1. Concepto de SHELL en Linux

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 interpretes	–	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*. En los laboratorios de prácticas para consultar ese fichero se puede ejecutar la orden *ypcat passwd*. 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 puede programarse usando ficheros de texto que contienen órdenes que interpretará en la ejecución de los mismos. Estos ficheros de texto se llaman **scripts**, **shell scripts** o **guiones shells**. Una vez creados, estos guiones shells 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, en cierto sentido, equivalente a crear archivos .BAT en MS-DOS, con la diferencia de que en UNIX/Linux es mucho más potente.

Una definición de guiones shells podría ser la dada a continuación:

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 formará 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
```

En este caso el carácter # del principio de la línea no es un comentario.

La programación de shell es una de las herramientas más apreciadas por todos los administradores y muchos usuarios 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:

- Comandos del sistema UNIX/Linux, por ejemplo *ls*, *cut*, 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 *Perl*, *TCL* o *Python*.

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.

2. Funcionamiento del Shell

Supongamos que tenemos el siguiente guión shell:

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

al ejecutarse el proceso que se sigue es el siguiente:

1. El shell */bin/bash* hace 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.
6. Cuando finaliza el proceso hijo, hace que el padre se despierte.

3. Variables y parámetros

3.1. 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 = y 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"
```

Las variables se dividen en dos tipos:

- **Variables locales**: son aquéllas 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:

```
$ export mils        # Convierte la variable mils en
                     # variable de entorno
$ export var=valor   # Crea la variable, le asigna "valor"
                     # y la exporta a la vez
```

La orden *set* muestra todas las variables (locales y de entorno).
La orden *env* muestra sólo las variables 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. El siguiente ejemplo modifica el prompt, utilizando colores, el nombre del usuario aparece en color verde, y el resto en negro:

```
PS1='[\033[0;32;48m\u\033[0;30;48m@\h \W ] \ $'
```

- *PS2*: prompt secundario.
- *LOGNAME*: nombre del usuario.
- *HOME*: directorio de trabajo ("home") del usuario actual, es el argumento que la orden *cd* toma por defecto.
- *PATH*: caminos usados para ejecutar órdenes o programas. El directorio actual no está incluido en la búsqueda por defecto (en MS_DOS sí) y si se desea debe formar parte de la variable PATH. Por ejemplo:

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

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

- *SHELL*: shell actual.
- *IFS*: el *Separador Interno de Campo* que se emplea para la división de palabras tras la expansión y para dividir líneas en palabras con la orden interna *read*. El valor predeterminado es "<espacio><tab><nueva-línea>".

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.
- *$-*: las opciones actuales suministradas para esta invocación del shell.
- *$**: todos los argumentos del shell comenzando por el *$1*. Cuando la expansión ocurre entre comillas dobles, se expande a una sola palabra con el valor de cada parámetro separado por el primer carácter de la variable especial *IFS*. Esto es, "$*" es equivalente a "$1c$2c...", donde c es el primer carácter del valor de la variable *IFS*. Si *IFS* no está definida, los parámetros se separan por espacios. Si *IFS* es la cadena vacía, los parámetros se juntan sin ningún separador.
- *$@*: igual que el anterior, excepto cuando va entrecomillado. Cuando la expansión ocurre dentro de comillas dobles, cada parámetro se expande a una palabra separada. Esto es, "$@" es equivalente a "$1" "$2"...

3.2. 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} ${12} ....
```

- 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 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
```

3.3. 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. Por ejemplo:

  ```
  echo $fecha
  ```

  muestra el contenido de la variable "fecha", si existe. En caso de que no esté 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 definida, 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
```

3.4. 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 usar * y @ es que *${nombre_array[*]}* expande los elementos del array como si fueran una única palabra, mientras que *${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. Mientras 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
```

4. Caracteres especiales y de entrecomillado

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**, (no confundir con el código ASCII cuyo valor es 27 en decimal o 33 en octal), el cual preserva el valor literal del siguiente carácter que lo acompaña, con la excepción de <nueva-línea>. Si aparece un par \<nueva-línea> y la barra invertida no está ella misma entre comillas, el \<nueva-línea> se trata como una continuación de línea (esto es, se quita del flujo de entrada y no se tiene en cuenta). Por ejemplo:

```
$ ls \
> -l

Es lo mismo que ejecutar "ls -l"
```

