SEMINARIO 06053

PROGRAMACIÓN AVANZADA EN SHELL (shellscripts)

Ramón M. Gómez Labrador

(ramon.gomez@eii.us.es)

Abril de 2.005

Nota importante: El presente seminario se oferta dentro del plan de formación para personal informático de la Universidad de Sevilla para el año 2006 y toda su documentación asociada está bajo licencia Creative Commons con reconocimiento (http://creativecommons.org/licenses/by/2.5/deed.es).

1ª edición: Curso 03-55 Programación Avanzada en *Shell*, octubre 2.003.
2ª edición: Curso 05-08 Programación Avanzada en *Shell*, abril 2.005.
Esta 3ª edición divide el curso en 2 seminarios:

1. Seminario 06013 Programación Avanzada en *Shell* (línea de comandos), septiembre 2.006.
2. Seminario 06053 Programación Avanzada en *Shell* (*shellscripts*), septiembre 2.006.

06013 Programación Avanzada en *Shell* (*shellscripts*)

ÍNDICE

1. Introducción

1.1. La línea de mandatos.

La línea de mandatos es el interfaz del usuario con el sistema, que permite personalizar el entorno de operación y ejecutar programas y guiones.

El formato típico de una línea consta de una orden y unos modificadores y parámetros opcionales, aunque puede incluir algunos caracteres especiales, que modifican el comportamiento típico.

```
Mandato [Modificador ...][Parámetro ...]
```

Un caso especial es el de las líneas que comienzan por la almohadilla (**#**), que se consideran comentarios. También puede insertarse un comentario en mitad de una línea, a la derecha de una orden.

```
# Esto es un comentario
ls -al    # lista el contenido del directorio actual
```

Pueden agruparse varias órdenes en la misma línea separadas por el punto y coma (**;**), que se ejecutan siempre en secuencia. Asimismo, si un mandato es muy largo o engorroso, puede usarse el carácter de escape (\) para continuar escribiéndolo en la línea siguiente.

```
cd /var/log; grep –i error *
find /tmp /var/tmp ! –user root –type f \
     -perm +2 –print
```

1.2. Edición y ejecución de un guión.

Un guión interpretado por BASH es un fichero de texto normal que consta de una serie de bloques de código formados por líneas de mandatos que se ejecutan en secuencia. El usuario debe tener los permisos de modificación (escritura) en el directorio –para crear un nuevo programa– o sobre el propio fichero, para modificar uno existente.

Como un programa binario, el usuario debe tener permiso de ejecución en el fichero del guión, y se lanza tecleando su camino completo junto con sus opciones y parámetros. Asimismo, como veremos en el Capítulo 4, si el programa se encuentra en un directorio incluido en la variable de entorno **PATH**, sólo se necesita teclear el nombre, sin necesidad de especificar el camino.

El proceso completo de edición y ejecución de un guión es el siguiente:

```
vi prueba.sh                    # o cualquier otro editor de textos
chmod u+x prueba.sh             # activa el permiso de ejecución
./prueba.sh                     # ejecuta el guión
prueba.sh                       # si está en un directorio de $PATH
```

Existe una manera especial para ejecutar un guión, precediéndolo por el signo punto, que se utiliza para exportar todas las variables del programa al entorno de ejecución del usuario (para más información sobre las variables de entorno revisar el Capítulo 4). El siguiente ejemplo ilustra el modo de ejecutar

```
apachectl start                 # Ejecución normal de un guión.
. miprofile                     # Ejecución exportando las variables.
source miprofile                # Equivalente a la línea anterior.
```

Un "*script*" puede –y debe– comenzar con la marca **#!** para especificar el camino completo y los parámetros del intérprete de mandatos que ejecutará el programa. Esta marca puede usarse para ejecutar cualquier intérprete instalado en la máquina (BASH, BSH, PERL, AWK, etc.).

El siguiente cuadro muestra un pequeño ejemplo de guión BASH.

```
#!/bin/bash
# ejemplo1: informe de la capacidad de la cuenta

echo "Usuario: $USER"
echo "Capacidad de la cuenta:"
du –hs $HOME          # suma total del directorio del usuario
```

1.3. Recomendaciones de programación.

Como cualquier otro programa, un guión BASH puede requerir un cierto mantenimiento, que incluya modificaciones, actualizaciones o mejoras del código. Por lo tanto, el programador debe ser precavido y desarrollarlo teniendo en cuenta las recomendaciones de desarrollo típicas para cualquier programa.

Una práctica ordenada permite una verificación y comprensión más cómoda y rápida, para realizar las modificaciones de forma más segura y ayudar al usuario a ejecutar el programa correctamente. Pare ello, seguir las siguientes recomendaciones.

- El código debe ser fácilmente legible, incluyendo espacios y sangrías que separen claramente los bloques de código
- Deben añadirse comentarios claros sobre el funcionamiento general del programa principal y de las funciones, que contengan: autor, descripción, modo de uso del programa, versión y fechas de modificaciones.

- Incluir comentarios para los bloques o mandatos importantes, que requieran cierta aclaración.
- Agregar comentarios y ayudas sobre la ejecución del programa.
- Depurar el código para evitar errores, procesando correctamente los parámetros de ejecución.
- No desarrollar un código excesivamente enrevesado, ni complicado de leer, aunque ésto haga ahorrar líneas de programa.
- Utilizar funciones y las estructuras de programación más adecuadas para evitar repetir código reiterativo.
- Los nombres de variables, funciones y programas deben ser descriptivos, pero sin confundirse con otros de ellos, ni con los mandatos internos o externos; no deben ser ni muy largos ni muy cortos.
- Todos los nombres de funciones y de programas suelen escribirse en letras minúsculas, mientras que las variables acostumbran a definirse en mayúsculas.

