Introducción:

Editores: vi, vim, nano, (ya veremos los gráficos como gedit o sublime text). El Shell Profile:

- Cuando se ejecuta un shell interactivo, se leen y ejecutan los comandos de los siguientes ficheros por orden: /etc/profile (si existe), el ∼/.bash_profile, ∼/.bash_login y el ∼/.profile, por este orden (en algunas distribuciones cambian).
- Cuando salimos del shell interactivo, se ejecutan el ∼/.bash_logout y el /etc/bash.bash_logout (si existe).
- Cuando se hace login (vía consola ssh, etc.) el .bash_profile se ejeucta para configurar el shell antes de que salga el prompt. Pero si ejecutas un xterm dentro de Gnome o KDE, entonces es el .bashrc el que se ejecuta (aunque también lo hace cuando se ejeucta una instancia de /bin/bash).
- A efectos prácticos, configurar el .bash_profile para configuraciones de nuestro terminal.
- Fíjese que el "." denota que el archivo es oculto (visualizarlos con ls -la).
- ∼ es equivalente a $HOME
- Para ejecutar un script:

```
$ chmod u+x script.sh # derechos de ejecución
$ ./script.sh
$ sh script.sh
$ . script.sh
$ bash script.sh
```

Un **shell** es un intérprete de órdenes, y un intérprete de órdenes es el programa que recibe lo que se escribe en la terminal y lo convierte en instrucciones para el sistema operativo. Básicamente permite a los usuarios comunicarse con el sistema operativo y darle órdenes. En otras palabras, el objetivo de cualquier intérprete de órdenes es ejecutar los programas que el usuario teclea en el prompt del mismo.

El **prompt** es una indicación que muestra el intérprete para anunciar que espera una orden del usuario. Cuando el usuario escribe una orden, el intérprete la ejecuta. En dicha orden, puede haber programas internos o externos. Los programas internos son aquellos que vienen incorporados en el propio intérprete, como, por ejemplo, echo, cd, o kill. Mientras que los externos son programas separados, un ejemplo son todos los programas que residen en los directorios /bin, /usr/bin, etc., como ls, cat, o cp.

En el mundo UNIX/Linux existen tres grandes familias de shells: sh, csh y ksh. Se diferencian entre sí, básicamente, en la sintaxis de sus órdenes y en la interacción con el usuario. En la siguiente tabla se muestran las tres grandes familias de shells, con el nombre correspondiente y posibles clones de cada uno:

Tipo de Shell	**Shell Estándar**	**Clones libres**
AT&T Bourne shell	sh	ash, bash, bash2
Berkeley "C" shell	csh	tcsh
AT&T Korn shell	ksh	pdksh, zsh
Otros	-	esh, gush, nwsh

Por defecto, cada usuario tiene asignado un shell, establecido en el momento de creación de su cuenta, y que se guarda en el fichero /etc/passwd. Puedes hacer (cat /etc/passwd para verlo).

El shell asignado a un usuario se puede cambiar de dos maneras: editando manualmente dicho fichero (que tiene que ser realizado por el administrador del sistema), o bien con el programa chsh (que lo puede ejecutar el propio usuario). Los shells están en el directorio /bin. Por ejemplo, para hacer que el shell por defecto sea /bin/bash se ejecutaría:

```
$ chsh -s /bin/bash
```

Una de las principales características del shell es que se puede programar usando ficheros de texto que contienen órdenes que interpretará la ejecución de los mismos. Estos ficheros de texto se llaman: **scripts, shell scripts o guiones shell**.

Una vez creados, estos shell scripts pueden ser ejecutados tantas veces como se desee, realizando las operaciones que se programaron. El shell ofrece construcciones y facilidades para hacer más sencilla su programación. La programación shell en Unix/Linux es equivalente a los archivos .BAT en Windows/MS-DOS, con la diferencia que en Unix/Linux son mucho más potentes.

