Versión multipágina

Programación Shell-script en Linux

Departamento de Ingeniería Telemática (DIT)

Universidad de Sevilla

Francisco José Fernández Jiménez

Fco. Javier Muñoz Calle

ÍNDICE

1 Objetivos y alcance

1.1 Introducción

El intérprete de comandos o shell es un programa que permite a los usuarios interactuar con el sistema, procesando las órdenes que se le indican. Los comandos invocables desde el shell pueden clasificarse en internos (corresponden en realidad a órdenes interpretadas por el propio shell) y externos (corresponden a ficheros ejecutables externos al shell). Además de comandos, los shells ofrecen otros elementos para mejorar su funcionalidad, tales como variables, funciones o estructuras de control. El conjunto de comandos internos y elementos disponibles, así como su sintaxis, dependerá del shell concreto empleado.

Además de utilizar el shell desde la línea de comandos (basada en el prompt como la indicación del shell para anunciar que espera una orden del usuario), puede emplearse para la interpretación de shell-scripts. Un shell-script o "guión de órdenes" es un fichero de texto que contiene un conjunto de comandos y órdenes interpretables por el shell.

En los S.O.'s Unix existen múltiples implementaciones de shell (en Windows, el equivalente serían los programas "command.com" o "cmd.exe"). Atendiendo al shell del que proceden y a su similitud sintáctica (incluyendo sus comandos internos), los shells de Unix pueden clasificarse en dos grandes familias (existen algunos shell adicionales, de uso residual y dentro de los shells de una misma familia también existen diferencias, pero gran parte de su sintaxis es común):

1. `sh` (Bourne Shell): este shell fue usado desde las primeras versiones de Unix (Unix Versión 7). Recibe ese nombre por su desarrollador, Stephen Bourne, de los Laboratorios Bell de AT&T. A raíz de él han surgido múltiples shells, tales como `zsh` (Z shell), `ash` (almquist shell), `bash` (Bourne again shell), `dash` (Debian almquist shell) o `ksh` (Korn shell). Por su mayor uso pueden destacarse:
    - `bash`: fue desarrollado para ser un superconjunto de la funcionalidad del Bourne Shell (en la que incluye funcionalidades de ksh y csh), siendo el intérprete de comandos asignado por defecto a los usuarios en las distribuciones de Linux, por lo que es el shell empleado en la mayoría de las consolas de comandos de Linux. Se caracteriza por una gran funcionalidad adicional a la del Bourne Shell. Como ficheros personales de los usuarios emplea `$HOME/.bashrc` y `.bash_profile`.
    - `dash` (Debian almquist shell), derivado directo de ash, se caracteriza por ser mucho más ligero (depende de menos bibliotecas) y rápido que otros shells, tales como bash, aunque con menos características funcionales. El fichero personal del usuario es `$HOME/.profile`.

- `ksh` (Korn shell): destaca por sus funciones avanzadas para manejar archivos, pudiendo competir con lenguajes de programación especializados tales como awk o perl.

Bourne Shell ha llegado a convertirse en un estándar de facto de tal modo que todos los sistemas Unix tienen, al menos, una implementación del Bourne Shell (o un shell compatible con él), ubicada en `/bin/sh`. En el caso concreto de los S.O.'s UNIX Linux, no existe ninguna implementación del Bourne Shell, manteniéndose la entrada `/bin/sh` (así como su manual `man sh`) como un enlace simbólico a una implementación de shell compatible. En concreto:

- En algunos S.O's Linux se enlaza a `/bin/bash`.
- En Debian, desde Debian Lenny "5.0" (en Ubuntu desde la versión 6.10) se enlaza a `/bin/dash`: de este modo, estas distribuciones pretenden hacer uso de:
  - `dash`: para los shell scripts empleados en el arranque del sistema (cuyo shebang tradicionalmente siempre ha sido `#!/bin/sh`), aprovechando la mayor rapidez de este intérprete.
  - `bash`: en las consolas de comandos y scripts de servicios de los usuarios, de modo que éstos puedan aprovechar la mayor funcionalidad que éste intérprete ofrece.

2. `csh` (C shell): caracterizado por presentar una sintaxis muy parecida a la del lenguaje de programación C. Como shell derivados destaca `tcsh`. Estos shell cuentan con un nivel de uso muy inferior respecto a los de la familia Bourne Shell.

   Para intentar homogeneizar esta diversidad de shells, el IEEE definió un estándar de "intérprete de comandos" bajo la especificación POSIX 1003.2 (también recogida como ISO 9945.2). La creación de dicho estándar se basó en la sintaxis que presentaban múltiples shells de la familia Bourne shell (el propio Bourne Shell de Unix Versión 7, implementaciones en UNIX System V y BSD, así como `ksh`). Esto ha llevado a que la gran mayoría de los shells derivados del Bourne Shell, tales como `bash`, `dash` o `ksh`, den soporte a este estándar POSIX (mientras que los derivados del `csh` no). En concreto:

   - `bash`: respeta completamente el estándar POSIX, sobre el que añade un número considerable de extensiones (estructura select, arrays, mayor número de operadores,...).
   - `dash`: conforme al estándar POSIX IEEE 1003.2, sólo ampliado con algunas extensiones Berkeley. De forma precisa, tal como indica el manual del intérprete dash de su sistema (`man dash`), la línea de desarrollo de `dash` pretende que éste satisfaga el estándar POSIX de shells IEEE 1003.2, propósito casi logrado en la actualidad (sólo presenta algunas características faltantes, como la variable `$LINENO`).

TAREAS

1. Ejecute en su sistema Debian el comando `ls -l /bin/sh`, y compruebe cómo efectivamente corresponde a un enlace simbólico al intérprete `/bin/dash`. Asimismo, consulte el manual `man sh`, comprobando cómo efectivamente obtiene el manual del intérprete `dash`.
2. Ejecute los comandos `ls -l /bin/dash` y `ls -l /bin/bash` para comprobar cómo efectivamente dispone de dichos shells en su sistema.

Puede obtener las versiones instaladas mediante `dpkg -s dash` y `bash –version`.

3. La mayoría de los shell-scripts implicados en el proceso de arranque del sistema se encuentran en la carpeta `/etc/init.d/`. Por ejemplo, mire el contenido del fichero `/etc/init.d/rc` (encargado del arranque de los servicios) y compruebe cómo la primera línea de dicho script es `#!/bin/sh`, lo que indica que es interpretado por el shell `/bin/dash`.
4. Analice el fichero `/etc/passwd` y compruebe cómo los usuarios `dit` y `root` tienen asignado el shell `bash`. Abra una consola de comandos con el usuario `dit` y ejecute el comando `ps ax | grep bash`, comprobando cómo el proceso del intérprete de comandos usado efectivamente corresponde al shell bash (proceso `-bash`). Abra una sesión con el usuario `root` y realice la misma prueba, comprobando cómo igualmente se está usando bash.

