Programación shell

1. Estructura y ejecución de un script

El shell no es solamente un simple intérprete de comandos, sino que dispone de un verdadero lenguaje de programación; en particular, con gestión de las variables, control del flujo y bucles, operaciones sobre las variables, funciones...

Se agrupan todas las instrucciones y los comandos dentro de un script. Durante su ejecución, cada línea se leerá una por una y se ejecutará. Una línea puede componerse de comandos internos o externos, de comentarios o estar vacía. Es posible diseñar varias instrucciones consecutivas siempre separadas por ; o relacionadas de manera condicional por **&&** o **||**. El **;** es el equivalente de un **salto de línea**.

Por convención, los nombres de los scripts del shell terminan en general (no es obligatorio) por «sh» para el Bourne Shell y el Bourne Again Shell; «.ksh» para el Korn Shell y «csh» para el C Shell.

Para que un script sea ejecutable directamente:

```
$ chmod u+x miscript
```

Para ejecutarlo:

```
$ ./miscript
```

Para evitar el ./:

```
$ PATH=$PATH:.
$ miscript
```

Observe que se coloca el punto en última posición en el PATH. Ponerlo en primera posición puede representar un riesgo para la seguridad: se ha colocado un nuevo

comando **ls** modificado en su directorio. Imagine los daños con un comando **passwd**.

Cuando se inicia un script, se crea un nuevo shell hijo que va a ejecutar cada uno de los comandos. Si es un comando interno, el nuevo shell lo ejecuta directamente. Si es un comando externo binario, se creará un nuevo hijo para ejecutarlo. En el caso de un script de shell, se inicia un nuevo shell hijo para leer este nuevo shell línea por línea.

Una línea de comentario siempre empieza con el carácter **#**. Se puede colocar un comentario al final de una línea que ya comporta comandos.

```
# La línea siguiente efectúa un ls
ls # La línea en cuestión
```

La primera línea reviste una importancia particular, ya que permite especificar qué shell va a ejecutar el script, esta línea se llama **shebang**:

```
#!/bin/bash
#!/bin/ksh
```

En el primer caso, es un script Bourne Again; en el otro, un script Korn. También puede pasar opciones a los comandos.

Si no sabemos dónde se encuentra el binario del intérprete de comandos, puede usar el comando **env** de esta manera:

```
#!/usr/bin/env bash
```

2. Argumentos de un script

a. Parámetros de posición

Los parámetros de posición son también variables especiales utilizadas cuando se pasan parámetros a un script.

Variable	Contenido
$0	Nombre del comando (del script).
$1-9	$1,$2,$3... Los nueve primeros parámetros pasados al script.
$#	Número total de parámetros pasados al script.
$*	Lista de todos los parámetros en formato "$1 $2 $3 ...".
$@	Lista de los parámetros en forma de elementos distintos "$1" $2" "$3" ...

```
$ cat param.sh
#!/bin/bash

echo "Nombre: $0"
echo "Número de parámetros: $#"
echo "Parámetros: 1=$1 2=$2 3=$3"
echo "Lista: $*"
echo "Elementos: $@"

$ param.sh juanito jorgito jaimito
Nombre: ./param.sh
Número de parámetros: 3
Parámetros: 1=juanito 2=jorgito 3=jaimito
Lista: juanito jorgito jaimito
Elementos: juanito jorgito jaimito
```

La diferencia entre **$@** y **$*** no salta a la vista. Retome el ejemplo anterior con una pequeña modificación:

```
$ param.sh juanito "jorgito jaimito"
Nombre: ./param.sh
```

```
Número de parámetros: 2
Parámetros: 1=juanito 2=jorgito jaimito 3=
Lista: juanito jorgito jaimito
Elementos: juanito jorgito jaimito
```

Esta vez, sólo se pasan dos parámetros. Sin embargo, las listas parecen visualmente idénticas. En realidad, si la primera contiene, en efecto:

```
"juanito jorgito jaimito"
```

La segunda contiene:

```
"juanito" "jorgito jaimito"
```

O sea, dos elementos. En el primer ejemplo, tenía:

```
"juanito" "jorgito" "jaimito"
```

b. Redefinición de los parámetros

Además de listar las variables, la instrucción **set** permite redefinir el contenido de las variables de posición. Con:

```
set valor1 valor2 valor3 ...
```