2. Expansiones.

Como se ha observando en el seminario anterior, la línea de comandos se divide en una serie de elementos que representan cierto significado en la semántica del intérprete. La **expansión** es un procedimiento especial que se realiza sobre dichos elementos individuales.

BASH dispone de 8 topos de expansiones, que según su orden de procesado son [2]:

- **Expansión de llaves**: modifica la expresión para crear cadenas arbitrarias.
- **Expansión de tilde**: realiza sustituciones de directorios.
- **Expansión de parámetro y variable**: tratamiento general de variables y parámetros, incluyendo la sustitución de prefijos, sufijos, valores por defecto y otros operaciones con cadenas.
- **Sustitución de mandato**: procesa el mandato y devuelve su salida normal.
- **Expansión aritmética**: sustituye la expresión por su valor numérico.
- **Sustitución de proceso**: comunicación de procesos mediante tuberías con nombre de tipo cola (FIFO).
- **División en palabras**: separa la línea de mandatos resultante en palabras usando los caracteres de división incluidos en la variable `IFS`.
- **Expansión de fichero**: permite buscar patrones con comodines en los nombres de ficheros.

El resto del capítulo describe cada uno de los tipos de expansiones.

2.1. Expansión de llaves.

La expansión de llaves es el preprocesado de la línea de comandos que se ejecuta en primer lugar y se procesan de izquierda a derecha. Se utiliza para generar cadenas arbitrarias de nombre de ficheros, los cuales pueden o no existir, por lo tanto puede modificarse el número de palabras que se obtienen tras ejecutar la expansión.

El formato general es el siguiente:

Formato	Descripción
[*Pre*]**{***C1*,*C2*[,...]**}**[*Suf*]	El resultado es una lista de palabras donde se le añade a cada una de las cadenas de las llaves –y separadas por comas– un prefijo y un sufijo opcionales.

Para ilustrarlo, véanse los siguientes ejemplos.

```
echo a{b,c,d}e             # ➔ abe ace ade
mkdir $HOME/{bin,lib,doc}  # Se crean los directorios:
                           # $HOME/bin, $HOME/lib y $HOME/doc.
```

Ejercicio 2.1: interpretar la siguiente orden.

```
mkdir ${HOME}/{bin,lib,doc}
```

2.2. Expansión de tilde.

Este tipo de expansión obtiene el valor de un directorio, tanto de las cuentas de usuarios, como de la pila de directorios accedidos. Los formatos válidos de la expansión de tilde son:

Formato	Descripción
~[*Usuario*]	Directorio personal del usuario indicado. Si no se expresa nada **$HOME**.
~+	Directorio actual (**$PWD**).
~-	Directorio anterior (**$OLDPWD**).

Véase este pequeño programa:

```
#!/bin/bash
# capacidad – muestra la capacidad en KB de la cuenta del
#             usuario indicado
# Uso:  capacidad usuario

echo "Usuario: $1"
ls –ld ~$1
du –hs ~$1
```

Es recomendable definir un alias en el perfil de entrada del usuario para cambiar al directorio anterior, ya que la sintaxis del comando es algo engorrosa. Para ello, añadir la siguiente línea al fichero de configuración **~/.bashrc**.

```
alias cda='cd ~-'
```

2.3. Expansión de parámetro o de variable.

Permite la sustitución del contenido de la variable siguiendo una amplia variedad de reglas. Los distintos formatos para la expansión de parámetros son [1]:

Formato	Descripción
${!*Var***}**	Se hace referencia a otra variable y se obtiene su valor (**expansión indirecta**).
${*Parám***:-***Val***}**	Se devuelve el parámetro; si éste es nulo, se obtiene el **valor por defecto**.
${*Parám***:=***Val***}**	Si el parámetro es nulo se le **asigna el valor por defecto** y se expande.
${*Parám***:?***Cad***}**	Se obtiene el parámetro; si es nulo se manda un **mensaje de error**.
${*Parám***:+***Val***}**	Se devuelve el **valor alternativo** si el parámetro no es nulo.
${***Parám***:***Inic***}** **${*Parám***:***Inic***:***Long***}**	**Valor de subcadena** del parámetro, desde el punto inicial hasta el final o hasta la longitud indicada.
${!*Pref****}**	Devuelve los nombres de variables que empiezan por el prefijo.
${#***Parám***}** **${#*Matriz***[*]}**	El **tamaño** en caracteres del parámetro o en elementos de una matriz.
${***Parám***#***Patrón***}** **${*Parám***##***Patrón***}**	Se elimina del valor del parámetro la mínima (o la máxima) comparación del patrón, comenzando por el principio del parámetro.
${***Parám***%***Patrón***}** **${*Parám***%%***Patrón***}**	Se elimina del valor del parámetro la mínima (o la máxima) comparación del patrón, buscando por el final del parámetro.
${***Parám/Patrón/Cad***}** **${*Parám//Patrón/Cad***}**	En el valor del parámetro se reemplaza por la cadena indicada la primera comparación del patrón (o todas las comparaciones).

