PROGRAMACIÓN DEL SHELL

1 INTRODUCCIÓN

Recordemos que es el shell: es un programa que se ejecuta automáticamente cuando se inicia una sesión UNIX. Su función principal es la de servir de interlocutor entre el núcleo del sistema operativo y el usuario: se encarga de leer cada orden que se teclea e interpretar lo que se ha solicitado.

Una característica añadida del shell es que funciona como un lenguaje de programación: permite crear ficheros de programas o procesos shell. El contenido de estos ficheros es:

* Comandos UNIX.
* Variables.
* Comandos del lenguaje de programación shell.

Antes de pasar a desarrollar cada parte, vamos a ver las distintas formas de ejecutar un proceso shell.

2 CÓMO EJECUTAR UN PROCEDIMIENTO SHELL

Existen tres formas básicas:

- **sh** *nombre del proceso shell*
- Se le activa al archivo que contiene el proceso shell el permiso de ejecución. Una vez hecho esto, basta con escribir el nombre del fichero para que el proceso shell se ejecute.
- **.** *nombre del proceso shell*

Las dos primeras formas son equivalentes: cada vez que se ejecuta el proceso se abre un nuevo shell que nace y muere con el proceso. Si se ejecuta de la tercera forma esto no ocurre: se ejecute desde el mismo shell que se abrió con la sesión de trabajo, o sea, desde el que se invocó al proceso. El resultado más visible de estas dos formas de operar se produce sobre las variables, ya que estas sólo tienen valor dentro del shell en el que fueron definidas. Por lo tanto, si ejecutamos el proceso de cualquiera de las dos primeras formas, las variables nacen y mueren con el proceso. Si se ejecuta de la tercera forma descrita, las variables definidas en el proceso permanecerán activas hasta que cerremos la sesión de trabajo (mueren con el shell).

3 COMENTARIOS EN PROGRAMAS SHELL

Un comentario en un proceso shell se identifica porque comienza por el carácter #. Desde ese carácter hasta el final de la línea, es ignorado por el shell. Es conveniente comentar un

programa de cara a su mejor legibilidad.

Ej

```
# Guion: practica1
# Autor: Pepe Pérez
# Fecha: 09-03-2005
# Ultima modificacion: 16-03-2005
# Objetivo: Distinguir las tres formas de ejecución de un guión o shell
# Descripcion: asignar valor a  variables y mostrarlas
echo dentro
var1=hola
echo $var1
exit
var2=adios
echo $var2
echo fuera
```

4 VARIABLES

Existen dos tipos de variables:

* Las variables de entorno.
* Argumentos para los procedimientos shell.

4.1 Variables de entorno

•**Asignación de valor**. Sintaxis: nombre de variable=valor.

• **Uso de la variable**. El valor asignado puede recuperarse precediendo el nombre de la variable con el signo $.

Estas variables pueden crearse y usarse tanto dentro de un proceso shell, como en el modo interactivo desde el prompt del sistema.

Ej.

```
$ fruta=pera
$ comida=sopa
$ echo $fruta $comida
pera sopa
$
```

Una variable sólo tendrá valor dentro del proceso en el cual fue creada.

Para observar el valor de todas las variables definidas en un determinado instante, hay que ejecutar la orden **set**.

4.1.1 Asignaciones especiales

* El valor asignado a una variable puede intervenir en otra variable. Es aconsejable en este caso, y a veces necesario[1], identificar el nombre de esta variable encerrándola entre corchetes.

Ej. *$ preposicion=para*
$ objeto=${preposicion}caidas
$ echo $objeto
paracaidas
$

* Cuando en el valor asignado existan varias palabras separadas por espacios, hay que usar comillas para preservar estos espacios en la definición de la variable

Ej. *$ s="ls -l"*
$ $s /usr
obtendremos el listado largo del contenido del directorio /usr
$

* Se puede asignar como valor de una variable, la salida de una orden UNIX. En este caso, después del igual se encierra la orden UNIX entre caracteres acento grave (`).

