir el `\*` para que coincida con todos los archivos en el directorio actual.

\*\*Ejemplo práctico:\*\*

Si en tu directorio actual tienes estos archivos: `archivo1.txt`, `archivo2.txt`, `script.sh`

```bash

# Esto NO funciona como esperas:

expr 5 \* 3

# El shell interpreta \* como "todos los archivos" y lo convierte en:

expr 5 archivo1.txt archivo2.txt script.sh 3

# Resultado: expr: error de sintaxis

```

\*\*Solución - Escapar el \*:\*\*

Hay dos formas de evitar que el shell interprete el `\*` como comodín:

1. \*\*Usando backslash (\):\*\*

```bash

expr 5 \\* 3 # Resultado: 15

```

2. \*\*Usando comillas:\*\*

```bash

expr 5 "\*" 3 # Resultado: 15

```

\*\*¿Por qué no pasa con otros operadores?\*\*

Los otros operadores matemáticos (`+`, `-`, `/`, `%`) \*\*NO\*\* tienen significado especial para el shell, por eso no necesitan escaparse:

```bash

expr 5 + 3 # ✓ Correcto - no necesita escape

expr 10 - 4 # ✓ Correcto - no necesita escape

expr 15 / 3 # ✓ Correcto - no necesita escape

expr 7 % 2 # ✓ Correcto - no necesita escape

expr 6 \\* 2 # ✓ Correcto - ¡sí necesita escape!

```

\*\*En resumen:\*\*

- `\*` = comodín para el shell → \*\*necesita escape\*\*

- `+`, `-`, `/`, `%` = no son comodines → \*\*no necesitan escape\*\*

\*\*Ejemplo en script:\*\*

```bash

#!/bin/bash

num1=6

num2=3

# Multiplicación CORRECTA (con escape)

resultado=$(expr $num1 \\* $num2)

echo "6 \* 3 = $resultado" # Muestra: 6 \* 3 = 18

# Esto daría ERROR:

# resultado=$(expr $num1 \* $num2) # ❌ Error de sintaxis

```

Claro, aquí tienes la explicación del \*\*punto 7\*\* en formato texto:

---

\*\*7. El comando test \_expression\_ 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.\*\*

\*\*Comando TEST:\*\*

El comando `test` evalúa expresiones y devuelve un \*\*código de salida\*\* (0 para verdadero, 1 para falso). Se puede usar de dos formas equivalentes:

```bash

test expresión

# o

[ expresión ]

```

\*\*Importante:\*\* Los corchetes deben tener \*\*espacios\*\* alrededor: `[ espacioexpresiónespacio ]`

\*\*Tipos de expresiones disponibles:\*\*

\*\*Evaluación de ARCHIVOS:\*\*

- `-e archivo` - Verifica si el archivo \*\*existe\*\*

- `-f archivo` - Verifica si es un \*\*archivo regular\*\* (no directorio)

- `-d archivo` - Verifica si es un \*\*directorio\*\*

- `-r archivo` - Verifica si es \*\*legible\*\*

- `-w archivo` - Verifica si es \*\*escribible\*\*

- `-x archivo` - Verifica si es \*\*ejecutable\*\*

- `-s archivo` - Verifica si el tamaño es \*\*mayor que cero\*\*

- `-L archivo` - Verifica si es un \*\*enlace simbólico\*\*

\*\*Evaluación de CADENAS de caracteres:\*\*

- `-z cadena` - Verifica si la cadena está \*\*vacía\*\* (longitud cero)

- `-n cadena` - Verifica si la cadena \*\*NO está vacía\*\*

- `cadena1 = cadena2` - Verifica si las cadenas son \*\*iguales\*\*

- `cadena1 != cadena2` - Verifica si las cadenas son \*\*diferentes\*\*

\*\*Evaluaciones NUMÉRICAS:\*\*

- `num1 -eq num2` - Equal: verifica si son \*\*iguales\*\*

- `num1 -ne num2` - Not equal: verifica si son \*\*diferentes\*\*

- `num1 -lt num2` - Less than: verifica si \*\*num1 < num2\*\*

- `num1 -le num2` - Less or equal: verifica si \*\*num1 <= num2\*\*

- `num1 -gt num2` - Greater than: verifica si \*\*num1 > num2\*\*

- `num1 -ge num2` - Greater or equal: verifica si \*\*num1 >= num2\*\*

\*\*Operadores LÓGICOS:\*\*

- `! expresión` - NOT: niega la expresión

- `expresión1 -a expresión2` - AND: ambas deben ser verdaderas

- `expresión1 -o expresión2` - OR: al menos una debe ser verdadera

\*\*Ejemplos prácticos:\*\*

```bash

#!/bin/bash

# Evaluación de archivos

if [ -f "mi\_archivo.txt" ]; then

echo "El archivo existe y es regular"

fi

if [ -d "/home/usuario" ]; then

echo "Es un directorio"

fi

# Evaluación de cadenas

nombre="Juan"

if [ -n "$nombre" ]; then

echo "La variable nombre no está vacía"

fi

if [ "$nombre" = "Juan" ]; then

echo "El nombre es Juan"

fi

# Evaluación numérica

edad=25

if [ $edad -ge 18 ]; then

echo "Es mayor de edad"

fi

# Combinación de expresiones

if [ -f "archivo.txt" -a -r "archivo.txt" ]; then

echo "El archivo existe y es legible"

fi

# Usando test explícitamente

if test -f "archivo.txt"; then

echo "Archivo encontrado (usando test)"

fi

