UT 013.
INTRODUCCIÓN AL SHELL SCRIPTING

Sistemas informáticos
CFGS DAW

Álvaro
Maceda
a.macedaarranz@edu.gva.es

2022/2023

Versión:230324.1242

Licencia

Nomenclatura

A lo largo de esta unidad se utilizarán diferentes símbolos para distinguir elementos importantes dentro del contenido. Estos símbolos son:

 Importante

 Atención

 Interesante

ÍNDICE

UT 013. INTRODUCCIÓN AL SHELL SCRIPTING

1. ¿QUÉ ES UN SCRIPT DE SHELL ?

Un shell script es un programa destinado a ser ejecutado por un shell. Recuerda que el shell es el "lugar" donde ejecutamos las órdenes en la consola, por lo que se utilizan para automatizar cosas que habitualmente haremos utilizando la consola.

Los scripts del shell están compuestos de texto ASCII y pueden crearse utilizando un editor de texto, un procesador de textos o una interfaz gráfica de usuario. Estos scripts consisten en una secuencia de comandos en un lenguaje que el shell puede interpretar. Los shell scripts son capaces de soportar varias funciones, incluyendo bucles, variables, sentencias if/then/else, arrays y atajos. Una vez finalizado el script, se guarda normalmente con extensión .sh.

En este curso, trabajaremos con scripts de shell `bash`. Otros shells, como `zsh`, tienen otras capacidades y los scripts escritos para un shell no siempre se ejecutarán en otro shell.

1.1 Un ejemplo de script de shell

Este es un ejemplo sencillo de un script de shell. No te preocupes si aún no entiendes cómo funciona:

```
#!/usr/bin/env bash

# Inicializar el contador de directorios a cero
dir_count=0

# Recorre los archivos del directorio actual para
archivo en *
do
  # Comprueba si el fichero es un
  directorio if [ -d "$fichero" ]
  entonces
    echo "$archivo es un directorio"
    ((dir_count++)) # incrementa la cuenta de
  directorios else
    echo "$archivo no es un directorio"
  fi
hech
o

# Imprime el recuento de ficheros y directorios
echo "Encontrados $file_count archivos y $dir_count directorios"
```

El script comienza con una línea especial, el shebang. Después de eso, tienes comentarios (texto después del símbolo `#`) comandos (como `echo`) estructuras de control (`for`, `do`, `in`, `if`, `else...`) y variables (`dir_count`).

Para ejecutar el script, es necesario asignar al archivo permisos de ejecución y lanzarlo desde el shell. Suponiendo que se llame `simple_ejemplo.sh`:

```
./simple_ejemplo.sh
```

Una cosa importante es que **el script** de **shell también funcionará si escribimos el mismo texto directamente en el terminal**, porque lo que hacemos con un script de shell es almacenar en un archivo una serie de comandos para no tener que copiarlos una y otra vez al intentar hacer algo en el sistema.

1.1.1 shebang

El shebang (también llamado hashbang o pound bang) es la primera línea de un archivo de script que comienza con los caracteres `#!` (almohadilla y signo de exclamación). Se utiliza para especificar el intérprete o shell que debe utilizarse para ejecutar el script. Cuando se ejecuta un archivo de script, el sistema operativo lee la línea shebang para determinar el intérprete o shell que debe utilizarse para ejecutar los comandos del script. Por ejemplo, una línea shebang de `#!/bin/bash` indicaría que se debe utilizar el intérprete de comandos Bash para interpretar y ejecutar los comandos del script.

Deberías usar `#!/usr/bin/env bash` por portabilidad: diferentes *nixes ponen bash en diferentes lugares, y usar `/usr/bin/env` es una solución para ejecutar el primer `bash` encontrado en el PATH.

2. COMANDO S

La parte principal de un script de shell son los comandos que queremos ejecutar. Son los mismos comandos que utilizarías en una consola para realizar operaciones. Sin embargo, para usarlos correctamente en un script necesitas saber un poco más sobre cómo son procesados por el shell.