Ej. *$ hoy=`date`*
$ echo $hoy
Mon Jan 16 17:30:19 MET 1995
$

4.1.2 Asignación externa a variables del shell: lectura de valores con **read**

Los valores para las variables del shell se pueden capturar a través del teclado o de otros ficheros mediante la sentencia **read**, con un funcionamiento similar a la lectura de una variable en cualquier lenguaje de programación.

$ echo “Dame un valor: “
$ read entrada
$ echo “He leido $entrada”

También se pueden leer varios datos en varias variables dentro de una misma sentencia **read**. Tan sólo hay que poner las variables una tras otra en el mismo mandato **read**:

[1] Será necesario cuando no haya otra forma de diferenciar a la variable del resto de la cadena de caracteres.

$ echo “Dame ahora dos valores: “
$ read entrada1 entrada2
$ echo “He leido dos entradas: $entrada1 y $entrada2”

El alcance de una variable es la del guión en la que está definida (o en el shell en el que esté definida). Si queremos que una variable esté definida para todo el entorno tendremos que utilizar el mandato **export.**

4.1.3 Variables shell predefinidas

El shell tiene predefinidas una serie de variables de entorno. Algunas de ellas pueden modificarse, asignándolas otro valor distinto al de por defecto. Otras son de sólo lectura, pueden se usadas, pero no modificadas. Las variables predefinidas modificables más interesantes son:

* HOME. Su valor por defecto es la ruta del directorio del usuario. Es el argumento por defecto de la orden cd.
* PATH. Es el grupo de directorios donde el sistema busca los ficheros ejecutables.
* PS1. Define la cadena de caracteres que aparecen como prompt del sistema. Por defecto su valor es $.

4.2 **Argumentos para los procedimientos shell**

Es un tipo de variables que se transmite al proceso shell cuando este es invocado. Es un argumento para el proceso.

- **Asignación de valo**r. Son argumentos que se añaden a la derecha del nombre del proceso cuando éste es invocado.

- **Uso de la variable**. Son referenciadas dentro del proceso shell como:

  * $1 para el primer parámetro, aquel que se encuentra justo a la derecha del nombre del proceso shell.
  * $2 para el segundo parámetro.
  * $3 para el tercer parámetro.
  * Y así sucesivamente.

*Ej. Supongamos que tenemos un proceso que se llama **mostrar**. Si el proceso se invoca:*

mostrar Pedro Juan Pepe

Entonces:

** El valor del primer argumento será Pedro y se referencia como $1.*
** El valor del segundo argumento será Juan y se referencia como $2.*
** El valor del tercer argumento será Pepe y se referencia como $3.*

*Si el contenido del proceso **mostrar** es, por ejemplo:*

```
echo argumento1=$1
echo argumento2=$2
echo argumento3=$3
```

Al ejecutarle de la forma indicada, el resultado será la siguiente salida por pantalla:

```
argumento1=Pedro
argumento2=Juan
argumento3=Pepe
```

El nombre del propio proceso es considerado un argumento y puede ser referenciado con $0.

Hay un límite de nueve argumentos que pueden ser referenciados: de $1 a $9. Sin embargo, existe la orden shift que produce un desplazamiento hacia la derecha en la referenciación de los argumentos:

* Si ejecuto una vez shift, la referenciación de los argumentos es:

  - $1 para el segundo.
  - $2 para el tercero.
  - ...

* Si la ejecuto dos veces, la referenciación será:

  - $1 para el tercero.
  - $2 para el cuarto.
  - ...

* Y así sucesivamente.

De esta forma es posible escribir un procedimiento que pueda tratar más de 9 argumentos. Otro modo de acceder a todos los argumentos es con $*. Esta variable se expande a todos los argumentos especificados cuando se invocó el procedimiento shell.

El parámetro $# toma el valor del número total de argumentos especificados.

Ej. Si invocamos al proceso mostrar de la forma indicada, $#=3.