1.2 Objetivo

El objetivo es introducir el uso del lenguaje de programación de shell-scripts. Consecuentemente, nos centraremos en la sintaxis de shell propuesta por el estándar POSIX IEEE 1003.2 lo que, conforme a lo antes indicado, se traduce en que todo lo visto aquí podrá ser utilizado en dash, bash y cualquier otro shell que satisfaga dicho estándar. Sobre la base de este documento podrá profundizarse analizando las funcionalidades adicionales añadidas por cada shell, siendo de especial interés, dado su uso, las aportadas por bash.

1.3 Documentación de apoyo

- Estándar de "intérprete de comandos" de la especificación POSIX (IEEE Standard 1003.1 2004/ISO 9945.2)
- The Linux Command Line (Openbook)". E. William, Jr. Shotts. San Francisco, 2011.
- Linux bible". Christopher Negus. Indianapolis, IN. Wiley, 2011. ISBN: 9780470929988.
- Pro Linux System Administration". James Turnbull, Peter Lieverdink, Dennis Matotek. Berkeley, CA. Apress, 2009. ISBN: 978-1-4302-1912-5.
- Linux Network Administrator's Guide (Openbook)". Olaf Kirch, Terry Dawson. O'Reilly, 2000. ISBN: 1-56592-400-2
- Web oficial de Dash
- Manual de referencia de Bash

Descarga de los shell-scripts de esta página

Para que no tenga que escribir los scripts que se muestran en esta página a mano, puede descargarse dichos scripts así como la solución a los ejercicios propuestos en el fichero comprimido `shellscripts.tar.gz` .
Descomprímalo con el comando:

```
tar xfvz ./shellscripts.tar.gz
```

En el directorio `scripts` están los ejemplos y en el directorio `aplicacion` los ejercicios propuestos resueltos.

Descarga en PDF (versión original)

Puede descargarse también la versión original de este documento en formato PDF: `FAST_t3-practica.pdf` .

2 Estructura básica de shell-scripts. Invocación

En su forma más básica, un shell-script puede ser un simple fichero de texto que contenga uno o varios comandos. Para ayudar a la identificación del contenido a partir del nombre del archivo, es habitual que los shell scripts tengan la extensión ".sh", por lo que seguiremos este criterio (pero recuerde que es algo meramente informativo y opcional). Por ejemplo, el siguiente fichero sería un shell-script:

`script.sh`

```
echo "Contenido carpeta personal:"
ls ~/
```

TAREAS

1. Compruebe que el fichero "script.sh" tiene permisos de ejecución generales (si no los tuviese, para asignárselos bastaría ejecutar "chmod +x script.sh").
2. Invoque el script para que sea interpretado, usando por ejemplo el comando:

   `cd ./script.sh`

Además de comandos, los shell-scripts pueden contener otros elementos, aportados por el shell para mejorar la funcionalidad de los scripts. De forma resumida, la

estructura básica de un shell-script es la siguiente:

script_ejemplo.sh

```
#!/bin/dash                                  <-- Shebang
# Esto no se interpreta                      <-- Comentarios
echo Hola                                    <-- Contenido
ps w
echo "Proceso lee el script: $$"
```

Como contenido del script pueden utilizarse múltiples elementos (comandos, variables, funciones, estructuras de control, comentarios,...) que se analizarán en el siguiente apartado.

El "shebang" permite especificar el intérprete de comandos con el que deseamos que sea interpretado el resto del script cuando se usa invocación implícita (ver más adelante). La sintaxis de esta línea es la secuencia `#!` seguida del ejecutable del shell deseado, sobre lo que deben realizarse la siguientes advertencias:

- Es imprescindible que sea la primera línea del script, ya que, en caso contrario, sería interpretado como un comentario (comienza con el carácter `#`).
- Puede haber espacios entre `#!` y el ejecutable del "shell".
- El shebang no es obligatorio (cuando se usa invocación implícita, si no se indica se intentará usar el mismo tipo de shell desde el que se ha invocado el script).

> **Sintaxis estricta**
>
> La sintaxis de los shell-scripts se caracteriza por ser bastante estricta en su escritura, especialmente en lo que se refiere a la inserción u omisión de espacios en blanco entre las palabras especiales. Tenga esto muy en cuenta a la hora de escribir los scripts que se proponen.

La utilización del shebang está condicionada por la forma en que sea invocado el shell-script, existiendo 3 opciones:

- Explícita: escribiendo explícitamente qué shell se desea invocar y pasando como argumento el nombre del script, cargándose en memoria un nuevo proceso para dicho shell (subshell o proceso shell hijo del shell padre responsable de la línea de comandos desde la que se ha invocado el script). En este caso se ignora el shebang.
- Implícita: invocando al script como si fuera un ejecutable, lo que requiere asignar permisos de ejecución al script. Se lee el shebang para determinar qué shell deberá usarse para leer el script, cargándose en memoria un proceso hijo (subshell) para dicho shell (si el script no presenta shebang, para el subshell se utilizará el mismo tipo de shell que el encargado de la línea de comandos desde la que se ha hecho la invocación). Tenga en cuenta que los shell-scripts son

ficheros de texto leídos por el intérprete de comandos, esto es, se interpretan, NO se ejecutan. La asignación del permiso de ejecución a un shell-script es una utilidad del sistema de ficheros para acelerar la invocación de scripts, pero cuyo funcionamiento interno es cargar un nuevo proceso de shell (subshell) para que éste interprete el script.

- Implícita con `.` (equivale a importar): el script será interpretado por el mismo proceso del shell responsable de la línea de comandos desde la que se está invocando el script (luego aquí no se abre ningún subshell). Consecuentemente, en este caso también se ignora el shebang.

En los casos en los que se crean subshells, salvo que se fuerce lo contrario (con `su -c` por ejemplo), el subshell pertenecerá al mismo usuario al que pertenecía el shell padre que lo ha creado. El usuario al que pertenece el proceso shell que interpreta un script condiciona las operaciones que se podrán hacer desde dentro del script (por ejemplo, si el shell pertenece al usuario `dit` , el script no podrá modificar el fichero `/etc/passwd` , mientras que si el shell pertenece al superusuario `root` , sí podrá hacerlo). Tenga en cuenta este aspecto para determinar qué operaciones puede realizar dentro de un script, y con qué usuario debe invocarlo.