2.1 Cómo se interpreta una orden

El shell no ejecuta el comando tal y como lo introducimos en la línea de comandos Esto ocurre no sólo para los scripts, sino también para el comando que introducimos directamente en un shell.

Los pasos para ejecutar un comando son algo así:

1. Dividir el comando en tokens
2. Sustituir variables por sus valores
3. Realiza la sustitución de comandos
4. Realizar expansión comodín
5. Ejecutar el comando

Veremos qué ocurre cuando intentemos ejecutar un comando como éste:

```
ls          -l$HOME/*       |grep "$(fecha +"%b %e")"
```

2.1.1 Dividir int fichas

El primer paso es dividir la orden en "palabras" separadas en los caracteres de espacio en blanco. El shell ignora la cantidad de espacios en blanco (espacios y TAB) entre las distintas palabras de una línea de órdenes. Cualquier espacio en blanco no entrecomillado crea una nueva palabra.

En este caso, el shell dividirá el comando y lo verá como una lista de tokens como ésta:

```
ls, -l, $HOME/*, |, grep, "$(fecha +"%b %e")"
```

Tenga en cuenta que los caracteres dentro de las comillas no se dividen. El comando anterior, entonces, será equivalente a esto:

```
ls -l $HOME/* | grep "$(fecha +"%b %e")"
```

El shell también se da cuenta de la `|` y ejecutará dos comandos diferentes estableciendo una tubería entre ellos. Esto es un proceso transparente para nosotros.

2.1.2 Sustituir variables

El siguiente paso es reemplazar las variables con sus valores. En un shell, podemos tener variables que contienen algunos datos. Las variables se indican con un símbolo `$` delante. En el ejemplo anterior, `$HOME` es una variable y será sustituida por su valor. Suponiendo que la variable `$HOME` tuviera el valor
`/home/usuario`, El comando en este punto se parece a esto:

```
ls -l /home/user/* | grep "$(fecha +"%b %e")"
```

2.1.3 Sustitución de comandos

La sustitución de órdenes es una función que permite utilizar la salida de una orden como entrada para otra orden o asignarla a una variable. La sustitución de comandos se consigue encerrando el comando que se va a ejecutar dentro de `$(comando)` o backticks: `` `comando` ``. Ambas sintaxis son casi equivalentes, pero suele preferirse la sintaxis `$()` por su legibilidad.

En el ejemplo anterior, tenemos una sustitución de comando: `$(fecha +"%b %e")`. Será sustituido por el resultado de ejecutar ese comando. Si ejecutamos eso en la consola obtendríamos, por ejemplo:

```
echo $(fecha +"%b %e") # El comando echo es para visualizar el resultado

mar 20 # Esta será la fecha actual
```

Así, la expresión `$(fecha +"%b %e")` se sustituirá por mar 20. En este punto, el comando se ve así:

```
ls -l /home/user/* | grep "mar 20"
```

Tenga en cuenta que las comillas "fuera" de la expresión `$()` no se sustituyen.

2.1.4 Ampliación de comodines

En Linux, la expansión de comodines se refiere al proceso de utilizar caracteres especiales para construir patrones glob. Estos patrones representan varios nombres de archivo o directorios a la vez.

Los caracteres comodín más utilizados son:

- `*`: Representa cero o más caracteres. Por ejemplo, `ls *.txt` listará todos los archivos con un `.txt`
  en el directorio actual.
- `?` (signo de interrogación): Representa un único carácter. Por ejemplo, `ls archivo?.txt` mostrará los archivos `archivo1.txt`, `archivo2.txt`, etc., pero no `archivo10.txt`.

- `[]` (corchetes): Representa un rango de caracteres o un conjunto de caracteres. Por ejemplo, `ls archivo[1-3].txt` mostrará los archivos `archivo1.txt`, `archivo2.txt` y `archivo3.txt`, pero no `archivo4.txt`.