BASH proporciona unas potentes herramientas para el tratamiento de cadenas, sin embargo la sintaxis puede resultar engorrosa y requiere de experiencia para depurar el código. Por lo tanto, se recomienda crear

guiones que resulten fáciles de comprender, documentando claramente las órdenes más complejas.

Unos ejemplos para estudiar:

```
# Si el 1er parámetro es nulo, asigna el usuario que lo ejecuta.
USUARIO=${1:-`whoami`}

# Si no está definida la variable COLUMNS, el ancho es de 80.
ANCHO=${COLUMNS:-80}

# Si no existe el 1er parámetro, pone mensaje de error y sale.
: ${1:?"Error: $0 fichero"

# Obtiene la extensión de un fichero (quita hasta el punto).
EXT=${FICHERO##*.}

# Quita la extensión “rpm” al camino del fichero.
RPM=${FICHRPM%.rpm}

# Cuenta el nº de caracteres de la variable CLAVE.
CARACTERES=${#CLAVE}

# Renombra el fichero de enero a Febrero.
NUEVO=${ANTIGUO/enero/febrero}

# Añade nuevo elemento a la matriz (matriz[tamaño]=elemento).
matriz[${#matriz[*]}]="nuevo"
```

Ejercicio 2.2: interpretar los siguientes mandatos.

```
DATOS="ls –ld $1"
TIPO=${DATOS:1}
PERM=${DATOS:2:9}
```

```
if [ ${#OPCIONES} -gt 1 ]; then ...
```

```
f="$f${1%$e}"
```

2.4. Sustitución de mandato.

Esta expansión sustituye el mandato ejecutado –incluyendo sus parámetros– por su salida normal, ofreciendo una gran potencia y flexibilidad de ejecución a un “*shellscrpit*”. Los formatos válido son:

Formato	Descripción
$(*Mandato***)**	Sustitución literal del mandato y sus parámetros.
`*Mandato*`	Sustitución de mandatos permitiendo caracteres de escape.

Cuando la sustitución de mandatos va en una cadena entre comillas dobles se evita que posteriormente se ejecute una expansión de ficheros.

El siguiente programa lista información sobre un usuario.

```
#!/bin/bash
# infous – lista información de un usuario.
# Uso:  infous usuario
TEMPORAL=`grep "^$1:" /etc/passwd 2>/dev/null`
USUARIO=`echo $TEMPORAL | cut -f1 -d:`
echo "Nombre de usuario: $USUARIO"
echo -n "Identifidor (UID): "
echo $TEMPORAL | cut -f3 -d:
echo -n "Nombre del grupo primario: "
GID=`echo $TEMPORAL | cut -f4 -d:`
grep ":$GID:" /etc/group | cut -f1 -d:
echo "Directorio personal: "
ls -ld `echo $TEMPORAL | cut -f6 -d:`
```

Ejercicio 2.3: interpretar la salida de la siguiente orden:

```
echo "$(basename $0): \"$USER\" sin permiso de ejecución." >&2
```

2.5. Expansión aritmética.

La expansión aritmética calcula el valor de la expresión indicada y la sustituye por el resultado de la operación. El formato de esta expansión es:

Formato	Descripción
$((*Expresión***))** **$[***Expresión***]**	Sustituye la expresión por su resultado.

Véase el siguiente ejemplo:

```
# Cuenta el nº de espacios para centrar una cadena
#     espacios = ( ancho_pantalla – longitud (cadena) ) / 2.
ESPACIOS=$(( (ANCHO-${#CADENA})/2 ))
```

Ejercicio 2.4: interpretar la siguiente línea, algo difícil de leer.

```
if [ ${#cad} -lt $[${#1}-1] ]; then ...
```

2.6. Sustitución de proceso.

La sustitución de proceso permite utilizar un fichero especial de tipo cola para intercambiar información entre 2 procesos, uno que escribe en la cola y el otro que lee de ella en orden (el primero en llegar es el primero en salir). Los formatos válidos para esta expansión son:

Formato	Descripción
Fich **<(***Lista***)** *Descr***<(***Lista***)**	La lista de órdenes escribe en el fichero para que éste pueda ser leído por otro proceso.
Fich **>(***Lista***)** *Descr***>(***Lista***)**	Cuando otro proceso escribe en el fichero, el contenido de éste se pasa como parámetro de entrada a la lista de órdenes.

Ejercicio 2.5: interpretar la siguiente línea.

```
comm <(sort enero.dat) <(sort febrero.dat)
```

2.7. División en palabras.

Una vez que se hayan realizado las expansiones previas, el intérprete divide la línea de entrada en palabras, utilizando como separadores los caracteres especificados en la variable de entorno **IFS**. Para evitar problemas de seguridad generados por un posible "Caballo de Troya", el administrador debe declarar esta variable como de sólo lectura y establecer unos valores fijos para los separadores de palabra; que por defecto éstos son espacio, tabulador y salto de línea. Una secuencia de varios separadores se considera como un único delimitador.

Por ejemplo, si se ejecuta el siguiente mandato:

```
du –hs $HOME
```

el intérprete realiza las sustituciones y –antes de ejecutar la orden– divide la línea en las siguientes palabras.

```
"du" "-hs" "/home/ramon"
```

2.8. Expansión de fichero.

Si algunas de las palabras obtenidas tras la división anterior contiene algún caracteres especial conocido como **comodín** (*****, **?** o **[**), ésta se trata como un patrón que se sustituye por la lista de nombres de ficheros que cumplen dicho patrón, ordena da alfabéticamente [2]. El resto de caracteres del patrón se tratan normalmente.

