1 Programación scripts shell.

1.1 Nuestro primer script.

Los scripts no son más que ficheros de texto ASCII puro, que pueden ser creados con cualquier editor del que dispongamos (vi, nano, gedit, emacs, etc.). Cread un fichero de texto con nombre primero.sh, con el siguiente contenido:

```
#!/bin/bash
echo "Hola Mundo"
```

La primera línea sirve para indicar que shell utilizamos (en nuestro caso bash) y donde puede ser encontrado en nuestro sistema (para saberlo, podemos hacer locate bash). La segunda línea de nuestro script, simplemente utiliza el comando para escribir en pantalla (echo) y escribe la línea Hola Mundo.

Una vez creado el fichero, debemos darle permisos de ejecución, mediante el comando chmod a+x primero.sh, posteriormente para ejecutarlo debemos llamarlo como ./permiso.sh (el punto barra es para indicarle que lo busque en el directorio actual, ya que dicho directorio no estará seguramente incluido en el PATH del sistema).

Si queremos ejecutar un script para comprobar como funciona sin hacerlo ejecutable, podemos hacerlo mediante el comando source primero.sh que permite lanzar un script no ejecutable. La orden source también puede ejecutarse simplemente escribiendo un punto (. primero.sh).

Sin embargo, la orden source solo debe usarse para comprobar el script, una vez que tengamos el script completo y depurado debemos darle sus permisos de ejecución correspondientes.

Las comillas dobles que hemos usado para escribir Hola Mundo no son necesarias, y podeis comprobar como quitandolas el proceso se ejecuta exactamente igual. Sin embargo, es una buena práctica encerrar siempre los textos entre comillas dobles, y en caso de que contengan caracteres especiales (como el *, el $, etc.), es mejor usar comillas simples, que son más potentes que las comillas dobles. Probad lo siguiente:

```
echo esto es un asterisco * sin comillas
echo esto es un dólar y tres letras $ABC sin comillas
echo "esto es un asterisco * entre comillas dobles"
echo 'esto es un asterisco * entre comillas simples'
echo "esto es un dólar y tres letras $ABC entre comillas dobles"
echo 'esto es un dólar y tres letras $ABC entre comillas simples'
```

Si tenemos que ejecutar varias líneas y queremos escribirlas en una sola, podemos hacerlo usando el símbolo punto y coma para indicar que lo siguiente es otra línea, aunque este en la misma:

```
echo Hola ; pwd ; echo Adios  # esto son tres líneas escrita en una sola.
```

También podemos hacer lo contrario, escribir una sola línea en varias. Para ello usamos el carácter contrabarra cuando queramos que nuestra línea se "rompa" y continué en la línea de abajo.

```
echo "Esto es un ejemplo \
de una línea escrita realmen\
te en tres" # esto es una línea escrita en tres.
```

En este ultimo ejemplo he aprovechado para mostraros como se pueden usar comentarios en los scripts. Basta con usar el símbolo almohadilla (#) donde queramos, todo lo que quede a la derecha de dicho símbolo es un comentario. Si usamos # como primer carácter de una línea, toda la línea es de comentario.

1.2 Uso de variables en los scripts. Expansiones.

Las variables de los shell scripts son muy simples, ya que no tienen tipo definido ni necesitan ser declaradas antes de poder ser usadas. Para introducir valor en una variable simplemente se usa su nombre, y para obtener el valor de una variable se le antepone un símbolo dólar.

```
#!/bin/bash
DECIR="Hola Mundo"
echo $DECIR
```

Este script realiza exactamente la misma función que el anterior, pero usando una variable.

Cualquier valor introducido en una variable se considera alfanumérico, asi que si realizamos lo siguiente:

```
NUMERO=4    # No se debe dejar ningún espacio en la asignación.
echo NUMERO+3
```

obtendremos por pantalla la cadena de caracteres 4+3.

En Linux podemos usar varias expansiones en las líneas de comandos, que son especialmente útiles en los scripts. La primera expansión consiste en usar $( ). Esta expansión permite ejecutar lo que se encuentre entre los paréntesis, y devuelve su salida.

```
echo pwd    # escribe por pantalla la palabra pwd
echo $(pwd) # ejecuta la orden pwd, y escribe por pantalla su resultado.
```

Así, por ejemplo, la siguiente instrucción copia el fichero /etc/network/interfaces en el directorio actual con el nombre red190306.conf (suponiendo que estamos en la fecha 19 de Marzo de 2006.

```
NOMBRE_FICHERO="red"$(date +%d%m%y)".conf"
cp /etc/network/interfaces $NOMBRE_FICHERO
```

Como es lógico, es perfectamente posible no usar variables en el ejemplo anterior y hacerlo todo en una línea, pero es una buena practica no complicar excesivamente cada una de las líneas del script, ya que nos permitirá una modificación mucho más simple y la depuración en caso de que existan errores suele ser bastante más rápida, al menos mientras nuestro nivel de programación no sea bastante alto.