Definición 0.1. Shell Script, guión shell: Es un fichero de texto ejecutable que contiene órdenes ejecutables por el shell.

Un guión shell puede incluir **comentarios**, para ello se utiliza el carácter # al inicio del texto que forma el comentario.

En un guión shell se puede indicar el tipo de shell con el que se debe de interpretar o ejecutar, indicando en la primera línea del mismo lo siguiente:

$\#!/bin/bash$

Aunque en este caso, el carácter # del principio no es un comentario al seguirle el signo !. Aquí, el programa que preprocesa el script, selecciona el /bin/bash por esta primera línea.

La programación de shell es una de las herramientas más funcionales por los administradores y usuario de Unix/Linux, ya que permite automatizar tareas complejas, órdenes repetitivas y ejecutarlas con una sola llamada al script, o hacerlo automáticamente a horas escogidas sin intervención de personas. A continuación se muestran una serie de ejemplos de distintas tareas que se suelen automatizar con scripts:

- Tareas administrativas: algunas partes de los sistemas UNIX son guiones shell. Para poder entenderlos y modificarlos es necesario tener alguna noción sobre la programación de scripts.
- Tareas tediosas que sólo se van a ejecutar una o dos veces, no importa el rendimiento del programa resultante pero sí conviene que su programación sea rápida.
- Hacer que varios programas funcionen como un conjunto de forma sencilla.
- Pueden ser un buen método para desarrollar prototipos de aplicaciones más complejas que posteriormente se implementarán en lenguajes más potentes.

Conocer a fondo el shell aumenta tremendamente la rapidez y productividad a la hora de utilizarlo, incluso sin hacer uso de guiones shell.

Los guiones shells pueden utilizar un sin número de herramientas como:

- Los guiones shells pueden utilizar un sin número de herramientas como: top, ps, grep, etc.
- Funciones internas del shell, por ejemplo echo, let, etc.
- Lenguaje de programación del shell, por ejemplo if /then/else/fi, etc
- Programas y/o lenguajes de procesamiento en línea, por ejemplo awk, sed, Perl.
- Programas propios del usuario escritos en cualquier lenguaje.

Si un guión shell se queda pequeño para lo que queremos hacer, existen otros lenguajes interpretados mucho más potentes como Python o Perl.

El intérprete de órdenes seleccionado para realizar estas prácticas es el Bourne Again Shell, cuyo ejecutable es /bin/bash. El resto del contenido de este documento está centrado en este intérprete de órdenes.

Funcionamiento del Shell

Supongamos que tenemos el siguiente guión Shell:

```
#!/bin/bash
clear
date
```

al ejecutarse el proceso es el siguiente:

1 El shell /bin/bash hacer un fork.

2 El proceso padre espera mientras no termina el nuevo hijo.

3 El proceso hijo hace un fork y un exec para ejecutar la orden clear, a continuación ejecuta un wait para esperar a que termine la ejecución de clear.

4 Una vez que ha terminado la orden clear, el proceso hijo repite los mismos pasos pero esta vez ejecutando la orden date.

5 Si quedasen órdenes por ejecutar se seguiría el mismo procedimiento.

5 Cuando finaliza el proceso hijo, hace que el padre se despierte.

Variables y Parámetros

Variables

Cada shell tiene unas variables ligadas a él, a las que el usuario puede añadir tantas como desee. Para dar un valor a una variable se usa:

```
variable=valor
```

donde variable es el nombre de la misma. Es interesante destacar que no puede haber un espacio entre el nombre de la variable, el signo ȳ el valor. Por otra parte, si se desea que el valor contenga espacios, es necesario utilizar comillas. Para obtener el valor de una variable hay que ante ponerle a su nombre el carácter $. Por ejemplo, para visualizar el valor de una variable:

```
echo $variable
```

