

**PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ
FACULTAD DE INGENIERÍA**

**SECCIÓN DE INGENIERÍA INFORMÁTICA
SISTEMAS OPERATIVOS**

MATERIAL PARA EL PRIMER LABORATORIO

Que es el *shell* ?

El *shell* es un programa que ofrece una interfaz al sistema Operativo. Este lee las entradas y ejecuta los programas que se le especifiquen. Mientras los programas son ejecutados, las salidas son mostradas hacia un dispositivo que por defecto es la pantalla. Por esta razón también es conocido como *interprete de comandos*.

Pero el *shell* es más que un interprete de comandos. Es sobre todo un potente lenguaje de programación que posee sentencias condicionales, lazos, funciones y muchas construcciones análogas a otros lenguajes de programación. Por supuesto, este lenguaje es interpretado y no compilado.

Tipos de *shell*

Existe una variedad de *shell*, entre los más conocidos tenemos: *Bourne shell*, *C shell*, *Korn shell*, *Bash*, *Z shell*, *Almquist shell*, *Debian Almquist shell*.

Para un mayor detalle puede leer: https://es.wikipedia.org/wiki/Shell_de_Unix

¿Qué tipo de *shell* tengo disponible en mi sistema?

```
alulab@minix:~$ cat /etc/shells
# /etc/shells: valid login shells
/bin/sh
/bin/bash
/bin/rbash
/bin/dash
alulab@minix:~$
```

¿Qué *shell* y versión estoy usando?

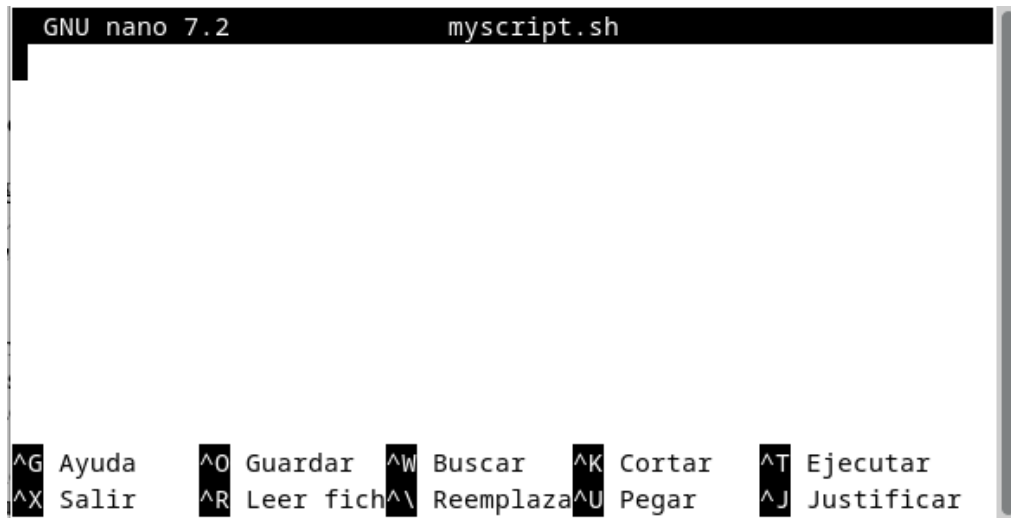
```
alulab@minix:~$ echo $SHELL
/bin/bash
alulab@minix:~$ echo $BASH_VERSION
5.0.16(1)-release
alulab@minix:~$
```

¿Que es un *shell script*?

Es un programa que es ejecutado por el *shell* de forma interpretada. Este programa debe ser escrito en texto plano en un archivo, para esto usted puede hacer uso de cualquier editor de textos. Le recomendamos hacer uso de editores sin entorno gráfico. Entre los más sencillos tenemos *pico*, *nano*, hasta algunos más sofisticados como *vim* o *emacs*.

Mi primer shell script

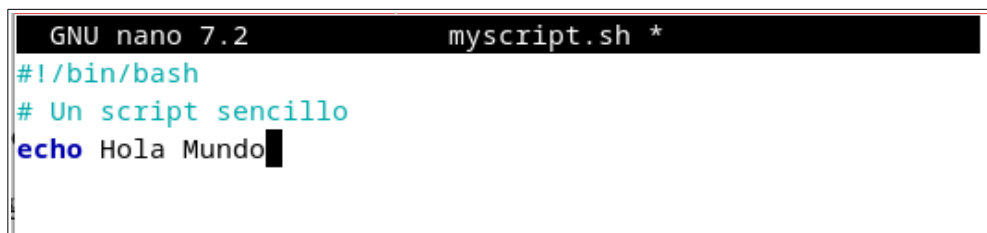
En la presente guía se usará el editor *nano*. En la línea de comando escriba `nano myscript.sh` y presiones <Enter>



```
GNU nano 7.2 myscript.sh
```