Encerrar caracteres entre **comillas simples** preserva el valor literal de cada carácter entre las comillas. Una comilla simple no puede estar entre comillas simples, ni siquiera precedida de una barra invertida.

```
$ echo 'caracteres especiales: $, &, `, ", (, ),;,|,<,>,\...'
caracteres especiales: $, &, `, ", (, ),;,|,<,>,\...
```

Encerrar caracteres entre **comillas dobles** preserva el valor literal de todos los caracteres, con la excepción de $,` , y \. Los caracteres $ y ` mantienen su significado especial dentro de comillas dobles. La barra invertida mantiene su significado especial solamente cuando está seguida por uno

de los siguientes caracteres: $, ‘, ", o <nueva-línea>. Una comilla doble puede ser entrecomillada entre otras comillas dobles precediéndola de una barra invertida. Por ejemplo:

```
$ echo "caracteres especiales: \$, &, \‘, \", (, ),;,|,<,>,\\..."
caracteres especiales: $, &, ‘, ", (, ),;,|,<,>,\...
```

Los parámetros especiales $* y $@ tienen un significado especial cuando están entre comillas dobles, (véase el apartado de “Variables y parámetros”).

Las palabras de la forma $’cadena’ se tratan de forma especial. La palabra se expande a cadena, con los caracteres protegidos por barra invertida reemplazados según especifica el estándar ANSI/ISO de C. Las secuencias de escape con barra invertida, si están presentes, se descodifican como sigue:

- \a: alerta (campana)
- \b: espacio-atrás
- \e: un carácter de escape (ESC)
- \f: nueva página
- \n: nueva línea
- \r: retorno de carro
- \t: tabulación horizontal
- \v: tabulación vertical
- \\: barra invertida
- \nnn: el carácter cuyo código es el valor octal nnn (de uno a tres dígitos)
- \xnnn: el carácter cuyo código es el valor hexadecimal nnn

El resultado traducido es entrecomillado con comillas simples, como si el signo de dólar no hubiera estado presente.

A continuación se muestra, de forma resumida, el significado de estos caracteres especiales:

- \ : elimina el significado especial del siguiente carácter
- $: muestra el valor de la variable que le prosiga. La variable se puede meter entre llaves si enlaza con más caracteres. Por ejemplo:

```
$ echo ${var}pepe
```

- ’ ’ Dentro de las comillas simples todos los caracteres son interpretados literalmente, ninguno de los caracteres especiales conserva su significado dentro de ellas, eliminando el significado especial de todo lo que va en medio.
- “ ” En general los caracteres especiales que hay en su interior no son interpretados, con la excepción de \, ‘ y $.

4.1. 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
```

5. Estructuras de control

5.1. 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 main prac.c
then
        echo encontrada la palabra clave main
else
        echo no encontrada la palabra clave main
fi
```

La sintaxis de la orden *case*:

```
case $var in
v1)     ..              # Ejemplo [Ss][Ii]*)
..
;;
v2|v3)  ..
..
;;
*)      ..              # Caso por defecto
;;
esac
```

Las expresiones regulares que hay en v1, v2 y v3 siguen el mismo patrón que los comodines de los ficheros. Un ejemplo de su funcionamiento podría ser:

```
case $var in
     1)   echo La variable var es un uno
     ;;
     2)   echo La variable var es un dos
     ;;
     *)   echo La variable var no es ni un uno ni un dos
     ;;
esac
```

En el apartado "Ejemplos de guiones shell" se pueden encontrar más ejemplos del uso de estas dos estructuras de control.

5.2. WHILE y UNTIL

También es posible ejecutar bloques de órdenes de forma iterativa dependiendo de una condición. 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
```

Un ejemplo del funcionamiento de ambos sería:

```
# Muestra todos los parámetros
while [ ! -z $1 ]
do
      echo Parámetro: $1
      shift
done

# El siguiente también muestra todos los parámetros
until [ -z $1 ]
do
      echo $1
      shift
done
```

En el apartado "Ejemplos de guiones shell" se pueden encontrar otros ejemplos del uso de estas dos estructuras de control.

5.3. FOR

Con la orden *for* se ejecutan bloques de órdenes, permitiendo que en cada iteración una determinada 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
```

5.4. 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.