Un ejemplo del uso de las variables sería:

```
$ mils=`ls -l`    # Se crea una nueva variable
$ mils            # No hace nada, buscará el ejecutable
                        # mils, que no existe
$ $mils           # Ejecutará la orden "ls -l"
$ echo $mils     # Muestra el contenido de la variable
                       # mils, i.e., "ls -l"
$ fecha=$(date) # También se pueden asignar órdenes de esta forma.
```

Las variables se dividen en dos tipos:

- Variables locales: son aquellas que no son heredadas por los procesos hijos del shell actual (cuando se realiza un fork).
- Variables de entorno: Estas variables son heredadas por los procesos hijos cuando se ejecuta un fork.

La orden export convierte una variable local en variable de entorno:

```
$ set    # comando que visualiza todas las variables, locales y de entorno
$ env  # comando que visualiza las variables de entorno
$ export mils  # Convierte la variable mils en variable de entorno
$ export var=valor       #Crea la variable, le asigna "valor"
$ unset variable # elimina la variable de entorno
```

Con la orden unset se pueden restaurar o eliminar variables o funciones. Por ejemplo, la siguiente instrucción elimina el valor de la variable mils:

```
$ unset mils
```

Además de las variables que puede definir el programador, un shell tiene definidas, por defecto, una serie de variables, las más importantes son:

- PS1: prompt primario. Por defecto:

  PS1=[\u@\h \W]\$

- PS1: prompt secundario.
- LOGNAME: nombre de usuario.
- HOME: directorio de trabajo del usuario actual. (cd $HOME ó cd ∼).
- PATH: rutas utilizadas para ejecutar órdenes o programas. El directorio actual no está incluido en la búsqueda por defecto (en MSDOS sí) y si se desea debe formar parte de la variable PATH. Por ejemplo:

  ```
  $ PATH=/bin:/home/user:/usr/bin/:.:
  $ PATH=$PATH:/usr/sbin:
  ```

- PWD: directorio activo.
- TERM: el tipo de la terminal actual.
- SHELL: shell actual.

Las siguientes variables son muy útiles al programar los guiones shells:

- $?: esta variable contiene el valor de salida de la última orden ejecutada. Es útil para saber si una orden ha finalizado con éxito o ha tenido problemas. Un '0' indica que no ha habido errores, otro valor indica que sí ha habido errores.
- $!: identificador de proceso de la última orden ejecutada en segundo plano.
- $$: el identificador de proceso (PID) de este shell, útil para incluirlo en nombres de ficheros para hacerlos únicos.
- $-: las opciones actuales suministradas para esta invocación del shell. (se pueden consultar mediante **info bash –index-search=set**)
- $*: todos los argumentos del shell comenzando por el $1. Esto es, "$*" es equivalente a "$1 $2. . . .", donde los parámetros se separan por espacios.
- $@: igual que el anterior.

Parámetros

Como cualquier programa, un guión shell puede recibir parámetros en la línea de órdenes para tratarlos durante su ejecución. Los parámetros recibidos se guardan en una serie de variables que el script puede consultar cuando lo necesite. Los nombres de estas variables son:
$1 $2 $3 ... $10 $11 ...

- La variable $0 contiene el nombre con el que se ha invocado al script, es decir el nombre del programa.
- $1 contiene el primer parámetro.
- $2 contiene el segundo parámetro.
- ...

A continuación se muestra un sencillo ejemplo de un guión shell que muestra los cuatro primeros parámetros recibidos:

```
#!/bin/bash
echo El nombre del programa es $0
echo El primer parámetro recibido es $1
echo El segundo parámetro recibido es $2
echo El tercer parámetro recibido es $3
echo El cuarto parámetro recibido es $4
```

La orden shift mueve todos los parámetros una posición a la izquierda, esto hace que el contenido del parámetro $1 desaparezca, y sea reemplazado por el contenido de $2, $2 es reemplazado por $3, etc.