^G Ayuda ^O Guardar ^W Buscar ^K Cortar ^T Ejecutar
^X Salir ^R Leer fich ^\ Reemplaza ^U Pegar ^J Justificar

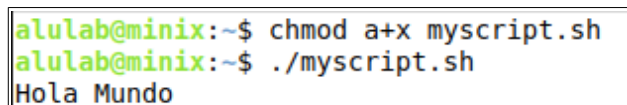
a continuación escriba el siguiente contenido:



```
GNU nano 7.2 myscript.sh *
#!/bin/bash
# Un script sencillo
echo Hola Mundo
```

Para grabar presione Ctrl+O, en la parte inferior se le pedirá que confirme el nombre. Presione Enter. A continuación presione Ctrl+x para salir de *nano*.

Recuerde que antes de ejecutarlo tiene que otorgarle permisos de ejecución (+x) con la orden `chmod`. Después lo puede ejecutar desde la línea de ordenes.



```
alulab@minix:~$ chmod a+x myscript.sh
alulab@minix:~$ ./myscript.sh
Hola Mundo
```

La explicación de por qué se usa `./` antes del nombre, es porque el *shell* busca en el PATH el nombre del archivo para ejecutarlo. Pero en el PATH no se encuentra el directorio actual. Por ese motivo se le debe de indicar `./` No es una buena practica agregar el directorio actual al PATH.

El shebang (#!)

A la secuencia de caracteres `#!` se le conoce como *shebang* y su misión es indicarle la ruta del programa (o interprete) que debe ser usado para ejecutar (o interpretar) las líneas del archivo texto que se encuentran a continuación. En este caso le está diciendo que quien debe interpretar las líneas de nuestro *script* es `/bin/bash`

Comentarios

Como se puede apreciar en el *script* de arriba, cualquier línea precedida por un # se considera un comentario.

Variables

Variables escalares son aquellas que solo pueden contener un valor. Se definen de la siguiente forma:

name=value

donde name es el nombre de la variable y value es el valor que debería tomar.

Los valores también pueden ser tomados como entrada desde el teclado usando la orden read.

A continuación un ejemplo sencillo en *myscript2.sh*

```
GNU nano 6.2      myscript2.sh
#!/bin/bash
# Saludo de bienvenida
curso="1INF29 Sistemas Operativos"
echo "Ingresa tu nombre"
read nombre
echo "Hola $nombre bienvenido(a) a $curso"
```

```
alulab@minix:~$ chmod a+x myscript2.sh
alulab@minix:~$ ./myscript2.sh
Ingresa tu nombre
Alejandro
Hola Alejandro bienvenido(a) a 1INF29 Sistemas Operativos
alulab@minix:~$ █
```

Observe que no hay espacios ni antes, ni después del símbolo "=", de lo contrario se emitirá un mensaje de error. El siguiente ejemplo en la línea de ordenes trata de mostrar este punto.

```
alulab@minix:~$ usuario = alejandro
-bash: usuario: command not found
alulab@minix:~$ usuario= alejandro
-bash: alejandro: command not found
alulab@minix:~$ usuario=alejandro
alulab@minix:~$ echo $usuario
alejandro
alulab@minix:~$ █
```

Los nombres de las variables solo pueden contener letras (a hasta la z o A hasta Z), números (0 hasta 9), y el carácter de subrayado (_). Además, los nombres de las variables solo pueden empezar con una letra o subrayado.

Arreglos, son aquellas que pueden contener más de un valor, con un solo nombre y un índice.

```
alulab@minix:~$ frutas[1]=manzana
alulab@minix:~$ frutas[2]=piña
alulab@minix:~$ frutas[3]=durazno
alulab@minix:~$ echo ${frutas[*]}
manzana piña durazno
alulab@minix:~$ frutas=uva
alulab@minix:~$ echo ${frutas[*]}
uva manzana piña durazno
alulab@minix:~$ echo ${frutas[2]}
piña
alulab@minix:~$
```

Observe en el ejemplo de arriba, se crea un arreglo de 3 entradas y se asigna valores. Para acceder al segundo valor se emplea `${frutas[2]}`. Si se desea acceder a todos los valores: `${frutas[*]}`. Note que si se asigna a una variable sin índice, pero con el mismo nombre que el arreglo, esta asume el arreglo en la posición 0. En el ejemplo de arriba `frutas=uva` es equivalente a `frutas[0]=uva`.