En el ejemplo, tenemos un carácter comodín `*`. Suponiendo que en `/home/usuario` hay tres ficheros
`archivo1`, `archivo2` y `archivo3`, se producirá la expansión y el comando quedará así:

```
ls -l /home/usuario/archivo1 /home/usuario/archivo2 /home/usuario/archivo3 |
grep "mar 20"
```

2.1.5 Ejecutar el comando

Una vez que tengamos la versión final del comando, el shell lo ejecutará. Tendremos el mismo resultado que si hubiéramos introducido la versión inicial (o cualquier versión intermedia) del comando.

2.2 V ariables

En Bash, una variable es un nombre que representa un valor. Las variables se utilizan para almacenar datos que pueden ser utilizados por scripts, programas o comandos.

Para crear una variable en Bash, basta con darle un nombre y asignarle un valor. Para leer el valor de una variable, se antepone un `$` al nombre de la variable. La sintaxis básica es:

```
A=valor
B=$A
echo $A,$B

valor,valor
```

Bash difiere de muchos otros lenguajes de programación en que no categoriza sus variables por "tipo". En Bash, las variables son esencialmente cadenas de caracteres, pero su capacidad para realizar operaciones aritméticas y comparaciones depende del contexto. Si el valor de una variable consiste únicamente en dígitos, Bash permite que se realicen tales operaciones y comparaciones.

2.2.1 Variables locales y de entorno

En Bash, hay dos tipos de variables: locales y de entorno. Las variables locales sólo están disponibles dentro del shell o script actual, mientras que las variables de entorno están disponibles para cualquier proceso hijo que se cree desde el shell actual. Para crear una variable de entorno, se utiliza el comando export.

Por ejemplo, si tenemos `script1.sh`:

```
#!/bin/bash
mi_var="¡Hola, mundo!"

./script2.sh
```

Y `script2.sh`:

```
/bin/bash
echo $my_var
```

Si ejecutamos `script1.sh` saldrá una cadena vacía porque `$my_var` no existe en `script2.sh`. Si añadimos un `export $my_var` a `script1.sh` antes de llamar a `script2.sh`, la salida será `"Hello, World!"`.

2.3 Expresiones aritméticas

En los scripts de shell, existe una forma de realizar operaciones matemáticas con valores enteros. Bash proporciona varios operadores aritméticos como la suma (`+`), la resta (`-`), la multiplicación (`*`), la división (`/`), el módulo (`%`) y la exponenciación (`**`), que pueden utilizarse para realizar operaciones matemáticas básicas.

Para realizar una evaluación numérica en los scripts de shell, debe encerrar la expresión matemática entre paréntesis dobles `$((expresión))`. Por ejemplo, para sumar dos números en scripts de shell, puede utilizar la siguiente sintaxis:

```
a=5
b=10
c=$((a + b))
echo $c

15
```

El comando `let` es otra forma de realizar operaciones aritméticas en los scripts del shell. Permite evaluar expresiones aritméticas y almacenar el resultado en una variable.

```
a=5
b=10
let c=a+b
echo $c

15
```

El comando `let` puede utilizarse con varios operadores aritméticos como `+`, `-`, `*`, `/`, `%`, `**` y `+=`, `-=` etc. He aquí algunos ejemplos:

```
a=5
b=10
let c=a+b       # adición
let d=a-b       # resta
let e=a*b       # multiplicación
let f=a/b       # división
let g=a%b       # Módulo
let h=a**b      # exponenciación
let a+=5        # incremento por
let b-=5        # decremento
```

2.4 Quo tes

En las secuencias de comandos del shell, las comillas se utilizan para controlar el modo en que el shell interpreta y expande variables, patrones glob y otros caracteres especiales en comandos y secuencias de comandos. Existen tres tipos de comillas en los scripts del shell: uno de ellos son las comillas inversas (`) utilizadas en la sustitución de comandos. Los otros dos son las comillas simples y las comillas dobles.