Los patrones válidos son:

Formato	Descripción
*	Equivale a cualquier cadena de caracteres, incluida una cadena nula.
?	Equivale a cualquier carácter único.
[*Lista*]	Equivale a cualquier carácter que aparezca en la lista. Pueden incluirse rangos de caracteres separados por guión (-). Si el primer carácter de la lista es ^, se comparan los caracteres que no formen parte de ella.

La siguiente tabla describe algunos ejemplos.

```
# Listar los ficheros terminados en .rpm
ls *.rpm
# Listar los ficheros que empiecen por letra minúscula y tengan
#      extensión .rpm
ls [a-z]*.rpm
# Listar los ficheros que empiezan por ".b", ".x" y ".X"
ls .[bxX]*
# Listar los ficheros cuya extensión tenga 2 caracteres
ls *.??
```

Ejercicio 2.6: evaluar la siguiente orden.

```
cd /var/log; tar cvf copialog.tar syslog.[0-9] messages.[0-9]
```

3. Programación estructurada.

Las estructuras de programación se utilizan para generar un código más legible y fiable. Son válidas para englobar bloques de código que cumplen un cometido común, para realizar comparaciones, selecciones, bucles repetitivos y llamadas a subprogramas.

3.1. Listas de mandatos.

Los mandatos BASH pueden agruparse en bloques de código que mantienen un mismo ámbito de ejecución. La siguiente tabla describe brevemente los aspectos fundamentales de cada lista de órdenes.

Lista de órdenes	Descripción
Mandato **&**	Ejecuta el mandato en 2º plano. El proceso tendrá menor prioridad y no debe ser interactivo..
Man1 **|** *Man2*	Tubería. Redirige la salida de la primera orden a la entrada de la segundo. Cada mandato es un proceso separado.
*Man1***;** *Man2*	Ejecuta varios mandatos en la misma línea de código.
Man1 **&&** *Man2*	Ejecuta la 2ª orden si la 1ª lo hace con éxito (su estado de salida es 0).
Man1 **||** *Man2*	Ejecuta la 2ª orden si falla la ejecución de la anterior (su código de salida no es 0).
(*Lista***)**	Ejecuta la lista de órdenes en un subproceso con un entorno común.
{ *Lista***; }**	Bloque de código ejecutado en el propio intérprete.
Línea1 **** *Línea2*	Posibilita escribir listas de órdenes en más de una línea de pantalla. Se utiliza para ejecutar mandatos largos.

Ya se ha comentado previamente la sintaxis de algunas de estas combinaciones de mandatos. Sin embargo, el siguiente epígrafe presta atención especial a las listas de órdenes condicionales.

3.1.1. Listas condicionales.

Los **mandatos condicionales** son aquellos que se ejecutan si una determinada orden se realiza correctamente –lista Y (**&&**)– o si se produce algún error –lista O (**||**)–.

A continuación se comentan algunos ejemplos más sacados del fichero de configuración `/etc/rc.d/rc.sysinit`

```
# Si se ejecuta correctamente “converquota”, se ejecuta “rm”.
/sbin/convertquota -u / && rm -f /quota.user
#
# Si da error la ejecución de “grep”, se ejecuta “modprobe”.
grep 'hid' /proc/bus/usb/drivers || modprobe hid 2> /dev/null
```

Ejercicio 3.1: interpreta la siguiente lista de órdenes.

```
loadkeys $KEYMAP < /dev/tty0 > /dev/tty0 2>/dev/null && \
    success "Loading def keymap" || failure "Loading def keymap"
```

3.2. Estructuras condicionales y selectivas.

Ambas estructuras de programación se utilizan para escoger un bloque de código que debe ser ejecutado tras evaluar una determinada condición. En la estructura condicional se opta entre 2 posibilidades, mientras que en la selectiva pueden existir un número variable de opciones.

3.2.1. Estructuras condicionales.

La **estructura condicional** sirve para comprobar si se ejecuta un bloque de código cuando se cumple una cierta condición. Pueden anidarse varias estructuras dentro del mismo bloques de código, pero siempre existe una única palabra **`fi`** para cada bloque condicional.

La tabla muestra cómo se representan ambas estructuras en BASH.

Estructura de programación	Descripción
`if` *Expresión*; **`then`** *Bloque*; `fi`	**Estructura condicional simple**: se ejecuta la lista de órdenes si se cumple la expresión. En caso contrario, este código se ignora.

if *Expresión1*;
then Bloque1;
[**elif** *Expresión2*;
  then *Bloque2*;
  ...]
[**else** *BloqueN*;]
fi
	Estructura condicional compleja: el formato completo condicional permite anidar varias órdenes, además de poder ejecutar distintos bloques de código, tanto si la condición de la expresión es cierta, como si es falsa.

Aunque el formato de codificación permite incluir toda la estructura en una línea, cuando ésta es compleja se debe escribir en varias, para mejorar la comprensión del programa. En caso de teclear la orden compleja en una sola línea debe tenerse en cuenta que el carácter separador (**;**) debe colocarse antes de las palabras reservadas: **then**, **else**, **elif** y **fi**.

