2

PROGRAMACIÓN DE LA SHELL DE UNIX/LINUX

Concepto de guión shell

- Un **guión shell** o *shell script* es un programa escrito con comandos de la shell e instrucciones condicionales, bucles y variables.

- La primera línea de un script debe ser el *tipo de shell* que va a interpretarlo, aunque esto opcional si dicho tipo de shell está definido y en *estado activo* en el entorno del sistema.

Shell	***Comando***
C	!#/bin/csh
Bash	!#/bin/bash
Korn	!#/bin/ksh

Excepto esta primera línea, las que empiezan con el carácter '#' son comentarios.

Concepto de guión shell (II)

- Para interactuar con los shell script se utilizan los comandos:
  - **read :** lee una variable desde el teclado
  - **echo :** escribe texto y valores de variables
- Para averiguar cuál de las shells está por defecto activa se accede a la variable de entorno **SHELL** (más adelante hablaremos de esto) mediante la siguiente orden:
  - echo $SHELL

❑ Concepto de guión shell (III)

- Cuando se inicia la sesión, Linux genera una *shell de usuario*. En esta shell, el usuario puede ejecutar comandos, declarar variables y ejecutar scripts.

- Cuando se ejecuta un script, el sistema vuelve a generar otra shell, denominada **subshell**. *Entonces, en este momento el usuario cuenta con dos shells*.

- *Las variables definidas en una shell son locales a esa shell. Ninguna otra shell podrá hacer referencia a esa variable.*

- Al ejecutar un script, éste se convierte en un *subproceso* de la shell donde se lanzó y, como tal proceso, tiene un único **PID** (*Número de Identificación de Proceso*) que lo identifica.

❑ Concepto de guión shell (III)

- Para averiguar el PID del script en ejecución o de cualquier proceso se utiliza la orden **ps.**

- Todo proceso además tiene un proceso padre (identificado por PPID), excepto aquel proceso identificado con el PID 1.

- Un script puede ejecutarse en primer o en segundo plano:
  - **Primer plano (foreground):** se lanza introduciendo el nombre del script. Hasta que éste no termine no se devolverá el control de la shell al usuario.
  - **Segundo plano (background):** se lanza igual que en foreground pero añadiendo a continuación un *ampersand* (**&**). El control de la shell pasa inmediatamente al usuario. *Es para procesos de* ***larga duración***.

❑ Formas de ejecutar un guión shell

1. Anteponiendo ***sh, source*** o ***bien . (punto)*** al nombre del guión :

```
$> sh nom_script
$> source nom_script
$> . nom_script
```

2. Dando permiso de ejecución al guión y, a continuación, invocándolo con su nombre anteponiendo la ruta donde se encuentra el script:

```
$> chmod +x nom_script
$> ./nom_script     o    $> ruta/nom_script
```

(*) Si el directorio donde se encuentra nom_script está referenciado en PATH, entonces se podría ejecutar introduciendo simplemente su nombre:
$> nom_script.

Script sencillo

- En el siguiente ejemplo, la primera línea nos está indicando que es un programa para la shell *bash*:

```
#!/bin/bash
# ejemplo1. Primer script en Linux
echo Primer script
```

- Después de salvar el código anterior con el nombre *ejemplo1*, este archivo carece de permisos de ejecución. Se los damos.

```
$> chmod  +x  ejemplo1
```

- Ahora lo ejecutamos:

$> ./ejemplo1 (*foreground*) o $> ./ejemplo1& (*background*)

Script algo más complejo

```
#!/bin/bash
# ejemplo2. Script que no termina
while true
do
        echo “No termino” > /dev/null
done
```

Este script entra en un bucle infinito. Genera un mensaje de salida hacia el dispositivo */dev/null*, el cual hace las funciones de sumidero, es decir, todo lo que se escribe ahí se pierde.

- Después de salvarlo y hacerlo ejecutable, podemos lanzarlo en segundo plano mediante la orden ./ejemplo2& y ver su PID y el de la shell que lo generó mediante la orden **ps**.

