1.1. Que es el interprete de ordenes?

Nos damos cuenta cuando estamos en el interprete de ordenes (shell en ingles) porque normalmente se despliega su símbolo de espera "$" o "#", donde se puede ingresar comandos, que luego son ejecutados por el sistema.
El interprete de ordenes es un programa que recibe nuestra orden (comando), realiza tareas como: reemplaza nombre de archivos y variables, redireccionamiento de entrada-salida, localiza el archivo ejecutable e inicia el programa. Esta ubicado entre el kernel y el usuario, atendiendo las ordenes que este ultimo ejecuta y ocultando la complejidad del kernel para ejecutarlas.

1.2. Porque es importante saber utilizar el interprete de ordenes ?

Es un programa que utilizamos a menudo, ya sea ejecutando comandos desde consola o mediante una Xterm cuando estamos en X. El sistema contiene cientos de scripts escritos para el interprete de ordenes, por ejemplo en el arranque se ejecutan varios script, para inicializar servicios.
Podríamos pensar que con lo que sabemos ya nos basta, ejecutar comandos es sencillo, pero un conocimiento mas profundo de este programa nos ayudaría en muchas tareas cotidianas.

1.3. Cuando se ejecuta el interprete de ordenes ?

1- Cuando arranca el sistema, se carga el kernel en memoria y el primer programa en ejecutarse es el init, el cual lee su archivo de configuración /etc/inittab y ejecuta los script de inicialización del sistema.
2- Luego por cada terminal habilitada se ejecuta el programa getty, el cual monitorea la terminal en espera de ingreso de caracteres. esto se especifica en el archivo /etc/inittab de la siguiente manera:

```
1:2345:respawn:/sbin/getty 38400 tty1
2:23:respawn:/sbin/getty 38400 tty2
3:23:respawn:/sbin/getty 38400 tty3
4:23:respawn:/sbin/getty 38400 tty4
5:23:respawn:/sbin/getty 38400 tty5
6:23:respawn:/sbin/getty 38400 tty6
```

Se puede observar seis terminales habilitadas (desde tty1 hasta tty6), por ejemplo si queremos habilitar la tty7 deberemos agregar la linea:

```
7:23:respawn:/sbin/getty 38400 tty7
```

3- Cuando el proceso getty detecta ingreso de caracteres, llama al programa login, el cual nos pide usuario y contraseña para autentificarnos.
4- El proceso login verifica en el archivo /etc/shadow (o /etc/passwd si no tenemos el shadow habilitado) si la contraseña es correcta, si es asi, ejecuta el programa indicado para dicho usuario en el archivo /etc/passwd.
Cada usuario se define en el archivo /etc/passwd en una linea, veámoslo a través de un ejemplo :

```
ariel:x:1000:101::/home/ariel:/bin/bash
```

Esta linea esta formada por campos que se separan mediante el carácter ":" y los mas importante para nuestro caso son: el primero que indica el nombre del usuario (ariel en este caso) y el ultimo que es el programa que ejecutara login luego de autentificar el usuario (/bin/bash para el ejemplo).
Podríamos modificar este ultimo para que login ejecute otro programa en vez de bash, como por ejemplo:

```
ariel:x:1000:101::/home/ariel:/bin/mc
```

Se ejecutara el Midnight Comander (clon del conocido Norton Comander) en vez del interprete de ordenes bash.
Existen varios interpretes de ordenes disponibles para su uso, cada uno tiene sus características particulares, los mas comunes son :

bash - El mas utilizado y el que trataremos en este texto, es la versión GNU del Bourne Shell.
ksh - Es la versión GNU del Korn Shell.
tcsh - Version GNU y compatible con el C Shell de Berkeley

5- Si la contraseña es incorrecta login devuelve el control de la terminal al getty.

Estos 3 interpretes tienen diferencias en su funcionamiento, a partir de ahora cuando nos referimos a "interprete de ordenes" será exclusivamente a bash.