5 COMANDOS DEL LENGUAJE DE PROGRAMACIÓN SHELL

Los distintos comandos que veremos en este apartado, además de formar parte de un

proceso shell almacenado en un fichero, pueden ser escritos y ejecutados desde el prompt del sistema. Las peculiaridades de cada uno, al ser usados de esta forma, serán descritas en el apartado correspondiente.

5.1 **echo**

Sintaxis: **echo** *mensaje*.

Muestra por pantalla la cadena de caracteres especificada en *mensaje*.

5.1.1 Uso de echo con variables

En cualquier posición del *mensaje* puede ir referenciada una cualquiera de las variables vistas. En su uso normal la variable será sustituida por su valor. Si se requiere que esto no ocurra, basta con encerrar el *mensaje* entre comillas simples (').

5.1.2 Secuencias de caracteres especiales

El *mensaje* puede incluir las siguientes de caracteres especiales:

* \c. Normalmente echo termina su salida con una nueva línea. Esta secuencia lo evita. Hay que encerrar *mensaje* entre comillas.

Ej. Si el contenido de proceso es:

```
echo "hola \c"
echo María
```

Su ejecución, produciría la siguiente salida pro pantalla:

```
hola María
```

* \n. Produce un cambio de línea y un retorno de carro. Hay que encerrar *mensaje* entre comillas.

```
Ej.   $ echo "mi nombre es:\nCarlos"
      mi nombre es:
      Carlos
      $
```

* \t. Produce un salto de tabulación. Hay que encerrar *mensaje* entre comillas.

```
Ej.   $ echo "mi nombre es:\tCarlos"
      mi nombre es:        Carlos
      $
```

* \$. Cuando queremos que aparezca el carácter $, hay que usar esta secuencia para evitar que $ sea interpretado como el símbolo que sirve para referenciar una variable.

Ej. *$ precio=10*
$ echo el precio del producto es de \$ $precio
el precio del producto es de $ 10

5.2 **Ejecución condicional con if**

Se utiliza para la ejecución condicional de un conjunto de ordenes: si se cumple una determinada condición se ejecutan, si no se cumple no se ejecutan. Sintaxis:

if *condición*
then
Grupo de órdenes a ejecutar si la condición se cumple
fi

Aunque no tiene mucho sentido, normalmente, su ejecución desde el prompt del sistema, sirve para ilustrar el proceso, ya que éste es bastante común, como veremos en siguientes sentencias. Una vez escrito desde el prompt la orden if con la condición, aparece una cadena secundaria de petición de orden (>). Se sigue escribiendo el resto de la sentencia, y en el momento que se introduzca la palabra clave de fin de sentencia if, o sea, fi, el shell la ejecutará. Una vez acabada esta ejecución se vuelve al prompt del sistema.

5.2.1 Especificación de la condición. Orden **test**

Para especificar la condición se utiliza la orden test. Básicamente, los argumentos de test forman una expresión, si la expresión es cierta, test devuelve el valor 0; si la comprobación falla, la orden devuelve un valor distinto de 0.

Hay tres clases principales de comprobaciones a efectuar:

* test sobre valores numéricos.
* test sobre ficheros.
* test sobre cadenas de caracteres.

En cualquiera de los tres casos, los distintos valores se pueden referenciar mediante variables. En este caso, si la variable puede no tener ningún valor (un argumento inexistente, por ejemplo) es aconsejable encerrarla entre comillas, si no, el sistema dará un mensaje de error en el test, en caso de suceder la situación citada. Aunque no sea necesario en todas las ocasiones, lo mejor para evitar errores es poner siempre las comillas.

- test sobre valores numéricos. Examina la relación entre dos números. La forma general de la expresión comprobada es:

test *N <primitiva> M*

Donde *N* y *M* son los valores numéricos a comprobar y las primitivas que pueden usarse son:

* -eq N = M.
* -ne N ≠ M.
* -gt N > M.
* -lt N < M.
* -ge N ≥ M.
* -le N ≤ M.