El efecto conseguido con $(orden) se puede conseguir también usando la tilde invertida `orden`.

Otra expansión que podemos usar es $(( )) (símbolo dólar pero con dos paréntesis). Los dobles paréntesis podemos sustituirlos si queremos por corchetes. Esta expansión va a tratar como una expresión aritmética lo que este incluido entre los paréntesis, va a evaluarla y devolvernos su valor.

```
NUMERO=4
```

```
echo $(($NUMERO+3))     # sería lo mismo poner echo $[$NUMERO+3]
```

obtenemos en pantalla el valor 7.

Aprovecho para explicar el comando let, que nos permite realizar operaciones aritmeticas como la anterior, pero sin tener que usar expansiones ni dolares para las variables.

```
NUMERO=4
let SUMA=NUMERO+3
echo $SUMA
```

obtenemos el mismo valor 7, y como vemos no hemos usado ni dólar, ni paréntesis.

Los operadores aritméticos que podemos usar para realizar operaciones son:

Operadores aritméticos	
+	Suma
-	Resta
*	Multiplicación
/	División
%	Modulo (resto)

1.3 Funciones.

Usar funciones en los scripts es muy simple. Basta con usar la siguiente estructura al principio del script:

```
function nombre_función {
      líneas de la función
}
```

Estas líneas de la función no se ejecutarán al procesar el script, sino que solo se ejecutarán cuando en el cuerpo del script usemos el nombre_funcion. Ejemplo:

```
#!/bin/bash
function doble {
      echo "voy a doblar el valor de numero"
      let NUMERO=NUMERO*2
}
$NUMERO=3
echo ' $NUMERO vale : ' $NUMERO     # comillas simples
doble                               # llamamos a la función
echo ' $NUMERO vale : ' $NUMERO
```

Podría parecer en una lectura rápida del script anterior, que estamos pasando por `let NUMERO=NUMERO*2` antes de asignarle el valor 3. No es cierto, ya que aunque veamos esas líneas físicamente anteriores a la asignación, solo serán procesadas cuando en el script escribimos doble.

Por defectos, todas las variables que usemos son globales, es decir, que las funciones y el script las comparten, pueden modificar sus valores, leer las modificaciones realizadas, etc. Sin embargo, en determinadas ocasiones nos puede interesar que las variables sean locales a la función, es decir, que si la función modifica su valor el script no se entera…

```
#!/bin/bash
function saludo {
      NOMBRE=”Jose Antonio”
      echo “Hola señor $NOMBRE encantado de conocerle”
}
NOMBRE=”Juana”
saludo
echo “En el script principal, mi nombre es $NOMBRE”
```

En este ejemplo, vemos como nos aparece `En el script principal, mi nombre es Jose Antonio`, ya que cuando en la función se modifica NOMBRE, se modifica en todo el ámbito del programa.

```
#!/bin/bash
function saludo {
      local NOMBRE=”Jose Antonio”
      echo “Hola señor $NOMBRE encantado de conocerle”
}
NOMBRE=”Juana”
saludo
echo “En el script principal, mi nombre es $NOMBRE”
```

Vemos como ahora, al anteponer local a la variable NOMBRE en la función, las modificaciones que se realicen sólo afectan a la propia función, por lo que en pantalla vemos como aparece `En el script principal, mi nombre es Juana`.