2.4.1 Comillas simples

Las comillas simples (') conservan el valor literal de todos los caracteres que contienen. Esto significa que las variables, los patrones glob y otros caracteres especiales entre comillas simples se tratan como texto sin formato y no son expandidos ni evaluados por el shell. Por ejemplo:

```
echo 'Hola $USUARIO'

Hola $USER
```

Observe que la variable `$USER` no se expande porque está entre comillas simples.

2.4.2 Comillas dobles

Las comillas dobles (`"`)permiten que ciertos caracteres especiales que contienen sean expandidos o evaluados por el shell. En concreto, las variables entre comillas dobles se sustituyen por sus valores, y algunos caracteres especiales como `$`, `*` y `? se` expanden para coincidir con nombres de archivo y realizar otras funciones. Por ejemplo:

```
echo "Hola $USUARIO"

Hola usuario1
```

Tenga en cuenta que la variable `$USER` se expande a su nombre de usuario.

2.4.3 Citas de nidificación

¿Cuál crees que será la salida de este comando?

```
echo ""Hola""
```

Las comillas en el shell no funcionan como en otros lenguajes de programación. En la programación shell funcionan "habilitando" y "deshabilitando" el modo comillas. Cuando escribimos una comilla entramos en el "modo comillas", y estará habilitado hasta que el shell encuentre otra comilla. Así, en el ejemplo anterior, sucederá algo como esto:

```
echo "(entra en modo comillas)"(sale del modo comillas)Hola"(entra en modo
comillas)"(sale del modo comillas)
```

No hay texto dentro del "modo comillas", por lo que las comillas no tienen ningún efecto.

Si queremos imprimir una comilla, podemos utilizar el carácter de escape `\`. El carácter que sigue a `\` no se interpretará como un carácter especial:

```
echo \"Hola\"

"Hola"
```

Y también:

```
echo \$HOME

$HOME
```

Las comillas también pueden anidarse unas dentro de otras para controlar cómo el shell expande y evalúa comandos y variables. Por ejemplo:

```
echo "Hola '$USUARIO'"

Hola 'usuario1'
```

Tenga en cuenta que las comillas simples dentro de las comillas dobles conservan el valor literal de la variable $USER, que luego se expande fuera de las comillas simples.

3. ESTRUCTURA DE CONTROL S

Para utilizar toda la potencia de la programación shell, al igual que en otros lenguajes, necesitamos utilizar estructuras de control. En este curso, veremos las más simples: sentencias condicionales y bucles. Para utilizarlos, primero tendremos que ver cómo expresar condiciones en bash.

3.1 Condiciones ions

En Bash, una condición es una expresión que puede ser evaluada como verdadera o falsa, también conocida como valores booleanos. Las condiciones se utilizan para controlar el flujo de un script y realizar ciertas acciones sólo si se cumple una determinada condición.

Existen dos formas de expresar condiciones en bash: sintaxis de corchete simple y sintaxis de corchete doble.

3.1.1 Sintaxis de un paréntesis

Es la sintaxis soportada más antigua. Equivale a ejecutar el comando `test` en la consola: ese comando devolverá `0` si se cumple la condición, y un valor distinto en caso contrario.

Con la sintaxis de corchetes simples puedes:

- Realiza comparaciones aritméticas. Por ejemplo, esto será cierto: `[ 1 -lt 5]`
- Compara cadenas. Por ejemplo, `"$VAR_1" == "patata"` comprobará si patata es el contenido de la variable `VAR_1`.
- Comprueba las condiciones de los archivos. Por ejemplo, `[ -d ruta ]` comprobará si la ruta es un directorio

En Linux, los comandos devuelven `0` si tienen éxito y cualquier otro valor en caso contrario. Por lo tanto, si las condiciones son verdaderas, la salida del comando será `0`.