Un ejemplo sería el siguiente:

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

En pantalla aparecería:

```
1) Ver contenido directorio actual
2) Salir
#?
```

Si se selecciona la primera opción, 1, se mostraría el mensaje: "Ha seleccionado la opción: Ver contenido directorio actual", y se ejecutaría la orden *ls* en el directorio actual, y volvería a pedir la siguiente elección.

Si por el contrario se pulsase un 2, seleccionando la segunda opción, aparecería el mensaje: "Ha seleccionado la opción: Salir", y saldría del *select*.

5.5. 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
```

6. Generación de nombres de ficheros

Cuando se le indica al shell el nombre de un fichero algunos caracteres tienen un significado especial. Estos caracteres son interpretados por el shell antes de proceder a ejecutar lo que se le ha indicado. Son los siguientes caracteres:

- ? : cualquier carácter excepto el "."inicial.
- * : 0..n caracteres (excepto el "." inicial).
- [ ] : clase de caracteres. Cualquier carácter de esa clase.
- [!] : clase de caracteres negada. Todos los caracteres excepto los de la clase.
- [^] : niega la expresión que le sigue.
- { } : contiene varias expresiones separadas por comas y corresponde a cualquiera de ellas.

7. Entrada/salida estándar y redirección

La filosofía de UNIX/Linux es en extremo modular. Se prefieren las herramientas pequeñas con tareas puntuales a las meta-herramientas que realizan todo. Para hacer el modelo completo es necesario proveer el medio para ensamblar estas herramientas en estructuras más complejas. Esto se realiza por medio del redireccionamiento de las entradas y las salidas.

Todos los programas tienen por defecto una **entrada estándar** (teclado) y dos salidas: la **salida estándar** (pantalla) y la **salida de error** (pantalla). En ellos se puede sustituir la entrada, salida y error por otro dispositivo. Con esto se consigue que los datos de entrada para ese programa se lean de un archivo, y los de salida (estándar o error) vayan a otro archivo.

Esas entrada, salida y error estándar se asocian a los programas mediante tres ficheros con los cuales se comunican con otros procesos y con el usuario. Estos tres ficheros son:

- **stdin**: entrada estándar, a través de este descriptor de fichero los programas reciben datos de entrada. Normalmente **stdin** está asociado a la entrada de la terminal en la que está corriendo el programa, es decir, al **teclado**. Cada descriptor de fichero tiene asignado un número con el cual podemos referirnos a él dentro de un script, en el caso de **stdin** es el 0. ¡OJO! dentro de un programa en C no es lo mismo usar **stdin** que el descriptor 0.
- **stdout**: salida estándar, es el descriptor de fichero en el que se escriben los mensajes que imprime un programa. Normalmente estos mensajes aparecen en la terminal para que los lea el usuario. Su descriptor de fichero es el número 1.
- **stderr**: salida de error, es el descriptor de fichero al que se escriben los mensajes de error que imprime el programa. Normalmente coincide con **stdout**. Tiene como descriptor de fichero el número 2.

Estos tres descriptores de fichero pueden redireccionarse, consiguiendo comunicar unos procesos con otros, de forma que trabajen como una unidad, haciendo cada uno una tarea especializada, o simplemente almacenando los datos de salida en un fichero determinado, o recibiendo los datos de entrada de un fichero concreto. Hay varios operadores para redireccionar la entrada y las salidas de un programa de distintas maneras:

- > : redirecciona **stdout** a un fichero, si el fichero existe lo sobrescribe:

```
who > usuarios.txt; less usuarios.txt
```

- >> : redirecciona **stdout** a un fichero, si el fichero existe añade los datos al final del mismo.
- 2 > : redirecciona la salida de error al fichero que se indique:

```
find / -type d -exec cat {} \; 2>errores.txt
```

- 2 >> : la misma idea que el anterior, pero si el fichero ya existe, los datos se añaden al final del mismo, no borrando el contenido que ya tiene.
- n>&m : redirecciona el descriptor de fichero n al descriptor de fichero m, en caso de que n se omita, se sobrentiende un 1 (**stdout**):

```
$ cat file directorio >salida.txt 2>&1
# Redirecciona stderr a stdout, y ésta al fichero salida.txt
```

- < : lee la entrada estándar de un fichero:

```
$ grep cadena < fichero.txt
# busca "cadena" dentro de un fichero
```

- | : redirecciona la salida de una orden a la entrada de otra orden: (véase el apartado de "Tuberías o filtros").