TAREAS

1. En una consola de comandos ejecute el comando `ps w` y localice el proceso asociado al shell responsable de la línea de comandos desde la que está trabajando.
2. Invoque dicho script mediante los distintos métodos de invocación. Para cada uno de ellos, analice la salida obtenida para determinar en cada caso cuál es el proceso shell que está interpretando el script, el usuario al que pertenece dicho proceso y si se ha abierto un subshell o no:
   - Explícita:

     `/bin/sh script_ejemplo.sh`

     `/bin/dash script_ejemplo.sh`

     `/bin/bash script_ejemplo.sh`
   - Implícita con ".":

     `. script_ejemplo.sh`
   - Implícita: compruebe que el script tiene permiso de ejecución y ejecútelo con:

     `./script_ejemplo.sh`

     Modifique el script eliminando el shebang, y vuelva a ejecutarlo. Analice si hay alguna diferencia respecto a la ejecución anterior.

Conforme se ha indicado en la introducción, si bien tanto bash como dash siguen el estándar POSIX, especialmente bash añade múltiples extensiones particulares, no soportadas por otros shells como dash. Consecuentemente, cada vez que diseñemos un script deberemos tener en cuenta el shell o shells que lo soportan, asegurando que sea invocado por uno de ellos. Para que se haga una idea de la importancia de este

aspecto, considere los dos scripts siguientes, basados en el uso de la estructura `for` (que se usará más adelante):

script_estandar.sh

```
#!/bin/sh for VAR in 0 1 2 3
do
   echo $VAR
done
```

script_bash.sh

```
#!/bin/bash
for ((VAR=0 ; VAR<4 ; VAR++ ))
do
   echo $VAR
done
```

Ambos scripts realizan la misma funcionalidad, pero `script_estandar.sh` está escrito bajo la sintaxis POSIX, mientras que `script_bash.sh` utiliza una sintaxis no estándar soportada por bash.

TAREAS

1. Invoque el guión `script_estandar.sh` (recuerde que debe disponer de ellos ya creados en el directorio `scripts/`) mediante los siguientes comandos, debiendo comprobar que todas funcionan correctamente y con igual resultado (la primera y segunda llamada realmente son la misma, usando el shell `dash`):

   `/bin/sh script_estandar.sh`

   `/bin/dash script_estandar.sh`

   `/bin/bash script_estandar.sh`

2. Ahora invoque el guión `script_bash.sh` mediante los siguientes comandos:

   `/bin/dash script_bash.sh`

   `/bin/bash script_bash.sh`

   Podrá comprobar cómo, debido a la sintaxis no estándar del script, la segunda invocación funciona, pero la primera (que emplea dash) da un error de sintaxis.

3 Sintaxis de Shell-scripts

En este apartado se describe el lenguaje de comandos shell definido en el estándar POSIX. Veremos el funcionamiento general del shell, así como su sintaxis. Puede encontrar la descripción detallada del estándar POSIX en el siguiente enlace: POSIX Shell Command Language

Aquí se resumen las características más utilizadas. Se recomienda acudir al estándar para obtener una descripción más detallada y exhaustiva.

3.1 Funcionamiento general del shell

El lenguaje shell es un lenguaje interpretado, en el que se leen líneas de texto (terminadas en \n), se analizan y se procesan. Las líneas a interpretar son leídas de:

- La entrada estándar (teclado por defecto). En este caso el shell se dice que es un shell interactivo.
- Un fichero shell-script.
- Los argumentos, con la opción `-c` al ejecutar el shell. Ejemplo: `bash -c "ls -l"`

Con las líneas leídas, el shell realiza los siguientes pasos (en este orden):

1. Se dividen las líneas en distintos elementos: palabras y operadores. Los elementos se separan usando espacios, tabuladores y operadores. El carácter `#` sirve para incluir un comentario, que se elimina del procesamiento.
2. Se distingue entre comandos simples, comandos compuestos y definiciones de función.
3. Se realizan distintas expansiones y sustituciones (ver más adelante). Se detecta el comando a ejecutar y los argumentos que se le van a pasar.
4. Se realizan las redirecciones de entrada/salida y se eliminan los elementos asociados a las redirecciones de la lista de argumentos. Las redirecciones de entrada/salida ya se han explicado en un tema anterior (si desea ampliar información, consulte el anexo).
5. Se ejecuta el elemento ejecutable, que podría ser una función, un comando interno del shell, un fichero ejecutable o un shell-script, pasando los argumentos como parámetros posicionales (ver más adelante).
6. Opcionalmente, se espera a que termine el comando y se guarda el código de salida.

> **Advertencia**
>
> A lo largo de la memoria se utilizará la palabra "ejecutar" para referirse a la ejecución de un programa binario (un programa compilado, por ejemplo), al inicio de la interpretación de un script o a la invocación de una función o comando interno. Se considerarán, por tanto, elementos ejecutables, programas binarios y scripts (con el permiso de ejecución), comandos internos y funciones.

Cuando se escriben comandos desde el teclado y se intenta introducir un elemento que está formado por más de una línea, una vez que teclee la primera línea y pulse `Intro`, el shell mostrará el indicador secundario de petición de orden `>` (en lugar del prompt), solicitándole que continúe escribiendo el elemento. Cuando el intérprete dé por concluida la introducción del elemento, la interpretará, volviendo a mostrar el prompt de la línea de comandos. Si utiliza el cursor `↑` para intentar ver el comando introducido, en general verá cómo el shell ha rescrito la entrada para aplicarle la sintaxis con la que todo el elemento es escrito en una sola línea. Si desea cancelar la introducción de una línea (o líneas) sin necesidad de borrar lo que lleva escrito, puede pulsar Ctrl-C.

TAREAS

Escriba el contenido del script `script_estandar.sh` visto en el apartado 2 directamente en una consola de comandos.

3.2 Entrecomillado y carácter de escape

El shell tiene una lista de caracteres que trata de manera especial (operadores) y una serie de palabras reservadas (palabras que tienen un significado especial para el Shell). Puede ver un listado de caracteres especiales y palabras reservadas en los apartados 2.2 y 2.4 del estándar.

Cuando queremos utilizar un carácter especial del shell o una palabra reservada del lenguaje sin que sea interpretada como tal o prevenir una expansión o sustitución indeseada (las expansiones y sustituciones se verán en un apartado posterior) es necesario indicárselo al shell mediante las comillas (simples o dobles) o el carácter de escape. Por ejemplo, para escribir el nombre de un fichero que contenga espacios, para pasar el símbolo `<` como argumento a un programa.

- El carácter de escape `\` : indica que el siguiente carácter debe preservar su valor literal. El carácter de escape se elimina de la línea una vez procesado. Si aparece al final de una línea, significa "continuación de línea" e indica que el comando continúa en la siguiente línea (puede ser utilizado para dividir líneas muy largas).
- Comillas simples `' '` : todo texto 'entrecomillado' con comillas simples mantendrá su valor literal, no se producirá ninguna expansión ni sustitución y será considerado como una única palabra.
- Comillas dobles `"` : es equivalente a usar comillas simples, salvo que en este caso sí se hacen expansiones y sustituciones (todas menos las expansiones de tilde y ruta y la sustitución de alias que veremos más adelante).