2-INGRESO DE COMANDOS SIMPLES, LARGOS Y MÚLTIPLES.

2.1. Comandos simples:

Lo mas común es ingresar comandos simples, que normalmente tienen las siguientes formas:
$ nombre-del-ejecutable
o
$ nombre-del-ejecutable -opciones archivos
La primer palabra es considerada como el nombre del ejecutable y las siguientes son argumentos, la mayoría toman primero como argumento una serie de opciones y luego los archivos sobre el cual operara.
Por ejemplo:

```
$ ls -l archivo1 archivo2
-rw-rw-r--  1 ariel  ariel      0 Jul  3 03:56 archivo1
-rw-rw-r--  1 ariel  ariel      0 Jul  3 03:56 archivo2
```

2.2 Comandos largos:

A veces el comando se torna tan largo que sobrepasa la longitud de la terminal, cuando esto sucede automáticamente continuamos escribiendo en la linea inferior, pero puede quedarnos entrecortado, y difícil de leer.
El carácter de nueva linea es el que le dice al interprete donde finaliza el comando, este se indica pulsando la tecla ENTER, pero si ingresamos un carácter "\ " antes de pulsar ENTER, este pierde su significado y podemos seguir escribiendo el comando en la linea inferior.
Ejemplo:

```
# ls -l archivo1 archivo2 \
archivo3 archivo4 \
archivo5
-rw-rw-r--  1 ariel  ariel      0 Jul  3 03:56 archivo1
-rw-rw-r--  1 ariel  ariel      0 Jul  3 03:56 archivo2
-rw-rw-r--  1 ariel  ariel      0 Jul  3 03:56 archivo3
-rw-rw-r--  1 ariel  ariel      0 Jul  3 03:56 archivo4
-rw-rw-r--  1 ariel  ariel      0 Jul  3 03:56 archivo5
```

2.3 Ejecución de comandos múltiples:

Veremos como ejecutar varios programas con un solo comando.
Supongamos que queremos ejecutar tres programas escribiendo solo una linea, estos se pueden ejecutar secuencialmente, ósea que el interprete espere que finalice el primero para ejecutar el segundo y asi sucesivamente, o podemos hacer que los tres programas se ejecuten simultáneamente (los 3 juntos).
Para esto existen 3 caracteres especiales: el punto y coma (;) , el ampersand (&) y el pipe (|).

Veamos un ejemplo donde se ejecutan tres programas secuencialmente:

```
$ ls -l ; who ; ps
```

Se ejecutan los tres programas uno tras otro, ósea es equivalente a:

```
$ ls -l
$ who
$ ps
```

Con la salida de cada comando intercalada.

Veamos otro ejemplo, donde se note la ejecución secuencial:

```
$ sleep 5; echo "pasaron 5 segundos" ; ls -l
pasaron 5 segundos
-rw-rw-r--  1 ariel  ariel       0 Jul  3 03:56 archivo1
-rw-rw-r--  1 ariel  ariel       0 Jul  3 03:56 archivo2
-rw-rw-r--  1 ariel  ariel       0 Jul  3 03:56 archivo3
-rw-rw-r--  1 ariel  ariel       0 Jul  3 03:56 archivo4
-rw-rw-r--  1 ariel  ariel       0 Jul  3 03:56 archivo5
```

sleep solo nos produce un retardo de 5 segundos, luego se ejecuta el comando **echo** y el **ls**.

Para el caso de ejecutar programas simultáneamente, estos pueden ejecutarse cada uno independientemente del resultado de los otros o pueden estar interconectados entre ellos, ósea que la salida de un proceso puede ser utilizada por otro.
El carácter "&" es el utilizado para ejecutar procesos simultáneamente sin interconexiones.