La variable $# contiene el número de parámetros que ha recibido el script. Como se indicó anteriormente $* o $ contienen todos los parámetros recibidos. La variable $ es útil cuando queremos pasar a otros programas algunos de los parámetros que nos han pasado.

Según todo esto, un ejemplo sencillo de guión shell que muestra el nombre del ejecutable, el número total de parámetros, todos los parámetros y los cuatro primeros parámetros es el siguiente:

```
#!/bin/bash

        echo El nombre del programa es $0
        echo El número total de parámetros es $#
        echo Todos los parámetros recibidos son $*
        echo Todos los parámetros recibidos son $@
        echo El primer parámetro recibido es $1
        shift
        echo El segundo parámetro recibido es $1
        shift
        echo El tercer parámetro recibido es $1
        echo El cuarto parámetro recibido es $2
```

Reglas de evaluación de variables

A continuación se describen las reglas que gobiernan la evaluación de las variables del guión shell:

- $var: significa el valor de la variable o nada si la variable no está definida, no muestra nada.
- ${var}: igual que el anterior excepto que las llaves contienen el nombre de la variable a ser sustituida.
- ${var-thing}: el valor de var si está definida, si no thing.
- ${var=thing}: valor de var si está definida, si no thing y el valor de var pasa a ser thing.
- ${var?message}: si definida, $var; si no, imprime el mensaje en el terminal del shell. Si el mensaje esta vacío imprime uno estándar.
- ${var+thing}: thing si $var está definida, si no nada.

El siguiente ejemplo muestra cómo podemos usar una variable, asignándole un valor en caso de que no esté definida. Esto es muy útil para trabajar con variables numéricas que no sabemos si están o no definidas.

```
$ echo El valor de var1 es ${var1}
        # No está definida, no imprimirá nada
$ echo El valor de la variable es ${var1=5}
        # Al no estar definida, le asigna el valor 5
$ echo Su nuevo valor es $var1
        # Su valor es 5
```

Pero si lo que queremos es usar un valor por defecto, en caso de que la variable no esté definida, pero sin inicializar la variable, se puede utilizar el siguiente ejemplo:

```
$ echo El valor de var1 es ${var1}
       # No está definida, no imprimirá nada
$ echo El valor de la variable es ${var1-5}
       # Al no estar definida, utiliza el valor 5
$ echo El valor es $var1
       # Su valor sigue siendo nulo, no se ha definido
```

Por otro lado, si lo que queremos es usar el valor de la variable, y en caso de que no esté definida, imprimir un mensaje, podemos usar lo siguiente:

```
$ echo El valor de var1 es ${var1}
       # No está definida, no imprimirá nada
$ echo El valor de la variable es ${var1? No está definida...}
       # Al no estar definida, se muestra en pantalla el mensaje
$ echo El valor es $var1
       # Su valor sigue siendo nulo, no se ha definido
```

Este último ejemplo nos muestra cómo utilizar un valor por defecto si una variable está defini- da, o "nada", sino está definida:

```
$ var1=4  # Le asigna el valor 4
$ echo El valor de var1 es ${var1}
       # El valor mostado será 4
$ echo El valor de la variable es ${var1+5}
       # Al estar definida, se utiliza el valor 5
$ echo El valor es $var1
       # Su valor sigue siendo 4
```