Componentes en un shell script

- Variables
  - Locales
  - Del entorno
  - Especiales
  - Parámetros o argumentos
- Operadores
  - De comparación aritmética
  - De comparación de cadenas
  - De comparación de archivos
- Sentencias de control
  - Alternativa
    - If, If ... Else, If .. Elif, case
  - Repetitiva
    - While, until, case

Variables

- Al igual que en cualquier otro lenguaje de programación, al definir una variable se le asigna una *zona de memoria* a la que se accede localmente mediante un *nombre*.

- En shell script **no se definen tipos de variables**.

- En shell script las variables se crean al mismo tiempo que son asignadas o leídas desde algún soporte.

- Se asignan a través del símbolo **=**
  -A=1 B=Ana (no hay espacios en blanco alrededor del =)
- Se accede al valor de una variable anteponiéndole el símbolo **$**
  -echo $A $B

Variables (II)

- Hay que tener en cuenta que la mayoría de los metacaracteres (*, $, ?, etc.) son interpretados por el comando *echo*. Para evitar esta situación se utilizan estos otros caracteres:

  - **‘ ‘ (comillas simples): impide que se interprete cualquier carácter**
  - **“ “ (comillas dobles): impide que se interprete cualquier carácter excepto $, “ “, ‘ ‘ y **

  echo $A $B → 1 Ana

  echo ‘\$A $B’ → \$A $B (no se interpreta ni el carácter $ ni \ )

  echo “ ‘$A*’ $B” → ‘1*’ Ana (se interpretan los caracteres ‘ y $)

Ejemplo de variables

- El siguiente ejemplo muestra cómo se asignan variables en un shell script y una forma de presentar resultados.

```
#!/bin/bash
x=2
y=Luis
read z
echo $y regaló $x libros a $z
```

En la instrucción ***read z*** se pide al usuario que introduzca por teclado un valor para z. Se introduce el valor ***Ana***.
A continuación se muestra por pantalla el texto ***Luis regaló 2 libros a Ana***
(*) No puede haber espacios en blanco alrededor del signo =

Variables especiales

- Estas variables informan sobre el estado de un proceso:

  - **$n** : valor del argumento *n* (*n* < 10) en del shell script. $0 representa el nombre del propio shell script.
  - **${n}** : ídem que $n pero con *n* >= 10
  - **$*** : conjunto de todos los argumentos
  - **$#** : número de argumentos introducidos
  - **$?** : código de retorno o error generado por la última orden ejecutada. Vale 0 si la orden se ejecutó con éxito y distinto de 0 si ocurrió algún error.
  - **$$** : Número de identificación del proceso (PID)
  - **$!** : Número del último proceso invocado por la shell (que podrá o no ser este shell script).

Variables de entorno

- Hay dos áreas de memoria para almacenar variables:
  - **Área local de datos**
  - **Entorno**

- Cuando se asigna una variable en una shell, por defecto, es local y, por tanto, es privada a la shell.

- Las variables del entorno están disponibles para las subshells. A este conjunto de variables se accede con la orden **env**.

- Existe un conjunto de variables de entorno ya predefinidas y que se pueden utilizar para averiguar la *ruta de trabajo actual, el nombre de la máquina de trabajo, tipo de shell por defecto, el valor del prompt, la ruta de búsqueda de órdenes, etc.*

Variables de entorno (II)

- La orden **export <nom_variable>** permite que la variable previamente asignada "*nom_variable*" sea accesible desde la propia shell y demás subshells.

- La orden **export** sin parámetros presenta el conjunto de variables de entorno exportadas.

- Una variable exportada *NO* es una variable global, sino una copia de la variable original. Podrá ser modificada en la subshell pero volverá a tener su anterior valor cuando se salga de la subshell. Es el equivalente a la *"llamada por valor".*

- La orden **set** muestra todas las variables del shell en el área de datos local y en el entorno.