No puede utilizar `<` o `>` dentro de la sintaxis de corchetes simples, porque se interpretará como redirección. En la tabla siguiente se ofrece un resumen de las condiciones más utilizadas.

3.1.2 Sintaxis de doble corchete

Esta sintaxis es una versión más potente de la sintaxis de corchetes simples, pero es menos portable que la sintaxis de corchetes simples (no funcionará en otros shells) Permite comparaciones avanzadas de cadenas y concordancia de patrones. También permite usar `<` y `>` para comparar (pero no `<=` o `>=`)

Presenta otras diferencias con respecto a la sintaxis de corchetes simples:

- Las expresiones pueden agruparse entre paréntesis dobles, lo que facilita la lectura de expresiones complejas. Por ejemplo: `[[ 3 == 3 && (2 == 2 && 1 == 1) ]]`
- Los corchetes dobles también admiten la coincidencia de patrones, donde el comodín `*` puede utilizarse para coincidir con cualquier carácter de una cadena. Por ejemplo, para comprobar si una variable contiene una c podemos usar `[[ $nombre = *c* ]] .`

- Bash no realiza la división de palabras dentro de corchetes dobles. Por ejemplo, si una variable tiene un valor de cadena que contiene espacios, Bash no dividirá la cadena en palabras. Esto puede ser útil en los casos en que desee probar una variable que contiene una ruta de archivo completa u otra cadena compleja.

Por ejemplo, si `FILENAME="fichero inexistente"` podemos usar `[[ ! -e $nombrefichero ]]`. Eso generará un error en la sintaxis de paréntesis simples. También tiene mejor soporte para variables vacías.

3.1.3 Operadores booleanos

Los operadores booleanos se utilizan para combinar o modificar condiciones en expresiones lógicas. Los tres operadores booleanos disponibles en Bash son NOT, AND y OR.

El operador `-a` se utiliza para realizar una operación AND:

```
[[ 1 > 2 -a 1 < 3 ]] # Falso
[[ 1 < 2 -a 1 < 3 ]] # Verdadero

[ 1 -gt 2 -a 1 -lt 3 ] # Falso
[ 1 -lt 2 -a 1 -lt 3 ] # Verdadero
```

La opción `-o` se utiliza para las operaciones OR:

```
[[ 1 > 2 -o 1 < 3 ]] # Verdadero
[[ 1 > 2 -o 1 > 3 ]] # Falso

[ 1 -gt 2 -o 1 -lt 3 ] # Verdadero
[ 1 -gt 2 -o 1 -gt 3 ] # Falso
```

También puedes implementar AND y OR con los operadores de shell `&&` y `||` (hablaremos de ellos más adelante) Ten en cuenta que, para utilizar estos operadores, necesitas cerrar la comparación en lugar de incluirlos dentro de la comparación si estás utilizando la sintaxis de corchete simple:

```
# AND
[ 1 -gt 2 ] && [ 1 -lt 3 ] # Falso
[[ 1 > 2 && 1 < 3 ]]       # Falso

# O
[ 1 -gt 2 ] || [ 1 -lt 3 ] # Verdadero
[[ 1 > 2 || 1 < 3 ]]       # Verdadero
```

¡El operador not se representa mediante el signo de exclamación !

```
[[ 1 < 5 ]] # true
¡! [[ 1 < 5 ]] # false
[[ ! 1 < 5 ]] # false
¡! [[ ! 1 < 5 || 1 > 3 ]] # Verdadero

[ 1 -lt 5 ] # true
¡! [ 1 -lt 5 ] # false
[ ! 1 -lt 5 ] # false
¡! [ ! 1 -lt 5 -o 1 -gt 3 ] # Verdadero
```

3.1.4 Reglas básicas para las condiciones

Existen ciertas reglas que le ayudarán a redactar correctamente las condiciones:

- Mantenga espacios entre los corchetes y la comparación real
  Malo:

  ```
  [1 -lt 2]
  ```