Ej.

```
usuarios=`who | wc -l`
if test "$usuarios" -gt 8
  then
      echo más de 8 personas conectadas
fi
```

Desde el prompt del sistema:

```
$ usuarios=`who | wc -l`
$ if test "$usuarios" -gt 8
> then
>echo más de 8 personas conectadas
> fi
```

(si hay más de 8 pesonas conectadas saldrá el mensaje, en caso contrario no habrá salida por pantalla de ningún mensaje)

```
$
```

El shell solo trata números enteros. Los números 1.0 y 1.3, por ejemplo, serán iguales.

• test sobre ficheros. Se refiere a la existencia o no de ficheros y de las propiedades de éstos. La forma general de la expresión de la condición es:

test <*primitiva*> *nombre del fichero*

Las primitivas más comunes empleadas en este tipo de comprobaciones son:

* -s Verifica que el fichero existe y no está vacío.
* -f Verifica que el fichero es ordinario (no es directorio).
* -d Verifica que el fichero es un directorio.
* -w Verifica que el fichero puede ser escrito.
* -r Verifica que el fichero puede ser leído.

Ej. Supongamos que el primer argumento introducido al invocar un determinado proceso ha de ser un directorio. Lo primero que habría que hacer antes de usarlo es comprobar se no se ha producido ningún error en la ejecución del proceso y el valor de ese primer argumento es un directorio válido. Esto se haría de la siguiente forma:

```
if test -d "$1"
  then
      echo directorio válido
      ls -l $1
fi
```

En la práctica suele ser más útil analizar condiciones de error. Para negar el sentido dado a las verificaciones anteriores, se emplea el signo de admiración, que es el operador unario de negación. Se sitúa antes de la primitiva a negar:

test ! *<primitiva> nombre del fichero*

Ej. En el ejemplo anterior el análisis de si se ha cometido un error sería:

```
if test ! -d "$1"
  then
      echo directorio no valido
fi
```

- test sobre cadena de caracteres. Se subdividen en dos grupos:

* Comprobaciones que comparan dos cadenas de caracteres. Sintaxis:

test *S <primitiva> R*

Donde S y R son dos cadenas de caracteres. Hay dos primitivas que pueden aplicarse:

* = Comprueba que las cadenas son iguales.
* != Comprueba que las cadenas no son iguales.

Ej. Un determinado proceso queremos que sólo se ejecute cuando el primer parámetro tiene un valor determinado, que suponemos es "permiso". Esto se haría de la siguiente manera, donde al principio se comprueba la existencia del primer argumento:

```
if test "$1" = ""
  then
      echo hay que introducir un argumento al ejecutar el proceso
      exit
fi
if test $1[2] != permiso
  then
      echo la palabra clave no era correcta. permiso de ejecución denegado
      exit
fi
(resto de ordenes del proceso que se ejecutan si el primer argumento es permiso)
```

* Comprobaciones para verificar la presencia o ausencia de una cadena. El formato de estas expresiones es:

test *<primitiva> S*

[2] En este caso, no es necesario encerrar $1 entre comillas, puesto que nunca llegamos a este punto del proceso si la variable tiene valor nulo, o sea, si no se introduce un parámetro. No pasaría nada si se pusieran.

Donde *S* es una cadena de caracteres. Las primitivas disponibles son:

* -z Comprueba si la cadena *S* tiene longitud cero.
* -n Comprueba si la cadena *S* tiene longitud mayor que cero.

Ej. La primera comprobación del ejemplo anterior (si se había introducido o no el primer argumento) se puede hacer también de las siguientes maneras:

a)
```
if test -z "$1"
  then
       echo hay que introducir un argumento al ejecutar el proceso
       exit
fi
...
```

b)
```
if test ! -n "$1"
  then
      echo hay que introducir un argumento al ejecutar el proceso
      exit
fi
...
```

• Combinación de test. Operadores -a y -o. Los operadores -a y -o permiten combinar varias expresiones a comprobar en una orden test única. El -a representa a la operación "y" lógica: el resultado de la comprobación es cierto si ambas expresiones son ciertas. El -o representa a la operación "o" lógica: el resultado de la comprobación es cierto si alguna de las dos expresiones (o ambas) es cierta.