También se pudo crear el arreglo de la siguiente forma, para obtener el mismo efecto:

```
frutas=(uvas manzana piña durazno)
```

Variables de solo lectura, es una variable cuyo valor no puede ser cambiada después que ha sido definida. Se pueden declarar variables de solo lectura, tanto las variables escalares como las de tipo arreglo.

```
alulab@minix:~$ fruta=kiwi
alulab@minix:~$ readonly fruta
alulab@minix:~$ echo $fruta
kiwi
alulab@minix:~$ fruta=fresa
-bash: fruta: readonly variable
alulab@minix:~$
```

Eliminando variables

Para decirle al *shell* que remueva una variable, tanto escalar como arreglo, de la lista de variables que éste registra, se emplea:

```
unset frutas
```

Una variable de solo lectura no puede ser eliminada y su valor persiste mientras el *shell* persista.

Variables del *shell* y del entorno

Cuando el *shell* inicia un programa, este le pasa al programa un conjunto de variables llamadas de entorno (*environment*). El entorno es usualmente un pequeño subconjunto de variables definidas en el *shell*.

Las variables hasta ahora estudiadas son variables locales. Las variables locales son variables cuyo valor está restringido a un único *shell*. Las variables locales no son pasados a programas iniciados por el *shell*.

Adicionalmente a las variables locales y las variables del entorno, hay una tercera categoría llamadas variables del *shell*. Estas son variables especiales configuradas por el *shell* para un funcionamiento adecuado del mismo. Algunas variables del *shell* son variables del entorno, mientras que otras son variables locales.

<i>Attribute</i>	<i>Local</i>	<i>Environment</i>	<i>Shell</i>
Accessible by child processes	No	Yes	Yes
Set by users	Yes	Yes	No
Set by the shell	No	No	Yes
User modifiable	Yes	Yes	No
Required by the shell	No	No	Yes

“ Una comparación entre variables locales, del entorno y del *shell*

Exportando variables del entorno

Las variables del entorno son solo variables locales que han sido colocadas en el entorno vía el comando

```
export name
```

La variable especificada por *name* es colocada en el entorno. El proceso de colocar variables en el entorno es a menudo referida como exportar la variable.

```
alulab@minix:~$ export MYVAR="Sistemas Operativos"
alulab@minix:~$ env | grep MYVAR
MYVAR=Sistemas Operativos
alulab@minix:~$
```

El comando `env` sin opciones, muestra el nombre y valor de cada variable del entorno. En este ejemplo se usa `grep` para filtrar solo aquellas líneas que contengan `MYVAR`

Variables del *Shell*

Las variables del *shell* son variables que el *shell* establece durante la inicialización y lo usa internamente. A continuación algunas de las variables del *shell* más conocidas:

PWD	Indica el actual directorio de trabajo
PATH	Indica la ruta para búsqueda de comandos. Es una lista de directorios separadas por dos puntos en el cual el <i>shell</i> busca los comandos.
HOME	Indica el directorio asignado por el sistema al usuario actual.

```

alulab@minix:~$ set | head -n 12
BASH=/bin/bash
BASHOPTS=checkwinsize:cmdhist:complete_fullquote:expand_aliases:extglob:extquote
:force_ignore:globasciiranges:histappend:interactive_comments:login_shell:prog
omp:promptvars:sourcepath
BASH_ALIASES=()
BASH_ARGC=([0]="0")
BASH_ARGV=()
BASH_CMDS=()
BASH_COMPLETION_VERSION=([0]="2" [1]="10")
BASH_LINENO=()
BASH_REMATCH=()
BASH_SOURCE=()
BASH_VERSION=([0]="5" [1]="0" [2]="16" [3]="1" [4]="release" [5]="x86_64-pc-lin
ux-gnu")
BASH_VERSION='5.0.16(1)-release'
alulab@minix:~$ █

```

El comando `set` sin opciones, muestra el nombre y valor de cada variable del *shell*. En el ejemplo de arriba la salida de `set` ha sido tomada como entrada, debido al *pipe*, (`|`) para la orden `head` que filtra solo las primeras 12 líneas de lo que se le envió.

Sustitución

Cuando el *shell* encuentra una expresión que contiene uno o más meta-caracteres, éste realiza sustituciones en la expresión. Meta-caracteres son caracteres que tienen un significado especial en el *shell*. Sustitución es el proceso por el cual el *shell* convierte una cadena conteniendo meta-caracteres en una cadena diferente que es el resultado de interpretar los meta-caracteres. Por ejemplo en el apartado anterior se empleó el meta-caracter `$` para acceder al valor de una variable en un proceso conocido como sustitución de variable. Adicionalmente el *shell* puede llevar a cabo otras muchos tipos de sustituciones.