Arrays

La shell permite que se trabaje con arrays (o listas) mono dimensionales. Un array es una colección de elementos todos del mismo tipo, dotados de un nombre, y que se almacenan en posiciones contiguas de memoria. El primer elemento del array está numerado con el 0. No hay un tamaño límite para un array, y la asignación de valores se puede hacer de forma alterna. La sintaxis para declarar un array es la siguiente:

```
nombre_array=(val1 val2 val3 ...)    #Crea e inicializa un array
nombre_array[x]=valor                #Asigna un valor al elemento x

Para acceder a un elemento de la lista se utiliza la siguiente sintaxis:

     ${nombre_array[x]}  # Para acceder al elemento x
     ${nombre_array[*]}  # Para consultar todos los elementos
     ${nombre_array[@]}  # Para consultar todos los elementos

La diferencia entre user * y @ es que en
${nombre_array[*]} # expande los elementos del array como
                        # si fueran una única palabra,
                        # mientras que ${nombre_array[@]} se expande para formar
                       # cada elemento del array una palabra distinta.

Si al referenciar a un array no se utiliza subíndice se considera
que se está referenciando a su primer elemento.

Para conocer el tamaño en bytes del array se utiliza

${nombre_array[x]} # donde x puede ser un subíndice, o bien los caracteres * ó @.
```

Es interesante destacar la diferencia entre ejecutar las siguientes órdenes:

```
$ aux=`ls `
$ aux1(`ls `)
```

En el primer caso, la variable aux contiene la salida de ls como una secuencia de caracteres. Mien- tras que en el segundo caso, al haber utilizado los paréntesis, aux1 es un array, y cada entrada está formada por los nombres de fichero devueltos por la orden ls.

Supongamos que el directorio actual tenemos los siguientes ficheros: a.latex, b.latex, c.latex, d.latex, e.latex f.latex, observe el resultado de ejecutar las órdenes anteriores:

```
$ ls
    a.latex b.latex c.latex d.latex e.latex f.latex
$ aux=`ls`
$ echo $aux
    a.latex b.latex c.latex d.latex e.latex f.latex
$ echo ${aux[0]}
    a.latex b.latex c.latex d.latex e.latex f.latex
$ aux1=(`ls`)
$ echo ${aux1[0]}
a.latex
```

Entrecomillado y caracteres especiales

El entrecomillado se emplea para quitar el significado especial para el shell de ciertos meta caracteres o palabras. Puede emplearse para que caracteres especiales no se traten de forma especial, para que palabras reservadas no sean reconocidas como tales, y para evitar la expansión de parámetros.

Los meta caracteres (un meta carácter es uno de los siguientes caracteres | & ; () < > espacio tab) tienen un significado especial para el shell y deben ser protegidos o entrecomillados si quieren representarse a sí mismos. Hay 3 mecanismos de protección: el carácter de escape, comillas simples y comillas dobles.

Una barra inclinada inversa no entrecomillada (\) es el carácter de escape, por ejemplo:

```
$ echo -e "Caracteres␣especiales\n"  # comilla simple también vale
También podemos hacer para escribir en varias líneas:
$ echo -e \
> "hola\n"
hola
```

Sustitución de órdenes

Poner una cadena entre comillas invertidas, o bien entre paréntesis precedida de un signo $, supone ejecutar su contenido como una orden y sustituir su salida, forzando al shell a ejecutar antes lo que va entre las comillas. La sintaxis es:

```
`orden` ó
$(orden)
```

Este proceso se conoce como sustitución de órdenes. A continuación se muestran varios ejemplos:

```
$ aux=`ls -lai`   # Ejecuta ls -lai y después lo asigna a aux
$ echo $aux       # Muestra el contenido de aux
$ fecha=$(date)   # Ejecuta date y almacena el valor en fecha
$ echo $fecha     # Muestra el contenido de fecha
```

Hay que tener en cuenta que el shell antes de ejecutar una orden, trata el significado especial de los caracteres especiales y de entrecomillado, así como los de generación de nombres de fichero. Por ejemplo:

```
$ var=`ls -al`  # Primero ejecuta y luego le asigna el valor
$ echo $var     # Muestra el contenido de la variable
$ echo `date`   # Primero se ejecuta date y luego echo
```

