Tema 8

Programación Shell Scripts

Shell-Script

- Objetivo:

  El shell-script (ó interprete de comandos) es la principal interfase entre el usuario y el S.O.

- Funcionamiento:

  Interpreta las ordenes introducidas por los usuarios y las pasa al S.O para su procesamiento.

- Tipo de comandos:
  - Internos.

  Comandos cuyo código está incluido dentro del interprete (cd, pwd, set,…)
  - Externos

  Se invoca a programas externos al propio interprete (cp, chown, ls, cut,…)

Localización comandos externos

- La búsqueda de los comandos externos se realiza examinando los directorios especificados en la variable de entorno PATH.
  - Consulta ruta de búsqueda:

> echo $PATH
/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin:/usr/openwin/bin:
/usr/local/bin:/usr/ccs/bin:/usr/ucb:

  - Modificación ruta de búsqueda (~ = $HOME):

> PATH=~/bin/:$PATH:.

> echo $PATH
~/bin:/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin:/usr/openwin/bi
n: /usr/local/bin:/usr/ccs/bin:/usr/ucb:.

Variables de entorno del interprete de comandos

- El funcionamiento del interprete de comandos se ve afectado por la configuración de ciertas variables de entorno.
- Principales variables de entorno del shell:
  - PATH **#ruta búsqueda comandos externos**
  - HOME **#directorio inicial del usuario.**
  - USER **#usuario activo.**
  - PWD **#directorio actual**
  - MAIL **#nombre del archivo donde el correo es almacenado**
- Los valores iniciales de estas variables se pueden modificar en los ficheros de arranque del interprete.

Ficheros arranque del shell

- Preparan el entorno de ejecución, que cada usuario necesita para utilizar el sistema.
- Ficheros arranque configuración (interactivo (nuevo terminal) y de login):
  - /etc/profile
  - ~/.bash_profile
  - ~/.bash_login
  - ~/.profile
- Ficheros arranque configuración (interactivo y no login):
  - ~/.bashrc
- Ficheros finalización configuración:
  - ~/.bash_logout

Programación scripts

- La mayoría de los interpretes proporcionan un lenguaje de programación propio que sirve para escribir programas (shell-scripts) capaces de automatizar y realizar distinta tareas.
- Estos scripts (guiones) pueden invocar a cualquier comando que se pueda utilizar en modo interactivo a partir del prompt.
- Existen una serie de comandos adicionales, específicamente diseñados para soportar la programación dentro del shell (variables, estructuras de control, funciones, …).

Formato ficheros script

- Los ficheros de script son ficheros texto con los permisos de ejecución activados:

  chmod 0755 nombre_script

- La primera línea del script se utiliza para especificar la ubicación del interprete de comandos que se debe utilizar para su ejecución:

  #!/bin/bash

  resto_comandos

  .....

- La ejecución de los scripts se realiza de la misma forma que con cualquier otro ejecutable:

  > nombre_script

Variables

- Bash permite la declaración y utilización de variables, definidas como cadenas de caracteres.
- Las variables pueden ser de tres tipos:
  - Variables de entorno.
    - Modificables (vistas en las primeras transparencias)
    - No modificables.
  - Variables definidas por el usuario.
  - Variables de solo lectura (constantes).

Variables de entorno no modificables

- Estas variables pueden ser consultadas pero no modificadas.
- Hacen referencia a información que el programador no podrá modificar directamente durante la ejecución del script:
  - Argumento de línea de órdenes.
    - $0 # Nombre script
    - $1-$9 # Argumentos de entrada 1-9
    - $* # Todos los argumentos.
    - $@ # Todos los argumentos.
    - $# # Número de argumentos.
  - Información relativa al proceso.
    - $$ # Id del proceso actual (script).
    - $! # Id del proceso en background más reciente.
    - $? # Estado de salida de la orden más reciente.

Ejemplo:

- VerArg:

```
#!/bin/bash
echo "Este script, llamado $0, tiene $# argumentos."
echo "Los argumentos son: $*"
ls -la
echo "El ls ha devuelto el siguiente código: $?"
touch /etc/passwd
echo "El cmd touch ha devuelto el siguiente código: $?"
echo "$$: Ejecuto top en background"
top &
echo "$$: Elimino el proceso top, cuyo id es $!"
kill -9 $!
echo "Código kill $?"
exit 0
```

Ejecución ejemplo:

> ./VerArg uno dos 3 4

Este script, llamado ./VerArg, tiene 4 argumentos.
Los argumentos son: uno dos 3 4
total 32
drwxr-xr-x 2 juan eup 4096 Feb 10 20:09 .
drwxr-xr-x 3 juan eup 4096 Feb 10 19:56 ..
-rwxr-xr-x 1 juan eup 351 Feb 10 20:09 VerArg
-rwxr-xr-x 1 juan eup 225 Feb 10 20:04 VerArg~
El ls ha devuelto el siguiente código: 0
touch: cannot change times on /etc/passwd
El cmd touch ha devuelto el siguiente código: 1
29346: Ejecuto top en background
29346: Elimino el proceso top, cuyo id es 29349
Codigo kill 0

Variables definidas por el usuario

- Son variables creadas e inicializadas por el usuario.
- Ámbito de actuación:
  - Locales al programa que las ha creado.
  - Pueden convertirse en variables de entorno mediante su exportación ($ export variable).
- Su declaración no es obligatoria.
- Tipos permitidos:
  - Cadenas
  - Enteros

Por defecto todas las variables son cadenas.