Sustitución de variables

-Sustitución de valor predeterminado

La primera forma de sustitución permite que un valor por defecto sea sustituido cuando el valor de una variable es *null* o no esté establecida. La sintaxis es la siguiente:

`${param:-word}`

Aquí *param* es el nombre de la variable y *word* es el valor por defecto. La sustitución se lleva a cabo solo cuando *param* no está establecida. Por tanto, *word* no es asignada a *param*; el *shell* solo reemplaza la expresión con *word*. El siguiente ejemplo ilustra el comportamiento

```

alulab@minix:~$ MIFRUTA=NARANJA
alulab@minix:~$ FRUTA=${MIFRUTA:-MANZANA}
alulab@minix:~$ echo Mi fruta es $MIFRUTA, FRUTA es $FRUTA
Mi fruta es NARANJA, FRUTA es NARANJA
alulab@minix:~$ unset MIFRUTA
alulab@minix:~$ FRUTA=${MIFRUTA:-MANZANA}
alulab@minix:~$ echo Mi fruta es $MIFRUTA, FRUTA es $FRUTA
Mi fruta es , FRUTA es MANZANA
alulab@minix:~$

```

-Asignación de valor predeterminado

La segunda forma asigna un valor a una variable cuando su valor es *null*. La sintaxis es:

`${param:=word}`

Aquí *param* es el nombre de la variable y *word* es el valor a asignar si el valor de la variable es *null*. El siguiente ejemplo ilustra el comportamiento:

```
alulab@minix:~$ unset FRUTA
alulab@minix:~$ echo FRUTA es $FRUTA
FRUTA es
alulab@minix:~$ echo FRUTA es ${FRUTA:=MANZANA}
FRUTA es MANZANA
alulab@minix:~$
```

-Error de valor nulo

Algunas veces sustituir o asignar valores preestablecidos puede ocultar problemas en un *script* del *shell*. Con el fin de señalar tales problemas en partes críticas de un *script*, se puede usar la tercera forma de sustitución de variable que envía un mensaje al error estándar cuando la variable no está establecida y el *shell* si no es iterativo, acaba. La sintaxis es:

`${param:?msg}`

Aquí, *param* es el nombre de la variable y *msg* es el mensaje a ser impreso en STDERR

Si un *script* o función del *shell* requiere que una determinada variable sea establecida para una ejecución apropiada, esta forma de sustitución de variable puede ser usada. Por ejemplo, la siguiente expresión causa que el *shell* termine, si no es iterativo, si la variable `$HOME` no está establecida.

```
: ${HOME:?}"Tu directorio home no está definido"}
```

En este ejemplo, además de usar la sustitución de variable descrita anteriormente, hace uso del comando `no-op` (forma corta de `no operation`), `∴`. Este comando no realiza trabajo alguno, solo evalúa los argumentos que le son pasados.

- Sustituir cuando se establece

Esta forma es usada para sustituir un valor cuando una variable es establecida. La sintaxis es como sigue:

`${param:+word}`

Aquí *param* es el nombre de la variable y *word* es el valor a sustituir si la variable está establecida. Si la variable no está establecida entonces nada se sustituye. Esta forma no altera el valor de la variable. Es comúnmente usada por *scripts* para indicar que el *script* está corriendo en modo de depuración.

```
alulab@minix:~$ DEBUG=YES
alulab@minix:~$ echo ${DEBUG:+"Debug está activo"}
Debug está activo
alulab@minix:~$ echo $DEBUG
YES
alulab@minix:~$
```

TAREA

Investigue el significado de las siguientes sustituciones de variables

- a) `${param:offset}`
`${param:offset:len}`
- b) `${#param}`
- c) `${param#word}`
`${param##word}`
- d) `${param%word}`
`${param%%word}`

Sustitución de ordenes

Sustitución de ordenes es el mecanismo por el cual el *shell* ejecuta un conjunto de ordenes y luego sustituye su salida en lugar de la orden. La sustitución de ordenes se lleva a cabo cuando un orden se proporciona de la siguiente forma:

``orden``

Donde *orden* puede ser un simple orden, una tubería, o una lista. Observe con detalle que se trata de las comilla invertida, no es comilla simple. A continuación algunos ejemplos

```
alulab@minix:~$ DATE=`date`
alulab@minix:~$ echo $DATE
Sat 05 Sep 2020 05:53:47 AM EDT
alulab@minix:~$ USERS=`who | wc -l`
alulab@minix:~$ echo $USERS
1
alulab@minix:~$ UP=`date ; uptime`
alulab@minix:~$ echo $UP
Sat 05 Sep 2020 05:54:49 AM EDT 05:54:49 up 6:28, 1 user, load average: 0.05, 0.02, 0.00
alulab@minix:~$
```

También se puede usar sustitución de ordenes para proveer argumentos a otras ordenes. Por ejemplo:

```
alulab@minix:~$ grep `id -un` /etc/passwd
alulab:x:1001:1001:Alumno de Informática,,,:/home/alulab:/bin/bash
alulab@minix:~$
```