$1 tomará como contenido valor1; $2, valor2, y así sucesivamente.

```
$ cat param2.sh
#!/bin/bash
echo "Antes:"
echo "Número de parámetros: $#"
echo "Parámetros: 1=$1 2=$2 3=$3 4=$4"
```

```
echo "Lista: $*"
set alfred oscar romeo zulu
echo "después set alfred oscar romeo zulu"
echo "Número de parámetros: $#"
echo "Parámetros: 1=$1 2=$2 3=$3 4=$4"
echo "Lista: $*"

$ ./param2.sh juanito jorgito jaimito donald gilito
Antes:
Número de parámetros: 5
Parámetros: 1=juanito 2=jorgito 3=jaimito 4=donald
Lista: juanito jorgito jaimito donald gilito
después set alfred oscar romeo zulu
Número de parámetros: 4
Parámetros: 1=alfred 2=oscar 3=romeo 4=zulu
Lista: alfred oscar romeo zulu
```

c. Reorganización de los parámetros

El comando **shift** permite modificar la posición de los parámetros. Una simple llamada desplaza todos los parámetros una posición, suprimiendo el primero: $2 se convierte en $1, $3 se convierte en $2, y así sucesivamente. El $1 original desaparece. $#, $* y $@ se vuelven a definir en consecuencia.

El comando **shift** seguido de un valor n efectúa un desplazamiento de n elementos. Así, con shift 4 $5 se convierte en $1, $6 se convierte en $2...

```
$ cat param3.sh
#!/bin/bash
set alfred oscar romeo zulu
echo "set alfred oscar romeo zulu"
echo "Número de parámetros: $#"
echo "Parámetros: 1=$1 2=$2 3=$3 4=$4"
echo "Lista: $*"
shift
echo "Después de un shift"
echo "Número de parámetros: $#"
echo "Parámetros: 1=$1 2=$2 3=$3 4=$4"
```

```
echo "Lista: $*"

$ ./param3.sh
set alfred oscar romeo zulu
Número de parámetros: 4
Parámetros: 1=alfred 2=oscar 3=romeo 4=zulu
Lista: alfred oscar romeo zulu
Después de un shift
Número de parámetros: 3
Parámetros: 1=oscar 2=romeo 3=zulu 4=
Lista: oscar romeo zulu
```

d. Salida de script

El comando **exit** permite terminar un script. Por defecto, el valor devuelto es 0 (no error), pero se puede especificar cualquier otro valor de 0 a 255. Puede recuperar el valor de salida mediante la variable $?.

```
$ exit 1
```

3. Entorno del proceso

Solamente las variables exportadas son accesibles por un proceso hijo. Si desea visualizar el contexto relacionado con un hijo (en un script por ejemplo), utilice el comando **env** como se vio anteriormente.

El comando **env** permite volver a definir también el contexto del proceso a ejecutar. Puede ser útil cuando el script debe acceder a una variable que no está presente en el entorno del padre, o que no se desea exportar. La sintaxis es:

```
env var1=valor var2=valor ... comando
```

En el caso de bash, env no es indispensable.

```
var1=valor var2=valor ... comando
```

Si la primera opción es el signo -, entonces se debe suprimir todo el entorno existente para sustituirlo por las nuevas variables y los nuevos valores.

```
$ unset a
$ ./ver_a.sh
a=
$ a=jojo ./ver_a.sh
a=jojo
$ echo a=$a
a=
```

4. Sustitución de comando

El mecanismo de sustitución permite colocar el resultado de comandos simples o complejos en una variable. Debe situar los comandos que va a ejecutar entre dos acentos graves «`» ([Alt Gr] 7 en un teclado de PC, tecla £ en un teclado Mac).

```
$ mi_unix=`uname`
$ echo ${mi_unix}
Linux
$ maquina=`uname -a | cut -d" " -f5`
echo $maquina
SMP
```

Cuidado: sólo se asigna a la variable el canal de salida estándar. El canal de error estándar sale siempre por pantalla en este caso. Si quiere recuperar la salida del canal de error, use una redirección.

```
$ valor=`cat archivo_inexistente 2>&1`
$ echo $valor
```

```
cat: archivo_inexistente: No existe el archivo o el directorio
```

Los acentos graves no son siempre ideales para estos tratamientos de comando. En efecto, si trabaja con varios niveles, debe cerrar los que están dentro de los niveles superiores. Asimismo, el bash permite utilizar en su lugar la sintaxis **$(...)**, que no tiene este problema.

```
$ mi_unix=$(uname)
$ echo ${mi_unix}
Linux
$ maquina=$(uname -a | cut -d" " -f5)
echo $maquina
SMP
```

No confunda las llaves y los paréntesis. Las primeras aíslan las variables, mientras que los segundos efectúan la sustitución de los comandos.