Creación/modificación variables (I)

- Las variables se modifican mediante el operador de asignación.

  <nombre variable>=<valor asignación>

- Ejemplos:

```
> persona=Juan
> echo persona
  persona
> echo $persona
  Juan
> persona="Juan      García      García"
> echo $persona
   Juan      García      García
> echo "$persona"
   Juan García García
> echo '$persona'
  $persona
```

Creación/modificación variables (II)

- Ejemplos:

```
> ficheros=Fic*
> echo $ficheros          # Muestra los ficheros del directorio actual
      Fic1.txt             # que coinciden con el patrón introducido.
      Fic2.out
> echo “$ficheros”
       Fic*
> numero=123              # Los número se gestionan como cadenas
> echo $numero
       123
> declare -i edad=42
> echo $edad
       42
> edad=“joven”
> echo $edad
      0
```

Borrar / visualizar variables

- Las variables existen mientras el shell-script en el que fueron creadas exista.
- Para borrar las variables se les puede asignar un valor nulo ó se puede utilizar el comando unset.
- Para visualizar todas las variables definidas en un script se pueden utilizar los comandos set ó env.
- Ejemplos:

```
> persona=Juan
> echo $persona
  Juan
> unset persona   ó    persona=
> echo $persona
```

Exportar variables

- Las variables locales a un shell-script se pueden exportar para lograr que sean visibles al resto de procesos hijos (comandos ó shell-scripts).
- Sintaxis:

export <variable>

- Ejemplos:

- CamVar:

```
#!/bin/bash
echo “H:  Var=$Var”
Var=Modificada
echo “H:Var = $Var”
exit 0
```

- Principal:

```
#!/bin/bash
Var=Original
echo “P: Var=$Var”
export Var
CamVar
echo “P:Var = $Var”
exit 0
```

Ejecución:

```
Principal
P: Var = Original
H: Var = Original
H: Var = Modificada
P: Var = Original
```

Variables no modificables

- Se puede impedir la modificación del valor de una variable, declarándola como no modificable.
- Se utiliza el comando readonly ó declare -r
- Ejemplos:

```
> persona=Juan
> echo $persona
 Juan
> readonly persona    # o declare -r persona
> persona=Pedro
 bash: persona : readonly variable
```

Vectores / Listas

- Bash soporta el tipo de datos de listas, las cuales son gestionadas como un vector.
- Las listas tienen las siguientes características:
  - Todos los elementos de la lista tienen el mismo tipo.
  - El primer elemento se sitúa en la posición 0.
  - No existe una tamaño limite ó dimensión máxima definida.
  - Los elementos se pueden insertar de forma no contigua.

Definición listas

- Para definir una lista se utiliza la siguiente sintaxis:

variable = (valor1, valor2 ... valorN)

[subindice=]cadena)

- El acceso a los elementos de las listas se realiza especificando el índice entre [].
- Ejemplos:

> películas = (“El silencio de los corderos”, “Dune”, [100]=“Blade Runner”)

> echo ${películas[0]}

El silencio de los corderos

> echo ${películas[100]}

Blade Runner

> echo ${películas[*]}

El silencio de los corderos Dune Blade Runner

Tamaño elementos/lista

- Mediante el operador # se puede acceder al tamaño de la lista ó la longitud de un determinado elemento
- Ejemplos:

```
> echo ${#películas}      # Muestra la longitud del primer
 19                          elemento (el [0] está implícito)

> echo ${#películas[100]}  # Muestra la longitud del elemento
 12                            100.

> echo ${#películas [*]}  # Muestra el tamaño de la lista.
 3
```

Sustitución de comandos

- Asignar a una variable el resultado (salida pantalla) de un comando de UniX.
- La sustitución se puede realizar incluyendo el comando entre ` ` (acento grave) o bien entre $().
- Ejemplos:

```
> diractual=`pwd` # Obtenemos el directorio actual
> echo $diractual
  /home/nando/aso
> echo “El directorio actual es $(pwd)”
   El directorio actual es /home/nando/aso
> procesos=$(ps)
> echo "$procesos"
  PID TTY          TIME CMD
21932 pts/0    00:00:00 bash
21985 pts/0    00:00:00 ps
> procesos=( `ps` )     # El resultado puede asignarse como vector
> echo "${procesos[@]} ”
PID TTY TIME CMD 21932 pts/0 00:00:00 bash 21985 pts/0 00:00:00 ps
```

Comando exec

- Equivalente a las llamadas a sistema execl. Superpone el código del comando especificado sobre el proceso que está ejecutando el shell-script.
- Una vez completado el comando especificado en la orden exec, no es posible volver al proceso que hizo la llamada y por lo tanto el resto del código del script no se ejecuta.
- Sintaxis:

exec comando

- Ejemplo fecha:

```
#!/bin/bash
echo “Inicio $0”
exec date
echo “Final $0”
```

```
> fecha
“Inicio fecha”
 lun feb 13 19:14:21 CET 2011
```

Operaciones aritméticas

- Los valores de las variables de los shell se almacenan en forma de cadenas.
- Para poder realizar operaciones aritméticas ó lógicas es preciso su traducción a entero (largo), realizar la operación y volver a convertir el resultado a una cadena.
- Existen varias maneras para realizar de forma automática este proceso:
  - Orden let.
  - Expansión $((<expresión>))
  - Orden expr.