```

\*\*Uso en condiciones IF:\*\*

```bash

if [ expresión ]; then

# Código si es verdadero

else

# Código si es falso

fi

```

\*\*Verificación del resultado:\*\*

```bash

[ -f "archivo.txt" ]

echo $? # 0 si existe, 1 si no existe

```

\*\*Consideraciones importantes:\*\*

- Siempre usar espacios alrededor de los corchetes

- Usar comillas alrededor de variables de cadena: `[ "$var" = "valor" ]`

- Para números usar operadores `-eq`, `-lt`, etc. (NO usar `=`, `>`, `<`)

Claro, aquí tienes la explicación del \*\*punto 8\*\* en formato texto:

---

\*\*8. Estructuras de control. Investigue la sintaxis de las siguientes estructuras de control incluidas en shell scripting: if, case, while, for, select\*\*

\*\*Estructura IF (condicional):\*\*

```bash

# Sintaxis básica

if [ condición ]; then

# comandos si condición es verdadera

elif [ otra\_condición ]; then

# comandos si otra\_condición es verdadera

else

# comandos si todas las condiciones son falsas

fi

# Ejemplo

if [ $edad -ge 18 ]; then

echo "Eres mayor de edad"

elif [ $edad -ge 13 ]; then

echo "Eres adolescente"

else

echo "Eres niño"

fi

```

\*\*Estructura CASE (selección múltiple):\*\*

```bash

# Sintaxis básica

case $variable in

patrón1)

# comandos para patrón1

;;

patrón2)

# comandos para patrón2

;;

patrón3|patrón4)

# comandos para patrón3 o patrón4

;;

\*)

# comandos por defecto (opcional)

;;

esac

# Ejemplo

case $opcion in

"start")

echo "Iniciando servicio..."

;;

"stop")

echo "Deteniendo servicio..."

;;

"restart"|"reload")

echo "Reiniciando servicio..."

;;

\*)

echo "Opción no válida"

;;

esac

```

\*\*Estructura WHILE (bucle mientras):\*\*

```bash

# Sintaxis básica

while [ condición ]; do

# comandos a repetir

# (importante modificar condición para evitar loop infinito)

done

# Ejemplo

contador=1

while [ $contador -le 5 ]; do

echo "Iteración número $contador"

contador=$((contador + 1))

done

# Lectura de archivo línea por línea

while read linea; do

echo "Línea: $linea"

done < archivo.txt

```

\*\*Estructura FOR (bucle para):\*\*

```bash

# Sintaxis básica - lista de valores

for variable in valor1 valor2 valor3 ...; do

# comandos usando $variable

done

# Ejemplos

for fruta in manzana pera naranja; do

echo "Fruta: $fruta"

done

# Con rangos numéricos

for i in {1..5}; do

echo "Número: $i"

done

# Con incremento

for i in {1..10..2}; do

echo "Número: $i" # 1, 3, 5, 7, 9

done

# Con parámetros del script

for parametro in "$@"; do

echo "Parámetro: $parametro"

done

# Sintaxis estilo C

for ((i=0; i<5; i++)); do

echo "Iteración: $i"

done

```

\*\*Estructura SELECT (menú interactivo):\*\*

```bash

# Sintaxis básica

select variable in opción1 opción2 opción3 ...; do

case $variable in

opción1)

# comandos para opción1

;;

opción2)

# comandos para opción2

;;

# ...

\*)

echo "Opción no válida"

break

;;

esac

done

# Ejemplo

echo "Selecciona una fruta:"

select fruta in Manzana Pera Naranja Salir; do

case $fruta in

Manzana)

echo "Seleccionaste Manzana"

;;

Pera)

echo "Seleccionaste Pera"

;;

Naranja)

echo "Seleccionaste Naranja"

;;

Salir)

echo "Saliendo..."

break

;;

\*)

echo "Opción no válida"

;;

esac

done