Veamos el ejemplo anterior, pero ejecutándose los tres programas simultáneamente :

```
$ sleep 5 & echo "pasaron 5 segundos" & ls -l
[1] 349
[2] 350
pasaron 5 segundos
-rw-rw-r--  1 ariel  ariel       0 Jul  3 03:56 archivo1
-rw-rw-r--  1 ariel  ariel       0 Jul  3 03:56 archivo2
-rw-rw-r--  1 ariel  ariel       0 Jul  3 03:56 archivo3
-rw-rw-r--  1 ariel  ariel       0 Jul  3 03:56 archivo4
-rw-rw-r--  1 ariel  ariel       0 Jul  3 03:56 archivo5
[2] Done              echo "pasaron  5 segundos"
```

El programa **echo** y **ls** no esperaron a que finalice el **sleep**, como sucedía anteriormente. **sleep** y **echo** se ejecutaron en background, pero ls en foreground.
Si queremos que **ls** también se ejecute en background le agregamos un '&' al final:

```
$ sleep 5 & echo "pasaron 5 segundos" & ls -l &
```

2.4 Intercomunicación entre procesos.

En el ejemplo anterior, se ejecutaban 3 programas simultáneamente, pero sin intercomunicarse entre ellos, ósea que sus salidas y entradas eran independientes entre si.
Puede surgir la necesidad de que un proceso necesite los datos que entrega otro, ambos ejecutándose simultáneamente, esto podemos indicárselo al interprete mediante el pipe (|), de la siguiente manera:

```
$ programa1 | programa2 | comando3
```

La salida estandart de **programa1** es enviada a la entrada estandart del **programa2**, la salida estandart de **programa2** es enviada a la entrada estandart de **programa3**.

La salida estandart en una terminal es la pantalla y la entrada estandart es el teclado, podemos redireccionar a ambas, de manera que logramos la intercomunicación entre comandos.

Veamos algunos ejemplos:

```
$ ls -l | more
```

El programa **ls** por defecto escribe en la salida estandart, pero mediante el pipe redireccionamos su salida a la entrada estandart de **more**.
El programa **more** lee de su entrada estandart, no sabe si los datos provienen del teclado o son generados por otro proceso.
El interprete de ordenes realiza la interconexión y el kernel maneja el flujo de datos entre los procesos.

$ ls -l /etc | grep ^d | sort | more

ls imprime los archivos que se encuentran en /etc en formato largo (-l), el programa grep recibe la salida de **ls** e imprime en su salida estandart solo las lineas que empiecen con "d", ósea los directorios, luego el programa **sort** ordena la salida y more nos muestra el resultado por paginas, dando como resultado algo asi:

```
drwxr-s---   2 root     dip         4096 Jul  7 19:55 chatscripts
drwxr-xr-x   2 root     root        4096 Dec 27  1995 rc.boot
drwxr-xr-x   2 root     root        4096 Jul  7 19:55 ae
drwxr-xr-x   2 root     root        4096 Jul  7 19:55 network
drwxr-xr-x   2 root     root        4096 Jul  7 19:55 security
drwxr-xr-x   2 root     root        4096 Jul  7 19:56 default
drwxr-xr-x   2 root     root        4096 Jul  7 22:57 console-tools
drwxr-xr-x   2 root     root        4096 Jul  7 23:04 menu
drwxr-xr-x   2 root     root        4096 Jul  9 06:45 nmh
drwxr-xr-x   2 root     root        4096 Jul  9 06:48 cron.d
drwxr-xr-x   2 root     root        4096 Jul  9 06:48 cron.monthly
drwxr-xr-x  14 root     root        4096 Jul  7 23:44 X11
```

3. INICIO DE PROGRAMAS.

Cuando bash termina de interpretar nuestro comando, realiza los siguientes pasos para comenzar la ejecución del programa:
-Busca el archivo ejecutable en los directorios indicados en la variable de entorno PATH.
-Si lo encuentra, inicia un subshell (shell hijo) el cual se encarga de la ejecución del programa.
-El subshell le envía la orden al kernel.
El subshell puede modificar variables de entorno, sin afectar el entorno del shell hijo.