Hay que resaltar la versatilidad para teclear el código de la estructura condicional, ya que la palabra reservada **then** puede ir en la misma línea que la palabra **if**, en la línea siguiente sola o conjuntamente con la primera orden del bloque de código, lo que puede aplicarse también a la palabra **else**).

Puede utilizarse cualquier expresión condicional para evaluar la situación, incluyendo el código de salida de un mandato o una condición evaluada por la orden interna test. Este último caso se expresa colocando la condición entre corchetes (formato: **[** *Condición* **]**).

Véanse algunos sencillos ejemplos de la estructura condicional simple extraídos del "*script*" `/etc/rc.d/rc.sysinit`. Nótese la diferencia en las condiciones sobre la salida normal de una orden –expresada mediante una sustitución de mandatos– y aquellas referidas al estado de ejecución de un comando (si la orden se ha ejecutado correctamente o si se ha producido un error).

```
# La condición más simple escrita en una línea:
# si RESULT>0; entonces rc=1
if [ $RESULT -gt 0 ]; then rc=1; fi
#
# La condición doble:
# si la variable HOSTNAME es nula o vale "(none)"; entonces ...
if [ -z "$HOSTNAME" -o "$HOSTNAME" = "(none)" ]; then
    HOSTNAME=localhost
fi
#
# Combinación de los 2 tipos de condiciones:
# si existe el fichero /fastboot o la cadena "fastboot" está
#    en el fichero /proc/cmdline; entonces ...
if [ -f /fastboot ] || grep -iq "fastboot" /proc/cmdline \
                              2>/dev/null; then
       fastboot=yes
fi
```

Obsérvese los ejemplos para las estructuras condicionales complejas, basados también en el programa de configuración `/etc/rc.d/rc.sysinit`.

```
# Estructura condicional compleja con 2 bloques:
# si existe el fichero especificado, se ejecuta; si no, se da
#    el valor “no” a la variable NETWORKING
if [ -f /etc/sysconfig/network ];
then
    . /etc/sysconfig/network
else
    NETWORKING=no
fi
# Estructura anidada:
# si rc=0; entonces ...; si no, si rc=1; entonces ...; en caso
#    contrario; no se hace nada.
if [ "$rc" = "0" ]; then
       success "$STRING"
       echo
elif [ "$rc" = "1" ]; then
       passed "$STRING"
       echo
fi
```

Ejercicio 3.2: describir el comportamiento de la siguiente estructura.

```
if [ -f /etc/sysconfig/console/default.kmap ]; then
   KEYMAP=/etc/sysconfig/console/default.kmap
else
  if [ -f /etc/sysconfig/keyboard ]; then
    . /etc/sysconfig/keyboard
  fi
  if [ -n "$KEYTABLE" -a -d "/usr/lib/kbd/keymaps" -o -d "/lib/kbd/keymaps" ];
then
     KEYMAP=$KEYTABLE
  fi
fi
```

3.2.2. Estructura selectiva.

La **estructura selectiva** evalúa la condición de control y, dependiendo del resultado, ejecuta un bloque de código determinado. La siguiente tabla muestra el formato genérico de esta estructura.

Estructura de programación	Descripción

case *Variable* **in**
  [**(**]*Patrón1*) *Bloque1* ;;
  ...
esac
	Estructura selectiva múltiple: si la variable cumple un determinado patrón, se ejecuta el bloque de código correspondiente. Cada bloque de código acaba con “**;;**”.
	La comprobación de patrones se realiza en secuencia.

Las posibles opciones soportadas por la estructura selectiva múltiple se expresan mediante patrones, donde puede aparecer caracteres comodines, evaluándose como una expansión de ficheros, por lo tanto el patrón para representar la opción por defecto es el asterisco (*). Dentro

de una misma opción pueden aparecer varios patrones separados por la barra vertical (|), como en una expresión lógica O.

Si la expresión que a comprobar cumple varios patrones de la lista, sólo se ejecuta el bloque de código correspondiente al primero de ellos, ya que la evaluación de la estructura se realiza en secuencia.

Obsérvese el siguientes ejemplos:

```
# Según el valor de la variable UTC:
# - si es “yes” o “true”, ...
# - si es “no” o “false”, ...
case "$UTC" in
   yes|true)   CLOCKFLAGS="$CLOCKFLAGS --utc";
               CLOCKDEF="$CLOCKDEF (utc)";
   ;;
   no|false)   CLOCKFLAGS="$CLOCKFLAGS --localtime";
               CLOCKDEF="$CLOCKDEF (localtime)";
   ;;
esac
```

Ejercicio 3.3: describir el modo de ejecución de la siguiente estructura selectiva.

```
case "$SO" in
     AIX)   echo –n "$US: "
            lsuser -ca expires $US|fgrep -v "#"|cut -f2 -d:`"
           ;;
     SunOS) echo "$US: `logins -aol $US|cut -f7 -d:`"
           ;;
     Linux) echo "$US: `chage -l $US|grep Account|cut -f2 -d:`"
           ;;
     *)     echo "Sistema operativo desconocido" ;;
esac
```

3.3. Bucles.

Los bucles son estructuras reiterativas que se ejecutan repetitivamente, para no tener que teclear varias veces un mismo bloque de código. Un bucle debe tener siempre una condición de salida para evitar errores provocados por bucles infinitos.

La siguiente tabla describe las 2 órdenes especiales que pueden utilizarse para romper el modo de operación típico de un bucle.