Ej. Las dos comprobaciones a realizar en los ejemplos anteriores se reduciría a una, usando esta combinación de comprobaciones.

```
if test -z "$1" -o "$1" != permiso
  then
      echo permiso denegado
      exit
fi
(resto de ordenes del proceso que se ejecutan si el primer argumento es permiso)
```

5.2.2 La sentencia **else**

Añadida a la sentencia if, permite la ejecución condicionada de uno de entre dos grupos de ordenes. Si una condición se cumple, se ejecutará un grupo de órdenes, si no se cumple el otro. Sintaxis:

if *condición*

```
then
    Grupo de órdenes que se ejecutan si la condición se cumple.
else
    Grupo de órdenes que se ejecutan si la condición no se cumple.
fi
```

Ej. Antes de listar un directorio introducido como parámetro comprobamos que este existe.

```
if test -z "$1" -o ! -d "$1"
 then
    echo primer argumento inexistente, o directorio no válido
 else
    ll $1
fi
```

5.2.3 La sentencia **elif**

Es una combinación de else e if: si no se cumple la condición, se comprueba una nueva condición antes de ejecutar un determinado grupo de órdenes. En realidad no es necesaria la introducción de esta nueva sentencia, ya que se pueden anidar varias sentencias condicionales if, pero aligera notación.

Ej. El ejemplo anterior, para diferenciar la situación de error cuando no existe el primer argumento, de cuando existe pero no es válido, se podría realizar de la siguiente manera:

```
if test "$#" -eq 0
 then
    echo primer argumento inexistente
 else
    if test ! -d "$1"
      then
         echo el directorio no es válido
      else
         ll $1
    fi
fi
```

Esto mismo se puede hacer usando la sentencia elif que aligera notación:

```
if test "$#" -eq 0
 then
    echo primer argumento inexistente
 elif test ! -d "$1"
    then
       echo el directorio no es válido
```

```
    else
        ll $1
fi
```

5.3 **Ejecución selectiva usando la sentencia case**

Cuando dependiendo del valor de una determinada variable, hay que elegir entre varios grupos de órdenes a ejecutar, habría que anidar varias sentencias if-else. El uso de la sentencia case facilita esta tarea. Sintaxis:

case *cadena* **in**
*cadena_1***)** *si "cadena" es igual a "cadena_1" ejecutar todas estas órdenes hasta ";;", e ignorar el resto de los casos* **;;**
*cadena_2***)** *si "cadena" es igual a "cadena_2" ejecutar todas estas órdenes hasta ";;", e ignorar el resto de los casos* **;;**
. . .
esac

Se puede hacer uso de las capacidades de correspondencia de patrones del shell aplicando los metacaracteres ? y *.

Ej. Supongamos que tenemos un proceso shell personalizado, esto es, al ser invocado ha de incluirse como primer argumento el usuario del proceso. Dependiendo de quien sea éste el proceso realiza una serie de operaciones diferentes. Si el argumento no es reconocido como usuario permitido, se indica esta eventualidad y se finaliza la ejecución:

```
case $1 in
juan)
        echo hola juan
        (resto de ordenes para ese usuario);;
pedro)
        echo hola pedro
        (resto de ordenes para ese usuario);;
... (resto de usuarios permitidos)

*)
        echo usuario desconocido. Permiso de ejecución denegado;;
esac
```

5.4 **Formación de bucles con la sentencia for**

Produce la ejecución repetida de un conjunto de órdenes. Sintaxis:

for *variable* **in** *lista de valores*
do
    grupo de órdenes a ejecutar
done

Donde:

* *variable*, es el nombre con se referencia a cada uno de los valores indicados en *lista de valores*, dentro de las órdenes especificadas entre **do** y **done**. Donde se quiera usar su valor, la variable se referencia de la forma ya conocida: $*variable*.
* **do**, indica el principio del bucle.
* **done**, indica el final del bucle.