Bien:

```
[ 1 -lt 2 ]
```

- Termina la línea antes de añadir una nueva palabra clave como `entonces` (verás `si` y `entonces` más adelante): Mal:

```
si [ 1 -lt 2 ] entonces
```

Bien:

```
if [ 1 -lt 2 ]; then

si [ 1 -lt 2 ]
entonces
```

- Entrecomillar variables de cadena utilizadas en condiciones para evitar errores causados por espacios y/o nuevas líneas. Malo:

```
[ $STR1 = $STR2 ]
```

Bien:

```
[ "$STR1" = "$STR2" ]
```

3.1.5 Cuadro de condiciones

Condiciones numéricas

Operador	Descripción
`-eq`	Igual a
`-ne`	No es igual a
`-gt`	Mayor que
`-lt`	Menos de
`-ge`	Mayor o igual que
`-le`	Inferior o igual a

Tenga en cuenta que puede utilizar `<` y `>` con la sintaxis de doble corchete. Por ejemplo, `[[ 1 < 2 ]]` o `[[ 5 > 3 ]]`.

Condiciones de las pruebas de archivos

Operador	Descripción
`-d`	Directorio
`-e`	Existe (también -a)
`-f`	Fichero normal
`-h`	Enlace simbólico (también -L)

`-r`	Legible por usted
`-w`	Puede escribirlo usted mismo
`-x`	Ejecutables por ti
`-s`	No vacío

Condiciones basadas en cadenas

Operador	Descripción
`"str1" = "str2"`	Devuelve verdadero si las cadenas son iguales. Es necesario añadir espacios en blanco enmarcando el `=`.
`"str1" == "str2"`	Similar a `=`. Válido sólo en sintaxis de doble corchete.
`"str1" != "str2"`	Devuelve true si las cadenas no son iguales
`"str1" > "str2"`	Devuelve true si str1 es mayor que str2 según el orden lexicográfico.
`"str1" < "str2"`	Devuelve true si str1 es menor que str2 según el orden lexicográfico.
`-z "cadena"`	Devuelve true si la longitud de la cadena es 0.
`-n "cadena"`	Devuelve true si la longitud de la cadena no es 0.
`"str1" =~ regex`	Devuelve verdadero si la cadena especificada coincide con la expresión regular ampliada.

3.2 Frases condicionales

En bash, la sentencia if/then/else se utiliza para la ejecución condicional de comandos. Esta es la sintaxis básica:

```
si [ condición1 ]
entonces
  # comandos a ejecutar si la condición1 es
verdadera elif [ condición2 ]
entonces
  # comandos a ejecutar si condición1 es falsa y condición2 es verdadera
else
  # comandos a ejecutar si tanto la condición1 como la condición2 son
falsas fi
```

Tenga en cuenta que la palabra clave then debe estar en una línea diferente. Puede utilizar el separador de comandos de shell `;` para poner el `entonces` en la misma línea:

```
si [ condición1 ]; entonces
  # comandos a ejecutar si la condición1 es
verdadera elif [ condición2 ]; then
  # comandos a ejecutar si condición1 es falsa y condición2 es verdadera
else
  # comandos a ejecutar si tanto la condición1 como la condición2 son
falsas fi
```

3.2.1 Condicionales de una línea (&& y ||)

En bash, puede utilizar `&&` y `||` como abreviaturas para escribir condicionales de una línea.