```
who | grep pilar
```

8. Tuberías o Filtros

En la línea de órdenes la integración entre diferentes programas se realiza mediante la interconexión de las entradas y salidas a través de pipes o tuberías.

Una tubería o pipe es una combinación de varias órdenes que se ejecutan simultáneamente, donde el resultado de la primera se envía a la entrada de la siguiente. Es como si se hiciese una redirección de la salida estándar de la primera orden, dirigida hacia la entrada estándar de la segunda. Esta tarea se realiza por medio del carácter barra vertical "|".

Por ejemplo si se quieren ver, poco a poco, todos los archivos que hay en el directorio */usr/bin*, se ejecuta lo siguiente:

```
$ ls /usr/bin | more
```

De este modo, la salida del programa *ls* (listado de todos los archivos del directorio */usr/bin*) irá al programa *more* (modo paginado, es decir, muestra una pantalla y espera a que pulsemos una tecla para mostrar la siguiente).

Dentro de esta estructura se han construido una serie de programas conocidos como "filtros" los cuales realizan procesos básicos sobre textos, son los siguientes:

Filtros	**Función**
sort fichero	Ordena las líneas de un texto. ls | sort # Ordena las líneas obtenidas de la orden ls
cut	Corta secciones de una línea cut -cn1-n2 #Corta desde el carácter n1 a n2 cut -fn1-n2 -dX #Corta del campo n1 al n2, donde X es el delimitador
grep	Busca en la entrada que se le especifica líneas que concuerden o coincidan con el patrón dado. grep [-inv] patrón fichero # i ignora mayúsculas, n con nº línea, v inverso grep -q patrón fichero # verifica si encuentra el patrón o no, sin mostrar # las líneas que coinciden.
od	Convierte archivos a forma octal u otras.
paste	Une líneas de diferentes archivos.
tac	Concatena e imprime archivos invertidos.
tee	Lee la entrada estándar y escribe en la salida estándar y en los ficheros que se le indican orden 1 | tee fichero | orden2 #recibe los resultados de orden1, los escribe en fichero y los pasa a orden2
tr	Traduce o borra caracteres. tr "c1" "c2" # traduce el carácter c1 por el c2 tr -s "c" # traduce 1 ó más apariciones de c por una sola aparición Esta orden es muy útil para aplicar antes de cut.
uniq	Elimina líneas repetidas. cat bd.txt | uniq #Elimina líneas repetidas de bd.txt
wc [-wlc]	Cuenta bytes, palabras y líneas.

A continuación se muestran algunos ejemplos del uso de estos programas junto con las tuberías.

- La siguiente orden cuenta cuantos ficheros hay en el directorio actual:

```
$ ls | wc -l
```

- Esta orden muestra el número de directorios que hay en el directorio actual:

```
$ ls -l | grep ^d | wc -l
```

- Usando las órdenes *who* ,*sort*, *uniq* y *cut* podemos saber qué usuarios están trabajando en el sistema en este momento, el listado se muestra ordenado:

```
$ who | cut -f1 -d " "| sort | uniq
```

- La siguiente orden elimina todos los espacios en blanco del fichero "basura.txt", lo guarda en otro fichero llamado "basura.sinblancos", y lo muestra por pantalla:

```
$ cat basura.txt | tr -s " " | tee basura.sinblancos
```

- Esta última orden obtiene el identificador de todos los procesos ejecutados por el usuario "pilar", y los muestra en una única línea:

```
$ ps aux |grep ^pilar |tr -s " " |cut -f2 -d " " |tr "\n" " "
```

Algunos filtros han llegado a ser tan complejos que son en sí un lenguaje de procesamiento de texto, de búsqueda de patrones, de construcción de scripts, y muchas otras posibilidades. Entre ellos podemos mencionar herramientas tradicionales en UNIX/Linux como *awk* y *sed* y otras más modernas como *Perl*.

9. 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"
```