4. REDIRECCIONAMIENTO DE ENTRADA-SALIDA.

Un programa que escribe en pantalla, podemos hacer que su salida vaya a un archivo, o también un programa que lee de teclado, podemos hacer que lea desde un archivo, todo eso sin que se de cuenta que realmente esta leyendo o escribiendo a archivo.
Estas tareas de redireccionamiento son realizadas por el interprete y se indican con los signos ">" o "<". El signo ">" se utiliza para redireccionar la salida estandart de un comando a un archivo y el "<" para que un comando lea desde un archivo en vez de su entrada estandart (teclado).
Ejemplos:
$ ls -l > file.txt
Si visualizamos el contenido del archivo file.txt veremos la salida del comando "ls -l". El comando **ls** solo recibe la opción **-l**, pero no se entera del redireccionamiento, el subshell que maneja la ejecución del programa recibe el nombre del archivo que reemplaza a la salida estandart, entonces cuando **ls** trata de escribir a la salida estandart el subshell redirecciona al archivo.
¿ Que pasaría si el archivo file.txt ya existía antes de ejecutar el comando anterior ?
Perdemos el contenido del archivo porque es sobrescrito.
Podemos utilizar también el ">>" para agregar la salida al archivo y no perder el contenido que ya tenia, si el archivo no existe funciona igual que el ">".
ls -l /dev/etc > file.txt ; ls -l / > file.txt
El archivo file.txt solo contendrá la salida del segundo comando, porque la salida del primer comando es sobrescrita, si queremos que esto no suceda:
ls -l /dev/etc > file.txt ; ls -l / >> file.txt
Ahora si file.txt contendrá la salida de ambos comandos.
El símbolo ">" o ">>" seguido por el nombre del archivo pueden ir indistintamente detrás o delante del comando.
> file.txt ls -l

Veamos ahora ejemplos de redireccionamiento de la entrada estandart, recordemos que se realizan con el signo "<" seguido por el nombre del archivo.

sort < file.txt
Aquí sort (programa que ordena las lineas) recibe los datos del archivo file.txt, no diferencia si estos datos provienen de un archivo o fueron tecleados por el operador. Este comando es equivalente a:

cat file.txt | sort

Solo que este ultimo es menos eficiente porque utiliza un proceso mas para realizar la misma tarea. La intercomunicación entre procesos es un caso particular del redireccionamiento de entrada-salida.

El símbolo "<" seguido por el nombre del archivo pueden ir indistintamente detrás o delante del comando.

< file.txt sort

5. VARIABLES.

Una variable es un lugar de memoria donde se almacena un dato para un uso posterior. El interprete tiene la capacidad de almacenar variables.
Pasa asignar un valor a una variable se realiza precediendo el nombre de la variable con un signo igual y su valor, veamos un ejemplo:
$ DIR=/home/ariel
Luego para poder utilizar el valor almacenado en la variable, debemos anteponer el nombre de la variable con el signo "$".
$ echo $DIR
/home/ariel
Como la gran mayoría de los nombres de los comandos son con minúsculas, para diferenciar los comandos de las variables, se utilizan mayúsculas para los nombres a las variables, pero pueden contener minúsculas.
Existen cuatro tipos de variables: variables definidas por el usuario, variables parámetros, variables especiales y variables de entorno.

5.1. Variables definidas por el usuario.

Son el caso del ejemplo anterior, su nombre solo debe contener caracteres alfanuméricos y el guión bajo (_), excepto el primer carácter no debe ser un dígito (0 a 9).
Ejemplos:
$ NOMBRE=Pepe
$ EDAD=20
$ echo Hola $NOMBRE
Hola Pepe
$ NOMBRE2=$NOMBRE
$ echo $NOMBRE2
Pepe
Se puede asignar valor a mas de una variable en una única linea:
$ NOMBRE=Pepe EDAD=20
$ echo $NOMBRE tiene $EDAD
Pepe tiene 20
La asignación se realiza de izquierda a derecha.
$ X=1 Y=$X
$ echo $Y
1

Para quitarle el valor a una variable podemos utilizar el comando **unset**.