Variables de entorno (III)

- Alguna de las variables de entorno más importantes son las siguientes:

-**PATH (*):** camino de búsqueda de órdenes
-**HOME:** directorio de trabajo del usuario
-**USER:** usuario que estableció la sesión
-**PWD:** ruta completa del directorio de trabajo actual
-**LOGNAME:** nombre del usuario que ejecuta la shell
-**PS1, PS2, PS3, PS4:** prompts
-**SHELL:** shell que está ejecutándose
-**TERM:** tipo de terminal.

(*) Esta variable es sumamente importante, ya que aquí están definidos todos los directorios con los ficheros ejecutables. El sistema se encargará de lanzar los ejecutables según el orden aquí definidos.

Ejemplo de variables de entorno

- **Export <variable1> ... <variableN>:** exporta al entorno variables ya asignadas.
  Si no se utilizan parámetros, muestra una relación de las variables exportadas.
- **Env:** muestra la lista de variables del entorno.

```
#!/bin/bash
FICHENT=$HOME/Practica2/alumnos.dat
export FICHENT
...
```

En la primera instrucción se declara la variable ***FICHENT*** que representa el nombre del archivo ***Practica2/alumnos.dat*** ubicado en el directorio de trabajo del usuario representado por la variable de entorno ***HOME***. A continuación se exporta FICHENT con lo que dicho archivo puede ser accesible desde otras subshells.

Ejemplo de argumentos

- Un guión shell soporta argumentos o parámetros de entrada. Éstos se sitúan a la derecha del nombre del guión en la línea de órdenes.
- Se referencian desde el interior del script mediante las variables especiales $i con i=1, 2, ..., NumeroArgumentos.

- NomScript pam1 parm2 ... parmN
  $0 $1 $2 $N

$> ./EjemploParm 1 casa

```
echo "Numero de argumentos = $#“  → 2
echo '$0=' $0   → EjemploParm
echo '$1=' $1  → 1
echo '$2=' $2  → casa
echo ‘$3=‘$3  →
echo '$*=' $*  → 1 casa
```

Operadores Aritméticos

Operador	*Significado*
+	Suma
-	Resta
*	Multiplicación
/	División
%	Resto

OPERADORES

Operadores de comparación aritméticos

Operador	*Significado*
lt (<)	*Anterior*
gt (>)	*Posterior*
le (<=)	*Anterior o igual*
ge (>=)	*Posterior o igual*
eq (==)	*Igualdad*
ne (!=)	*Desigualdad*

Operadores de comparación de cadenas

Operador	*Significado*
s1 = s2	*s1 es igual a s2*
s1 != s2	*s1 no es igual a s2*
s1 < s2 (alfabéticamente)	*s1 es anterior a s2*
s1 > s2 (alfabéticamente)	*s1 es posterior a s2*
-n s1	*s1 no es nulo (contiene al menos uno o más caracteres)*
-z s1	*s1 es nulo*

OPERADORES

Operadores de comparación de archivos

Operador	*Significado*
-a	*Archivo existente*
-s	*Archivo existente y no vacío*
-f	*Archivo ordinario*
-d	*Directorio*
-b	*Dispositivo de bloques*
-c	*Dispositivo de caracteres*
-p	*Tubería*
-l	*Enlace simbólico*
-w	*Archivo con permiso de escritura*
-r	*Archivo con permiso de lectura*
-*x*	*Archivo con permiso de ejecución*
F1 –ef F2	*Los ficheros F1 y F2 son enlaces al mismo archivo*
-N	*El archivo ha sido modificado desde la última lectura*
F1 –nt F2	*F1 es más reciente que F2*
F1 –ot F2	*F1 es más antiguo que F2*

Orden test

Estos tres operadores de comparación necesitan de la orden **test**. Este comando tiene un código de retorno igual a 0 cuando el test es cierto y diferente en caso contrario. Los argumentos de test serán, por tanto, numéricos, cadenas de caracteres o archivos.