1.4 Estructuras condicionales

if

La principal estructura condicional de los scripts en shell es el if:

```
if [ expresión ]; then
      realizar si expresión es verdadera
fi
```

La expresión es cualquier expresión lógica que produzca un resultado verdadero o falso. Si estamos operando con cadenas alfanuméricas, los operadores que podemos utilizar son los siguientes:

Operadores de comparación de cadenas alfanuméricas	
Cadena1 = Cadena2	Verdadero si Cadena1 o Variable1 es IGUAL a Cadena2 o Variable2
Cadena1 **!=** Cadena2	Verdadero si Cadena1 o Variable1 NO es IGUAL a Cadena2 o Variable2
Cadena1 **<** Cadena2	Verdadero si Cadena1 o Variable1 es MENOR a Cadena2 o Variable2
Cadena1 **>** Cadena2	Verdadero si Cadena1 o Variable1 es MAYOR a Cadena2 o Variable2
-n Variable1	Verdadero si Cadena1 o Variable1 NO ES NULO (tiene algún valor)
-z Variable1	Verdadero si Cadena1 o Variable1 ES NULO (esta vacía o no definida)

Los anteriores operadores sólo son validos para comparar cadenas, si queremos comparar valores numéricos, hemos de usar los siguientes:

Operadores de comparación de valores numéricos.	
Numero1 **-eq** Numero2	Verdadero si Numero1 o Variable1 es IGUAL a Numero2 o Variable2
Numero1 **-ne** Numero2	Verdadero si Numero1 o Variable1 NO es IGUAL a Numero2 o Variable2
Numero1 **-lt** Numero2	Verdadero si Numero1 o Variable1 es MENOR a Numero2 o Variable2
Numero1 **-gt** Numero2	Verdadero si Numero1 o Variable1 es MAYOR a Numero2 o Variable2
Numero1 **-le** Numero2	Ver. si Numero1 o Variable1 es MENOR O IGUAL a Numero2 o Variable2
Numero1 **-ge** Numero2	Ver. si Numero1 o Variable1 es MAYOR O IGUAL a Numero2 o Variable2

Veamos un ejemplo de una estructura if, tanto con valores de cadena como con valores numéricos.

```
NOMBRE="Jose Antonio"
if [ $NOMBRE = "Jose Antonio" ]; then
      echo Bienvenido Jose Antonio
fi

NUMERO=12
if [ $NUMERO -eq 12 ]; then
      echo Efectivamente, el número es 12
fi
```

Si usamos operadores de comparación numéricos con valores de cadena, el sistema nos dará un error parecido al siguiente: "integer expected"

La estructura if podemos ampliarla usando la construcción else (en caso contrario) y elif (en caso contrario si…).

La estructura simple de else es la siguiente:

```
if [ expresión 1 ]; then
      realizar si expresión 1 es verdadera
else
      realizar si expresión 1 es falsa
fi
```

Una estructura con elif (else if) tiene la siguiente forma:

```
if [ expresión1 ]; then
      realizar si expresión1 es verdadera
elif [ expresión2 ]; then
      realizar si expresión1 es falsa, pero es verdadera expresión2
elif [ expresión3 ]; then
      realizar si exp1 y exp2 son falsas, pero es verdadera expresión3
else
      realizar si todas las expresiones anteriores son falsas
fi
```

Hay que tener muchísimo cuidado con los espacios en blanco, y seguramente durante nuestros primeros scripts casi todos los errores vendrán por haberlos usado mal en las estructuras if. Hay que recordar que los corchetes llevan espacios en blanco tanto a izquierda como derecha, que el punto y coma sin embargo va pegado al corchete cerrado, y que SIEMPRE hay que poner espacios en blanco las expresiones.

Veamos algunos errores muy comunes, para que no los cometáis.

`if [3 -eq 5 ]; then`	`bash: [3: command not found.` `Hemos usado [3 en lugar de [ 3`
`if [ "Jose" -eq "Jose ]; then`	`Bash: [: jose: integer expression expected.` `Debíamos haber usado =`
`if [ 3 = 4 ]; then`	`Esto no nos devolverá error, y parece que funciona, pero en realidad no es asi, hay que usar -eq`
`if [ 3 > 4 ]; then`	`Esto devuelve verdadero. Sirva como prueba que no hay que usar operadores de cadena para comparar números.`
`If [ jose=Antonio ]; then`	`No hemos dejado espacios en la condición =. Esta expresión da como valor verdadero. Mucho cuidado con este error, que nos puede volver locos en depuración.`

Otro error muy común es el siguiente:

```
#!/bin/bash
PROFESOR="Juana"
if [ $PROFSOR = "Juana" ]; then
      echo "Hola Juana"
fi
```

Este programa nos devuelve por pantalla el siguiente error:

```
bash: [: unary operador expected
```

Que traducido resulta, me he encontrado un [(corchete abierto) y luego un operador (el =) sin nada en medio, y eso no funciona.