$ X=1
$ echo $X
1

```
$ unset X
$ echo X
```

Variables no modificables (solo lectura).

Para asegurarse que el valor de una variable no sea modificado, se puede indicar como de solo lectura de la siguiente manera:

```
$ readonly variable
```

Ejemplo:

```
$ X=1
$ readonly X
$ X=2
bash: X: readonly variable
$ echo $X
1
```

5.2 Variables parámetros.

Como vimos al principio del texto, cuando el interprete procesa un comando, la primer palabra es el nombre del ejecutable y las siguientes son argumentos.
Cuando el ejecutable es un script para bash, los parámetros son pasados al script mediante las variables parámetros, el primer parámetro será la variable $1, el segundo $2 y asi hasta $9.
Los nombres de las variables son de 1 a 9, el signo "$" es para poder leer su valor.
Veamos un ejemplo de un script en bash que visualiza los dos primeros parámetros pasados.
Editamos un archivo llamado script y le agregamos lo siguiente:

```
echo $1
echo $2
```

Luego lo hacemos ejecutable:

```
$ chmod +x script.
```

Y por ultimo lo probamos:

```
$ script parametro1 parametro2
parametro1
parametro2
```

El nombre del ejecutable se almacena en la variable $0.
Pero si los parámetros pasados son mas de 9, ¿ como accedemos al parámetro numero 10 y superiores ? ver el próximo punto.

5.3 Variables especiales.

Son variables que siempre están presentes en el interprete y se inicializaron cuando se ejecuto.
Veamos como se llaman estas variables y que contienen:

S* Contiene todos los parámetros pasados al scritp, si le pasamos mas de 9 parámetros estarán incluidas en esta variable.

$# Contiene el numero de parámetros pasados al script.

$? Contiene el estado de finalización del ultimo comando ejecutado, el cual es cero si el comando finalizo exitosamente o 1 si hubo algún error.

$$ Contiene el PID del proceso actual.

5.4 Variables de entorno.

Son variables que utilizan los programas para obtener información del usuario. Cualquier programa puede utilizar variables de entorno, normalmente en la documentación del programa (pagina del manual) se indica que variables de entorno usara.

Veamos el nombre y su uso de algunas variables de entorno comunes:

HOME

Esta variable se inicializa cuando se ejecuta el bash y contiene el directorio home del usuario (/home/usuario)

Por ejemplo cuando al comando cd no le inidicamos el directorio, ósea hacemos "cd", este comando lee la variable de entorno HOME y realiza "cd $HOME".

PATH

Cuando ejecutamos bash, lo primero que hace es ejecutar los scripts /etc/profile, $HOME/.bash_profile y $HOME/.profile, durante la ejecución de estos, una de las tareas que realiza es cargarle un valor a la variable PATH, la cual inidica los directorios donde bash buscara los archivos ejecutables.

```
$ echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/bin/X11:/usr/games
```

PS1

Se inicializa cuando se ejecuta bash y contiene el valor del símbolo de espera (prompt) de bash, que normalmente es "$" para un usuario ordinario o "#" para el superusuario.

PS2

Contiene el símbolo de espera del shell secundario (shell hijo), normalmente es ">".

MAIL

Especifica la ruta completa del archivo de la casilla de correo del usuario.

MAILCHECK

Especifica cada cuanto tiempo se verificara si hay nuevo correo en el archivo de la casilla de correo (60).

TERM

Especifica el tipo de terminal que se esta utilizando (vt100, ansi, etc).