&& Operador

`&&` es el operador lógico AND, lo que significa que el comando que sigue a `&&` sólo se ejecutará si el comando que precede a `&&` sale con un código de estado `0` (éxito):

```
comando1 && comando2
```

En este caso, si `el comando1` tiene éxito (sale con un código de estado `0`), se ejecutará `el comando2`. Si `comando1` falla (sale con un código de estado distinto de cero), el `comando2` no se ejecutará. Por ejemplo:

```
¡! [ -d directorio_1 ] && mkdir directorio_1
```

Comprobará si el directorio existe. Si no, se creará el directorio ejecutando la segunda instrucción. Puede utilizar cualquier instrucción, no sólo la sintaxis de paréntesis simples o dobles:

```
apt update && apt upgrade
```

Esto actualizará el sistema sólo si el comando de actualización tiene éxito.

|| Operador

`||` es el operador lógico OR, lo que significa que el comando que sigue a `||` sólo se ejecutará si el comando que precede a `||` sale con un código de estado distinto de cero (fallo):

```
comando1 || comando2
```

En este caso, si `el comando1` falla (sale con un código de estado distinto de cero), se ejecutará el `comando2`. Si el comando1 tiene éxito (sale con un código de estado `0`), el `comando2` no se ejecutará.

El mismo ejemplo anterior podría escribirse utilizando ||:

```
[ -d directorio_1 ] || mkdir directorio_1
```

También puede utilizar cualquier comando con ||:

```
que git || sudo apt-get install git
```

Esto comprobará si `git` está instalado, y lo instalará si no lo está.

Puede combinar los dos operadores. El operador `&&` tiene mayor precedencia que el operador `||`, lo que significa que `&&` se evalúa primero. Por ejemplo:

```
grep "patrón" miarchivo.txt && echo "patrón encontrado" || echo "patrón no
encontrado"
```

En este caso, si `grep "patrón" miarchivo.txt` encuentra correctamente el patrón en `miarchivo.txt`, el patrón de mensaje `encontrado se imprimirá` en la consola. En caso contrario, se imprimirá el mensaje patrón `no encontrado`.

3.2.2 Exposición del caso

La sentencia `case` es una estructura de control en bash que permite probar una variable contra un conjunto de patrones. Se utiliza a menudo para implementar interfaces de tipo menú en las que el usuario selecciona una opción de una lista de opciones.

Esta es la sintaxis básica de la sentencia case:

```
case variable in
  patrón1)
    comandos1
    ;;
  patrón2)
    comandos2
    ;;
...
  patrónN)
    comandosN
    ;;
  *)
    comandos-por-defecto
    ;;
esac
```

En esta sintaxis, `variable` es la variable que desea comprobar, y `patrón1`, `patrón2,` etc., son los patrones con los que desea comparar. Cada patrón va seguido de un bloque de comandos que se ejecutarán si el patrón coincide con la variable. El operador `;;` se utiliza para marcar el final de cada bloque de comandos.

Puedes usar patrones bash como

- `*`: Coincidirá con todo
- `opción1|opción2`: Coincidirá si el valor de la variable es la opción1 o la opción2.
- `*.txt`: Coincidirá con los textos que terminen en `.txt`
- `archivo??.txt`: coincidirá con `archivo01.txt`, `archivo02.txt`, etc.

La sentencia `case` intenta hacer coincidir la variable con los patrones en orden, de arriba a abajo. Si se encuentra una coincidencia, se ejecutan los comandos para ese patrón y la sentencia case termina. Si no se encuentra ninguna coincidencia, se ejecuta el bloque default-commands, si está presente.

4. MATERIA COMPLEMENTARIA L

Shell scripting:

https://bash.cyberciti.biz/guide/Main_Page

Evaluación de la línea de comandos:

https://docstore.mik.ua/orelly/unix/upt/ch08_05.htm

https://www.oreilly.com/library/view/learning-the-bash/1565923472/ch07s03.html

Tipos de variables:

https://tldp.org/LDP/abs/html/untyped.html

Condiciones:

https://acloudguru.com/blog/engineering/conditions-in-bash-scripting-if-statements

Hoja de comparación:

https://kapeli.com/cheat_sheets/Bash_Test_Operators.docset/Contents/Resources/Documents/índice

Sintaxis simple y doble

https://acloudguru.com/blog/engineering/conditions-in-bash-scripting-if-statements#h-1-single-sintaxis de corchetes

https://developer.ibm.com/tutorials/l-bash-test/