El signo **!** (negación) invierte el sentido de la opción.

test $a -ge $b (cierto si *a* es mayor o igual que *b*)

test $cad = Ana (cierto si el valor de *cad* coincide con la cadena *Ana*)

test –z $cad (cierto si *cad* es una cadena vacía)

test –f arch (cierto si *arch* es un archivo ordinario)

test arch1 –nt arch2 (cierto si *arch1* es más reciente que *arch2)*

test ! –d arch (cierto si *arch* no es un directorio).

ENTRECOMILLADO Y EXPANSIÓN

Entrecomillado

En Linux/Unix existen muchos caracteres con significados especiales: $, ?, *, <, >, etc. Si se quiere que estos caracteres dejen de tener este significado especial, se necesita un mecanismo especial denominado **entrecomillado**.

- **Backslash (\ )**: quita el significado especial al carácter que le sigue.
- **Comillas simples (‘*Cadena*’)**: siempre conserva el valor literal de cada uno de los caracteres de la cadena.
- **Comillas dobles (“*Cadena*”)**: conserva el valor de literal de la cadena, excepto para los caracteres dólar ($), comilla simple ('), comilla doble (“), backslash (\) y el acento grave (`)
- **Acentos graves (`Cadena`)**: si Cadena representa una orden entonces se ejecuta dicha orden.

Ejemplo de entrecomillado

A=date B=prisa (A es la cadena “date” pero también podría ser el comando *date*)

echo $A $B → date prisa

echo \$A $B → $A prisa ($ no es interpretado debido al escape \ )

echo ‘$A $B’ → $A $B (los $ no son interpretados)

echo ‘\$A $B’ → \$A $B (ni $ ni \ son interpretados)

echo “\$A $B” → \$A prisa (\ no es interpretado y $ sí)

echo “ ‘$A’ $B” → ‘date’ prisa (interpreta comillas simples)

echo ‘$A’ $B → $A prisa (comillas simples no interpretan $)

echo “\”$A’ $B” → “date’ prisa (se interpreta el carácter \”)

echo `$A` → sáb feb 10 14:04:49 CET 2007 (ejecuta el comando date)

AGRUPAMIENTO DE ÓRDENES

La shell de Unix/Linux permite que el usuario introduzca varias órdenes en una línea. Esta posibilidad va a permitir que el usuario pueda expresar operaciones realmente complejas en una única línea.

- **Orden &:** ejecuta la orden en 2º plano o background.

- **Ord1 | Ord2:** tubería. Redirige la salida estándar de la primera orden a la entrada estándar de la segunda.

```
ls –l  |  tr –s ‘ ‘ |  cut –f1,5,6 –d’ ‘ |  tr ‘ ‘  ‘\t’
```

- **Ord1 ; Ord2:** ejecuta ambas órdenes en la misma instrucción.

```
date ;  sleep 5 ;  date
```

AGRUPAMIENTO DE ÓRDENES

- **Ord1 && Ord2:** ejecuta ambas órdenes si la primera lo hace con éxito. En caso contrario no se ejecuta la segunda orden.

  mv && ls (falla la orden *mv* y, por tanto, *ls* no se ejecuta)

  mv arch1 arch1.bak && ls (se ejecutan ambas)

- **Ord1 || Ord2:** ejecuta la segunda orden si falla la primera

  mv || ls (falla la orden *mv* pero *ls* sí se ejecuta)

  mv arch1 arch1.bak || ls (se ejecuta solo la primera)

- **Linea1 **

  Linea2: permite ejecutar órdenes que ocupan más de una línea en el script.

  echo "Este comando es demasiado largo y no cabe en \
  una sola línea"

AGRUPAMIENTO DE ÓRDENES

- **(Lista):** ejecuta una lista de órdenes generando una subshell.