La orden *read* puede leer varias variables a la vez. También se puede combinar el uso de *read* con *echo*, para mostrar un prompt que indique que es lo que se está pidiendo. Hay una serie de caracteres especiales para usar en *echo* y que permiten posicionar el cursor en un sitio determinado:

- \b: retrocede una posición (sin borrar)
- \f: alimentación de página
- \n: salto de línea
- \t: tabulador

Para que *echo* reconozca estos caracteres es necesario utilizar la opción "*-e*" y utilizar comillas dobles:

```
$ echo -e "Hola \t  ¿cómo estás?"
hola     como estás
```

- *cd*: cambia de directorio
- *pwd*: devuelve el nombre del directorio actual, equivale a leer el valor de la variable $PWD.
- *pushd / popd / dirs*: estas órdenes son muy útiles cuando un script tiene que navegar por un árbol de directorios:
  - *pushd*: apila un directorio en la pila de directorios.
  - *popd*: lo desapila y cambia a ese directorio.
  - *dirs*: muestra el contenido de la pila.
- *let arg* [*arg*]: cada arg es una expresión aritmética a ser evaluada:

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

  Si el último arg se evalúa a 0, let devuelve 1; si no, devuelve 0.
- *test*: permite evaluar si una expresión es verdadera o falsa, véase el apartado "La orden test".
- *export*: hace que el valor de una variable esté disponible para todos los procesos hijos del shell.
- [.|*source*] *nombre_fichero argumentos*: lee y ejecuta órdenes desde nombre_fichero en el entorno actual del shell y devuelve el estado de salida de la última orden ejecutada desde nombre_fichero. Si se suministran argumentos, se convierten en los parámetros cuando se ejecuta nombre_fichero. Cuando se ejecuta un guión shell precediéndolo de "." o *source*, no se crea un shell hijo para ejecutarlo, por lo que cualquier modificación en las variables de entorno permanece al finalizar la ejecución, así como las nuevas variables creadas.
- *exit*: finaliza la ejecución del guión, recibe como argumento un entero que será el valor de retorno. Este valor lo recogerá el proceso que ha llamado al guión shell.
- *fg*: reanuda la ejecución de un proceso parado, o bien devuelve un proceso que estaba ejecutándose en segundo plano al primer plano.
- *bg*: lleva a segundo plano un proceso de primer plano o bien un proceso suspendido.
- *wait*: detiene la ejecución hasta que los procesos que hay en segundo plano terminan.
- *true* y *false*: devuelven 0 y 1 siempre, respectivamente.

Nota: El valor 0 se corresponde con *true*, y cualquier valor distinto de 0 con *false*.

10. Evaluación aritmética

El shell permite que se evalúen expresiones aritméticas, bajo ciertas circunstancias. La evaluación se hace con enteros largos sin comprobación de desbordamiento, aunque la división por 0 se atrapa y se señala como un error. La lista siguiente de operadores se agrupa en niveles de operadores de igual precedencia, se listan en orden de precedencia decreciente.

-, +	Menos y más unarios
~	Negación lógica y de bits
**	Exponenciación
*, /, %	Multiplicación, división, resto
+, -	Adición, sustracción
<<, >>	Desplazamientos de bits a izquierda y derecha
<=, >= , <, >	Comparación
==, !=	Igualdad y desigualdad
&	Y de bits (AND)
^	O exclusivo de bits (XOR)
|	O inclusivo de bits (OR)
&&	Y lógico (AND)
||	O lógico (OR)
expre?expre:expre	Evaluación condicional
=, +=, -=, *=, /=, %=, &=, ^=, |= <<=, >>=	Asignación: simple, después de la suma, de la resta, de la multiplicación, de la división, del resto, del AND bit a bit, del XOR bit a bit, del OR bit a bit, del desplazamiento a la izquierda bit a bit y del desplazamiento a la derecha bit a bit.

Por ejemplo, la siguiente orden muestra en pantalla el valor 64

```
$ echo $((2**6))
```

Se permite que las variables del shell actúen como operandos: se realiza la expansión de parámetro antes de la evaluación de la expresión. El valor de un parámetro se fuerza a un entero largo dentro de una expresión. Una variable no necesita tener activado su atributo de entero para emplearse en una expresión.