Operadores aritméticos

- Bash soporta los siguientes operadores (ordenados según la precedencia de mayor a menor):
  - - , + menos, más unario (-3,+5).
  - !, ~ Not lógico, complemento a dos (! "$A" –eq ~3).
  - ** Exponente (A= 2**4)
  - *, /, % Multiplicación, división y resto.
  - +, - Suma y resta.
  - <<, >> Desplazamientos bit izquierda y derecha ($A<<2)
  - <=, >=, <> Comparadores mayor,menor, distinto.
  - ==, != Comparadores igual y distinto.
  - & AND bit a bit.
  - ^ OR exclusivo bit a bit (XOR)
  - | OR bit a bit
  - && AND lógico
  - || OR lógico
  - =, +=, *=, /=, %=, &=, ^=, |=, <<=, >>=

Comando let

- Permite evaluar expresiones aritméticas, devolviendo un 1 si el resultado de la expresión es nulo y un cero en caso contrario.
- Sintaxis:

let <lista-expresiones>

- Ejemplos:

> let “a = 8” “b = 13”
> let c=a+b
> echo $c
21

Expansión $(())

- Permite evaluar y devolver la expresión circunscrita entre los paréntesis interiores.
- Sintaxis:

  $((expresión))
- Ejemplos:

```
> a=8 b=13
> echo “a + b es igual a $((a+b))”
  a + b es igual a 21
> c=21
> d=$((c<<2))  # desplazamiento de 2 bits a la izquierda (= *4)
> echo $d
 84
```

Comando expr

- También se puede utilizar el comando expr para evaluar una expresión.
- Sintaxis:

Expr [substr | index | length] args

- Ejemplo:

```
> var1=10
> var1=`expr “$var1” + 1`
> echo $var1
 11
```

Manipulación cadenas (I)

- Bash permite diversas operaciones con variables de tipo cadena:
  - *${#cadena}*

Muestra la longitud cadena.

> cadena="abcABC123ABCabc"
> echo ${#cadena}
15

  - *${cadena:pos}*

Extrae la sub-cadena a partir de la posición pos.

> echo ${cadena:7}
23ABCabc

  - *${cadena:pos:long}*

Extrae la sub-cadena formada por long caracteres a partir de la posición pos.

> echo ${cadena:6:4}
123A

Manipulación cadenas (II)

- *${cadena#patron}*
- *${cadena## patron}*

Corta la cadena a partir del punto final de la subcadena más corta (#) / más grande (##) que concuerda con el patrón introducido.

```
> echo ${cadena#a*C}
  123ABCabc
> echo ${cadena##a*C}
  abc
```

- *${cadena%patron}*
- *${cadena%%patron}*

Corta la cadena (desde el final de la cadena al principio) a partir del punto final de la subcadena más corta (%) / más grande (%%) que concuerda con el patrón introducido.

```
> echo ${cadena%b*c}
  abcABC123ABCa
> echo ${cadena%%b*c}
  a
```

Manipulación cadenas (III)

- *${cadena/subcadena/reemplazo}*

  Reemplaza la primera ocurrencia de subcadena con la cadena de reemplazo.

  > echo ${cadena/abc/xyz}

  xyzABC123ABCabc

- *${cadena//subcadena/reemplazo}*

  Reemplaza todas las ocurrencias de subcadena con la cadena de reemplazo.

  > echo ${cadena/abc/xyz}

  xyzABC123ABCxyz

Comando Set

- Asigna a los valores de los argumentos de comando de línea ($1-$9) a los argumentos con que fue llamado set.
- ***Sintaxis:***

  set [-- aefhkntuvx] lista-argumentos
- Ejemplos:
  - PrSet:

```
#!/bin/bash
set do re mi
echo $3 $2 $1
set $(date)
echo $@
echo “$3”
```

Ejecución:

```
> PrSet
mi re do
Wed Feb 11 13:24:52 MET 2008
11
```

Lectura entrada estándar

- El comando read puede utilizarse para leer información del usuario y asignarla a variables del script.
- Permite distribuir la entrada del usuario entre una ó mas variables.
- Sintaxis:

read <lista_variables>

- Ejemplo:
- Lectura

```
#!/bin/bash
echo “Nombre y apellidos: ”
read Nom Apellidos
echo -n “Comando: ”
read cmd
$cmd
echo “Gracias $Apellidos, $Nom”
ls
```

Ejecución:

```
> Lectura
Nombre y apellidos:
Juan García García
Comando: pwd
/home/Juan/Pruebas
Gracias García García, Juan
LISTADO FICHEROS
```

Estructuras de control de flujo del shell-script

- Las estructuras de control de flujo permiten alterar el orden de ejecución de los comandos dentro de un script
- Aunque todos los interpretes de comandos soportan las mismas estructuras de control, el formato puede variar de uno a otro.
- Bash incluye las siguientes estructuras:
  - ***IfThen***
  - ***Case***
  - ***For***
  - ***While***
  - ***Until***

Estructura If Then

- Permite la ejecución condicional de un código en función del valor devuelto por la expresión condicional.
- *Sintaxis:*

  if <***expresión_test***>
  then
  comandos
  fi

- La expresión condicional se evalua a partir del resultado de un comando ó mediante el comando test.