Orden	Descripción
break	**Ruptura inmediata de un bucle** (debe evitarse en programación estructurada para impedir errores de lectura del código).

continue	**Salto a la condición del bucle** (debe evitarse en programación estructurada para impedir errores de lectura del código).

Los siguientes puntos describen los distintos bucles que pueden usarse tanto en un guión como en la línea de mandatos de BASH.

3.3.1. Bucles genéricos.

Los bucles genéricos de tipos “para cada” ejecutan el bloque de código para cada valor asignado a la variable usada como índice del bucle o a su expresión de control. Cada iteración debe realizar una serie de operaciones donde dicho índice varíe, hasta la llegar a la condición de salida.

El tipo de bucle **for** más utilizado es aquél que realiza una iteración por cada palabra (o cadena) de una lista. Si se omite dicha lista de valores, se realiza una iteración por cada parámetro posicional.

Por otra parte, BASH soporta otro tipo de bucle iterativo genérico similar al usado en el lenguaje de programación C, usando expresiones aritméticas. El modo de operación es el siguiente:

- Se evalúa la primera expresión aritmética antes de ejecutar el bucle para dar un valor inicial al índice.
- Se realiza una comprobación de la segunda expresión aritmética, si ésta es falsa se ejecutan las iteraciones del bucle. Siempre debe existir una condición de salida para evitar que el bucle sea infinito.
- como última instrucción del bloque se ejecuta la tercera expresión aritmética –que debe modificar el valor del índice– y se vuelve al paso anterior.

La siguiente tabla describe los formatos de los bucles iterativos genéricos (de tipo “para cada”) interpretados por BASH.

Bucle	Descripción
for *Var* [**in** *Lista*]**; do** *Bloque* **done**	**Bucle iterativo**: se ejecuta el bloque de mandatos del bucle sustituyendo la variable de evaluación por cada una de las palabras incluidas en la lista.
For ((*Exp1***;** *Exp2***;** *Exp3***))** **do** *Bloque* **done**	**Bucle iterativo de estilo C**: se evalúa *Exp1*, mientras *Exp2* sea cierta se ejecutan en cada iteración del bucle el bloque de mandatos y *Exp3* (las 3 expresiones deben ser aritméticas).

Véase algunos ejemplos:

```
# Se asigna a la variable “library” el camino de cada uno de
#    los archivos indicados en la expansión de ficheros y se
#    realizan las operaciones indicadas en el bloque de código.
for library in /lib/kernel/$(uname -r)/libredhat-kernel.so* ; do
        ln -s -f $library /lib/
        ldconfig -n /lib/
done
...
# Se establece un contador de hasta 20 iteraciones para
#    ejecutar el bucle.
for (( times = 1; times < 20; times++ )); do
    /usr/sbin/rpcinfo -p | grep ypbind > /dev/null 2>&1 && \
            ypwhich > /dev/null 2>&1
done
```

Ejercicio 3.4: evaluar el siguiente bucle:

```
for US in `cut -f2 -d: /home/cdc/*.lista`; do
    grep "^$US:" /etc/shadow | cut -f1-2 -d: >>$FICHTEMP
done
```

7.3.2. Bucles condicionales “mientras” y “hasta”.

Los bucles condicionales evalúan la expresión en cada iteración del bucle y dependiendo del resultado se vuelve a realizar otra iteración o se sale a la instrucción siguiente.

La siguiente tabla describe los formatos para los 2 tipos de bucles condicionales soportados por el intérprete BASH.

Bucle	Descripción
while *Expresión*; **do** *Bloque* **done**	**Bucle iterativo “mientras”**: se ejecuta el bloque de órdenes mientras que la condición sea cierta. La expresión de evaluación debe ser modificada en algún momento del bucle para poder salir.
until *Expresión*; **do** *Bloque* **done**	**Bucle iterativo “hasta”**: se ejecuta el bloque de código hasta que la condición sea cierta. La expresión de evaluación debe ser modificada en algún momento del bucle para poder salir.

Ambos bucles realizan comparaciones inversas y pueden usarse indistintamente, aunque se recomienda usar aquél que necesite una

condición más sencilla o legible, intentando no crear expresiones negativas. Véase el siguiente ejemplo:

```
# Mientras haya parámetros que procesar, ...
while [ $# != 0 ] ; do
        processdir "$1"
        shift
done
```

Ejercicio 3.5: convertir este bucle **`while`** en un bucle **`until`** y en uno **`for`** de estilo C.

```
i=5
while [ $i -ge 0 ] ; do
        if [ -f /var/log/ksyms.$i ] ; then
                mv /var/log/ksyms.$i /var/log/ksyms.$(($i+1))
        fi
        i=$(($i-1))
done
```

3.3.3. Bucle de selección interactiva.

La estructura **`select`** no es propiamente dicho un bucle de programación estructurada, ya que se utiliza para mostrar un menú de selección de opciones y ejecutar el bloque de código correspondiente a la selección escogida. En caso de omitir la lista de palabras, el sistema presenta los parámetros posicionales del programa o función. Este tipo de bucles no suele utilizarse.

La siguiente tabla describe el formato del bucle interactivo.

Bucle	Descripción
`select` *`Var`* [**`in`** *`Lista`*]**`;`** **`do`** *`Bloque1`* `...` **`done`**	**Bucle de selección interactiva**: se presenta un menú de selección y se ejecuta el bloque de código correspondiente a la opción elegida. El bucle se termina cuando se ejecuta una orden **`break`**. La variable **`PS3`** se usa como punto indicativo.