El entrecomillado de una cadena vacía (`''` o `""`) genera una palabra vacía (palabra que no tiene ningún carácter).

Vea el apartado 2.2 del estándar para obtener información detallada.

TAREAS

1. Ejecute los siguientes comandos en el terminal y analice los resultados:

```
cd
echo $PWD
echo \$PWD
echo '$PWD'
```

```
echo "$PWD"
echo hola \> a y b
#se crea el fichero a
echo hola > a y b
ls
cat a
#se crea el fichero 'a y b'
echo hola >"a y b"
ls
cat a\ y\ b
```

3.3 Parámetros y variables

Como en cualquier lenguaje de programación, en el lenguaje shell se pueden crear y utilizar variables, que aquí se llaman parámetros. Existen varios tipos de parámetros:

1. Si el nombre es un número se denominan parámetros posicionales.
2. Si el nombre es un carácter especial se denominan parámetros especiales.
3. El resto se denominan simplemente variables.

A continuación se detallan cada uno de estos tipos. Vea el apartado 2.5 del estándar para obtener más información.

3.3.1 Variables

El shell permite realizar las siguientes operaciones básicas con las variables:

Sólo Definición	`VAR=""` `VAR=`
Definición y/o Inicialización/Modificación	`VAR=valor`
Expansión (Acceso a Valor)	`$VAR` `${VAR}`
Eliminación de la variable	`unset VAR`

Sobre ello deben realizarse las siguientes observaciones respecto a:

1. Definición y uso:
   - Las variables sólo existen en el proceso shell en que son definidas (locales al proceso).
   - Las variables sólo son accesibles desde el momento de su definición hacia abajo del script, esto es, siempre deben definirse primero e invocarse después (no puede usarse una variable que es definida más adelante). Pueden utilizarse también dentro de funciones (ver más adelante) aunque se hayan declarado fuera de ellas. En el estándar POSIX todas las variables son globales, aunque existen variantes (como `bash`) que permiten la creación de variables locales a las funciones.

   No es necesario definir las variables previamente a su uso, ya que se crean al asignarles la primera vez un valor. Al definir una variable sin inicialización, su valor por omisión es la cadena nula, esto es, las siguientes entradas son equivalentes:

   `VAR=`

   `VAR=""`

   Con el comando `set` (sin argumentos) puede ver todas las variables (y funciones) definidas en el shell actual.

2. Nombrado: téngase en cuenta que Linux es "case sensitive" (sensible a mayúsculas y minúsculas) en general, lo que incluye el nombre de las variable. Así, `VAR` y `var` serán tomadas como dos variables independientes . Para el nombre de una variable puede usarse :
   - 1er carácter: una letra o el carácter de subrayado `_`.
   - 2º y posteriores caracteres: una letra, dígito o el carácter de subrayado.
3. Inicialización/Modificación del valor de una variable:
   - Es importante no incluir ningún espacio ni antes ni después del signo `=`. Si se hace, el shell intentará interpretar el nombre de la variable como un comando (recuerde que la sintaxis del shell es especialmente estricta en lo que a espacios se refiere).
   - El valor de una variable siempre es tomado por el shell como una cadena de caracteres.
   - Si el valor de una variable contiene caracteres especiales, espacios, u otros elementos que puedan ser malinterpretados por el shell, tendrá que ir entrecomillado o deberá incluir el carácter de escape donde sea necesario.
4. Expansión de una variable: el uso de las llaves `{}` sólo es necesario si justo tras el nombre de la variable se desean escribir más caracteres, sin añadir ningún espacio antes. Se verá más adelante con más detalle.

TAREAS

Mire el contenido del script `script_variables.sh`, que deberá contener lo siguiente:

`script_variables.sh`

```
echo "Mal definida":
echo "(orden no encontrada)":
VAR = 1
echo "Variables bien definidas:"
VAR=1
VAR1=2
var=3
echo "Variables: $VAR $VAR1 $var"
echo "Variable VAR: $VAR"
echo "Variable VAR1: $VAR1"
echo "VAR seguida de 1: ${VAR}1"
echo "Comillas dobles: $VAR"
echo 'Comillas simples: $VAR'
echo "Valor: $VAR-1"
```

Compruebe que dispone del permiso de ejecución general. Invóquelo y analice su funcionamiento.

3.3.1.1 Variables del shell

Existe un conjunto de variables que afectan al funcionamiento del shell. Muchas ya han sido analizadas en temas anteriores, por ejemplo: `HOME`, `PATH`, `LANG`,... Puede volver a ver una descripción de ellas en el apartado 2.5.3 del estándar.

Aquí vamos a destacar la variable `IFS` (Input Field Separators). El valor de esta variable es una lista de caracteres que se emplearán en el proceso de división de campos realizado tras el proceso de expansión (que se verá más adelante, en el apartado 3.4) y por el comando `read` (ver el apartado 3.8.1 de comandos internos). El valor por defecto es `<espacio><tab><nueva-línea>`. Podrá ver un ejemplo de uso de esta variable cuando realice los ejercicios propuestos (ver solución a script_user.sh).

3.3.2 Parámetros posicionales

Son los parámetros de la línea de comandos con la que se ha invocado al script (equivalen a la variable `argv` de C). Están denotados por un número y para obtener su valor se emplea `$X` o `${X}` para los parámetros del 1 al 9 y `${X}` para parámetros mayores (números de más de un dígito). Se pueden modificar con los comando `set` (los crea) y `shift` (los desplaza de posición).

3.3.3 Parámetros especiales

Son parámetros identificados por un carácter especial creados por el Shell y cuyo valor no puede ser modificado directamente. En esta tabla se muestran los parámetros especiales definidos en el estándar:

Parámetro especial	Valor
`$*`	Se expande a todos los parámetros posicionales desde el 1. Si se usa dentro de comillas dobles, se expande como una única palabra formada por los parámetros posicionales separados por el primer carácter de la variable `IFS` (si la variable `IFS` no está definida, se usa el espacio como separador y si está definida a la cadena nula, los campos se concatenan).
`$@`	Se expande a todos los parámetros posicionales desde el 1, como campos separados, incluso aunque se use dentro de comillas dobles.
`$0`	Nombre del shell o shell-script que se está ejecutando.
`$-` (guion)	Opciones actuales del shell (modificables con el comando set). Consulte las opciones disponibles con el comando `man dash` .
`$#`	Nº de argumentos pasados al script (no incluye el nombre del script).
`$?`	Valor devuelto por el último comando, script, función o sentencia de control invocado. Recuerde que, en general, cualquier comando devuelve un valor. Usualmente, cuando un comando encuentra un error devuelve un valor distinto de cero.