Sustitución aritmética

Sustitución aritmética permite realizar simple matemática entera usando el *shell*. La sustitución aritmética se lleva a cabo cuando se encuentra una expresión de la siguiente forma:

`$((exp))`

Si `exp` no evalúa a un entero, el valor de `exp` es truncado. Como se ilustra en el siguiente comando:

```
alulab@minix:~$ X=5
alulab@minix:~$ Y=2
alulab@minix:~$ echo $(( X/Y ))
2
alulab@minix:~$ █
```

El siguiente ejemplo ilustra las reglas de precedencia

```
alulab@minix:~$ A=5
alulab@minix:~$ B=3
alulab@minix:~$ C=2
alulab@minix:~$ D=4
alulab@minix:~$ echo $(( (A + B*C) - D) / C ))
3
alulab@minix:~$ █
```

TAREA

Busque en el manual en línea del *shell* y averigüe cuáles son las operaciones permitidas en la sustitución aritmética. Pruebe en la línea de comandos las operaciones menos comunes para usted.

Entrecomillado (*Quoting*)

En los párrafos anteriores se trató el tema de sustitución, el cual ocurre automáticamente cuando se ingresa un comando conteniendo meta-caracteres o un \$. La forma como el *shell* interpreta estos caracteres especiales generalmente es útil, pero a veces es necesario desactivar la sustitución que lleva a cabo el *shell* y permitir que cada carácter se represente por sí mismo. Desactivar el significado especial de caracteres es llamado entrecomillado (*quoting*), y puede ser realizado de tres formas:

Usando *backslash* (\)

Usando comilla simple (*single quote*) ('')

Usando comillas dobles (*double quote*) ("")

Entrecomillado con *backslash*

Para empezar, usaremos el comando `echo` para tener una mejor idea de como el *shell* trata los caracteres especiales.

```
alulab@minix:~$ echo Hola mundo
Hola mundo
alulab@minix:~$ echo Hola; mundo
Hola
-bash: mundo: command not found
alulab@minix:~$ echo Hola\; mundo
Hola; mundo
alulab@minix:~$ █
```

En el ejemplo de arriba mostrado se deseaba como salida `Hola; mundo`, sin embargo debido a que para el *shell* el punto y coma representa un carácter especial cuyo significado es: fin de la orden; toma lo que viene a continuación como una nueva orden. En este caso la palabra `mundo`. Por ese motivo el *shell* responde con: orden no encontrada. Para lograr el efecto deseado, antecedemos al punto y coma con un *backslash*, y de esta forma el significado especial queda deshabilitado, tal como se muestra arriba en el ejemplo.

A continuación otros ejemplos

```
alulab@minix:~$ echo Usted posee $120 dólares
Usted posee 20 dólares
alulab@minix:~$ echo Usted posee \$120 dólares
Usted posee $120 dólares
alulab@minix:~$ echo En Windows la unidad C:\ es mi disco duro
En Windows la unidad C: es mi disco duro
alulab@minix:~$ echo En Windows la unidad C:\\ es mi disco duro
En Windows la unidad C:\ es mi disco duro
alulab@minix:~$ █
```

Usando comillas simples

La siguiente orden contiene muchos meta-caracteres que deseamos que se ignoren

```
echo <-$1250.**>; (update?) [y|n]
```

Una opción es usar *backslash* anteponiéndolo a los meta-caracteres. Pero este camino no solo es engorroso sino que difícil de leer y comprender. Observe el resultado

```
echo \<-\$1250.\*\*>; \ (update\?) \[y|n\]
```

Una mejor opción es emplear el entrecomillado simple. De la siguiente forma

```
echo '<-$1250.**>; (update?) [y|n]'
```

Usando comillas dobles

A veces se desea que en una cadena los meta-caracteres \$ y ` sean evaluados por el shell, pero el resto aún deben ser ignorados. Por ejemplo la siguiente orden

```
echo $USER owes <-$1250.**>; [ as of (`date +%m/%d`) ]
```

Si se usa entrecomillado simple, obtendremos todo como una cadena y el *shell* no habrá evaluado ningún meta-caracteres. En este caso se debe de usar comillas dobles, esto permitirá que sean ignorados todos los meta-caracteres excepto el \$ y ` que serán procesados por el *shell*. En la línea siguiente se puede ver la forma correcta (trate de explicarse por qué el meta-caracter \)

```
echo "$USER owes <-\$1250.**>; [ as of (`date +%m/%d`) ]"
```

Control de flujo

La estructura de la sentencia *if-then-else* es la siguiente

```
if orden; then
    # hacer algo
else
    # hacer otra cosa
fi
```

```
GNU nano 6.2      myscript3.sh
#!/bin/bash

echo "Ingrese nombre de usuario"
read usuario

if grep $usuario /etc/passwd; then
    echo "El usuario $usuario existe"
else
    echo "El usuario $usuario no existe"
fi
```