6. SUSTITUCIÓN DE NOMBRES DE ARCHIVOS, VARIABLES Y COMANDOS.

El interprete de ordenes realiza varias sustituciones en la linea de comando antes de ejecutarla, estas pueden ser nombres de archivos, valores de variables o el resultado de la ejecución de un comando, en seguida veremos por separado estas tres tipos de sustituciones:

6.1. Sustitución de nombres de archivos.

Una vez ingresado el comando (cuando presionamos ENTER), el interprete busca los caracteres "*", "?" y "[...]" en los parámetros del comando, y reemplaza esos parámetros por nombres de archivos que se encuentran en el directorio de trabajo si se cumplen ciertas reglas que se explicaran a continuación:

Asterisco (*):

Los parámetros que contengan "*" serán reemplazados por nombres de archivos donde en la posición que se encuentra el asterisco tengan cualquier cadena de caracteres.

Ejemplo:

```
$ ls
diccionario.txt  documento1.txt  documento2.txt
documento3.txt  documento_bash.txt
documento_linux.txt
```

Listamos todos los archivos que están dentro del directorio de trabajo (excepto los que comienzan con ".")

```
$ echo doc*
documento1.txt  documento2.txt  documento3.txt
documento_bash.txt  documento_linux.txt
```

El comando **echo** podemos utilizarlo como un rudimentario resplazo del **ls**. Con esto verificamos que el interprete realiza la sustitución y que el comando echo no lee doc*, sino la sustitución que fue realizada por el interprete.

$ echo d*
diccionario.txt documento1.txt documento2.txt
documento3.txt documento_bash.txt
documento_linux.txt
$ echo *1*
documento1.txt
$ echo *bash.txt
documento_bash.txt
$ echo *_*
documento_bash.txt documento_linux.txt
$ echo documento1.txt*
documento1.txt
El "*" también puede ser remplazado por una cadena nula.
$ echo a*
a*
Como en el directorio no existe ningún archivo o directorio que empiece con "a", el interprete no realiza sustitución y el comando echo recibe "a*".

Notas:
Los nombres de archivos que comienzan con punto (.) no son utilizados en la sustitución, para que se produzca la sustitución debemos incluir un punto al inicio, por ejemplo para visualizar todos los archivos que comienzan con punto ejecutamos "ls .*".
El carácter * puede ser reemplazado por cadenas que contengan un punto (.), esto no era asi en el DOS, donde para listar todos los archivos había que ejecutar "dir *.*", en Linux ejecutando "dir *" o "dir" se visualizan todos los archivos excepto aquellos que comiencen con un punto (.).

Interrogación (?):

Solo puede ser remplazado por un único carácter.

Ejemplo:

$ ls
documento1.txt documento2.txt documento3.txt
documento10.txt documento20.txt documento30.txt

$ ls documento?.txt
documento1.txt documento2.txt documento3.txt

$ ls documento??.txt
documento10.txt documento20.txt documento30.txt

$ ls documento?.???
documento1.txt documento2.txt documento3.txt

$ ls documento?
ls: documento?: No such file or directory
¿ Que paso aquí ? Como bash no realizo sustitución, el comando **ls** recibe "documento?" y como no existe un archivo o directorio con ese nombre, nos da ese mensaje.

Corchetes ([...]):

Los corchetes sustituyen a un único carácter al igual que el "?", excepto que dentro de los corchetes se indica que valores pueden ser utilizados para la sustitución.

Los valores por los cuales se puede realizar la sustitución se indican todos juntos ([abc1234;#]) o mediante rangos de valores ([a-z] [0-9] [a-zA-Z]).

Ejemplos:

```
$ ls
documento1.txt  documento2.txt documento3.txt
documentoa.txt  documentob.txt documentoc.txt
documentoa1.txt  documentob1.txt documentoc1.txt
```

```
$ ls documento[1-3].txt
documento1.txt  documento2.txt documento3.txt
```

El mismo resultado se hubiera obtenido con los comandos: "ls documento[123].txt" o "documento[0-9].txt".

```
$ ls documento[a-c].txt
documentoa.txt  documentob.txt documentoc.txt
```

Lo misno hubiera sucedido con los comandos: "ls documento[abc].txt" o "ls documento[a-z].txt".

```
$ ls documento[a-c][1-3].txt
documentoa1.txt  documentob1.txt documentoc1.txt
```

```
$ ls documento[xyz]
ls: documento[xyz]: No such file or directory
```

6.2 Sustitución de variables.

En la linea de comando cuando el bash encuentra el signo "$", supone que lo siguiente es el nombre de una variable y sustituye su valor.