5. El programa test

El programa **test** permite evaluar escenarios de programación. Se puede recuperar el resultado de una condición con la variable $? (código retorno). Si el resultado es 0, entonces se ha cumplido la condición.

a. Pruebas en una cadena

- **test -z "variable"**: cero, retorno OK si la variable está vacía (p. ej.: test -z "$a").
- **test -n "variable"**: no cero, retorno OK si la variable no está vacía (texto cualquiera).
- **test "variable"= cadena**: OK si las dos cadenas son idénticas.

- **test "variable" != cadena**: OK si las dos cadenas son diferentes.

```
$ a=
$ test -z "$a" ; echo $?
0
$ test -n "$a" ; echo $?
1
$ a=Julio
$ test "$a" = Julio ; echo $?
0
```

Tenga cuidado con colocar correctamente las variables que contienen texto entre comillas, porque si la variable está vacía el sistema producirá un bug:

```
$ a=
$ b=pepito
$ [ $a = $b ] && echo "ok"
bash: [: =: unary operator expected
```

Mientras que

```
[ "$a" = "$b" ] && echo "ok"
```

no produce errores.

b. Pruebas sobre los valores numéricos

Se deben convertir las cadenas en valores numéricos para poder evaluarlas. Bash sólo gestiona valores enteros. La sintaxis es:

```
test valor1 opción valor2
```

y las opciones son las siguientes:

Opción	Función
`-eq`	Equal: igual
`-ne`	Not Equal: diferente
`-lt`	Less than: inferior
`-gt`	Greater than: superior
`-le`	Less or equal: inferior o igual
`-ge`	Greater or equal: superior o igual

```
$ a=10
$ b=20
$ test  "$a" -ne "$b" ; echo $?
0
$ test "$a" -ge "$b" ; echo $?
1
$ test "$a" -lt "$b" && echo "$a es inferior a $b"
10 es inferior a 20
```

c. Pruebas sobre los archivos

La sintaxis es:

```
test opción nombre_archivo
```

y las opciones son las siguientes:

Opción	Función
`-f`	Archivo normal.
`-d`	Un directorio.
`-c`	Archivo en modo carácter.
`-b`	Archivo en modo bloque.
`-p`	Tubería nombrada (named pipe).
`-r`	Permiso de lectura.
`-w`	Permiso de escritura.
`-x`	Permiso de ejecución.
`-s`	Archivo no vacío (al menos un carácter).
`-e`	El archivo existe.
`-L`	El archivo es un vínculo simbólico.
`-u`	El archivo existe, SUID-Bit activado.
`-g`	El archivo existe, SGID-Bit activado.

```
$ ls -l
-rw-r--r--  1 seb  users    1392 Ago 14 15:55 dump.log
lrwxrwxrwx  1 seb  users       4 Ago 14 15:21 vínculo_fic1 -> fic1
lrwxrwxrwx  1 seb  users       4 Ago 14 15:21 vínculo_fic2 -> fic2
-rw-r--r--  1 seb  users     234 Ago 16 12:20 lista1
-rw-r--r--  1 seb  users     234 Ago 13 10:06 lista2
-rwxr--r--  1 seb  users     288 Ago 19 09:05 param.sh
-rwxr--r--  1 seb  users     430 Ago 19 09:09 param2.sh
-rwxr--r--  1 seb  users     292 Ago 19 10:57 param3.sh
drwxr-xr-x  2 seb  users    8192 Ago 19 12:09 dir1
-rw-r--r--  1 seb  users    1496 Ago 14 16:12 resultado.txt
-rw-r--r--  1 seb  users    1715 Ago 16 14:55 pepito.txt
-rwxr--r--  1 seb  users      12 Ago 16 12:07 ver_a.sh
$ test -f vínculo_fic1 ; echo $?
1
$ test -x dump.log ; echo $?
1
$ test -d dir1 ; echo $?
0
```

d. Pruebas combinadas por criterios Y, O, NO

Puede efectuar varias pruebas con una sola instrucción. Las opciones de combinación son las mismas que para el comando **find**.