Revisando el programa anterior, vemos como nos hemos equivocado en el nombre de la variable, por lo cual $PROFSOR no tiene ningún valor (es nula) y por lo tanto al no valer nada, el programa lo que ve es lo siguiente: `if [ = "Juana" ].`

Hemos visto operadores aritméticos y operadores para cadena, pero en las expresiones podemos utilizar cualquier operación que nos devuelva un valor lógico (0 para verdadero). Por ejemplo, podemos usar la función test del bash, que funciona de la siguiente forma:

Operaciones condicionales usando test.	
-a fichero	Verdadero si fichero existe
-d fichero	Verdadero si fichero existe, y es un fichero de tipo directorio
-f fichero	Verdadero si fichero existe, y es un fichero regular.
-r fichero	Verdadero si fichero existe y se puede leer
-w fichero	Verdadero si fichero existe y se puede escribir
-x fichero	Verdadero si fichero existe y se puede ejecutar
fichero1 -nt fichero2	Verdadero si fichero1 es más nuevo que fichero2
fichero1 -ot fichero2	Verdadero si fichero1 es más viejo que fichero2

Si lo necesitamos, podemos anidar expresiones usando tanto and (y, &&) como or (o, ||).

```
if [ expresión1 ] && [ expresión2 ]; then
      se ejecuta si expresión1 Y expresión2 son verdaderas
fi

if [ expresión1 ] || [ expresión2 ]; then
      se ejecuta si expresión1 O expresión2 son verdaderas
fi
```

También podemos usar el operador not (!) para indicar una negación.

```
if ! [ expresión1 ]; then
      se ejecuta si expresión1 NO es verdadera
fi
```

Para hacer algunos ejercicios, vamos a aprovechar para explicar como le podemos pedir datos al usuario. Se hace con la orden read y es muy simple de usar:

```
read -p "texto de la pregunta" variable
```

La ejecución del script se parará, mostrará por pantalla el texto de la pregunta, y dejará que el usuario escriba la respuesta, cuando pulse INTRO la respuesta dada se introducirá como valor de variable.

read también puede ser usada sin el parámetro -p, de la forma read variable. También podemos hacer que lea un determinado número de caracteres, sin obligar a que el usuario pulse intro, con el parámetro -n número_de_caracteres. El parámetro -s silencia el eco (no se ve por pantalla lo que el usuario escribe).

Ahora que sabemos usar el read, hagamos por ejemplo un programa que nos permita indicar si un número introducido es par o impar.

```
#!/bin/bash
# parimpar.sh - script que nos pide un número e indica si es par o impar.
clear
read -p "Introduzca un número : " NUMERO
let RESTO=NUMERO%2
if [ $RESTO -eq 0 ]; then
      echo "El número $NUMERO es par"
else
      echo "El número $NUMERO es impar"
fi
```

Hagamos un script un poco más complicado… vamos a pedir al usuario un número de 3 cifras y vamos a indicar si es capicúa.

```
#!/bin/bash
# capicua.sh - script que nos pide un número de tres cifras e indica si es capicúa o no.
clear
read -n 3 -p  "Dame un número entre 100 y 999 (no pulses INTRO) : " NUMERO
echo  # este echo sirve para introducir un retorno de linea
if [ $NUMERO -lt 100 ]; then
      echo "Lo siento, has introducido un número menor de 100"
else
      PRIMERA_CIFRA=$(echo $NUMERO | cut -c 1)
      TERCERA_CIFRA=$(echo $NUMERO | cut -c 3)
      if [ $PRIMERA_CIFRA = $TERCERA_CIFRA ]; then
            echo "El número $NUMERO es capicúa."
      else
            echo "El número $NUMERO ni es capicúa ni ná".
      fi
fi
```

Es evidente que podíamos haber hecho este último script mucho más corto, por ejemplo usando una línea como:

```
if [ $(echo $NUMERO | cut -c 1) = $(echo $NUMERO | cut -c 3) ]; then
```

Pero eso ya queda al gusto de cada programador. A mi personalmente me gustan los programas que pueden entenderse simplemente echándole una ojeada al fuente, y lo aconsejo fuertemente al menos hasta que tengáis un nivel de programación muy alto. (Y aun entonces, facilita mucho la modificación posterior de los programas).

Cuando hacemos un script de varias líneas como el anterior, es posible que cometamos algún fallo. Una opción que podemos usar para depurar los scripts y encontrar rápidamente los errores, es añadir un -x en la llamada al bash de la primera línea. Esto hará que cada línea antes de ejecutarse sea mostrada por pantalla tal y como la esta interpretando el bash.