Las constantes con un 0 inicial se interpretan como números octales. Un 0x ó 0X inicial denota un número en hexadecimal. De otro modo, los números toman la forma [base#]n, donde base es un número en base 10 entre 2 y 64 que representa la base aritmética, y n es un número en esa base. Si base se omite, entonces se emplea la base 10. Por ejemplo:

```
$ let a=6#10+1
$ echo el valor de a es $a
  El valor de a es 7
```

Los operadores se evalúan en orden de precedencia. Las subexpresiones entre paréntesis se evalúan primero y pueden sustituir a las reglas de precedencia anteriores.

Existen tres maneras de realizar operaciones aritméticas:

1. Con *let lista_expresiones*, como se ha dicho anteriormente, se pueden evaluar las expresiones aritméticas dadas como argumentos. Es interesante destacar que esta orden no es estándar, sino que es específico del *bash*. A continuación se muestra un ejemplo de su uso:

```
$ 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

$ let c=2#101\|2#10
$ echo El valor de c es $c
El valor de c es 7
```

2. La orden *expr* sirve para evaluar expresiones aritméticas. Puede incluir los siguientes operadores: \(, \), *, \\, \+, \-, donde el carácter '\' se introduce para quitar el significado especial que pueda tener el carácter siguiente. Por ejemplo:

```
$ expr 10 \* \( 5 \+ 2 \)
70

$ i=`expr $i - 1`      #restará 1 a la variable i
```

3. Mediante *$((expresión))* también se pueden evaluar expresiones. Varios ejemplos de su uso serían:

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

  $ echo El resto de la división es: $((7%5))
  El resto de la división es: 2

  $ echo El valor es $((2#101|2#10))
  El valor de c es 7
```

11. 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 ]
```

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

- Para **números**: *arg1 OP arg2*, donde OP puede ser uno de los siguientes:

-eq	Igual a
-ne	Distinto de
-lt	Menor que
-le	Menor o igual que
-gt	Mayor que
-ge	Mayor o igual que

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.

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

-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.

- Además se pueden incluir **operadores lógicos** y **paréntesis**:

-o	OR
-a	AND
!	NOT
\(	Paréntesis izquierdo
\)	Paréntesis derecho

11.1. 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
```

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
```

Comparando dos cadenas:

```
#! /bin/bash
S1='cadena'
S2='Cadena'
if [ $S1!=$S2 ];
then
       echo "S1('$S1') no es igual a S2('$S2')"
fi
if [ $S1=$S1 ];
then
       echo "S1('$S1') es igual a S1('$S1')"
fi
```

En determinadas versiones, esto no es buena idea, porque si *$S1* o *$S2* son vacíos, aparecerá un error sintáctico. En este caso, es mejor: `x$1=x$2` o `"$1"="$2"`.

12. Órdenes simples, listas de órdenes y órdenes compuestas

12.1. Órdenes simples

Una orden simple es una secuencia de asignaciones opcionales de variables seguida por palabras separadas por blancos y redirecciones, y terminadas por un operador de control. La primera palabra especifica la orden a ser ejecutada. Las palabras restantes se pasan como argumentos a la orden pedida. Por ejemplo, si tenemos un shell script llamado *programa*, podemos ejecutar la siguiente orden:

```
$ a=9 programa
```

La secuencia de ejecución que se sigue es: se asigna el valor a la variable *a*, que se exporta a *programa*, i.e., *programa* va a utilizar la variable *a* con el valor 9.

El valor devuelto de una orden simple es su estado de salida, ó 128+n si la orden ha terminado debido a la señal n.

Un **operador de control** es uno de los siguientes símbolos:

&	&&	;	;;	(	)	|	<nueva-línea>

12.2. Listas de órdenes

Una lista es una secuencia de una o más órdenes separadas por uno de los operadores ;, &, &&, o ||, y terminada opcionalmente por uno de ;, &, o <nueva-línea>.

De estos operadores de listas, && y || tienen igual precedencia, seguidos por ; y &, que tienen igual precedencia.

Si una orden se termina mediante el operador de control &, el shell ejecuta la orden en segundo plano en un subshell. El shell no espera a que la orden acabe, y el estado devuelto es 0. Las órdenes separadas por un ; se ejecutan secuencialmente; el shell espera que cada orden termine. El estado devuelto es el estado de salida de la última orden ejecutada.