Criterio	Acción
`-a`	AND, Y lógico
`-o`	OR, O lógico
`!`	NOT, NO lógico
`( . . . )`	agrupación de las combinaciones. Se deben cerrar los paréntesis \(...\).

```
$ prueba -d "dir1" -a -w "dir1" && echo "dir1: directorio, derecho en escritura"
dir1: directorio, derecho en escritura
```

e. Sintaxis ligera

Se puede sustituir la palabra test por los corchetes abiertos y cerrados **[...]**. Hay que colocar un espacio antes y otro después de los corchetes.

```
$ [ "$a" -lt "$b" ] && echo "$a es inferior a $b"
10 es inferior a 20
```

El bash (y el ksh) integra un comando interno que viene a sustituir al programa test. En la práctica, el comando interno es completamente compatible con el comando externo, pero mucho más rápido, ya que no requiere la creación de un nuevo proceso. Para forzar el uso del comando interno, utilice los dobles corchetes **[[...]]**.

```
$ [[ "$a" -lt "$b" ]] && echo "$a es inferior a $b"
10 es inferior a 20
```

La sintaxis bash permite utilizar el operador ==, pero también && (-a, y) y || (-o, o) dentro de los corchetes:

```
$ i=1 ; j=2 ; [[ $i == 1 && $j == 2 ]] && echo OK
OK
```

6. if ... then ... else

La estructura **if then else fi** es una estructura de control condicional.

```
if <comandos_condición>
then
  <comandos ejecutados si condición cumplida>
elif <condición>
then
  <comandos ejecutados si condición cumplida> else
  <comandos ejecutados si última condición no realizada>
fi
```

Puede especificar **elif**, en realidad un else if. Si no se cumple la última condición, se prueba una nueva.

```
$ cat param4.sh
#!/bin/bash
if [ $# -ne 0 ]
then
      echo "$# parametros en linea de comandos"
else
      echo "Ningun parametro; set alfred oscar romeo zulu"
      set alfred oscar romeo zulu
fi

echo "Numero de parametros: $#"
echo "Parametros: 1=$1 2=$2 3=$3 4=$4"
echo "Lista: $*"

$ ./param4.sh titi pepito
2 parametros en linea de comandos
Numero de parametros: 2
Parametros: 1=pepito 2=titi 3= 4=
Lista: toto titi

$ ./param4.sh
Ningun parametro; set alfred oscar romeo zulu
Numero de parametros: 4
Parametros: 1=alfred 2=oscar 3=romeo 4=zulu
Lista: alfred oscar romeo zulu
```

La sintaxis cercana a C permite utilizar condiciones muy prácticas. Por ejemplo, usted

quiere saber si un archivo contiene un patrón de búsqueda utilizando grep, dentro de una condición if. Aquí hay dos formas de proceder:

```
grep -q 20 input
if [[ $? -eq 0 ]]
then
   echo "encontrado"
fi
```

Probamos el código de retorno del comando **grep**, que devolverá 0 si 20 se encuentra en la entrada. Está obligado a hacerlo en dos líneas y utilizando `-q` (quiet) para esconder la salida estándar. Ahora, la misma operación usando el comando **grep** dentro de una condición if directamente:

```
if grep -q 20 input
then
   echo "encontrado"
fi
```

La expresión devuelve 0 y es interpretada como verdadera por el shell. ¿No es esto más simple?

7. Evaluación múltiple

El comando **case ... esac** permite comprobar el contenido de una variable o de un resultado de manera múltiple.

```
case Valor in
   Modelo1) Comandos ;;
   Modelo2) Comandos ;;
   *) accion_defecto ;;
esac
```

El modelo es o bien un texto simple o bien uno compuesto de caracteres especiales. Cada

bloque de comandos relacionados con el modelo se debe terminar con dos puntos y coma. En cuanto se haya comprobado el modelo, se ejecuta el bloque de comandos correspondiente. El asterisco en la última posición (cadena variable) es la acción por defecto si no se establece ningún criterio; es opcional.