Comando Test

- El comando test permite evaluar una determinada condición, proporcionado un resultado de verdadero (0) ó falso (distinto de 0).
- Se pueden utilizar los siguientes formatos:

test <expresión>

[<expresión>]

- La expresión a evaluar permite comparar condiciones sobre valores enteros, cadenas, ficheros ó unir lógicamente varias expresiones.

Operadores de comparación I

- De archivos. Devuelven true si:
  - *-e archivo* el *archivo* existe
  - *-r archivo* el *archivo* puede leerse
  - *-w archivo* el *archivo* puede escribirse
  - *-x archivo* el *archivo* puede ejecutarse
  - *-f archivo* el *archivo* existe y es un archivo regular
  - *-d archivo* el *archivo* existe y es un directorio
  - *-h,-L archivo* el *archivo* existe y es un enlace simbólico
  - *-c archivo* el *archivo* existe y es un archivo especial de caracteres
  - *-b archivo* el archivo existe y es un archivo especial de bloques
  - *-p archivo* el *archivo* existe y es un pipe

Operadores de comparación II

- Cadenas:
  - *s1* s1 no es una cadena nula
  - *-z s1* longitud de la cadena *s1* es cero
  - *-n s1* longitud de la cadena *s1* no es cero
  - *s1 = s2* cadenas *s1* y *s2* son iguales
  - *s1 != s2* cadenas *s1* y *s2* son diferentes
- Números:
  - n1 -eq n2 los enteros n1 y n2 son iguales
  - n1 -ne n2 los enteros n1 y n2 no son iguales
  - n1 -gt n2 n1 es más grande que n2
  - n1 -ge n2 n1 es más grande ó igual a n2
  - n1 -lt n2 n1 es más pequeño que n2
  - n1 -le n2 n1 es más pequeño ó igual que n2

Operadores de comparación III

- Mediante los siguientes operadores se pueden componer expresiones de comparación más complejas:
  - *!* Negación
  - *(condición)* paréntesis para agrupar expresiones
  - *-a* operador lógico and
  - *-o* operador lógico or
- Ejemplos:

```
[ "$a" -gt "$b" -a -z "$c" ]  # La variable a es mayor que b
                              # y la variable c es una cadena vacía
[ ! -n “$var” ]    # La variable var es una cadena vacía
[ -f f1 -a -w f1 ]  # f1 es un fichero y puede escribirse
[ f1 -nt f2 ]   # El fichero f1 es más nuevo que f2
[ f1 -ot f2 ]   # El fichero f1 es más viejo que f2
[ f1 -ef f2 ]   # f1 y f2 son enlaces físicos a un mismo fichero
```

Otras estructuras condicionales

- Sintaxis IfElse:

```
if <expresión>
    then
        comandos
    else
        comandos
fi
```

- Sintaxis IfElif:

```
if <expresión>
 then
     comandos
    elif <expresión>
        then
     comandos
    else
        comandos
fi
```

Ejemplos

```
#!/bin/bash
if test $# -lt 2     # Comprobamos parámetros de entrada.
    then
     echo  “Se necesitan al menos dos parámetros”
     exit -1
fi
if [ $1 = $2 ]      # Comprobamos que no sean el mismo fichero.
    then
     echo “Los dos ficheros son iguales”
     exit -2
    elif cmp $1 $2    # Comando comparación ficheros.
     then
          echo "Los dos ficheros son iguales (Cod ret == 0)."
     else
          echo "Los dos ficheros son diferentes (Cod ret != 0)."
fi
exit 0
```

Estructura case

- Permite implementar un flujo alternativo con múltiples vías similar a múltiples if anidados.
- Sintaxis:

```
case test-string in
 patrón -1)
            comandos1
            ;;
 patrón - 2)
            comandos2
            ;;
 patrón - 3)
            comandos3
            ;;
 esac
```

Case: formato patrones

- Los patrones de cada una de las opciones del case siguen la mismas reglas de reconocimiento de ficheros.
- Se admiten los siguientes caracteres especiales:
  - * Identifica a cualquier carácter ó cadena.
  - ? Identifica que esa posición puede ser sustituida por cualquier carácter.
  - [...] Identifica un rango de valores.
  - | Identifica caracteres/cadenas alternativas (equivalente operador or)
- Ejemplos:

[0-9]|[a-f]|[A-F]) **# Digito hexadecimal.**

fic*.txt) **# Nombre ficheros con extensión txt y que empiecen por “fic”**

Ejemplo Case

- **EjemCase**

```
#!/bin/bash
echo “d: Ver fecha y hora actual”
echo “l: Listado archivos dir actual”
echo “q: Para salir de este programa”
echo “Elija opción:”
read opcion
case “$opcion” in
    d|D)  date
      ;;
    l|L)   ls
      ;;
    q|Q)  exit 0
      ;;
    *)      echo “Opción incorrecta”
      exit 1
      ;;
esac
```

```
> EjemCase
    d: Ver fecha y hora actual
    l: Listado archivos dir actual
    q: Para salir de este programa
    Elija opción:
    D
    Thu Feb 12 13:51:45 WET
2004
>
```