```

\*\*Ejemplo combinado con estructuras anidadas:\*\*

```bash

#!/bin/bash

# Menú interactivo con select y case

echo "Sistema de gestión de usuarios"

select opcion in "Listar usuarios" "Agregar usuario" "Eliminar usuario" "Salir"; do

case $opcion in

"Listar usuarios")

# Usando for para iterar

for usuario in $(who | cut -d' ' -f1); do

echo "Usuario conectado: $usuario"

done

;;

"Agregar usuario")

# Usando while para validación

while true; do

read -p "Nombre de usuario: " username

if [ -n "$username" ]; then

echo "Agregando usuario: $username"

break

else

echo "El nombre no puede estar vacío"

fi

done

;;

"Eliminar usuario")

echo "Función de eliminar usuario"

;;

"Salir")

echo "Hasta luego!"

break

;;

\*)

echo "Opción no válida, intenta de nuevo"

;;

esac

done

```

\*\*Consideraciones importantes:\*\*

- En `if` y `while`, siempre usar espacios alrededor de los corchetes

- En `case`, cada bloque termina con `;;`

- En `for`, la lista de valores puede venir de variables, comandos o rangos

- `select` es ideal para menús interactivos simples

- Usar `break` para salir de bucles y `continue` para saltar a la siguiente iteración

Claro, aquí tienes la explicación del \*\*punto 9\*\* en formato texto:

---

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

\*\*Sentencia BREAK:\*\*

La sentencia `break` se utiliza para \*\*salir inmediatamente\*\* de un bucle (for, while, until, select), interrumpiendo su ejecución normal y continuando con el código que está después del bucle.

\*\*Comportamiento:\*\*

- Termina la ejecución del bucle actual

- El control del programa pasa a la siguiente instrucción después del `done`

- Puede usarse para salir de bucles infinitos o cuando se cumple una condición específica

\*\*Parámetros que recibe:\*\*

`break` puede recibir un \*\*parámetro numérico opcional\*\* que indica cuántos niveles de bucles anidados se deben romper:

- `break` o `break 1` - Sale del bucle actual (por defecto)

- `break 2` - Sale de 2 niveles de bucles anidados

- `break n` - Sale de n niveles de bucles anidados

\*\*Ejemplos de BREAK:\*\*

```bash

# Break sin parámetro - sale del bucle actual

for i in {1..10}; do

if [ $i -eq 5 ]; then

break

fi

echo "Número: $i"

done

# Resultado: 1, 2, 3, 4

# Break con parámetro - sale de múltiples niveles

for i in {1..3}; do

echo "Bucle externo: $i"

for j in {1..3}; do

echo " Bucle interno: $j"

if [ $j -eq 2 ]; then

break 2 # Sale de ambos bucles

fi

done

done

# Resultado:

# Bucle externo: 1

# Bucle interno: 1

# Bucle interno: 2

```

\*\*Sentencia CONTINUE:\*\*

La sentencia `continue` se utiliza para \*\*saltar a la siguiente iteración\*\* del bucle actual, omitiendo el resto de comandos en la iteración actual pero continuando con el bucle.

\*\*Comportamiento:\*\*

- Salta los comandos restantes en la iteración actual

- Pasa inmediatamente a la siguiente iteración del bucle

- No termina el bucle, solo salta una iteración específica

\*\*Parámetros que recibe:\*\*

`continue` también puede recibir un \*\*parámetro numérico opcional\*\* que indica cuántos niveles de bucles anidados se deben saltar:

- `continue` o `continue 1` - Salta a la siguiente iteración del bucle actual (por defecto)

- `continue 2` - Salta a la siguiente iteración del bucle que está 2 niveles arriba

- `continue n` - Salta a la siguiente iteración del bucle que está n niveles arriba

\*\*Ejemplos de CONTINUE:\*\*

```bash

# Continue sin parámetro - salta iteración actual

for i in {1..5}; do

if [ $i -eq 3 ]; then

continue

fi

echo "Número: $i"

done

# Resultado: 1, 2, 4, 5 (omite el 3)

# Continue con parámetro - salta niveles anidados

for i in {1..2}; do

echo "Bucle externo: $i"

for j in {1..3}; do

if [ $j -eq 2 ]; then

continue 2 # Salta al siguiente del bucle externo

fi

echo " Bucle interno: $j"

done

done

# Resultado:

# Bucle externo: 1

# Bucle interno: 1

# Bucle externo: 2

# Bucle interno: 1

```

\*\*Ejemplo práctico combinado:\*\*

```bash

#!/bin/bash

# Procesar archivos, omitir algunos y salir bajo condición

contador=0

for archivo in \*; do

# Omitir directorios

if [ -d "$archivo" ]; then

continue

fi

# Omitir archivos ocultos

if [[ "$archivo" == .\* ]]; then

continue

fi

# Salir después de procesar 5 archivos

if [ $contador -eq 5 ]; then

echo "Límite alcanzado, saliendo..."

break

fi

echo "Procesando: $archivo"

contador=$((contador + 1))

done

```

\*\*Resumen de diferencias:\*\*

- \*\*`break`\*\* - Termina el bucle completamente

- \*\*`continue`\*\* - Salta a la siguiente iteración del mismo bucle

- Ambos pueden recibir un \*\*parámetro numérico\*\* para trabajar con bucles anidados