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

Para que esto funcione, es necesario que hagamos el script ejecutable, no es valido si lanzamos el script con la orden source o con el punto. Hay que ejecutar el script con ./script.

case

Hemos visto la principal estructura condicional que es el if, pero tenemos alguna otra a nuestra disposición, como el case.

```
case VARIABLE in
      valor1)
            Se ejecuta si VARIABLE tiene el valor1
            ;;
      valor2)
            Se ejecuta si VARIABLE tiene el valor2
            ;;
      *)
            Se ejecuta si VARIABLE no tiene ninguno de los valores anteriores
            ;;
esac
```

Veámoslo mediante otro ejemplo:

```
#!/bin/bash
# horóscopo.sh - script que nos pide el año de nacimiento y nos dice cual es el animal que nos
# corresponde en el horóscopo chino.
clear
read -p "En que año naciste (usa 4 cifras) : " ANO
let RESTO=ANO%12
case $RESTO in
```

```
      0) AN="el MONO" ;;
      1) AN="el GALLO" ;;
      2) AN="el PERRO" ;;
      3) AN="el CERDO" ;;
      4) AN="la RATA" ;;
      5) AN="el BUEY" ;;
      6) AN="el TIGRE" ;;
      7) AN="el CONEJO" ;;
      8) AN="el DRAGON" ;;
      9) AN="la SERPIENTE" ;;
      10) AN="el CABALLO" ;;
      11) AN="la CABRA" ;;
      *) echo "Imposible, el resto de dividir entre 12 va de 0 a 11 siempre.
esac
echo "Si naciste en $ANO según el horóscopo chino te corresponde" $AN
```

1.5 Estructuras iterativas o Bucles.

Las principales estructuras iterativas que podemos usar en shell scripts son for, while, until, y select.

for

La estructura for es la siguiente:

```
for variable in conjunto; do
      estas líneas se repiten una vez por cada elemento del conjunto,
      y variable va tomando los valores del conjunto
done
```

Ese conjunto que aparece en la estructura del for, es normalmente un conjunto de valores cualesquiera, separados por espacios en blanco o retornos de linea. Así, si queremos mostrar los días de la semana por pantalla este script lo haría:

```
#!/bin/bash
for dia in lunes martes miércoles jueves viernes sabado domingo; do
      echo el dia de la semana procesado es $dia
done
```

La "gracia" de ese conjunto de valores, es que podemos obtenerlo mediante la ejecución de ordenes que nos devuelvan una salida de ese tipo. Por ejemplo, si ejecutamos la orden

```
find ~ -iname "*sh" 2> /dev/null
```

Obtendremos una lista de todos los ficheros que terminen en sh (normalmente los scripts si es que decidimos usar esa extensión) que están dentro de nuestro directorio de usuario (~) y enviando los posibles mensajes de error de la orden a /dev/null (para no verlos por pantalla). Bien, pues la salida de esta orden es un conjunto que podemos procesar con un for:

```
#!/bin/bash
for programa in $( find ~ -iname "*sh" 2> /dev/null ); do
      echo "Uno de mis scripts :" $programa
done
```

Imaginemos que queremos copiar a un llaverito USB (montado en /media/usbdisk por ejemplo) todos los scripts que tengamos en nuestro directorio home, sin importar en que directorio estén, podríamos hacerlo fácilmente con este script:

```
#!/bin/bash
for programa in $( find ~ -iname "*sh" 2> /dev/null ); do
      echo "copiando el script :" $programa
      cp $programa /media/usbdisk
done
```

Ya que estamos, mejoremos el script anterior para que cree un directorio scripts en nuestro llaverito, pero únicamente si no existe.

```
#!/bin/bash
if ! [ -d /media/usbdisk/scripts ]; then
      mkdir /media/usbdisk/scripts
fi
for programa in $( find ~ -iname "*sh" 2> /dev/null ); do
      echo "copiando el script :" $programa
      cp $programa /media/usbdisk
done
```

while y until

Cuando no queremos recorrer un conjunto de valores, sino repetir algo mientras se cumpla una condición, o hasta que se cumpla una condición, podemos usar las estructuras while y until.