Carácter	Función
*	Cadena variable (incluso vacía)
?	Un solo carácter
[...]	Un intervalo de caracteres
[!...]	Negación del intervalo de caracteres
|	O lógico

```
$ cat case1.sh
#!/bin/bash
if [ $# -ne 0 ]
then
    echo "$# parametros en linea de comandos"
else
    echo "Ningun parametro; set alfred oscar romeo zulu"
    exit 1
fi

case $1 in
    a*)
        echo "Empieza por a"
        ;;
    b*)
        echo "Empieza por b"
        ;;
```

```
        fic[123])
            echo "fic1 fic2 o fic3"
            ;;
        *)
            echo "Empieza por cualquier"
            ;;
esac

exit 0
$ ./case1.sh "adios"
Empieza por a
$ ./case1.sh hola
Empieza por b
$ ./case1.sh fic2
fic1 o fic2 o fic3
$ ./case1.sh error
Enpieza por cualquiera
```

8. Introducción de cadena por el usuario

El comando **read** permite al usuario introducir una cadena y colocarla en una o varias variables. Se valida la cadena con [Intro].

```
read var1 [var2 ...]
```

Si se especifican varias variables, la primera palabra irá en var1; la segunda, en var2, y así sucesivamente. Si hay menos variables que palabras, las últimas palabras van en la última variable.

```
$ cat read.sh
#!/bin/bash
echo "Continuar (S/N)? \c"
read respuesta
echo "repuesta=$respuesta"
```

```
case $respuesta in
        S)
                echo "Sí, seguimos"
                ;;
        N)
                echo "No, paramos"
                exit 0
                ;;
        *)
                echo "Error de inserción (S/N)"
                exit 1
                ;;
esac
echo "Usted siguó. Inserte dos palabras o más :\"
read palabra1 palabra2
echo "palabra1=$palabra1\npalabra2=$palabra2"
exit 0
$ ./read.sh
Continuar (S/N)? S
respuesta=S
Sí, seguimos.
Usted siguió. Inserte dos palabras o más: hola amigos
palabra1=hola
palabra2=amigos
```

9. Los bucles

Permiten la repetición de un bloque de comandos ya sea un número limitado de veces o bien de manera condicional. Todos los comandos que se deben ejecutar en un bucle se colocan entre los comandos **do** y **done**.

a. Bucle for

El bucle **for** no se basa en un incremento de valor cualquiera, sino en una lista de valores, archivos...

```
for var in lista
do
   comandos a ejecutar
done
```

La lista representa un determinado número de elementos que se asignarán a var sucesivamente.

Con una variable

```
$ cat for1.sh
#!/bin/bash
for params in $@
do
      echo "$params"
done
$ ./for1.sh test1 test2 test3
test1
test2
test3
```

Lista implícita

Si no especifica ninguna lista a **for**, entonces es la lista de los parámetros la que está implícita. Ningún script anterior hubiera podido parecerse a:

```
for params
do
      echo "$params"
done
```

Con una lista de elementos explícita

Cada elemento colocado después del «in» se utilizará para cada iteración del bucle, uno tras otro.

```
$ cat for2.sh
#!/usr/bin/sh
for params in lista lista2
do
      ls -l $params
done
$ ./for2.sh
-rw-r--r--   1 oracle   system      234 Aug 19 14:09 lista
-rw-r--r--   1 oracle   system      234 Aug 13 10:06 lista2
```

Con criterios de búsqueda sobre nombres de archivos

Si uno o varios elementos de la lista corresponden a un archivo o a un grupo de archivos presentes en la posición actual de la estructura, el bucle for considera el elemento como un nombre de archivo.

```
$ cat for3.sh
#!/bin/bash
for params in *
do
      echo "$params \c"
      type_fic=`ls -ld $params | cut -c1`
      case $type_fic in
            -)    echo "Archivo normal" ;;
            d)    echo "Directorio" ;;
            b)    echo "modo bloque" ;;
            l)    echo "vinculo simbolico" ;;
            c)    echo "modo caracter" ;;
            *)    echo "otro" ;;
      esac

done
$ ./for3.sh
case1.sh Archivo normal
dump.log Archivo normal
for1.sh Archivo normal
for2.sh Archivo normal
for3.sh Archivo normal
vinculo_fic1 vinculo simbolico
```

```
vinculo_fic2 vinculo simbolico
lista Archivo normal
lista1 Archivo normal
lista2 Archivo normal
param.sh Archivo normal
param2.sh Archivo normal
param3.sh Archivo normal
param4.sh Archivo normal
read.sh Archivo normal
dir1 Directorio
resultado.txt Archivo normal
pepito.txt Archivo normal
ver_a.sh Archivo normal
```