4. Funciones.

Una función en BASH es una porción de código declarada al principio del programa, que puede recoger parámetro de entrada y que puede ser llamada desde cualquier punto del programa principal o desde otra función, tantas veces como sea necesario.

El uso de funciones permite crear un código más comprensible y que puede ser depurado más fácilmente, ya que evita posibles errores tipográficos y repeticiones innecesarias.

Los parámetros recibidos por la función se tratan dentro de ella del mismo modo que los del programa principal, o sea los parámetros posicionales de la función se corresponden con las variables internas `$0`, `$1`, etc.

El siguiente cuadro muestra los formatos de declaración y de llamada de una función.

Declaración	LLamada
[**function**] *NombreFunción* **()** **{** *Bloque* ... [**return** [*Valor*]] ... **}**	*NombreFunción* [*Parámetro1* ...]

La función ejecuta el bloque de código encerrado entre sus llaves y –al igual que un programa– devuelve un valor numérico. En cualquier punto del código de la función, y normalmente al final, puede usarse la cláusula **`return`** para terminar la ejecución y opcionalmente indicar un código de salida.

Las variables declaradas con la cláusula **`local`** tienen un ámbito de operación interno a la función. El resto de variables pueden utilizarse en cualquier punto de todo el programa. Esta característica permite crear funciones recursivas sin que los valores de las variables de una llamada interfieran en los de las demás.

En el ejemplo del siguiente cuadro se define una función de nombre `salida`, que recibe 3 parámetros. El principio del código es la definición de la función (la palabra **`function`** es opcional) y ésta no se ejecuta hasta que no se llama desde el programa principal. Asimismo, la variable `TMPGREP` se declara en el programa principal y se utiliza en la función manteniendo su valor correcto.

```
#/bin/bash
# comprus - comprueba la existencia de usuarios en listas y en
#           el archivo de claves (normal y NIS).
#           Uso: comprus ? | cadena
#                    ?: ayuda.
# Ramón Gómez - Septiembre 1.998.

# Rutina de impresión.
# Parámetros:
#       1 - texto de cabecera.
#       2 - cadena a buscar.
#       3 - archivo de búsqueda.
salida ()
        {
        if egrep "$2" $3 >$TMPGREP 2>/dev/null; then
           echo "       $1:"
           cat $TMPGREP
        fi
        }

## PROGRAMA PRINCIPAL ##

PROG=$(basename $0)
TMPGREP=/tmp/grep$$
DIRLISTAS=/home/cdc/listas

if [ "x$*" = "x?" ]
then
    echo "
Uso:       $PROG ? | cadena
Propósito: $PROG: Búsqueda de usuarios.
                  cadena: expresión regular a buscar.
"
    exit 0
fi
if [ $# -ne 1 ]; then
   echo "$PROG: Parámetro incorrecto.
Uso: $PROG) ? | cadena
        ?: ayuda" >&2
    exit 1
fi
echo

for i in $DIRLISTAS/*.lista; do
    salida "$1" "$(basename $i | sed 's/.lista//')" "$i"
done
salida "$1" "passwd" "/etc/passwd"
[ -e "$TMPGREP" ] && rm -f $TMPGREP
```

Ejercicio 4.1: desarrollar en BASH una función que reciba un único parámetro, una cadena de caracteres, y muestre el último carácter de dicha cadena. Debe tenerse en cuenta que el primer carácter de una cadena tiene como índice el valor 0.

Ejercicio 4.2: programar una función BASH que reciba una cadena de caracteres y que la imprima centrada en la pantalla. Tener en cuenta que

el ancho del terminal puede estar definido en la variable de entorno `COLUMNS`, en caso contrario usar un valor por defecto de 80 columnas.

5. Características especiales.

En última instancia se van a describir algunas de las características especiales que el intérprete BASH añade a su lenguaje para mejorar la funcionalidad de su propia interfaz y de los programas que interpreta.

Seguidamente se enumeran algunas de las características especiales de BASH y en el resto de puntos de este capítulo se tratan en mayor profundidad las que resultan más interesantes:

- Posibilidad de llamar al intérprete BASH con una serie de opciones que modifican su comportamiento normal.
- Mandatos para creación de programas interactivos.
- Control de trabajos y gestión de señales.
- Manipulación y personalización del punto indicativo.
- Soporte de alias de comandos.
- Gestión del histórico de órdenes ejecutadas.
- Edición de la línea de mandatos.
- Manipulación de la pila de últimos directorios visitados.
- Intérprete de uso restringido, con características limitadas.
- Posibilidad de trabajar en modo compatible con la norma POSIX 1003.2 [2].

5.1. Programas interactivos.

BASH proporciona un nivel básico para programar "*shellscripts*" interactivos, soportando instrucciones para solicitar y mostrar información al usuario.

Las órdenes internas para dialogar con el usuario se describen en la siguiente tabla.

Mandato	Descripción
read [**-p** "*Cadena*"] [*Var1 ...*]	**Asigna la entrada a variables**: lee de la entrada estándar y asigna los valoras a las variables indicadas en la orden. Puede mostrarse un mensaje antes de solicitar los datos. Si no se especifica ninguna variable, **REPLY** contiene la línea de entrada.
echo [**-n**] *Cadena*	**Muestra un mensaje**: manda el valor de la cadena a la salida estándar; con la opción `–n` no se hace un salto de línea.
printf *Formato Parám1 ...*	**Muestra un mensaje formateado**: equivalente a la función `printf` del lenguaje C, manda un mensaje formateado a la salida normal, permitiendo mostrar cadena y números con una longitud determinada.