La estructura del while es la siguiente:

```
while [ expresión ]; do
      estas líneas se repiten MIENTRAS la expresión sea verdadera
done
```

La estructura del until es la siguiente:

```
until [ expresión ]; do
      estas líneas se repiten HASTA que la expresión sea verdadera
done
```

Ambas estructuras, tanto while como until realmente realizan exactamente lo mismo, al efectuar la comprobación de la expresión en la primera línea, no como en otros lenguajes.

Veamos un ejemplo de un script usando la estructura while (mientras).

```
#!/bin/bash
#doble.sh - script que pide números y muestra el doble de dichos números.
# el script continua ejecutandose mientras que el número introducido no sea 0.
read -p "Dime un número (0 para salir) : " NUMERO
while [ $NUMERO -ne 0 ]; do
      echo "El doble de $NUMERO es :" $(($NUMERO*2))
      read -p "Dime un número (0 para salir) : " NUMERO
done
```

Ahora veamos como queda el script, usando la estructura until (hasta).

```
#!/bin/bash
#doble.sh - script que pide números y muestra el doble de dichos números.
# el script continua ejecutandose mientras que el número introducido no sea 0.
read -p "Dime un número (0 para salir) : " NUMERO
until [ $NUMERO -eq 0 ]; do
      echo "El doble de $NUMERO es :" $(($NUMERO*2))
      read -p "Dime un número (0 para salir) : " NUMERO
done
```

Otro ejemplo, vamos a mostrar por pantalla los número del 1 al 20

```
#!/bin/bash
NUMERO=1
until [ $NUMERO -gt 20 ]; do
      echo "Número vale :" $NUMERO
      let NUMERO=NUMERO+1
done
```

Existe una orden interesante en Linux que es seq, que nos permite mostrar una secuencia de números, cuyo formato es `seq primero incremento ultimo`. Esto nos permite realizar este tipo de estructuras con for.

```
#!/bin/bash
for I in $( seq 1 1 20 ); do
      echo "Número vale :" $NUMERO
done
```

select

La última estructura iterativa que vamos a ver es select. Esta nos permite realizar una iteración o bucle, pero presentando un menú por pantalla para que el usuario escoja una opción. Su estructura general es la siguiente:

```
select VARIABLE in conjunto_opciones; do
      Aquí variable toma el valor de una de las opciones del conjunto
done
```

Esta estructura como vemos es muy parecida a la del for, pero presenta la principal diferencia en que por definición se crea un bucle sin final, no hay un valor inicial y un valor limite, el bucle se repetirá eternamente, lo que nos obliga a salirnos del mismo por las bravas, bien con `break` que nos permite salirnos del bucle o con `exit` que nos permite salirnos del script entero.

Veamos un ejemplo:

```
#!/bin/bash
select OPCION in Chiste Refrán Proverbio Salir; do
      if [ $OPCION = "Chiste" ]; then     # quedaría más mono con case claro
            echo "Van dos por la calle y se cae el de en medio"
      elif [ $OPCION = "Refrán" ]; then
            echo "Quien cría cuervos tendrá muchos"
      elif [ $OPCION = "Proverbio" ]; then
            echo "Ten cerca a tus amigos, y mucho mas cerca a tus enemigos"
      else
            # Ha escogido Salir
            break
      fi
done
echo "Hemos acabado el menú con select"
```

Veremos como por pantalla nos muestra un menú parecido al siguiente:

```
      1) Chiste
      2) Refrán
      3) Proverbio
```

```
4) Salir
#?
```

Y automáticamente realiza un read (el #?) para pedir al usuario que escoja una opción mediante los números que ha antepuesto a cada una. Comprobad como al escoger la opción 4, en nuestro script realizamos un break, cuya misión es salirse del bucle. Veremos como en pantalla sale el mensaje de Hemos acabado… ya que break continua la ejecución del script justo debajo del done. Si en lugar de break hubiéramos utilizado exit, no veríamos este mensaje en pantalla, ya que se termina el script completo, y en muchas ocasiones incluso cierra el terminal (razón de más para no usar nunca exit).