Estructura For

- La estructura for permite implementar un bucle recorriendo todas las palabras almacenadas en una lista de argumentos.
- Sintaxis 1:

  for **<variable>** **[**in ***<lista-argumentos>***]**(*)
  do
  comandos
  done

  (*) Si no se especifica la lista de argumentos en el for se asume los parámetros de entrada del script.
- Sintaxis 2:

  for **((inicialización; condición_fin; incremento))**
  do
  comandos
  done

Ejemplos For

- **EjemFor1**

```
#!/bin/bash
for i in 1 2 3 4 5
    do
    echo “Bienvenido $i veces“
done
for ((  i=1 ;  i <= 5;  i++  ))
    do
    echo “Adios $i veces“
done
exit 0
```

> EjemFor1 a b c

```
Bienvenido 1 veces
Bienvenido 2 veces
Bienvenido 3 veces
Bienvenido 4 veces
Bienvenido 5 veces
Adios 1 veces
Adios 2 veces
Adios 3 veces
Adios 4 veces
Adios 5 veces
```

Ejemplos For anidados

```
#!/bin/bash
if [ $# -eq 0 ]
then
    echo "Error - Falta número“; exit 1
fi
## Bucle externo ##
for (( i = 1; i <= $1; i++ ))
do
    ## Bucle interno ##
    for (( j = 1 ; j <= 10; j++ ))
    do
        echo -n "`expr $i \* $j`"
    done
    echo ""
done
exit 0
```

Estructura While

- Ejecuta las ordenes del bucle mientras el resultado de evaluar la expresión siga siendo cierto.
- Sintaxis:

```
while <expresión>
do
  comandos
done
```

- Se pueden alterar las iteraciones de los bucles mediante los comandos:
  - break #Rompe el bucle.
  - continue #Se salta el resto de código de la # iteración actual de un bucle.

Ejemplos While

- **EjemWhile**

```
#!/bin/bash
if [ $# -eq 0 ]
then
    echo "Error - Falta número“
    exit 1
fi
n=$1
i=1
while [ $i -le 10 ]
do
    echo "$n * $i = `expr $i \* $n`“
    i=`expr $i + 1`
done
```

```
> EjemWhile 5
5 * 1 = 5
5 * 2 = 10
5 * 3 = 15
5 * 4 = 20
5 * 5 = 25
5 * 6 = 30
5 * 7 = 35
5 * 8 = 40
5 * 9 = 45
5 * 10 = 50
```

Estructura Until

- Ejecuta las ordenes del bucle mientras el resultado de evaluar la expresión sea falso.
- Sintaxis:

until ***<expresión>***
do
comandos
done

Ejemplos Until

```
#!/bin/bash
if [ $# -eq 0 ]
then
echo "Error - Falta numero"; exit 1
fi
echo "Intente adivinar un número entre 1-10"
Numero=$RANDOM
echo "RANDOM: $Numero"
let "Numero=(Numero%10) + 1"
echo "Numero: $Numero"
Veces=0
Intento=0
until [ "$Intento" -eq "$Numero" -o "$Veces" -ge $1 ]
do
echo "Introduzca un número: "
read Intento
let "Veces=$Veces+1"
echo "Veces = $Veces"
done
if [ "$Numero" -eq "$Intento" ]
then echo "Muy bien, lo acertaste!!"
else echo "Lo siento, vuelve a intertalo!!"
fi
```

Funciones

- La utilización de funciones en scripts es útil para acceder a código que se utiliza múltiples veces.
- Antes de poder utilizar una función, ésta debe estar definida (tanto el nombre como el código).
- Las funciones solo se pueden invocar después de que el script haya ejecutado el código correspondiente a su definición.

Ejemplo: **Prog**:

```
#!/bin/bash
funcion1
funcion1()
{
    echo "Prueba función 1"
}
funcion1
exit 0
```

Ejecución:

```
> Prog
./Prog: line 3: funcion1: command   not found
Prueba función 1
```

Funciones: Paso parámetros

- Las funciones reciben los parámetros de llamada utilizando el mismo mecanismo que los shell-scripts.
  - `$#  # Número de parámetros.`
  - `$1-$9  # Parámetros 1 a 9`
- Todos los parámetros se pasan por valor.
- El comando shift nos permite desplazar los parámetros una posición a la izquierda:

  $1<--$2, $2<--$3, ect…

Funciones: valores retorno

- Las funciones puede devolver valores, equivalentes al estado de salida (exit) de los comandos / programas.
- Se especifica el valor numérico que debe devolver la función mediante la sentencia return.
- El valor que devuelve una función se puede consultar mediante la variable de entorno $?
- Para devolver un valor no numérico se debe utilizar una variable previamente definida fuera de la función.