Estructuras de Control

IF y CASE

En un guión shell se pueden introducir condiciones, de forma que determinadas órdenes sólo se ejecuten cuando se cumplan unas condiciones concretas. Para ello se utilizan las órdenes if y case, con la siguiente sintaxis:

```
if [ expresión ]       #(No tiene porque ser un test)
then
    órdenes a ejecutar si se cumple la condición
elif [expresión]
then
     órdenes a ejecutar si se cumple la condición
             # (el bloque elif y sus órdenes son opcionales)
else
     órdenes a ejecutar en caso contrario
            # (el bloque else y sus órdenes son opcionales)
fi
```

La expresión a evaluar por if puede ser o un test o bien otras expresiones, como una lista de órdenes (usando su valor de retorno), una variable o una expresión aritmética, básicamente cualquier orden que devuelva un código en $?. Un ejemplo del funcionamiento de la orden if es:

```
if grep -q print threads1.py
then
        echo Palabra print encontrada
else
        echo Palabra no encontrada
fi
```

La sintaxis de la orden *case* con un ejemplo:

```
case $var in
     v1)      ..
     ..
     ;;
     v2|v3)   ..
     ..
     ;;
     *)       ..
     ;;
esac
```

```
#!/bin/bash
read var
case $var in
[Ss]) echo opción sí
;;
[N]|[n]) echo opción no
;;
1) echo opción 1
;;
DOS) echo opción DOS
```

```
;;
*) echo Ninguna opción # por defecto
;;
esac
```

WHILE y UNTIL

También es posible ejecutar bloques de órdenes de forma iterativa dependiendo de una condi- ción. La comprobación puede ser al principio o al final (while o until respectivamente). La sintaxis es:

```
while [ expresión ] # Mientras la expresión sea cierta ...
do
...
done
```

```
until [ expresión ] # Mientras la expresión sea falsa ...
do
...
done
```

FOR

Con la orden for se ejecutan bloques de órdenes, permitiendo que en cada iteración una deter- minada variable tome un valor distinto. La sintaxis es la siguiente:

```
for var in lista
do
        ..$var..
        ...
done
```

Por ejemplo:

```
for i in 10 30 70
do
            echo Mi número favorito es $i
            # ($i valdría 10 en la primera iteración, 30 en
            # la segunda y 70 en la tercera)
done
```

SELECT

La sintaxis de la orden select es:

```
select name [ in word ] ;
do
        ...
        list ;
        ...
done
```

select genera una lista de items al expandir la lista "word", presentando en la pantalla esta lista de items precedidos cada uno de un número. A continuación se presenta un prompt pidiendo que se introduzca una de las entradas.

Y se lee de la entrada estándar la opción elegida. Si la respuesta dada es uno de los números de la lista presentada, entonces la variable "name" toma el valor de esa opción, y se ejecuta con ella la lista de órdenes indicada.

Si la respuesta es vacía, se vuelve a presentar la lista, y si es EOF se finaliza. Además el número de opción seleccionada se almacena en la variable REPLY.

La lista de órdenes se ejecuta después de cada selección, mientras no se termine, bien con break, bien con EOF. El valor de salida de select será igual a valor de la última orden ejecutada.

Ejemplo de **select:**

```
#!/bin/bash
select respuesta in "Ver contenido directorio actual" \
            "Salir"
        do
            echo Ha seleccionado la opción: $respuesta
            case $REPLY in
            1) ls .
            ;;
            2) break
            ;;
            esac
done

$ ./select1.sh
1) Directorio_Actual
2) Salir
#?
```

BREAK y CONTINUE

Las órdenes break y continue sirven para interrumpir la ejecución secuencial del cuerpo del bucle.

break transfiere el control a la orden que sigue a done, haciendo que el bucle termine antes de tiempo.

continue transfiere el control a done, haciendo que se evalúe de nuevo la condición, prosiguiendo el bucle.

En ambos casos, las órdenes del cuerpo del bucle siguientes a estas sentencias, no se ejecutan. Lo normal es que formen parte de una sentencia condicional, como if.