Con un intervalo de valores

Existen dos métodos para contar de 1 a n con un bucle for. El primero consiste en utilizar una sustitución de comando con el comando **seq**. Su sintaxis básica requiere un parámetro numérico y cuenta de 1 al valor de este parámetro. El manual le mostrará que se puede iniciar en cualquier valor e incrementarse con cualquier valor.

```
$ seq 5
1
2
3
4
5
```

Usándolo con el bucle for, obtenemos este resultado:

```
$ for i in $(seq 5); do echo $i; done
1
2
3
4
5
```

El segundo método consiste en utilizar una sintaxis parecida a la del lenguaje C:

```
$ for ((a=1 ; a<=5 ; a++)); do echo $a; done
1
2
3
4
5
```

El tercer método consiste en utilizar sustituciones entre llaves:

```
for a in {1..5}; do echo $a; done
1
2
3
4
5
```

Cualquier comando que produzca una lista de valores puede usarse después de «in» gracias a la sustitución del comando. El bucle for tomará el resultado de este comando como una lista de elementos donde iterar.

Con una sustitución de comando

Se puede colocar cualquier comando que produzca una lista de valores a continuación del «in» con la ayuda de una sustitución de comando. El bucle for tomará el resultado de este comando como lista de elementos con la que iterar.

```
$ cat for4.sh
#!/bin/bash
echo "Lista de los usuarios en /etc/passwd"
for params in `cat /etc/passwd | cut -d: -f1`
do
      echo "$params "
done
$ ./for4.sh
Lista de los usuarios en /etc/passwd
```

```
avahi
bin
chrony
colord
daemon
dnsmasq
firebird
flatpak
gdm...
```

b. Bucle while

El comando **while** permite un bucle condicional «mientras». Mientras se cumpla la condición, se ejecuta el bloque de comandos. Si la condición ya no es válida, se sale del bucle.

```
while condición
do
   comandos
done
```

o:

```
while
bloque de instrucciones que forman la condición
do
   comandos
done
```

Por ejemplo:

```
$ cat while1.sh
#!/bin/bash
while
```

```
echo "¿Cadena? \c"
      read nombre
      [ -z "$nombre" ]
do
      echo "ERROR: no inserción"
done
echo "Usted insertó: $nombre"
```

Lectura de un archivo línea por línea

```
#!/bin/bash
cat pepito.txt | while read line
do
      echo "$line"
done
```

o:

```
#!/bin/bash
while read line
do
        echo "$line"
done < pepito.txt
```

Hay una enorme diferencia entre los dos versiones. En la primera, observe la presencia de la tubería (pipe): se ejecuta el bucle en un segundo proceso. ¡Por lo tanto, cualquier variable modificada en el interior de este bucle pierde su valor a la salida!

c. Bucle until

El comando **until** permite un bucle condicional «hasta». En cuanto se haya cumplido la condición, se sale del bucle.

```
until condición
```

```
do
    comandos
done
```

o:

```
until
bloque de instrucciones que forman la condición
do
   comandos
done
```

d. true y false

El comando **true** no hace más que volver a enviar 0. El comando **false** siempre devuelve 1. De esta manera, es posible realizar bucles sin fin. La única manera de salir de estos bucles es con un exit o un break.

Por convención, cualquier programa que no devuelve error devuelve 0, mientras que cualquier programa que devuelve un error, o un resultado que se debe interpretar, devuelve algo distinto a 0. Es lo contrario en lógica booleana.

```
while true
do
    comandos
    exit / break
done
```

e. break y continue

El comando **break** permite interrumpir un bucle. En este caso, el script continúa su ejecución después del comando **done**. Puede coger un argumento numérico que indique el número de bucles que saltar, en el caso de los bucles anidados (rápidamente ilegible).

```
while true
do
    echo "¿Cadena? \c"
    read a
    if [ -z "$a" ]
    then
       break
    fi
done
```

El comando **continue** permite saltarse líneas de una iteración e ir directamente a la siguiente. Puede coger un argumento numérico, que indica el número de bucles que se deben iniciar de nuevo (se remontan n bucles). El script se ejecuta con el comando **do**.

f. Bucle select

El comando **select** permite crear menús simples, con selección por número. La inserción se efectúa por el teclado con el prompt de la variable PS3. Si el valor insertado no es correcto, se efectúa un bucle y se visualiza de nuevo el menú. Para salir de un select, hay que utilizar un **break**.

```
select variable in lista_contenido
do
     tratamiento
done
```