  (sleep 1800; test -r arch && head -100 arch)&

F	S	UID	PID	PPID	C	PRI	NI	ADDR	SZ	WCHAN	TTY	TIME	CMD	
0	**S**	**1000**	**3390**	**3389**	**0**	**76**	**0**	**-**	**909**	**wait**	**pts/1**	**00:00:00**	**bash**	**shell inicial**
1	**S**	**1000**	**3559**	**3390**	**0**	**76**	**0**	**-**	**909**	**wait**	**pts/1**	**00:00:00**	**bash**	**subshell script**
1	**S**	**1000**	**3560**	**3559**	**0**	**76**	**0**	**-**	**909**	**wait**	**pts/1**	**00:00:00**	**bash**	**subshell de ( )**
0	S	1000	3562	3560	0	76	0	-	471	322413	pts/1	00:00:00	sleep	
0	R	1000	3564	3390	0	76	0	-	544	-	pts/1	00:00:00	ps	

- **{ Lista; }:** ejecuta una lista de órdenes dentro del shell

  {sleep 1800; test -r arch && head -100 arch; }&

F	S	UID	PID	PPID	C	PRI	NI	ADDR	SZ	WCHAN	TTY	TIME	CMD	
0	**S**	**1000**	**3390**	**3389**	**0**	**76**	**0**	**-**	**909**	**wait**	**pts/1**	**00:00:00**	**bash**	**shell inicial**
1	**S**	**1000**	**3559**	**3390**	**0**	**76**	**0**	**-**	**909**	**wait**	**pts/1**	**00:00:00**	**bash**	**subshell script**
0	S	1000	3562	3560	0	76	0	-	471	322413	pts/1	00:00:00	sleep	
0	R	1000	3564	3390	0	76	0	-	544	-	pts/1	00:00:00	ps	

El comando { } no genera subshell

Orden de precedencia

En el caso de la misma precedencia, el análisis de la línea se hace de izquierda a derecha. El orden de precedencia, de mayor a menor, es el siguiente:

- Listas con tuberías o cauces
- Listas Y y O (misma preferencia)
- Listas asíncronas y secuenciales (misma preferencia)

❑ Instrucción if

Su sintaxis es la siguiente:

```
if  [ condición ] ;
then
        # Código que se ejecuta
        # si condición es verdad
            comando1
            comando2
            ...
fi
```

❑ Instrucción if compuesta

Su sintaxis es la siguiente:

```
if  [ condición ] ;
then
        comando1
        comando2
        ...
else
        comando1
        comando2
        ...
fi
```

Ejemplo

```
#!/bin/bash
T1="cierto"
T2="falso"
if  [ "$T1" = "$T2" ] ;  then
        echo expresión evaluada como cierta
else
        echo expresión evaluada como falsa
fi
```

Instrucción if anidada

Su sintaxis es cualquiera de las siguientes:

```
if  [ condición1 ] ;  then
      comando1; comando2; …
else if  [ condición2 ] ;  then
      comando3; comando4; …
else if  [ condición3 ] ;  then
      comando5; comando6; …
else
      comando7
fi
fi
fi
```

```
if  [ condición1 ] ;  then
      comando1; comando2; …
elif  [ condición2 ] ;  then
      comando3; comando4; …
elif  [ condición3 ] ;  then
      comando5; comando6; …
else
      comando7
fi
```

Instrucción case

Su sintaxis es la siguiente:

```
case var | expresión  in
        patrón1) lista_de_comandos ;;
        patrón2) lista_de_comandos ;;
                ...
        *)   lista_de_comandos ;;
esac
```

Equivale a una instrucción *if* múltiple, con la salvedad de que sólo se ejecuta uno de los casos posibles. *) representa el patrón por defecto.

Ejemplo instrucción case

```
case tipocarnet  in

        A*)         echo Motocicletas ;;
        B[12] )     echo Turismos ;;
        C1 | C2)    echo Camiones ;;
        D)          echo Autobuses ;;
        E)          echo Remolques ;;
        *)          echo Categoría desconocida ;;
esac
```

(*) Nótese que los patrones admiten metacaracteres.

Bucles

- Un bucle es una estructura que repite un grupo de instrucciones cierto número de veces, o hasta que se cumple una condición.