Un par de ejemplos de su uso es:

```
# Muestra todos los parámetros, si encuentra una "f" finaliza
while [ $# -gt 0 ]
do
        if [ $1 = "f" ]
      then
                break
       fi
       echo Parámetro: $1
         shift
done

# Muestra todos los parámetros, si encuentra una "f"
# se lo salta y continúa el bucle
while [ $# -gt 0 ]
do
```

```
        if [ $1 = "f" ]
         then
                shift
                continue
         fi
        echo Parámetro: $1
         shift
done
```

Órdenes internas de Bash

Una orden interna del shell es una orden que el intérprete implementa y que ejecuta sin llamar a programas externos. Por ejemplo, echo es una orden interna de bash y cuando se llama desde un script no se ejecuta el fichero /bin/echo. Algunos de las órdenes internas más utilizadas son:

- echo: envía una cadena a la salida estándar, normalmente la consola o una tubería. Por ejemplo:
  echo El valor de la variable es $auxvar
- read: lee una cadena de la entrada estándar y la asigna a una variable, permitiendo obtener entrada de datos por teclado en la ejecución de un guión shell:

  ```
  echo -n "Introduzca un valor para var1: "
  read var1
  echo "var1 = $var1"
  ```

- let arg [arg]: cada arg es una expresión aritmética a ser evaluada:

  ```
  $ b=11
  $ let a=$b+7
  $ echo $a

  $ let a=6+7
  $ echo El resultado de la suma es $a
  El resultado de la suma es: 13
  $ let b=7%5
  $ echo El resto de la división es: $b
  El resto de la división es: 2
  ```

- Evaluación de expresiones:

  ```
  $ echo $((2**6))
  64
  $ echo El resultado de la suma es $((6+7))
  El resultado de la suma es: 13
  ```

La orden test

test es una orden que permite evaluar si una expresión es verdadera o falsa. Los tests no sólo operan sobre los valores de las variables, también permiten conocer, por ejemplo, las propiedades de un fichero.

Principalmente se usan en la estructura if/then/else/fi para determinar qué parte del script se va a ejecutar. Un if puede evaluar, además de un test, otras expresiones, como una lista de órdenes (usando su valor de retorno), una variable o una expresión aritmética, básicamente cualquier orden que devuelva un código en $?.

La sintaxis de test puede ser una de las dos que se muestran a continuación:

```
test expresión
[ expresión ]
```

Atención: Los espacios en blanco entre la expresión y los corchetes son necesarios. La expresión puede incluir operadores de comparación como los siguientes:

-eq	Igual a
-ne	no igual a
-lt	menor que
-le	menor o igual a
-gt	mayor que
-ge	mayor o igual

Es importante destacar que en las comparaciones con números si utilizamos una variable y no está definida, saldrá un mensaje de error. El siguiente ejemplo, al no estar la variable "e" definida, mostrará un mensaje de error indicando que se ha encontrado un operador inesperado.

```
if [ $e -eq 1 ]
then
    echo Vale 1
else
    echo No vale 1
fi
```

Por el contrario, en el siguiente ejemplo, a la variable "e" se le asigna un valor si no está definida, por lo que sí funcionaría:

```
if [ ${e=0} -eq 1 ]
       then
            echo Vale 1
       else
            echo No vale 1
fi
```

- Para caracteres alfabéticos o cadenas:

-z cadena	Verdad si la longitud de cadena es cero.
-n cadena ó cadena	Verdad si la longitud de cadena no es cero.
cadena1 == cadena2	Verdad si las cadenas son iguales. Se puede emplear = en vez de ==.
cadena1 != cadena2	Verdad si las cadenas no son iguales.
cadena1 < cadena2	Verdad si cadena1 se ordena lexicográficamente antes de cadena2 en la localización en curso.
cadena1 > cadena2	Verdad si cadena1 se clasifica lexicográficamente después de cadena2 en la localización en curso.