`$$`	PID del proceso shell que está interpretando el script.
`$!`	PID del último proceso puesto en segundo plano.

Mire el contenido del script `script_var-shell.sh`, que deberá contener lo siguiente:

TAREAS

`script_var-shell.sh`

```
#!/bin/sh
echo \$@=$@
echo \$*=$*
echo \$0=$0
echo \$1=$1
echo \$2=$2
echo Cambio parametros posicionales
set uno dos tres
echo \$1=$1
echo \$2=$2
echo Desplazo
shift
echo \$1=$1
echo \$2=$2
echo \$-=$-
echo \$#=$#
echo \$?=$?
firefox &
ps w
echo \$$=$$
echo \$!=$!
```

Compruebe que dispone del permiso de ejecución. Invóquelo el comando y analice la salida:

`./script_var-shell.sh arg1 arg2`

3.3.4 Exportación de variables

Cuando un proceso (proceso padre, como por ejemplo el shell) ejecuta otro proceso (proceso hijo, otro programa o script), el proceso padre, además de los parámetros habituales (`argc` y `argv` en C), le pasa un conjunto de variables de entorno al proceso hijo (cada lenguaje de programación tiene su método propio para obtenerlas y modificarlas). Las variables de entorno pasadas pueden ser utilizadas por el proceso hijo para modificar su comportamiento. Por ejemplo, un programa en C puede usar la

función `getenv` declarada en biblioteca estándar `stdlib.h` para obtener dichas variables (puede obtener más información ejecutando `man getenv`).

El comando interno del shell `export` permite que una variable (previamente definida o no) sea configurada para que su valor sea copiado a los procesos hijos que sean creados desde el shell actual (por ejemplo otros shell). Presenta la sintaxis:

`export VAR`

`export VAR=valor`

En este último caso, sí es posible añadir un espacio antes o después del signo `=`.

Debe advertirse que "exportación" significa "paso de parámetros por valor", esto es, en el proceso hijo se creará una variable de igual nombre que en el shell padre, y con igual valor, pero serán variables independientes (esto es, la modificación de valor de la variable en el proceso hijo no afectará al valor de la variable en el shell padre). **El proceso hijo no puede crear ni modificar variables del proceso padre**.

TAREAS

Mire el contenido de los siguientes scripts en su sistema:

script_padre.sh

```
#!/bin/bash
export VAR=a
echo $VAR
ps w
./script_hijo.sh
echo $VAR
```

script_hijo.sh

```
#!/bin/sh
echo $VAR
VAR=b
echo $VAR
ps w
```

Compruebe que dispone del permiso de ejecución. Ejecute el comando:

`./script_padre.sh`

Analice el resultado.

No debe confundirse la exportación con la invocación de shell-scripts mediante el mecanismo implícito basado en `.`. En este caso no hay ninguna copia de variables por valor, simplemente el script invocado es interpretado por el mismo shell.

TAREAS

Mire el contenido de los siguientes scripts en su sistema:

script1.sh

```
#!/bin/bash
VAR=a
echo $VAR
ps w
. script2.sh
echo $VAR
```

script2.sh

```
#!/bin/sh
echo $VAR
VAR=b
echo $VAR
ps w
```

Compruebe que dispone del permiso de ejecución general. Ejecute el siguiente commando y analice el resultado:

`./script1.sh`

En los S.O.'s Linux suele ser habitual encontrar scripts que se dedican exclusivamente a contener la inicialización de un conjunto de variables, o la definición de un conjunto de funciones. Otros scripts del sistema hacen uso del mecanismo de invocación implícito basado en `.`, para cargar o importar las variables o funciones definidas en dichos scripts.

cont_func.sh

```
#!/bin/sh
fun1(){...}
fun2(){...}
#...
```

cont_var.sh

```
#!/bin/sh
VARa=1
VARb=2
#...
```

`script_sistema.sh`

```
#!/bin/sh
. /dir/cont_var.sh
. /dir2/cont_fun.sh
fun1 $VARb
#...
```

Por ejemplo, visualice el contenido del script del sistema encargado del arranque de los servicios `/etc/init.d/rc`. Observe cómo contiene las líneas:

`. /etc/default/rcS`

`. /lib/lsb/init-functions`

TAREAS

Visualice el contenido de esos archivos `rcS` e `init-functions` y compruebe cómo sólo contienen definiciones de variables y funciones (la sintaxis para la definición de las funciones se analizará más adelante), respectivamente. De hecho, todos los ficheros de la carpeta `/etc/default/` son scripts dedicados exclusivamente a contener la inicialización de variables, siendo importadas desde otros scripts del sistema. Por ejemplo, observe cómo:

- La variable `VERBOSE` es inicializada en `rcS` y luego es usada por `rc`.
- La función `log_failure_msg()` es definida en `init-functions` y luego es usada por `rc`.

3.4 Expansiones y sustituciones

Como se vio en el apartado de funcionamiento general del shell (apartado 3.1), en un paso previo a la ejecución del elemento ejecutable se realizan una serie de expansiones y sustituciones. En este apartado se describe cuáles son y cómo se realizan. Puede ver una descripción detallada en el apartado 2.6 del estándar.

Existen los siguientes tipos de expansiones y sustituciones (que desarrollaremos más adelante):

1. Expansión de ~ (virgulilla o tilde de la ñ).
2. Expansión de parámetros o variables.
3. Sustitución de comando.
4. Expansión aritmética.
5. Expansión de ruta.

Aparte de estas expansiones está el concepto de alias (ver apartado 2.3.1 del estándar) que se utiliza para crear sinónimos de comandos y sólo se realiza en el elemento ejecutable de la línea antes de cualquier otra expansión o sustitución. Su utilización es muy limitada y puede conseguirse un comportamiento similar usando funciones que permiten además parámetros (las funciones se verán más adelante).

Puede ver los alias definidos usando el comando `alias`. Puede ver ejemplos de definición de alias en el fichero `~/.bashrc`.

El orden de ejecución de estas expansiones es el siguiente:

1. Primero se hacen en orden las siguientes expansiones:
   - Expansión de ~.
   - Expansión de parámetros o variables.
   - Sustitución de comando.
   - Expansión aritmética.
2. Los campos generados en el primer paso son divididos utilizando como separadores los caracteres de la variable `IFS`. Si la variable `IFS` es nula (cadena nula) no se produce la división. Si la variable no está definida, se utilizan por defecto los caracteres espacio, tabulador y nueva línea.
3. Se realiza a continuación la última expansión:
4. Expansión de ruta.
5. Por último, se eliminan las comillas simples o dobles que existan y se utilicen como tal (no se eliminan las comillas que hayan perdido su significado especial, por ejemplo usando el carácter de escape).