Ejemplo función:

```
#!/bin/bash
Maximo()
{
    mayor=0
    while [ $# -ne 0 ]
    do
     echo ”Procesando parámetro $1."
     if [ "$1" -ge "$mayor" ]
          then
          mayor=$1
     fi
     shift
    done
    return $mayor
}
Maximo 2 12 4 24 1
echo "Máximo entre 2 12 4 24 1 es: $?"
Maximo
echo "Máximo entre nada es: $?"
exit 0
```

- **Ejecución**

> ./func

Procesando parámetro 2.
Procesando parámetro 12.
Procesando parámetro 4.
Procesando parámetro 24.
Procesando parámetro 1.
Máximo entre 2 12 4 24 1 es: 24
Máximo entre nada es: 0

Listas de comandos

- Se permite lanzar la ejecución encadenada de una serie de comandos.

  La ejecución del siguiente comando de la lista depende del resultado devuelto por los anteriores y del tipo de lista utilizado.
- Listas comandos and:

  El siguiente comando se ejecuta si todos los anteriores devuelven cierto (han tenido éxito). Se finaliza la ejecución de los comandos cuando uno de ellos falla.

  Sintaxis: cmd1 && cmd2 && .. && cmdN
- Listas comandos or:

  El siguiente comando se ejecuta si todos los anteriores devuelven falso (no han tenido éxito). Se finaliza la ejecución de los comandos cuando uno de ellos tiene exito.

  Sintaxis: cmd1 || cmd2 || .. || cmdN

Ejemplo Lista comandos:

- **EjemCmdList**

```
#!/bin/bash
if [ ! -z "$1" ] && echo "Arg1 = $1" && \
   [ ! -z "$2" ] && echo ”Arg2 = $2”
then
    # Todos los cmds de la lista devuelven 0.
    echo ”Existen al menos 2 argumentos”
else
    echo ”No hay al menos 2 argumentos”
    exit 1
fi
[ "$1” == “a” ] && echo ”Arg1 correcto”
[ -f "$2" ] || echo ”Fichero $2 no existe.”
# Si existe el fichero comprimimos.
[ -f "$2" ] && ( tar -zcvf $2$(date +%F).tar.gz
    $2 ) && echo "fichero $2 comprimido"
exit 0
```

- **Ejecución:**

```
> EjemCmdList a
    Arg1=a
    No hay al menos 2 argumentos
> EjemCmdList a fic
    Arg1 = a
    Arg2 = fic
    Existen al menos 2 argumentos
    Arg1 correcto
```

Se crea el fichero fic11-02-04.tar.gz

Ejemplo:

- **EjemCmdList2**

```
#!/bin/bash
if [ ! -z "$1" ] && echo "Existe argv1 y es $1"
then
        echo "Existe argv1"
fi
[ -z "$2" ] && echo "No Existe argv2"
[ "$2" != "a" ] && echo "argv2 incorrecto"
[ ! -f "$2" ] && echo "No existe fichero $2"
[ ! -f "$2" ] || ( tar -zcvf $2$(date +%F).tar.gz $2 ) && echo "fichero $2
    comprimido"
exit 0
```

Depuración scripts

- Los shell-scripts se pueden depurar utilizando la opción -v ó la opción -x del bash.
  - -v:

Muestra cada línea del script antes de su ejecución y el resultado de dicha ejecución.

  - -x:

Muestra las líneas del script, después de que se haya realizado la sustitución de variables pero antes de su ejecución. También muestra el resultado de la ejecución.

Ejemplo depuración -v:

```
$ bash -v EjemCmdList2 a PrSet
#!/bin/bash
if [ ! -z "$1" ] && echo "Existe argv1 y es $1"
then
echo "Existe argv1"
fi
Existe argv1 y es a
Existe argv1
[ -z "$2" ] && echo "No Existe argv2"
[ "$2" != "a" ] && echo "argv2 incorrecto"
argv2 incorrecto
[ ! -f "$2" ] && echo "No existe fichero $2"
[ ! -f "$2" ] || ( tar -zcvf $2$(date +%F).tar.gz $2 ) && echo "fichero $2 comprimido"
date +%F
a PrSet
fichero PrSet comprimido
exit 0
```

Ejemplo depuración -x:

```
$ bash -x EjemCmdList2 a PrSet
+ '[' '!' -z a ']'
+ echo 'Existe argv1 y es a'
Existe argv1 y es a
+ echo 'Existe argv1'
Existe argv1
+ '[' -z PrSet ']'
+ '[' PrSet '!=' a ']'
+ echo 'argv2 incorrecto'
argv2 incorrecto
+ '[' '!' -f PrSet ']'
+ '[' '!' -f PrSet ']'
++ date +%F
+ tar -zcvf PrSet2011-11-20.tar.gz PrSet
a PrSet
+ echo 'fichero PrSet comprimido'
fichero PrSet comprimido
+ exit 0
```

Redirecciones

- En los shell-script se permite utilizar los operadores de redirección para redirigir la entrada/salida estándar ó la salida error de los comandos que se utilicen en los programas de script.
- El descriptor 1 representa la salida estándar y el descriptor 2 la salida de error.
- Operadores de redirección:
  - **>, >>** Redirección salida estándar de un comando hacia un fichero.
  - **<** Redireccionar fichero hacia entrada estándar de un comando
  - **2>,2>>** Redirección salida de error (stderr) de un comando hacia un fichero.