- Bucle for

Su sintaxis es la siguiente:

```
for  variable in lista
do
        comando1
        comando2
        …
done
```

- Ejemplo

```
#!/bin/bash
for fichero in  *.c
      echo “$fichero\n”
done
```

Escribe todos los archivos del directorio actual con extensión .c

❑ Bucles (II)

- Bucle while

```
while [ condición ]
    do
        comando1
        …
    done
```

```
#!/bin/bash
cont=0
while [ $cont -lt 10 ]
 do
      echo El contador es $cont
      cont = `expr $cont + 1`
 done
```

- Bucle until

```
until  [ condición ]
    do
        comando1
        …
    done
```

```
#!/bin/bash
cont=20
until [ $ cont -lt 10 ]
 do
      echo contador $cont
      cont = `expr $cont - 1`
 done
```

Funciones

Las funciones se utilizan para agrupar cierta parte del código que tiene una funcionalidad especial. Se declaran con la palabra reservada ***function***. Pueden ser llamadas desde cualquier punto del programa o desde otra función.
Pueden tener una cláusula **return** para devolver valores.

-Sintaxis: [function] nombre_funcion () {codigo_funcion }

- Para llamarlas, simplemente se escribe su nombre seguido opcionalmente de uno o más parámetros:

$> nombre_funcion parm1 ... parmN

Ejemplo de funciones

- Creamos un archivo denominado *misfunciones* con el siguiente contenido:

```
function  saludohola()
{
        echo Hola que tal  $1
}
function  saludoadios()
{
        echo Hasta luego  $1
}
```

- Ejecutamos el script: $> . *misfunciones*
- Podríamos verificar que son accesibles: $> set | grep saludo
- Ahora ya tenemos disponibles ambas funciones:

  $> saludohola Ana → Hola que tal Ana

  $> saludoadios Jose → Hasta luego Jose

❑ Tipo de shell a ejecutar

Ya hemos mencionado previamente que para que la shell interprete correctamente un script, la primera línea del mismo debe contener una instrucción que lo indique : #!/bin/tipo_shell

Esta primera línea se puede omitir si tenemos definida en nuestra sesión una *shell por defecto*. Existen una serie de variables de entorno que nos permiten averiguar la shell por defecto en la sesión e incluso establecerla.

- **SHELL:** variable de entorno con la shell por defecto en la sesión
- **BASH_VERSION:** variable de entorno con la versión bash
- **cat /etc/shells:** orden que nos muestra las shells definidas en nuestro sistema
- **chsh –s /bin/bash:** establece a *bash* como la shell por defecto.

❑ Alias

Los alias se emplean para invocar órdenes con un nombre diferente al utilizado normalmente. Representan un sinónimo del nombre de la orden.

Por ejemplo, puede ser interesante para un usuario acostumbrado a trabajar con DOS, utilizar cd.. y no cd .. (apreciar el espacio en blanco).

- **alias cd..="cd .."**
- **alias dir="ls –l"**

El alias permanece activo hasta que finalice la sesión o hasta que se emplee la orden **unalias**. Si queremos que se mantenga, deberemos configurarlo así en un archivo de inicialización.

❑ Archivos de configuración

Existen una serie de archivos de inicialización en el directorio de trabajo del usuario donde se pueden establecer configuraciones de variables del sistema, de entorno y alias y funciones.

- **.bash_profile:** es leído cada vez que el usuario entra en el sistema. Cualquier cambio realizado en este archivo no tendrá efecto hasta que se reinicie el sistema o bien cuando se ejecute el comando ***source .bash_profile.***

- **$HOME/.bashrc:** es leído cuando el usuario arranca una subshell. Se puede activar con el comando ***source .bashrc.***

- **.bash_logout**: es leído cuando nos salimos del sistema, se puede utilizar, por ejemplo, para borrar los archivos temporales generados en la última sesión.