Véase los siguientes ejemplos.

```
# Las siguientes instrucciones son equivalentes y muestran
#     un mensaje y piden un valor
echo –n "Dime tu nombre: "
read NOMBRE
#
read –p "Dime tu nombre: " NOMBRE
...
# Muestra los usuarios y nombres completos
#   (la modificación de la variable IFS sólo afecta al bucle)
while IFS=: read usuario clave uid gid nombre ignorar
do
  printf "%8s (%s)\n" $usuario $nombre
done </etc/passwd
```

Como ejercicio se propone explicar paso a paso el funcionamiento del bucle anterior y los valores que se van asignando a las distintas variables en cada iteración.

5.2. Control de trabajos.

Las órdenes para el control de trabajos permiten manipular los procesos ejecutados por el usuario o por un guión BASH. El usuario puede manejar varios procesos y subprocesos simultáneamente, ya que el sistema asocia un número identificador único a cada uno de ellos.

Los procesos asociados a una tubería tienen identificadores (PID) propios, pero forman parte del mismo trabajo independiente.

La siguiente tabla describe las instrucciones asociadas con el tratamiento de trabajos.

Mandato	Descripción
Mandato **&**	**Ejecución en 2º plano**: lanza el proceso de forma desatendida, con menor prioridad y con la posibilidad de continuar su ejecución tras la desconexión del usuario.
bg *%NºTrabajo*	**Retorna a ejecutar en 2º plano**: continúa la ejecución desatendida de un procesos suspendido.
fg *%NºTrabajo*	**Retorna a ejecutar en 1er plano**: vuelve a ejecutar el proceso asociado al trabajo indicado de forma interactiva.
jobs	**Muestro los trabajos en ejecución**, indicando el nº de trabajo y los PID de sus procesos.
kill *Señal PID1|%Trab1* ...	**Manda una señal a procesos o trabajos**, para indicar una excepción o un error. Puede especificarse tanto el nº de señal como su nombre.
suspend	**Para la ejecución del proceso**, hasta que se recibe una señal de continuación.
trap [*Comando*] [*Señal1* ...]	**Captura de señal**: cuando se produce una determinada señal (interrupción) en el proceso, se ejecuta el mandato asociado. Las señales especiales `EXIT` y `ERR` se capturan respectivamente al finalizar el "*script*" y cuando se produce un error en una orden simple.
wait [*PID1|%Trab1*]	**Espera hasta que termine un proceso o trabajo**: detiene la ejecución del proceso hasta que el proceso hijo indicado hay finalizado su ejecución.

Revisar los siguientes ejemplos.

```
# Se elimina el fichero temporal si en el programa aparecen las
3    señales 0 (fin), 1, 2, 3 y 15.
trap 'rm -f ${FICHTEMP} ; exit' 0 1 2 3 15
...
# Ordena de forma independiente las listas de ficheros de
#   usuarios, espera a finalizar ambos procesos y compara
#   los resultados.
(cat alum03.sal prof03.sal pas03.sal | sort | uniq > usu03) &
(cat alum04.sal prof04.sal pas04.sal | sort | uniq > usu04) &
wait
diff usu03 usu04
```

5.3. Intérprete de uso restringido.

Cuando el administrador del sistema asigna al usuario un intérprete de uso restringido, éste último utiliza un entorno de operación más controlado y con algunas características limitadas o eliminadas.

Existen 3 formas de ejecutar BASH en modo restringido:

```
rbash
bash –r
bash --restricted
```

BASH restringido modifica las siguientes características [2]:

- No se puede usar la orden **`cd`**.
- No se pueden modificar las variables `SHELL`, `PATH`, `ENV` ni `BASH_ENV`.
- No se pueden ejecutar mandatos indicando su camino, sólo se ejecutarán aquellos que se encuentren en los directorios especificados por el administrador..
- No se pueden especificar caminos de ficheros como argumento del mandato “**`.`**”.
- No se pueden añadir funciones nuevas en los ficheros de inicio.
- No se admiten redirecciones de salida.
- No se puede reemplazar el intérprete RBASH (no se puede ejecutar la orden **`exec`**).
- No se pueden añadir o quitar mandatos internos.
- No se puede modificar el modo de operación restringido.

6. Referencias.

1. B. Fox, C. Ramey: *"BASH(1)" (páginas de manuales de BASH v2.5b)"*. 2.002.
2. C. Ramey, B. Fox: "*Bash Reference Manual, v2.5b*". Free Software Foundation, 2.002.
3. Mike G, trad. G. Rodríguez Alborich: *"Programación en BASH – COMO de Introducción"*. 2.000.
4. M. Cooper: *"Advanced Bash-Scripting Guide, v2.1"*. Linux Documentation Project, 2.003.
5. R. M. Gómez Labrador: *"Administración Avanzada de Sistemas Linux (3ª edición)"*. Secretariado de Formación del PAS (Universidad de Sevilla), 2.004.

i. Proyecto GNU.: http://www.gnu.org/
ii. The Linux Documentation Project (TLDP): http://www.tldp.org/
iii. Proyecto HispaLinux (LDP-ES): http://www.hispalinux.es/