Ejemplos redirecciones

```
#!/bin/bash
# Redireccionamos la salida estándar del comando ls al fichero
# listado.txt
ls –l > listado.txt

# Redireccionamos la salida de error del comando grep al fichero
# err.txt
grep da * 2> err.txt

# Redireccionamos la salida estándar del comando grep a la salida
# de error
grep da * 1>&2

# Elimina todos los ficheros de core del sistema y eliminamos la
# salida de error del comando.
rm –f  $(find / -name core) 2> /dev/null
```

Tuberías

- Las tuberías nos van a permitir utilizar la salida de un comando como entrada de otro.
- Ejemplos:

```
# Listamos ficheros, de los cuales nos quedamos con
# aquellos que tienen la extensión txt y posteriormente
# los ordenamos
ls | grep ".txt" | sort

# Borramos ficheros core y ordenamos los ficheros de
# core eliminados
rm –f $(find / -name core) | sort > fics.txt
```

Ejemplo redirección (II)

- **VisUsuarios**

```
#!/bin/bash
# Visualizar usuarios
# definidos en el sistema.
IFS=:
while read nombre password
      uid resto
do
    echo -n "USUARIO con
    UID=$uid,
    Nombre=$nombre, "
    echo "resto=$resto"
done </etc/passwd
exit 0
```

> VisUsuarios | head -6

```
USUARIO con UID=0, Nombre=root,
resto=0:root:/root:/bin/bash
USUARIO con UID=1, Nombre=bin,
resto=1:bin:/bin:
USUARIO con UID=2, Nombre=daemon,
resto=2:daemon:/sbin:
USUARIO con UID=3, Nombre=adm,
resto=4:adm:/var/adm:
USUARIO con UID=4, Nombre=lp,
resto=7:lp:/var/spool/lpd:
USUARIO con UID=5, Nombre=sync,
resto=0:sync:/sbin:/bin/sync
USUARIO con UID=6,
Nombre=shutdown,
resto=0:shutdown:/sbin:/sbin/shutdown
```

Documento Here

- Esta característica de bash permite redirigir la entrada estándar de un comando dentro de un shell-script.
- Sintaxis:

```
comando << [-] marcador_entrada
  datos insertados
marcador_entrada
```

- Ejemplo:

```
#!/bin/bash
mail –s “Prueba Mail” $1 << MARCADOR
  Hola $1,
  Este es un e-mail de prueba generado por un shell-script.
  Disculpa las molestias.
MARCADOR
echo “Correo enviado a la dirección $1”
exit 0
```

Comandos

- Existen infinidad de comandos de UNIX que pueden ser de gran utilidad a la hora de realizar un script:
  - head
  - tail
  - cat
  - paste
  - sort
  - cut
  - find
  - grep
  - sed
  - awk

Comandos head/tail

- El comando head se utiliza para mostrar las primeras n líneas de un archivo.
- El comando tail permite mostrar las últimas n líneas de un archivo.
- Sintaxis:

  head [-cn]... [fichero]…

  tail [-cn]... [fichero]...
- Ejemplos:

  # Visualizar las primeras 10 líneas del fichero /etc/passwd:

  > head /etc/passwd

  # Visualizar los primeros 20 caracteres:

  > head -c20 /etc/passwd

  # Mostrar 5 últimas líneas fichero /etc/passwd:

  > tail –5 /etc/passwd

  # Mostrar las líneas del 7 al 9 de los ficheros de un directorio:

  > ls –l dir | head –9 | tail –3

Comando cat

- El comando cat permite concatenar diversos archivos, enviando el resultado a la salida estándar.
- Cuando solo se especifica un único archivo, el comando permite mostrarlo por pantalla
- Ejemplos:

```
# Concatenar ficheros f1, f2, f3 y el resultado se almacena
# en el fichero f4
> cat f1 f2 f3 > f4
# Visualizamos las últimas 15 líneas de un fichero
> cat fich | tail –15
```

Comando paste

- El comando paste permite unir archivos horizontalmente (imprime las líneas de cada uno de los ficheros implicados, separadas por tabuladores y terminadas por un salto de línea).

  paste [-s] [-d delim-list] [file...]

- Ejemplo:

```
> cat f1.txt
11 22 33
11 22 33
> cat f2.txt
aa bb cc
AA BB CC
> paste f1.txt f2.txt
11 22 33        aa bb cc
11 22 33        AA BB CC
```

```
> paste -s f1.txt f2.txt
11 22 33        11 22 33
aa bb cc        AA BB cc

> cat f1.txt f2.txt
11 22 33
11 22 33
aa bb cc
AA BB cc
```

Comando sort

- El comando **sort** permite ordenar las líneas de un archivo.

  sort [-nrtk]... [fichero]...
- A la hora de realizar la ordenación se pueden escoger diversas opciones:
  - Alfabética (por defecto) o numérica (-n)
  - Orden ascendente (por defecto) o descendente (-r).
  - Seleccionar separador de campos (-t).
  - Seleccionar los campos utilizados para realizar la ordenación (-k c_inicio, c_final).
- Ejemplos:

```
sort arch1 > arch2          # Ord. alfabética-ascendente
sort -n arch1 >arch2        # Ord. numérica-ascendente
sort -nr arch1 >arch2       # Ord. Numérica-descendente.
sort -t: -k 2,3 arch1 >arch2 # Ord. campos  2 a 3, separador
                             # de campos ":"
```