Si no se especifica `in lista_contenido` , se utilizarán y visualizarán los parámetros de posición.

```
$ cat select.sh
#!/bin/bash
PS3="Su elección:"
echo "¿Qué dato?"
select respuesta in Julio Román Francisco sale
do
```

```
if [[ "$respuesta" = "sale" ]]
        then
            break
        fi
        echo "Usted eligió $respuesta"
done
echo "Adiós."
exit 0

$ ./select.sh
¿Qué dato?
1) Julio
2) Román
3) Francisco
4) sale
Su elección :1
Usted eligió Julio
Su elección :2
Usted eligió Román
Su elección :3
Usted eligió Francisco
Su elección :4
Adiós.
```

10. Las funciones

Las funciones son trozos de scripts con nombre, directamente llamados por su nombre, que pueden aceptar parámetros y devolver valores. Los nombres de funciones siguen las mismas reglas que las variables, excepto que no se pueden exportar.

```
nombre_funcion ()
{
   comandos
   return
}
```

Para tener una mejor visibilidad, es posible añadir la palabra clave `function`:

```
function nombre_función ()
{
  comandos
  return
}
```

Las funciones pueden escribirse o bien en el script actual, o bien en otro archivo que puede incluirse en el entorno. Para ello, teclee:

```
. nombrearchivo
```

El punto seguido de un nombre de archivo carga su contenido (funciones y variables) en el contexto actual.

El comando **return** permite asignar un valor de vuelta a una función. Bajo ningún concepto se debe utilizar el comando **exit** para salir de una función porque esta instrución interrumpe también el script invocante.

```
$ cat funcion
ll ()
{
      ls -l $@
}
li ()
{
      ls -i $@
}
$ . funcion
$ li
 252 case1.sh     326 for4.sh      187 param.sh     897 resultado.txt
 568 dump.log     409 vinculo_fic1  272 param2.sh    991 pepito.txt
 286 funcion      634 vinculo_fic2  260 param3.sh    716 ver_a.sh
 235 for1.sh     1020 lista         42 param4.sh   1008 while1.sh
 909 for2.sh      667 lista1       304 read.sh
 789 for3.sh     1006 lista2       481 dir1
```

11. Cálculos y expresiones

a. expr

El comando **expr** permite efectuar cálculos sobre valores numéricos, comparaciones, así como la búsqueda en cadenas de texto.

Operador	Función
+	Suma.
-	Sustracción.
*	Multiplicación. Como el shell reconoce la estrella como comodín, hay que cerrarla con una contrabarra: *.
/	División.
%	Módulo.
!=	Diferente. Visualiza 1 si diferente, 0 en caso contrario.
=	Igual. Visualiza 1 si igual, 0 en caso contrario.
<	Inferior. Visualiza 1 si inferior, 0 en caso contrario.
>	Superior. Visualiza 1 si superior, 0 en caso contrario.
<=	Inferior o igual. Visualiza 1 si inferior, 0 en caso contrario.
>=	Superior o igual. Visualiza 1 si superior, 0 en caso contrario.
:	Búsqueda en una cadena. P. ej.: expr Julio: J* devuelve 1, ya que Julio empieza por J. Sintaxis particular: expr "Julio": ".*" devuelve la longitud de la cadena.

```
$ expr 7 + 3
10
$ expr 7 \* 3
21
$ a=$(expr 13 - 10)
$ echo $a
3
$ cat expr1.sh
#!/bin/bash
cumul=0
contador=0
nb_bucles=10
while [ "$contador" -le "$nb_bucles" ]
do
      cumul=$(expr $cumul + $contador)
      echo "$cumul=$cumul, bucle=$contador"
      contador=$(expr $contador + 1)
done
$ ./expr1.sh
cumul=0, bucle=0
cumul=1, bucle=1
cumul=3, bucle=2
cumul=6, bucle=3
cumul=10, bucle=4
cumul=15, bucle=5
cumul=21, bucle=6
cumul=28, bucle=7
cumul=36, bucle=8
cumul=45, bucle=9
cumul=55, bucle=10
$ expr "Julio César" : ".*"
11
```

b. Cálculos con bash

El bash propone una forma sencilla de cálculos sobre los enteros colocando la operación entre **$((...))**:

```
$ a=1
$ a=$((a+1))
$ echo $a
2
$ b=2
$ a=$((a*b))
$ echo $a
4
```

No necesita especificar los $ de los nombres de las variables entre los dobles paréntesis.