El grupo de órdenes especificadas entre **do** y **done**, se ejecuta para cada uno de los valores indicados en *lista de valores*. La ejecución de esta sentencia, al igual que el resto de bucles que veremos, desde el prompt del sistema, puede resultar un tarea de utilidad práctica; no es infrecuente encontrarnos con la necesidad puntual de ejecutar una serie de operaciones de forma repetitiva. El proceso que se sigue para su definicióin y ejecución es similar al referido en el caso de la sentencia if: una vez introducida la primera línea, aparece el prompt secundario (>), éste se cierra en cuanto al shell se le indique el final del bucle mediante la palabra clave done; en este momento el bucle es ejecutado, y a su finalización se retorna al prompt del sistema.

Ej.

```
 for fichero in *.txt *.datos
do
     sort $fichero > $fichero.ord
     echo el fichero $fichero ha sido ordenado
done
```

Ejecución desde el prompt del stma.:

```
$ for fichero in *.txt *.datos
> do
> sort $fichero > $fichero.ord
> echo el fichero $fichero ha ...
> done
( se ejecuta el bucle)
$
```

Este proceso ordenará el contenido de todos los ficheros con extensiones txt y datos del directorio actual. El fichero ordenado se almacena con el mismo nombre que el original pero añadiendo la extensión ord. Cada vez que se ordena un fichero, sale un mensaje indicando tal eventualidad.

Se permite la anidación de bucles.

5.4.1 Alteración de la ejecución de un bucle con **break** y **continue**

Estas dos sentencias pueden usarse para suspender un bucle incondicionalmente.

La sentencia **continue** no suspende el bucle, hace que el resto de las órdenes del bucle sean ignoradas, y vuelve al comienzo del bucle para efectuar una nueva iteración.

Ej. Supongamos que queremos crear un proceso que nos permite ver el contenido de unos determinados ficheros introducidos como argumentos al invocar al proceso. Antes de esto comprobaremos que el fichero existe, de no ser así, iremos al siguiente. Fijamos el número

máximo de argumentos a 3.

```
for fichero in $1 $2 $3
do
    if test ! -s "$fichero"
      then
          echo el fichero $fichero no existe
          continue
    fi
    cat $fichero
done
```

La sentencia **break** suspende el bucle por completo.

Ej. Queremos crear un proceso que ordene el contenido de tres ficheros introducidos como argumentos. El contenido de los tres ficheros ordenados debe ser almacenado en un único fichero llamado tres.ord. Si alguno de los tres ficheros no existe debe abandonarse la operación, indicando esta situación y borrando, si se creo, el fichero tres.ord.

```
for archivo in $1 $2 $3
do
    if test ! -s "$archivo"
      then
          echo $archivo archivo inexistente
          rm tres.ord
          break
      else
         sort $archivo >> tres.ord
    fi
done
```

5.5 **Formación de bucles con la sentencia while**

Se ejecuta un determinado grupo de órdenes mientras se cumpla una condición especificada. Sintaxis:

while *condición*
do
    Grupo de órdenes a ejecutar mientras la condición se cumpla
done

La condición se expresa de la misma forma que en el caso de la sentencia if. La alteración de la ejecución del bucle while se puede realizar, de la misma forma que en bucle for, con las sentencias **continue** y **break**. Su ejecución desde el prompt del sistema sigue un proceso idéntico al referido en el caso del bucle for.

Ej. While permite la ejecución de una operación sobre un número indefinido de argumentos:

```
while test  -n "$1"
do
     if test ! -s "$1"
       then
           echo $1 fichero inexistente
           shift
           continue
       else
           cat $1
           shift
     fi
done
```

5.6 **Creación de bucles con la sentencia until**

Se ejecuta un determinado grupo de órdenes hasta que se cumpla una condición especificada. Sintaxis:

until *condición*.
do
    Grupo de órdenes que se ejecutan hasta que se cumpla la condición
done

Tiene las mismas características que el bucle while, en cuanto a expresión de la condición, ruptura del bucle y ejecución desde el prompt del sistema.