Comando cut

- El comando **cut** se utiliza para separar de cada línea de un archivo uno ó más campos (columnas), ó partes del archivo.
- Los campos se delimitan, por defecto, por un tabulador, pero se permite especificar el separador de campos.
- Sintaxis:

  cut [-fcd]... [fichero]...
- Ejemplos:

  cut -f2 fic1 # Se muestra el segundo campo

  cut -c1-6 fic1 # Muestra las primeras 6 columnas (cars)

  # Mostar la versión del SO y el kernel

uname -a | cut -d" " -f1,3,11,12 **# -d" " significa separador de campos es " "**

Comando find

- Permite localizar archivos dentro del árbol de directorios de Linux que cumplan ciertas características.
- Sintaxis:

  find [camino...] [expresión]
- Ejemplos:

  # Buscar en el directorio actual todos los archivos o directorios de nombre perdido.

  > find . -name perdido

  # Busca, a partir del directorio /usr/people, todos los archivos que terminen en f

  > find /usr/people -name '*.f' -print

Comando Grep

- El comando **grep** permite buscar patrones dentro de ficheros e imprimir todas las líneas que contengan dicho patrón.
- Sintaxis:

  grep [-bchilnsvw] expresión-regular [fichero...]

  grep [-bchilnsvw] -e lista_patrones [-f fic_patrones] ... [fichero...]
- Principales opciones:
  - -c Solo imprimir el número de líneas que contienen el patrón.
  - -i No distinguir entre mayúsculas y minúsculas.
  - -n Anteponer a cada línea coincidente su número de línea dentro del fichero.
  - -v Imprimir las líneas que no contengan el patrón
  - -f fichero Fichero de patrones.

Grep: ejemplos

```
# Buscamos la entrada correspondiente al usuario root en el fichero de
    contraseñas.
> grep root /etc/passwd
root:x:0:1:Super-User:/:/sbin/sh
# Imprimir todas los procesos que contienen fernando
> ps -ef | grep fernando
fernando 11097 10928  0 13:15:04 pts/2    0:00 bash
fernando 10928 10924  0 12:46:09 pts/2    0:00 -sh
fernando 10627   148  0 11:29:38 ?        0:03 imapd
> ps -ef | grep -c fernando
4
>  ps -ef | grep -n fernando
48:fernando 11097 10928  0 13:15:04 pts/2    0:00 bash
50:fernando 10928 10924  0 12:46:09 pts/2    0:00 -sh
53:fernando 11248 11097  0 13:34:29 pts/2    0:00 bash
72:fernando 10627   148  0 11:29:38     ?       0:03 imapd
```

Expresiones regulares (I)

- Los dos siguientes comandos que vamos a tratar se basan en la utilización de expresiones regulares para la búsqueda de patrones.
- Admiten los siguientes caracteres especiales:
  - ^ Concordar con el inicio de línea
  - $ Concordar con el final de línea
  - . Representa un único carácter.
  - (car)* Concuerda con un número arbitrario de los caracteres (car)
  - (car)? Concuerda con 0 ó una instancia de (car)
  - [...] Concuerda con alguno de los caracteres dentro de los corchetes. Soporta rangos de caracteres.
  - [^...] Concuerda con alguno de los caracteres no incluido dentro de los corchetes.

Expresiones regulares (II)

- Repetición caracteres:
  - \<

Concordancia al inicio de una palabra

  - \>

Concordancia al final de una palabra

  - (carácter)\{m,n\}

Concuerda con m a nrepeticiones del carácter.

  - (carácter)\{m,\}

Concuerda con m ó más repeticiones del carácter.

  - (carácter) \{1,n\}

Concuerda con n ó menos repeticiones del carácter.

  - (carácter) \{,n\}

Concuerda con n repeticiones del carácter.

  - ()

Operador de grupo.

Ejemplos expresiónes regulares

Obtener el número de líneas de un fichero sin contar las líneas en blanco.

> grep –nv ‘^$’ fic

Listar solo los directorios.

> ls -l | grep ‘^d’

Buscar todos los ficheros con extensión diferente de txt

> ls -l | grep '[^\.txt]$‘

Obtener las líneas que al menos tienen 9 letras minúsculas consecutivas.

> grep ‘[a-z]\{9\}’ fic

Otros comandos

- wc
  Imprime el número de carácteres, líneas de un fichero.
- basename
  Obtiene el nombre del comando de un camino.
- dirname
  Obtiene la ruta del camino un comando.
- nohup
  Ejecución ininterrumpida de un comando.
- trap
  Ejecuta una función/comando cuando se recibe una señal.
- Ejemplos:

```
> wc -l /ect/password    # Resultado:  59 /etc/passwd
> basename /bin/ls       # Resultado: ls
> dirname /bin/ls        # Resultado: /bin
> nohup ls &          # Coloca la salida en nohup.out
# Deshabilitar el control-C (señal 2)
> trap ‘echo ”Control−C deshabilitado.“’ 2
```