Ejemplos:

```
$ NOMBRE=Pepe
$ echo Hola $NOMBRE
Hola Pepe
~$ ls $HOME/bin/
~/bin$ pwd
/home/pepe/bin
```

Las dos sustituciones tratadas anteriormente solo se realizan en los parámetros, la sustitución de variables se puede realizar en cualquier parte del comando.

```
$ dir=ls
$ $dir $HOME
GNUstep   nsmail   bin   doc   src   newsletter2.html
```

6.3 Sustitución de comandos.

Es un forma de pasar el resultado de un comando en los parámetros de otro comando, esto se realiza encerrando el comando entre comillas invertidas (``).

Ejemplos:

Para saber la cantidad de usuarios en el sistema podemos utilizar el comando:

```
$ who | wc -l
5
```

who lista cada usuario en un linea y "wc -l" cuenta la cantidad de lineas.

```
$ echo Hay `who | wc -l` usuarios utilizando el sistema.
Hay 5 usuarios utilizando el sistema.
```

```
$ date +"%H:%M:%S"
20:30:03
$ echo Este comando se ejecuto a la hora `date +"%H:%M:%S"`
Este comando se ejecuto a la hora 20:31:32
```

7. METACARACTERES.

Ya hemos aprendido el significado que les da el bash a unos caracteres especiales llamador **metacaracteres**, veamos un resumen de los mas utilizados:

Ejecución.	**Uso.**
;	Se utiliza para ejecutar dos programas en forma secuencial mediante un solo comando.
&	Ejecución de procesos en background.
|	Intercomunicación entre procesos.

Redir.	**Uso.**
> file	Redirecciona la salida estandart a un archivo.
>> file	Agrega de salida estandart a un archivo.
< file	Redirecciona un archivo a la entrada estandart.

Sustitución.	**Uso.**
$	Sustitución de variables.
?	Sustitución por un carácter de un nombre de archivo.
*****	Sustitución por una cadena de caracteres de un nombre de archivo
``	Sustitución de resultados de comandos.
[]	Sustitución por un carácter de un nombre de archivos y especificado entre los corchetes.

Anulación.	**Uso.**
""	
''	Quita significado a los metacaracteres que se encuentran en el interior.
\	Quita el significado al metacaracter que le sigue.
#	Comenta la linea.

Todos han sido estudiados durante el texto excepto la ultima categoría, que serán tratados a continuación:

7.1 Anulación de significado de los metacaracteres.

Vimos que bash recorre nuestro comando ingresado en busca de caracteres especiales o metacaracteres, luego interpreta sus significado y por ultimo pasa la orden al kernel para su ejecución. Existe la posibilidad de indicarle que no interprete los caracteres especiales, esto se puede realizar con los caracteres: barra invertida (\), apóstrofos (**''**) y comillas (**""**) para algunos metacaracteres.

Barra invertida (\):

No se interpreta el metacaracter que le sigue, ósea el programa recibe el metacaracter dentro de sus parámetros.

Ejemplos:

```
# echo *
documento1.txt documento2.txt documento3.txt documentoa.txt documentob.txt documentoc.txt
documentoa1.txt documentob1.txt documentoc1.txt
```

```
# echo \*
*
# echo $HOME
/home/ariel
# echo \$HOME
$HOME

# ls > archivo1.txt
# ls \> archivo2.txt
ls: >: No such file or directory
ls: archivo2.txt: No such file or directory
#echo \\
\
```

apóstrofos ('):

No se interpreta ningún metacaracter que se encuentre dentro de ellas.

```
$ echo '$*?<>'
$*?<>
```

Comillas (" "):

El shell interpretara dentro de las comillas los caracteres: **$, `..` y \,** los demás no son interpretados.