Ej El ejemplo anterior se podría realizar mediante un bucle until, de la siguiente manera:

```
# comienzo del bucle. Se ejecuta hasta que se hayan procesado todos los
# argumentos
until test  -z "$1"
do
# antes de operar se comprueba si el fichero pasado como argumento existe
     if test ! -s "$1"
       then
           echo $1 fichero inexistente
           shift
       else
# si el fichero existe, la operación sobre él del proceso es ver su contenido
           cat $1
           shift
     fi
done
```

5.7 **Ejecución de expresiones aritméticas. Sentencia expr**

La orden expr evalúa sus argumentos considerándolos como una expresión matemática y escribe el resultado sobre la salida estándar. Sintaxis:

expr *expresión aritmética*

Los operadores aritméticos que pueden utilizarse son:

+	suma
-	resta
*	producto. Debido a que este símbolo tiene un significado especial para el shell (metacarácter), cuando se emplee hay que hacerlo de cualquiera de las siguientes formas: *, "*" o '*'.
/	cociente
%	resto

Cada operador y cada operando forma un argumento distinto para expr, y por ello debe de haber espacios de separación entre éstos.

Ej.
```
$ expr 13 + 49
62
$ a=3
$ expr $a \* 10 / 2
15
$
```

6 **EL PROCESO SHELL .PROFILE**

Una de las últimas operaciones que realiza el sistema al conectarse un usuario, antes de que aparezca el prompt de petición de orden, es buscar en el directorio del usuario un fichero denominado **.profile**, si lo encuentra lo ejecuta automáticamente. En esta característica radica la importancia de este fichero: permite ejecutar ficheros y comandos, cada vez que un usuario se conecta al sistema.

En este fichero se sitúan, entre otras, operaciones tan importantes como:

- Definir el tipo de terminal usado.
- Definir la variable PATH.

7 **FUNCIONES SHELL**

Una función es similar en su efecto y contenido a un proceso shell, es decir, permite designar un grupo de órdenes del sistema y sentencias del lenguaje de programación shell, que se ejecutan cuando sean invocadas mediante un nombre único en la línea de ordenes. La diferencia radica en que, mientras un procedimiento shell debe estar contenido en un fichero, una función

puede estar definida simplemente para el shell, en este caso, morirá su definición con éste (de forma similar a lo que ocurre con las variables).

La sintaxis de definición de una función shell es:

nombre de la función **( )**
{
ordenes que queremos que se ejecuten al invocar a la función
}

Esta definición se puede realizar dentro de un fichero, o bien desde el prompt del sistema. En el primero de los casos para que la definición se active para el shell abierto al conectarse al sistema, el fichero que la contenga tiene que ejecutarse de la forma: **.** *fichero.* En el segundo caso aparecerá, una vez tecleado "*nombre de la función* **( )**", una cadena secundaria de petición de orden (>). Esta cadena desaparece, y vuelve el prompt del sistema al indicar el fin de la definición de la función con el símbolo "cerrar llave" (**}**) (es la secuencia de operaciones típica). Una vez hecho esto ya se está en disposición de ejecutar la función invocándola por su nombre La definición hecha de este modo es el caso, ya indicado, en el que la función muere con el shell, o sea, al cerrar la sesión en la que se definió.

Ej. *$ prueba( )*
> {
> echo ha ejecutado la función prueba
>}
$ prueba
ha ejecutado la función prueba
$

Para ver las definiciones de funciones activas, se emplea el mismo comando que para las variables: **set**.

El uso de funciones en vez de procedimientos no es realmente adecuado para especificar largas listas de órdenes del sistema, pero puede emplearse con ventaja en lugar de un procedimiento shell sencillo. Nos permite, por ejemplo, redefinir comandos, o evitar el engorro de tener que estar introduciendo largas listas de argumentos en órdenes de gran uso.

BIBLIOGRAFÍA

- Rachel Morgan y Henry McGilton, “Introducción al UNIX Sistema V”, Mc Graw Hill.
- Manual del sistema operativo.
- Syed M. Sarwar, Robert Koretsky, Syed A. Sarwar, “El libro de UNIX”, Addison Wesley, 2001.