Puede utilizar las expansiones en cualquier ubicación donde se pueda usar una cadena de texto (incluido el nombre de comandos), exceptuando las palabras reservadas del lenguaje (`if`, `else`, …).

3.4.1 Expansión de ~

Las apariciones de la virgulilla (o tilde de la ñ) dentro de una línea, que no se encuentren entrecomilladas, se expanden de la siguiente manera:

Variables	Valor
`~`	Se expande al valor de la variable `HOME`
`~login`	Si "login" es un nombre de usuario del sistema, se expande a la ruta absoluta del directorio de inicio de sesión de ese usuario. Si no, no se expande.

TAREAS

Comente la línea del script `script_var-shell.sh`, visto en el apartado 3.3, donde aparece la palabra "firefox". Ejecute los siguientes comandos observando el valor de las variables posicionales:

```
./script_var-shell.sh ~ ~root
```

```
./script_var-shell.sh ~noexiste ~dit
```

3.4.2 Expansión de parámetros y variables

El formato general para incluir una expansión de variables o parámetros, como se ha visto en apartados anteriores es:

```
${PAR}
```

Las llaves pueden omitirse, salvo cuando se trate de un parámetro posicional con más de un dígito o cuando se quiera separar el nombre de la variable de otros caracteres. Por ejemplo:

```
echo $PAR   #puede omitirse
```

```
echo ${10}   #no puede omitirse
```

```
${PAR}TEXTO  #no se omite
```

Si la expansión de parámetros ocurre dentro de comillas dobles, sobre el resultado no se realizará la expansión de ruta ni la división de campos (pasos b y c). En el caso del parámetro especial `@`, se hace la división de campos siempre.

Aparte de este formato general, se pueden utilizar otros formatos que permiten establecer valores por defecto, comprobar si la variable está definida, eliminar sufijos y prefijos, contar el número de caracteres, etc. Puede ver la lista de formatos junto con ejemplos en el apartado 2.6.2 del estándar. Por ejemplo, los siguientes:

`${PAR:-alternativo}`	Valor de la variable. Si la variable no tiene ningún valor, la construcción se sustituye por el valor `alternativo`.
`${PAR:=alternativo}`	Ídem al anterior, pero asignando el valor `alternativo` a la variable.
`${PAR%sufijo}`	Elimina el `sufijo` más pequeño del valor de la variable. `sufijo` es un patrón como los utilizados en la expansión de ruta. Si en vez de `%` se pone `%%` se elimina el `sufijo` más grande.
`${PAR#prefijo}`	Elimina el `prefijo` más pequeño del valor de la variable. `prefijo` es un patrón como los utilizados en la expansión de ruta. Si en vez de `#` se pone `##` se elimina el `prefijo` más grande.

Los siguientes scripts muestran un posible ejemplo de ambas construcciones:

script_expansion1.sh

```
#!/bin/sh
VAR=1
echo $VAR
unset VAR
echo ${VAR:-2}
echo $VAR
FICH=fichero.c
echo ${FICH%.c}.o
```

`script_expansion2.sh`

```
#!/bin/sh
VAR=1
echo $VAR
unset VAR
echo ${VAR:=2}
echo $VAR
FICH=/usr/bin/prueba
echo ${FICH##*/}
```

TAREAS Mire el contenido de los scripts anteriores en su sistema, invóquelos y analice los resultados.

3.4.3 Sustitución de comando

Permite que la salida estándar de un programa se utilice como parte de la línea que se va a interpretar.

Existen dos opciones, con el mismo funcionamiento:

`$(comando)`

`` `comando` ``

En el segundo caso se está utilizando la tilde francesa o acento grave, que no debe confundirse con las comillas simples. Para escribirla, hay que pulsar la tecla correspondiente a `` ` `` y pulsar espacio.

El shell ejecutará `comando`, capturará su salida estándar y sustituirá `$(comando)` por la salida capturada.

Por ejemplo, para almacenar en una variable el nombre de todos los ficheros con extensión .sh del directorio actual, podría escribir:

`` VAR=`ls *.sh` ``

O, por ejemplo, para matar el proceso con nombre `firefox-bin`, podría usar:

`kill -9 $(pidof firefox-bin)`

3.4.4 Expansión aritmética

El formato para realizar una expansión aritmética es el siguiente:

`$((expresión))`

Permite evaluar las cadenas indicadas en la expresión como enteros, admitiendo gran parte de los operadores usados en el lenguaje C, pudiendo usar paréntesis como parte de la expresión y el signo `-` para números negativos (a las cadenas que contengan letras se les asigna el valor `0`). Tras la evaluación aritmética, el resultado vuelve a ser convertido a una cadena. La conversión de un número a un carácter puede realizarse con `$'\xxx'` (en `bash`) o con `'\xxx'` (en `dash`), ambos con comillas simples, pero ello no está recogido en el estándar POSIX.

Si se usan variables en la expresión, no es necesario que vayan precedidas por el carácter `$` si ya contienen un valor entero válido (sí es necesario para los parámetros posicionales y especiales).

Puede ver más detalles en el apartado 2.6.4 del estándar.

TAREAS

Mire el contenido del script `script_expansion3.sh`, que deberá contener lo siguiente:

`script_expansion3.sh`

```
#!/bin/sh
VAR=1
VAR=$VAR+1
echo $VAR
RES1=$(($VAR))+1
echo $RES1
VAR=1
RES2=$((VAR+1)) #VAR no necesita $
echo $RES2
VARb=b
echo $(($VARb+1)) #VARb necesita $
```

Compruebe que dispone del permiso de ejecución. Invóquelo mediante el comando:

`./script_expansion3.sh`

Analice el resultado.

3.4.5 Expansión de ruta

Los campos que incluyan los caracteres `*`, `?` y `[` (asterisco, interrogación y apertura de corchetes) no entrecomillados serán sustituidos por la lista de ficheros que cumplan ese patrón. Si no hay ningún fichero con ese patrón no se sustituye nada. El uso de estos caracteres ya se ha utilizado en otros cursos. Puede ver más detalles en el apartado 2.13 del estándar.