Si la orden después del *if* se ejecuta con éxito (devuelve 0) ejecuta lo que se encuentra después del *then*, en caso contrario ejecuta lo que se encuentra después del *else*.

```
alulab@minix:~$ chmod a+x myscript3.sh
alulab@minix:~$ ./myscript3.sh
Ingrese nombre de usuario
alulab
alulab:x:1001:1002:Alumno de Informática,,,:/home/alulab:/bin/bash
El usuario alulab existe
alulab@minix:~$ ./myscript3.sh
Ingrese nombre de usuario
alejandro
alejandro:x:1000:1000:Alejandro Toribio Bello Ruiz,,,:/home/alejandro:/bin/bash
El usuario alejandro existe
alulab@minix:~$ █
```

Usualmente la lista dada a una sentencia *if* es una o más ordenes *test*. Una orden *test* tiene la siguiente sintaxis:

`test expr`

Donde *expr* es construido usando una de las opciones entendidas por *test*. Después de evaluar *expr*, *test* retorna ya sea 0 (verdadero) o 1 (falso). Los corchetes (`[]`) a menudo son usados como una forma corta para *test*.

`[expr]`

El espacio después del corchete izquierdo (`[`) y el espacio antes del corchete derecho (`]`) son obligatorios.

Hay tres tipos de expresiones entendidas por *test*

- a) Test de archivos
- b) Comparaciones de cadenas
- c) Comparaciones numéricas

a) Test de archivos

Estas prueban si un archivo se ajusta a un criterio particular. La sintaxis general es

```
test opción archivo
```

o

```
[ opción archivo ]
```

A continuación se muestran algunas de las opciones

-d <i>pathname</i>	True si el <i>pathname</i> existe y es un directorio.
-e <i>pathname</i>	True si el archivo o directorio especificado por <i>pathname</i> existe.
-f <i>archivo</i>	True si <i>archivo</i> existe y es un archivo regular
-r <i>pathname</i>	True si el archivo o directorio especificado por <i>pathname</i> existe y tiene permisos de lectura.
-s <i>archivo</i>	True si <i>archivo</i> existe y tiene un tamaño mayor a 0
-w <i>pathname</i>	True si el archivo o directorio especificado por <i>pathname</i> existe y tiene permisos de escritura
-x <i>pathname</i>	True si el archivo o directorio especificado por <i>pathname</i> existe y tiene permiso de ejecución.

```
GNU nano 6.2                                myscript4.sh
#!/bin/bash

if [ -d temporal ]; then
    echo "el directorio temporal ya existe"
else
    mkdir temporal
fi
```

```
alulab@minix:~$ ls
Descargas  Escritorio  Música      myscript3.sh  Plantillas  Vídeos
Documentos Imágenes    myscript2.sh myscript4.sh  Público
alulab@minix:~$ chmod a+x myscript4.sh
alulab@minix:~$ ./myscript4.sh
alulab@minix:~$ ls
Descargas  Escritorio  Música      myscript3.sh  Plantillas  temporal  Vídeos
Documentos Imágenes    myscript2.sh myscript4.sh  Público
alulab@minix:~$ ./myscript4.sh
el directorio temporal ya existe
alulab@minix:~$
```

b) Comparación de cadenas

La opciones de comparación de cadenas se presentan a continuación

<code>-z str</code>	True si <i>str</i> tiene longitud 0.
<code>-n str</code>	True si <i>str</i> no tiene longitud 0.
<code>str1 = str2</code>	True si <i>str1</i> y <i>str2</i> son iguales.
<code>str1 != str2</code>	True si <i>str1</i> y <i>str2</i> no son iguales.

```
GNU nano 7.2      myscript5.sh
#!/bin/bash

if [ "$HOME" = "/home/alulab" ]; then
    echo "Todo bien"
else
    echo "Algo anda mal"
fi
```

Las variables se entrecomillan para evitar el caso cuando son nulas. Si no se entrecomillan al momento de hacer la sustitución quedaría sin nada y generaría un error.

```
alulab@minix:~$ chmod a+x myscript5.sh
alulab@minix:~$ ./myscript5.sh
Todo bien
alulab@minix:~$
```

c) Comparaciones numéricas

Para hacer comparaciones numéricas debe de hacerlo de la siguiente forma.