Al igual que sucedía con el for, es perfectamente posible crear el conjunto mediante una instrucción. Asi por ejemplo, la instrucción ls nos devuelve un conjunto formado por todos los ficheros del directorio actual. Vamos a trabajar sobre esta idea:

```
#!/bin/bash
select FICHERO in $( ls ); do
      echo Has seleccionado el fichero $FICHERO
      echo Ahora podríamos preguntar si quieres borrarlo, copiarlo,
visualizarlo, etc.
done
```

Si ejecutáis ese script, veréis dos cosas: Como el conjunto esta formado por todos los ficheros del directorio actual, y como es imposible detener la ejecución del script, como no sea matando el proceso en primer plano con Control + C

Por cierto, si en el conjunto ponemos directamente el símbolo asterisco (*) veremos que tiene la misma función que un ls, devuelve el listado de ficheros del directorio actual.

```
select FICHERO in *; do
```

Hagamos otro ejemplo sobre el select, un poco más avanzado. Vamos a mostrar por pantalla un menú con todos los mp3 que existan en el directorio home del usuario actual, y vamos a dejar que escoja uno de ellos para reproducirlo. (Para ello uso un reproductor de mp3 desde línea de comandos, si no lo tenéis instalado lo podéis instalar con un apt-get install mpg321).

```
#!/bin/bash
clear
select MP3 in $( find . -iname "*mp3" ); do
        echo "Voy a reproducir el mp3 : " $MP3
        mpg321 $MP3 &> /dev/null
done
```

Volvemos a tener el problema de que la ejecución no se acabará nunca, a menos que la interrumpamos mediante control c. Vamos a arreglarlo forzando la opción Salir en el conjunto:

```
#!/bin/bash
clear
CONJUNTO=$(find . -iname "*mp3")
CONJUNTO=$CONJUNTO" Salir"
select MP3 in $CONJUNTO; do
        if [ $MP3 = "Salir" ]; then
                break
```

```
            fi
            echo "Voy a reproducir el mp3 : " $MP3
            mpg321 $MP3 &> /dev/null
done
```

Como siempre en Informática, este script tan sencillo se puede complicar hasta lo inimaginable. Por ejemplo, este script necesita para funcionar que los nombres de los archivos mp3 no contengan espacios en blanco, ya que el conjunto separa sus valores por este carácter. Asi, si tuviéramos una canción con nombre La Gasolina, veríamos que en el menú nos aparecen dos opciones 1)La y 2)Gasolina, por lo que el script como es obvio no funcionará. ¿Se os ocurre alguna manera de solucionarlo? (A mi si, :-P, por ejemplo podríamos ejecutar antes del select una orden find con una opción -exec que fuera sustituyendo todos los espacios en blanco por subrayados bajos mediante un sed, y al finalizar el script volveríamos a dejar los subrayados bajos como espacios en blanco).

1.6 Paso de parámetros.

Podemos pasar parámetros tanto a los scripts como a las funciones. Los parámetros en bash se indican como un símbolo dólar ($) seguido de un número o carácter. Los principales parámetros que se pueden usar son:

Parámetros	
$1	Devuelve el 1º parámetro pasado al script o función al ser llamado.
$2	Devuelve el 2º parámetro.
$3	Devuelve el 3º parámetro. (Podemos usar hasta $9).
$*	Devuelve todos los parámetros separados por espacio.
$#	Devuelve el número de parámetros que se han pasado.
$0	Devuelve el parámetro 0, es decir, el nombre del script o de la función.

Podemos entenderlo mucho mejor con un script como el siguiente:

```
#!/bin/bash
# parámetros.sh - script para demostrar el funcionamiento de los parámetros.
echo "El primer parámetro que se ha pasado es " $1
echo "El tercer parámetro que se ha pasado es " $3
echo "El conjunto de todos los parámetros : " $*
echo "Me has pasado un total de " $# " parámetros"
echo "El parámetro 0 es : " $0
```

Si hacemos este script ejecutable, y lo llamamos como:

```
./parámetros.sh  Caballo  Perro  675 Nueva Cork
```

obtendríamos por pantalla lo siguiente:

```
El primer parámetro que se ha pasado es Caballo
El tercer parámetro que se ha pasado es 675
El conjunto de todos los parámetros : Caballo Perro 675 Nueva Cork
Me has pasado un total de     5 parametros
El parámetro 0 es : ./parámetros.sh
```

Como he indicado antes, también podemos pasarle parámetros a las funciones, usando el mismo metodo y las mismas posibilidades que para los scripts completos.