TAREAS

Utilice el script `script_var-shell.sh`, visto en el apartado 3.3, pero modifíquelo eliminando las líneas con la palabra "firefox", para evitar que moleste en las siguientes pruebas. Ejecute los siguientes comandos observando el valor de la variable especial `@`:

```
./script_var-shell.sh s*_for?.sh
```

```
./script_var-shell.sh s*_for*.sh
```

```
./script_var-shell.sh s*_exp*.sh
```

```
./script_var-shell.sh s*_exp*[12].sh
```

```
./script_var-shell.sh s*_e*.sh
```

3.5 Comandos del shell

Un comando puede ser clasificado en las siguientes tipos de comandos (de menor a mayor nivel):

a. Comandos simples
b. Tuberías
c. Listas AND-OR
d. Listas
e. Listas compuestas
f. Comandos compuestos (o estructuras de control)
g. Definiciones de función

Cada uno de estos tipos se forma mediante la composición de elementos de los tipos inferior. Pero también se permite anidar distintos tipos de comandos (no todas las combinaciones son posibles como se verá más adelante), por lo que podríamos encontrarnos: tuberías de comandos compuestos, comandos compuestos formados por otros comandos compuestos, etc.

En general, el valor devuelto por un comando compuesto (tipo b y superiores) será el valor devuelto por el último comando simple ejecutado.

A continuación describiremos cada uno de los tipos. Puede ver más detalles en el apartado 2.9 del estándar.

3.5.1 Comandos simples

Un comando simple está formado por (todos los elementos son opcionales) una secuencia de asignación de variables y redirecciones (en cualquier orden) seguida de palabras (elemento ejecutable y sus argumentos) y redirecciones. A continuación se muestra la estructura genérica de un comando simple (los corchetes `[ ]` indican qué elementos son opcionales, que es la notación usada en el resto de este documento):

```
[VAR=v] [redir] [ejecutable argumentos] [redir]
```

Pudiendo ser el `ejecutable` programas "ejecutables" (comandos internos y ejecutables externos) e "interpretables" (funciones).

Es decir, en un mismo comando simple se puede hacer simultáneamente la asignación de variables y la ejecución de un programa. Cuando un comando simple no incluye un

programa a ejecutar, la asignación de variables afecta a todo el shell, de lo contrario la asignación sólo afecta al programa que se va a ejecutar.

Si en un mismo comando simple hubiera que hacer expansiones en las palabras y en la asignación de variables, primero se hace las expansiones de las palabras.

Ejemplos de comandos simples:

```
VAR=x
```

El anterior comando asigna el valor `x` a la variable `VAR` y afecta a todo el proceso shell actual.

```
VAR=x programa
```

Asigna el valor `x` a la variable `VAR` y afecta solo al programa.

```
VAR=y OTRA=z
```

```
VAR=x programa $VAR
```

```
echo $VAR
```

Se asigna el valor `y` a la variable `VAR` y el valor `z` a la variable `OTRA`, que afectan a todo el shell. A continuación, asigna el valor `x` a la variable `VAR` y afecta sólo al programa, al cual se le pasa como primer argumento `y`. A continuación se imprime `y` por pantalla.

```
VAR=x > fichero programa
```

```
VAR=x programa > fichero  #equivalente
```

Ambas líneas, asignan el valor `x` a la variable `VAR` que afecta solo al programa. Se ejecuta el programa y la salida estándar se redirige al archivo `fichero`. La redirección se realiza independientemente de que aparezca antes o después del programa. Si hubiera varias redirecciones se realizan en el orden de aparición en la línea, de izquierda a derecha.

3.5.2 Tuberías

Una tubería es una secuencia de uno o más comandos (simples o compuestos, pero no ningún tipo de lista) separados por el operador `|`. La salida estándar de un comando se conecta a la entrada estándar del siguiente comando (cada comando se ejecuta en otro subshell simultáneamente). Esta conexión se hace previamente a cualquier redirección. El formato es:

```
[ ! ] comando1 [ | comando2 … ]
```

Opcionalmente, se puede añadir delante el carácter `!` que hace la negación lógica del valor devuelto por el último comando, de tal manera que el valor devuelto por la tubería sería 1 si el último comando devuelve 0, o 0 en caso contrario.

3.5.3 Listas AND-OR

Una lista AND es una secuencia de tuberías (tenga en cuenta que una tubería puede ser sólo un comando simple) separadas por el operador `&&`. El formato es:

```
tuberia1 [ && tuberia2 … ]
```

Se van ejecutando las tuberías de izquierda a derecha hasta que una de ellas devuelva un valor distinto de cero. No se realiza ninguna expansión en una tubería hasta que el shell no determine que tiene que ejecutar dicha tubería (dependerá del resultado de la tubería anterior).

Una lista OR es una secuencia de tuberías separadas por el operador `||`. El formato es:

```
tuberia1 [ || tuberia2 … ]
```

Se van ejecutando las tuberías de izquierda a derecha hasta que una de ellas devuelva un valor cero. No se realiza ninguna expansión en una tubería hasta que el shell no determine que tiene que ejecutar dicha tubería.

Una lista AND-OR es el resultado de combinar listas AND y/o OR en una misma línea. Los operadores `&&` y `||` se evalúan con la misma prioridad de izquierda a derecha. Ejemplo:

```
tuberia1 || tuberia2 && tuberia3
```

3.5.4 Listas

Las listas son secuencias de una o más listas AND-OR separadas por los operadores `;` o `&`. Los operadores `;` y `&` no pueden aparecer seguidos (por ejemplo, daría error `prog1 & ; prog2`)

Según el operador las listas pueden ser secuenciales, asíncronas o mixtas (combinación de ambas).

3.5.4.1 Listas secuenciales

Se utiliza como separador el operador `;`. Se van ejecutando los distintos comandos secuencialmente (no se ejecuta un comando hasta que haya terminado el anterior). Cada lista AND-OR debe estar terminada por el operador `;` a excepción de la última donde es opcional. El formato es:

```
listaAND-OR1 [ ; listaAND-OR2 … ] [ ; ]
```

3.5.4.2 Listas asíncronas

Se utiliza como separador el operador `&`. Se van ejecutando los distintos comandos sin esperar a que el comando anterior termine (ejecución en segundo plano). El formato es:

```
listaAND-OR1 & [ listaAND-OR2 & ]
```

En este caso, a menos que se haga una redirección explícita de la entrada estándar, si un programa en segundo plano lee de la entrada estándar recibirá un error de fin de

fichero (EOF).

3.5.4.3 Listas mixtas

Son combinaciones de listas secuenciales y asíncronas. Por ejemplo:

```
#asíncrona y secuencial
lANDOR1 & lANDOR2 [ ; ]

#secuencial y asíncrona
lANDOR1 ; lANDOR2 &

#asíncrona y secuencial
lANDOR1 & lANDOR2 & lANDOR3 ; lANDOR4

#secuencial, asíncrona, secuencial
lANDOR1 ; lANDOR2 & lANDOR3
```

3.5.5 Listas compuestas

No es más que una secuencia de listas (apartado 3.5.4), separadas por el carácter de nueva línea (intros), terminada por el operador `;`, el operador `&`, el carácter de nueva línea (intro) o un comando compuesto. La utilidad de este tipo de listas se verá sobre todo cuando se expliquen los comandos compuestos.