<code>n1 -eq n2</code>	comprueba si $n1 = n2$
<code>n1 -ge n2</code>	comprueba si $n1 \geq n2$
<code>n1 -gt n2</code>	comprueba si $n1 > n2$
<code>n1 -le n2</code>	comprueba si $n1 \leq n2$
<code>n1 -lt n2</code>	comprueba si $n1 < n2$
<code>n1 -ne n2</code>	comprueba si $n1 \neq n2$

```
GNU nano 6.2      myscript6.sh
#!/bin/bash

num1=10
num2=11

if [ $num1 -ge $num2 ]; then
    echo "$num1 es mayor o igual que $num2"
else
    echo "$num2 es mayor o igual que $num1"
fi
```

```
alulab@minix:~$ chmod a+x myscript6.sh
alulab@minix:~$ ./myscript6.sh
11 es mayor o igual que 10
alulab@minix:~$
```

TAREA

- 1.- Investigue como obtener expresiones compuestas. Usando el complemento, el AND y el OR.
- 2.- Investigue el uso de la sentencia case

Lazos

El lazo while

```
while orden
do
    lista
done
```

```
GNU nano 6.2      myscript7.sh
#!/bin/bash

x=0
while [ $x -lt 10 ]
do
    echo $x
    x=`expr $x + 1`
done
```

```
alulab@minix:~$ chmod a+x myscript7.sh
alulab@minix:~$ ./myscript7.sh
0
1
2
3
4
5
6
7
8
9
alulab@minix:~$
```

```
GNU nano 6.2      myscript8.sh
#!/bin/bash

while read LINE
do
    case $LINE in
        *alulab*) echo $LINE ;;
    esac
done < /etc/passwd
```

```
alulab@minix:~$ chmod a+x myscript8.sh
alulab@minix:~$ ./myscript8.sh
alulab:x:1001:1002:Alumno de Informática,,,:/home/alulab:/bin/bash
alulab@minix:~$
```

El lazo until

```
until orden
do
    lista
done
```

```
GNU nano 6.2      myscript9.sh
#!/bin/bash

x=0
until [ $x -ge 10 ]
do
    echo $x
    x=`expr $x + 1`
done
```

Este programa es equivalente a *myscript7.sh* y producirá la misma salida

El lazo for

```
for var in val1 val2 . . . valN
do
    lista
done
```

```
GNU nano 6.2      myscript10.sh
#!/bin/bash

for FILE in $HOME/.bash*
do
    destino=${FILE##*.}
    cp $FILE $destino.bak
done
```

```
alulab@minix:~$ ls -l .bash*
-rw----- 1 alulab alulab  85 mar 15 14:30 .bash_history
-rw-r--r-- 1 alulab alulab 220 mar 15 12:11 .bash_logout
-rw-r--r-- 1 alulab alulab 3771 mar 15 12:11 .bashrc
alulab@minix:~$ ls -l *.bak
ls: no se puede acceder a '*.bak': No existe el archivo o el directorio
alulab@minix:~$ chmod a+x myscript10.sh
alulab@minix:~$ ./myscript10.sh
alulab@minix:~$ ls -l *.bak
-rw----- 1 alulab alulab  85 mar 15 15:43 bash_history.bak
-rw-r--r-- 1 alulab alulab 220 mar 15 15:43 bash_logout.bak
-rw-r--r-- 1 alulab alulab 3771 mar 15 15:43 bashrc.bak
```

TAREA

- 1.- Investigue como lograr lazos anidados con `while`
- 2.- Investigue como emplear lazos infinitos. Uso de `break` y `continue`

Ejercicio

- 1.- Escriba un *script* que emplee `while` o `until` para mostrar la siguiente salida

```
0
1 0
2 1 0
3 2 1 0
4 3 2 1 0
5 4 3 2 1 0
6 5 4 3 2 1 0
7 6 5 4 3 2 1 0
8 7 6 5 4 3 2 1 0
9 8 7 6 5 4 3 2 1 0
```

Parámetros

Muchas veces nuestros *scripts* van a tener que recibir argumentos. Estos argumentos son recibidos en parámetros que no son más que variables especiales del shell. A continuación una lista de ellas y su descripción