Asi por ejemplo:

```
#!/bin/bash
function mayor_edad {
        if [ $1 -ge 18 ]; then
                echo Si, es mayor de edad
        else
                echo No, es menor de edad
        fi
}
read -p "Dime la edad del que quiere entrar : " EDAD
echo voy a comprobar si puede entrar o no.
mayor_edad $EDAD
echo pues eso.
```

Como práctica, intentad modificad el script que hicimos explicando los if, que tenia como misión indicar si un numero introducido era capicúa o no. Modificadlo de tal modo que en lugar de pedir el número al usuario mediante un read, use directamente el número pasado como parámetro 1. Es decir, que se ejecutará asi: `./capicua.sh 767`

1.7 Valores devueltos por las órdenes.

Existe un parámetro especial, el $? que nos devuelve el valor del resultado de la ultima orden. Es decir, despues de ejecutar cualquier orden del sistema (o casi cualquier orden mejor dicho) podemos comprobar el valor de $? que tendrá un 0 si todo ha ido bien, y otro valor cualquiera en caso de que haya fallado. Comprobarlo es muy simple:

Desde la línea de comandos, haced un cd a un directorio que no existe, por ejemplo

```
cd /juegos/teto
```

y luego mirad el contenido de $? con un

```
echo $?
```

comprobareis como vale 1, es decir, indica que la última orden no funciono correctamente.

Ahora haced un cd a un directorio que si exista

```
cd /etc/network
```

y luego mirad el contenido de $? con un

```
echo $?
```

comprobareis como vale 0, es decir, indica que la última orden funciono sin problemas.

Este parámetro puede sernos muy útil realizando scripts, ya que nos permite una forma rápida y cómoda de ver si todo esta funcionando bien o no.

1.8 Algunos scripts completos de ejemplo

Hagamos un script que nos permita simular el juego ese de pensar un número y que el jugador lo adivine diciendo varios números, a los que se responderá únicamente si se han quedado cortos o se han pasado. Vamos a realizarlo llevando un control de cuantos intentos llevan y un contador de record que nos permitirá mostrar las 3 personas que lo han resuelto en menos intentos.

```
#!/bin/bash
# juego1.sh - script que permite jugar a adivinar un numero en varios
#                 intentos y llevando un control de los 3 mejores.
clear
# Si pasamos como parametro x borramos fichero record para empezar desde 0
if [ $# -ne 0 ]; then   # para controlar que se ha pasado al menos 1 para.
        if [ $1 = x ]; then
                echo "Borrando fichero de records."
                rm record.txt
        fi
fi
# Ahora vamos a leer el record del fichero
if [ -f record.txt ]; then
        POSICION=0
        for CAMPEON in $(cat record.txt); do
                let POSICION=POSICION+1
                NOMBRE_RECORD=$(echo $CAMPEON | cut -d: -f2)
                NUMERO_RECORD=$(echo $CAMPEON | cut -d: -f1)
                echo "En posicion $POSICION esta $NOMBRE_RECORD con\
                        $NUMERO_RECORD intentos"
        done
```

```
else
        echo '************************************************************'
        echo "No hay ningun record todavia. Aprovecha la oportunidad"
        echo '************************************************************'
fi
# comenzamos el juego en si mismo
CONTADOR=1
#let MINUMERO=RANDOM  # La variable $RANDOM nos da un numero aleatorio.
MINUMERO=3
echo ' ' ; echo ' '
echo '************************************************************'
read -p "Dime tu nombre : " NOMBRE
echo '************************************************************'
echo ' ' ; echo ' '
read -p "Llevas $CONTADOR intentos. Dime un numero : " NUMERO
until [ $NUMERO -eq $MINUMERO ]; do
        if [ $NUMERO -gt $MINUMERO ]; then
                echo "El numero que has metido es mayor"
        else
                echo "El numero que has metido es menor"
        fi
        let CONTADOR=CONTADOR+1
        read -p "Llevas $CONTADOR intentos. Dime un numero : " NUMERO
done
echo Hombreeee, por fin acertaste.
#grabamos el record en el fichero (primero los intentos y luego el nombre)
echo $CONTADOR:$NOMBRE >> record.txt
#ordenamos para dejar arriba los que lo han hecho en menos intentos
#y nos quedamos con las 3 primeras lineas
sort record.txt -g | head -3 > record.txt
```

Una cosa interesante que podéis ver en este script, es como hemos aprovechado las funciones de las ordenes Linux para simplificar el programa. Todo el tema de ordenación de los record, quedarnos solo con los 3 primeros, comprobar si el usuario actual ha batido algún record, etc, lo hemos realizado simplemente con una orden Linux, en este caso un head que viene de un sort. Si no lo hubiéramos hecho asi, el script sería mucho más largo y complicado.