Figure 1: cadenas

- En expresión se pueden incluir operaciones con ficheros, entre otras:

Ejemplos de uso

Uno de los usos más comunes de la variable $#, es validar el número de argumentos necesarios en un programa shell. Por ejemplo:

```
 if test $# -ne 2
       then
            echo "se necesitan dos argumentos"
            exit
fi
```

-e fichero	El fichero existe.
-r fichero	El fichero existe y tengo permiso de lectura.
-w fichero	El fichero existe y tengo permiso de escritura.
-x fichero	El fichero existe y tengo permiso de ejecución.
-f fichero	El fichero existe y es regular.
-s fichero	El fichero existe y es de tamaño mayor a cero.
-d fichero	El fichero existe y es un directorio.

Figure 2: ficheros

El siguiente ejemplo comprueba el valor del primer parámetro posicional. Si es un fichero (- f) se visualiza su contenido; sino, entonces se comprueba si es un directorio y si es así cambia al directorio y muestra su contenido. En otro caso, echo muestra un mensaje de error.

```
if test -f "$1"          # ¿ es un fichero ?
then
            more $1
elif test -d "$1"        # ¿ es un directorio ?
then
            (cd $1;ls -l|more)
else                     # no es ni fichero ni directorio
            echo "$1 no es fichero ni directorio"
fi
```

Depuración

Una buena idea para depurar los programas es la opción -x en la primera línea:

```
#!/bin/bash -x
```

Como consecuencia, durante la ejecución se va mostrando cada línea del guión después de sustituir las variables por su valor, pero antes de ejecutarla.

Otra posibilidad es utilizar la opción -v que muestra cada línea como aparece en el script (tal como está en el fichero), antes de ejecutarla:

```
#!/bin/bash -v
```

Otra opción es llamar al programa usando el ejecutable bash. Por ejemplo, si nuestro programa se llama prac1, se podría invocar como:

```
$ bash -x prac1
          ó
$ bash -v prac1
```

Ejercicios

(a) script1.sh que imprima "Cuál es tu nombre", se introduzca una variable por teclado, y luego imprima "Mi nombre es <tu_nombre>" y la fecha actual del sistema.

(b) script2.sh: muestra su número PID y después llama a un programa llamado **num** a través de del punto (. num). Cuando num termina su ejecución, la orden (.) devuelve el control al programa que lo llamó, el cual muestra el mensaje. (Nota: el fichero num lleva un sólo **echo "Soy num"**).

(c) script3.sh que copie un fichero en otro, controlando que el numéro de argumentos sea exactamente dos y que ambos parámetros sean ficheros.

(d) script4.sh: programa que que imprima en pantalla el contenido de un fichero de datos o el contenido de todos los ficheros de un directorio.

(e) script5.sh: haga un script que imprima un menú con 5 opciones que realice lo siguiente:

1) Ver directorio actual # lista el contenido del directorio actual
2) Copiar ficheros # le dos variables (que son ficheros) y copia de uno a otro
3) Editar ficheros # lee una variable (fichero) y abre el fichero con vi
4) Imprimir fichero
5) Salir del menú

Nota: Puedes hacerlo con select o con while.

(f) script6.sh: lee dos números del teclado e imprime su suma, (usando las órdenes read, printf y let).

(g) script7.sh: escribir un guión shell que, dado el "username" de un usuario, nos devuelva cuántas veces esa persona está conectada. (Usa: who, grep, wc).

(h) script8.sh: Escribir un guión shell que liste los directorios existentes en el directorio actual.

(i) script9.sh: script que pida por teclado un número PID de proceso y lo elimine.

(j) script10.sh: Escribir un guión shell llamado ver que para cada argumento que reciba realice una de las siguientes operaciones:

- si es un directorio ha de listar los ficheros que contiene,
- si es un fichero regular lo tiene que mostrar por pantalla,
- en otro caso, que indique que no es ni un fichero ni un directorio.