<code>\$0</code>	El nombre del <i>script</i> que está siendo ejecutando
<code>\$n</code>	Estas variables corresponden a los argumentos con los cuales un argumento es invocado. Donde <i>n</i> es un número decimal positivo correspondiente a la posición de un argumento (el primer argumento es <code>\$1</code> , el segundo argumento es <code>\$2</code> , etc).
<code>\$#</code>	El número de argumentos proporcionados al <i>script</i>
<code>\$*</code>	Todos los argumentos son entrecomillados con comillas dobles
<code>@</code>	Todos los argumentos son individualmente entrecomillados con comillas dobles
<code>?</code>	El estado de salida de la última orden
<code>\$\$</code>	El <i>pid</i> del <i>shell</i> actual que se está ejecutando

```
GNU nano 6.2      myscript11.sh
#!/bin/bash

echo "Mi script se llama $0"
echo "Fueron ingresados $# argumentos"

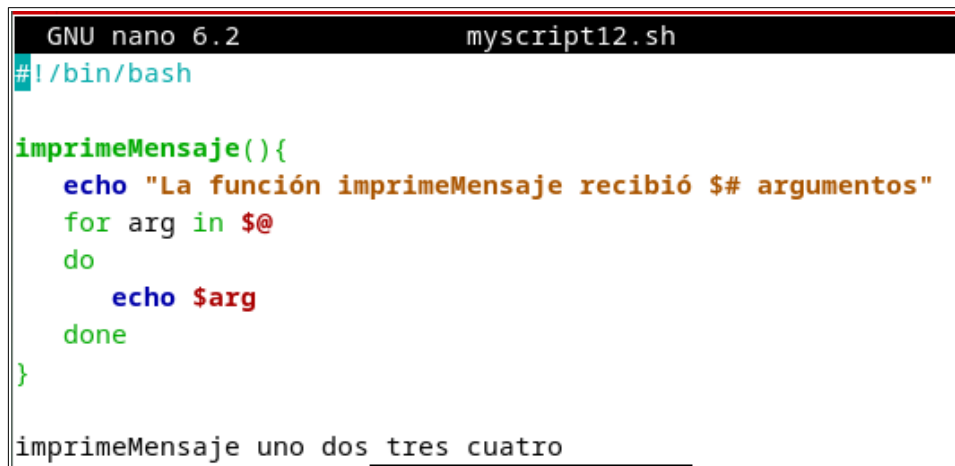
for arg in $*
do
    echo $arg
done
```

```
alulab@minix:~$ chmod a+x myscript11.sh
alulab@minix:~$ ./myscript11.sh uno dos tres
Mi script se llama ./myscript11.sh
Fueron ingresados 3 argumentos
uno
dos
tres
alulab@minix:~$
```

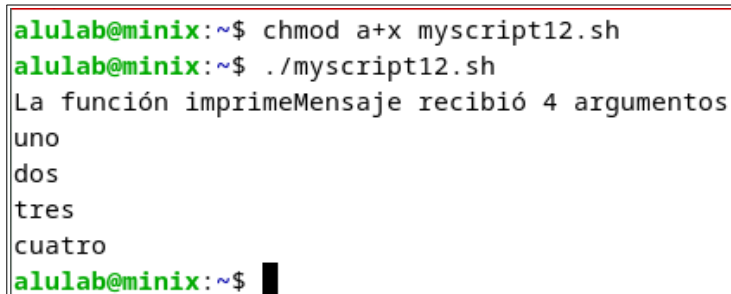

Funciones

Las funciones se definen como sigue

```
name () {  
    lista  
}
```



```
GNU nano 6.2 myscript12.sh  
#!/bin/bash  
  
imprimeMensaje(){  
    echo "La función imprimeMensaje recibió $# argumentos"  
    for arg in $@  
    do  
        echo $arg  
    done  
}  
  
imprimeMensaje uno dos tres cuatro
```



```
alulab@minix:~$ chmod a+x myscript12.sh  
alulab@minix:~$ ./myscript12.sh  
La función imprimeMensaje recibió 4 argumentos  
uno  
dos  
tres  
cuatro  
alulab@minix:~$
```

Ejercicios

- 1) Write a script that backs itself up, that is, copies itself to a file named *backup.sh*
- 2) Write a script that reads each line of a target file, then writes the line back to stdout, but with an extra blank line following. This has the effect of *double-spacing* the file.

Include all necessary code to check whether the script gets the necessary command-line argument (a filename), and whether the specified file exists.

When the script runs correctly, modify it to *triple-space* the target file.

Finally, write a script to remove all blank lines from the target file, *single-spacing* it.
- 3) Print (to stdout) all prime numbers between 60000 and 63000. The output should be nicely formatted in columns

4) The *atoi* function in C converts a string character to an integer. Write a shell script function that performs the same operation. Likewise, write a shell script function that does the inverse, mirroring the C *itoa* function which converts an integer into an ASCII character.

5) Solve a *quadratic* equation of the form $Ax^2 + Bx + C = 0$. Have a script take as arguments the coefficients, A, B, and C, and return the solutions to five decimal places.

6) Find the sum of all five-digit numbers (in the range 10000 - 99999) containing *exactly two* out of the following set of digits: { 4, 5, 6 }. These may repeat within the same number, and if so, they count once for each occurrence.

Some examples of *matching numbers* are 42057, 74638, and 89515.

7) A *lucky number* is one whose individual digits add up to 7, in successive additions. For example, 62431 is a *lucky number* ($6 + 2 + 4 + 3 + 1 = 16$, $1 + 6 = 7$). Find all the *lucky numbers* between 1000 and 10000.