Archivos de configuración (II)

Primero se ejecuta el archivo .bash_profile y, posteriormente, .bashrc

- # Inicio de ~/.bashrc.
  # Variables de entorno del sistema y programas de inicio # están en /etc/profile. Alias del sistema y funciones están en # /etc/bashrc.

  ```
  alias ll="ls -l --color"
  alias lal="ls -la --color"

  if [ -f "/etc/bashrc" ] ; then
          source /etc/bashrc
  fi
  # Fin de ~/.bashrc
  ```

Archivos de configuración (III)

```
# Inicio de ~/.bash_profile.
# Variables de entorno personales y programas de inicio
# Alias personales y funciones deberían estar en ~/.bashrc.
Variables  # de entorno del sistema y programas de inicio
deberían estar en          #  /etc/profile. Alias del sistema y
funciones están en /etc/bashrc.
if [ -f  "$HOME/.bashrc" ] ;  then
            source $HOME/.bashrc
fi
if [ -d  "$HOME/bin" ] ;  then
            append $HOME/bin
fi
unset append
# Fin de  ~/.bash_profile
```

EJEMPLOS

- Muestra el contenido de /etc/hosts si es que éste existe

```
if  test –f  /etc/hosts
then
     cat /etc/hosts
else
     echo Archivo no existente
fi
```

- Muestra el día de la semana

```
dia=`date | cut –c0-3`
case $dia in
    lun) echo Hoy es Lunes;;
    mar) echo Hoy es Marte;;
    …
   dom) echo Hoy es Domingo;;
esac
```

EJEMPLOS

- Realiza una copia de seguridad de un grupo de archivos

```
for ARCH  in  pr1.c  pr2.c  h1.h  lista.txt
do
      test  -f  $ARCH  &&  cp  $ARCH  $ARCH.bak
done
```

- Imprime un fichero si éste es un archivo ordinario

```
test  -f  archivo  &&  lpr archivo
```

- Script que procesa un archivo de entrada, generando otro de salida y a continuación ordena este último

```
(Script1  entrada  salida ;  sort  salida )&
```

EJEMPLOS

- Comprueba si existe el directorio Practicas y si no existe lo crea

```
test –d  $HOME/Practicas
if [ $? = 1 ]
then
      mkdir Practicas
fi
```

- Función que simula una barra de progreso al ejecutarse un programa

```
function puntos ()
{
   if  [ $1 ]    #  Nombre de progama a ejecutar introducido
   then
         INTERVALO=$1  # Considerar el intervalo como tal
    else
         INTERVALO=5    # Si no se ha pasado, considerar 5
    fi
   while true do
        echo ".\c" ;  sleep  $INTERVALO
    done
 }
```

RESUMEN COMANDOS

- **alias** : muestra todos los alias activados.
- **echo** : muestra texto y valores de las variables.
- **env** : informa de los valores y de las variables de entorno.
- **exit** : detiene la ejecución del programa del shell.
  exit [n] : asigna el valor *n* al código de retorno.
- **export** : informa de los nombres y valores de las variables exportadas en el shell actual.
  Export variable: traslada "variable" al entorno.
- **expr** : permite efectuar operaciones de aritmética simple.
  expr `arg1 op arg2 [op arg3 …]`.

RESUMEN COMANDOS

- jobs : muestra los trabajos ejecutándose.
- kill [-señal] PID [PID ...] : envía señales a uno o varios procesos identificador por su PID.
- ps -[opciones] : informa sobre los procesos que en esos momentos están ejecutándose en el sistema.
  - -e informa de todo los procesos que hay en el sistema
  - -f proporciona una lista completa de cada proceso
  - -l muestra un lista en formato largo .
- read : lee información de forma interactiva desde el terminal.

  Si hay más variables que palabras escritas, las variables que sobran por la derecha se asignarán a NULL.

  Si hay más palabras escritas que variables, todos los datos que sobran se asignarán a la última variable de la lista.

RESUMEN COMANDOS

- **set** : informa de los nombres y valores de las variables del shell en el área local de datos y en el entorno.

- **test** : sirve para evaluar expresiones. Genera un valor de retorno 0 si la expresión se evalúa como verdad y 1 en caso contrario.

- **unset** : borra el valor de todas las variables del área local de datos.

  unset variable: elimina el valor de “variable”.

- **unalias** : desactiva un alias.