Finalmente, como puede teclear valores como enteros, puede aplicar directamente sus cálculos sin tomar precauciones particulares:

```
$ typeset -i a
$ a=2+2
$ echo $a
4
```

c. Cálculos de números reales

Ni expr, ni las posibilidades internas de bash permiten efectuar cálculos con números reales. Para esto, debemos utilizar el comando **bc**, un languaje de cálculo. Se trata de un lenguaje de programación completamente aparte. bc escapa del ámbito de este libro, pero sin embargo aquí mostramos como efectuar algunos cálculos simples y comparaciones. El principio es enviar las expresiones de cálculo a la entrada estándar del comando. Observe la presencia del parámetro `-l` que define el uso de una librería matemática que permite el trabajo con números decimales:

```
$ echo "2+2" | bc -l
4
$ echo "1/3" | bc -l
.33333333333333333333
```

Podemos limitar la longitud de las cifras después de la coma (punto decimal) con scale.

```
$ echo "scale=2 ; 1/3" | bc -l
.33
```

Aquí por ejemplo como calcular el porcentaje con dos cifras después de la coma. Sea el siguiente script:

```
BLOCK_USAGE=1245786
BLOCK_LIMIT=1964856
WARNING=60
USAGE_PERCENT=$(echo "scale=2; ${BLOCK_USAGE}*100/${BLOCK_LIMIT}" | bc -l)
echo $USAGE_PERCENT
if (( $(echo "${USAGE_PERCENT}>=${WARNING}" |bc -l) ))
then
   echo "WARNING"
fi
```

Su ejecución mostrará:

```
63.40
WARNING
```

12. Una variable dentro de otra variable

Aquí tiene un ejemplo:

```
$ a=Julio
$ b=a
$ echo $b
a
```

¿Cómo visualizar el valor de a y no su nombre? Utilizando el comando **eval**. Este comando, situado al principio de la línea, intenta interpretar el valor de una variable precedida de dos «$» como si fuera otra variable.

```
$ eval echo \$$b
Julio

$ cat eval.sh
cpt=1
for i in a b c
do
   eval $i=$cpt
   cpt=$((cpt+1))
done
echo $a $b $c

$ ./eval.sh
1 2 3
```

13. Tratamiento de señales

El comando **trap** permite modificar el comportamiento del script en el momento de la recepción de una señal.

Comando	Reacción
trap '' señales	Ignora las señales. trap '' 2 3 ignora las señales 2 y 3.
trap 'comandos' señales	Para cada señal recibida, ejecuta los comandos indicados.
trap señales	Restaura las acciones por defecto para estas señales.

En el ejemplo siguiente, antes de haber mostrado su PID y esperar que se pulse la tecla [Entrar], **trap** impide la ejecución de [Ctrl] **C** (SIGINT) e intercepta la señal SIGTERM:

```
$ cat trap.sh
!#/bin/bash

salir()
{
        echo "Señal 15 recibido"
        exit 0
}

trap '' 2
trap exit 15
echo "Yo soy $$"

read
while true
do
        echo "Ctrl+C imposible!"
done

$ ./trap.sh

Yo soy 12456
Ctrl+C imposible!
Ctrl+C imposible!
Ctrl+C imposible!
Ctrl+C imposible!
Ctrl+C imposible!
Ctrl+C imposible!
Ctrl+C imposible!
Ctrl+C imposible!
Ctrl+C imposible!
Ctrl+C imposible!
...
```

Desde otra consola:

```
$ kill -2 12456 # ningun efecto
$ kill -15 12456 # SIGTERM
En la primera consola:
```

```
Ctrl+C imposible!
Ctrl+C imposible!
Ctrl+C imposible!
Ctrl+C imposible!
Ctrl+C imposible!
Señal 15 recibida
```

14. Comando «:»

El comando «**:**» suele ser totalmente desconocido para los usuarios de Unix. Siempre devuelve el valor 0 (éxito). Por lo tanto, se puede utilizar para sustituir al comando **true** en un bucle, por ejemplo:

```
while:
do
  ...
done
```

Este comando, colocado al principio de línea y seguido de otro comando, trata el comando y sus argumentos, pero no hace nada y siempre devuelve 0. Puede ser útil para probar variables.

```
$ : ls
$ : ${X:?"Error"}
X : Error
```