Los operadores de control && y || denotan listas *Y* (*AND*) y *O* (*OR*) respectivamente. Una lista *Y* tiene la forma:

```
orden1 && orden2
```

orden2 se ejecuta si y sólo si *orden1* devuelve un estado de salida 0.

Una lista *O* tiene la forma:

```
orden1 || orden2
```

orden2 se ejecuta si y sólo si *orden1* devuelve un estado de salida distinto de 0.

El estado de salida de las listas *Y* y *O* es el de la última orden ejecutada en la lista.

Dos ejemplos del funcionamiento de estas listas de órdenes son:

```
test -e prac.c || echo El fichero no existe
test -e prac.c && echo El fichero sí existe
```

La orden *test -e fichero* devuelve verdad si el fichero existe. Cuando no existe devuelve un 1, y la lista *O* tendría efecto, pero no la *Y*. Cuando el fichero existe, devuelve 0, y la lista *Y* tendría efecto, pero no la *O*.

Otros ejemplos de uso son:

```
sleep 1 || echo Hola     # echo no saca nada en pantalla
sleep 1 && echo Hola     # echo muestra en pantalla Hola
```

Un ejemplo de lista de órdenes encadenadas con ; es:

```
ls -l; cat prac.c; date
```

Primero ejecuta la orden *ls -l*, a continuación muestra en pantalla el fichero prac.c (*cat prac.c*), y por último muestra la fecha (*date*).

El siguiente ejemplo muestra el uso del operador de control & en una lista:

```
ls -l & cat prac.c & date &
```

En este caso, ejecuta las tres órdenes en segundo plano.

12.3. Órdenes compuestas

Las órdenes se pueden agrupar formando órdenes compuestas:

- { c1 ; ... ; cn; }: las n órdenes se ejecutan simplemente en el entorno del shell en curso, sin crear un shell nuevo. Esto se conocen como una **orden de grupo**.
- (c1 ; ... ; cn): las n órdenes se ejecutan en un nuevo shell hijo, se hace un *fork*.

¡OJO! Es necesario dejar los espacios en blanco entre las órdenes y las llaves o los paréntesis.

Para ver de forma clara la diferencia entre estas dos opciones lo mejor es estudiar qué sucede con el siguiente ejemplo:

Ejemplo	#!/bin/bash cd /usr { cd bin; ls; } pwd	#!/bin/bash cd /usr (cd bin; ls) pwd
Resultado	1.- Entrar al directorio /usr 2.- Listado del directorio /usr/bin 3.- Directorio actual: /usr/bin	1.- Entrar al directorio /usr 2.- Listado del directorio /usr/bin 3.- Directorio actual: /usr

Ambos grupos de órdenes se utilizan para procesar la salida de varios procesos como una sola. Por ejemplo:

```
$ ( echo bb; echo ca; echo aa; echo a ) | sort
a
aa
bb
ca
```

13. Funciones

Como en casi todo lenguaje de programación, se pueden utilizar funciones para agrupar trozos de código de una manera más lógica, o practicar la recursión.

Declarar una función es sólo cuestión de escribir:

```
function mi_func
{ mi_código }
```

Llamar a la función es como llamar a otro programa, sólo hay que escribir su nombre.

13.1. Ejemplo de funciones

```
#! /bin/bash

# Se define la función salir
function salir {
       exit
}

# Se define la función hola
function hola {
       echo ¡Hola!
}

hola        # Se llama a la función hola
salir       # Se llama a la función salir
echo petete
```

Tenga en cuenta que una función no necesita ser declarada en un orden específico.

Cuando ejecute el script se dará cuenta de que: primero se llama a la función "hola", luego a la función "salir", y el programa nunca llega a la línea "echo petete".

13.2. Ejemplo de funciones con parámetros

```
#!/bin/bash
function salir {
       exit
}

function e {
       echo $1
}


e Hola
e Mundo
salir
echo petete
```

Este script es casi idéntico al anterior. La diferencia principal es la función "e", que imprime el primer argumento que recibe. Los argumentos, dentro de las funciones, son tratados de la misma manera que los argumentos suministrados al script. (Véase el apartado "Variables y parámetros").

14. 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
```

Ambas opciones pueden ser utilizadas de forma conjunta:

```
#!/bin/bash -xv
      ó
$ bash -xv prac1
```