TAREAS

Mire el contenido del script `script_operadores.sh`, que deberá contener lo siguiente:

`script_operadores.sh`

```
#!/bin/sh
head -1 /etc/passwd && echo "Sin error1A" || echo "Con error1B"
head -1 /nofile && echo "Sin error2A" || echo "Con error2B"
echo "Comando dividido \
en dos líneas"
echo "Sin escapado: $$"
echo "Con escapado: \$\$"
echo "N º de proceso del shell bash:" `pidof bash`
```

Compruebe que dispone del permiso de ejecución. Invóquelo y analice su funcionamiento.

Desde la línea de comandos, cree listas y tuberías de todos los tipos vistos usando combinaciones de los comandos `ls`, `echo`, `cat` y `ps`.

3.5.6 Comandos compuestos o estructuras de control

Los comandos compuestos son lo que en otros lenguajes de programación se conocen como estructuras de control. Cada uno de estos comandos compuestos (o estructuras de control) están delimitados por una palabra reservada u operador de control al

principio y otro al final (terminador). Si se hace una redirección a continuación del terminador, en la misma línea, esa redirección se aplicará a todos los comandos que se encuentre en ese comando compuesto, a menos que se haga otra redirección explícita en un comando en concreto.

A continuación se hace un repaso por todas las estructuras de control disponibles en el estándar POSIX. Puede consultar el apartado 2.9.4 del estándar para más información.

3.5.6.1 Secuencial (agrupación de comandos)

La agrupación de comandos permite mejorar la legibilidad del código, aplicar una redirección a un conjunto de comandos y crear un subshell entre otras cosas.

Existen dos formas de agrupar comandos, con los siguientes formatos:

- `( lista-compuesta )`

  Se ejecuta la lista compuesta en un subshell. Los cambios que se produzcan en este subshell no afectarán al shell actual. Si la lista compuesta está terminada por el carácter de nueva línea, este carácter puede omitirse.

- `{ lista-compuesta }`

  Se ejecuta la lista compuesta en el shell actual. Recuerde que las listas compuestas están terminadas por los operadores `;`, `&` o nueva línea (el último comando debe estar separado de la llave de cierre por esos operadores).

En ambos casos, se permite añadir una redirección al final (detrás del `)` o `}`) que afectará a todos los comandos del grupo.

3.5.6.2 Condicional: if-elif-else

Presenta la siguiente sintaxis:

```
if lista-compuestaA1 then
    lista-compuestaB1
elif lista-compuestaA2 then
    lista-compuestaB2
...
else
  lista-compuestaN
fi
```

Las entradas `elif` tienen el significado de un `else` seguido de un nuevo `if`. Puede haber tantas entradas `elif` como se desee. Tanto las entradas `elif` como la entrada `else` son opcionales.

En esta estructura, lo primero que se hace es ejecutar la `lista-compuestaA1`, si el valor devuelto es 0 (ADVERTENCIA: 0 significa verdadero aquí), se ejecutaría `lista-compuestaB1` y terminaría la estructura `if`. Si el valor devuelto no es 0 se comprueba el siguiente `elif`. Si ninguna de las listas-compuestas A devuelve 0 se

ejecutaría el bloque del `else`. En otras palabras, las listas compuestas B solo se ejecutan si se ha comprobado su respectiva lista-compuesta A y ha devuelto 0.

A veces, para mejorar la legibilidad, las listas-compuestas se encierran entre llaves (agrupación de comandos) pero es opcional. Asimismo, `then` estará en una línea u otra dependiendo del operador (`&` , `;` o nueva línea) utilizado para terminar la lista compuesta. Si desea dejar una lista-compuesta vacía (no quiere realizar ninguna operación en un determinado caso), puede utilizar el comando `:` (comando nulo).

Por ejemplo, si tenemos un programa llamado `condicion` que devuelve 0 si algo es verdadero y 1 si es falso, los siguientes ejemplos son equivalentes:

<table>
<tr>
<td>

```
if condicion; then
 { comando1; comando2; }
fi
```

</td>
<td>

```
if condicion; then
 comando1; comando2;
fi
```

</td>
</tr>
<tr>
<td>

```
if condicion
then
 comando1;
 comando2;
fi
```

</td>
<td>

```
if condicion
then
 comando1
 comando2
fi
```

</td>
</tr>
<tr>
<td colspan="2">

```
if condicion; then comando1; comando2; fi
```

</td>
</tr>
<tr>
<td colspan="2">

```
if condicion; then { comando1; comando2; } fi
```

</td>
</tr>
</table>

Recuerde que si usa las llaves, debe separarlas del resto de elementos. Por ejemplo:

`if condicion; then {comando1;  comando2;} fi`

`if condicion; then{ comando1;  comando2;} fi`

`if condicion; then { comando1; comando2} fi`

darán error de sintaxis.

Respecto a la `condición` ; que puede usarse, basta cualquier lista compuesta que devuelva un valor (por ejemplo, pueden usarse los comandos internos `true` o `false`). El valor de una lista compuesta es el valor del último comando simple ejecutado en la lista compuesta.

Un programa habitual que se utiliza como condición es el programa **`test`** . El comando `test` se puede ejecutar de dos formas (ambas equivalentes):

`test expresion`

```
[ expression ]  #los [] deben estar separados
```

En la segunda forma los corchetes no son operadores ni indican que la expresión sea opcional, sino que realmente son el nombre del programa.

Puede ver la descripción del comando `test` según el estándar.

Como expresiones más habituales pueden usarse las siguientes:

Tipo	Expresión	Verdadera sí (devuelve 0)
Enteros (n1 y n2 se convierten a enteros)	`n1 -eq n2`	n1 = n2
	`n1 -ne n2`	n1 ≠ n1
	`n1 -gt n2`	n1 > n2
	`n1 -ge n2`	n1 ≥ n2
	`n1 -lt n2`	n1 < n2
	`n1 -le n2`	n1 ≤ n2
Cadenas	`"$VAR" = "cad"`	$VAR vale "cad". **Es conveniente, pero no necesario, poner la variable entre comillas por si tuviera espacios o estuviese vacía, para que al expandirse no dé error de sintaxis.**
	`"$VAR" != "cad"`	$VAR vale algo distinto de "cad".
	`-z "$VAR"` `"$VAR"`	$VAR está vacía. Equivale a `"$VAR" = ""`

Tipo	Expresión	Verdadera sí (devuelve 0)
	`-n "$VAR"`	$VAR no está vacía. Equivale a `"$VAR" != ""` o `! -z`
Ficheros	`-e "$FILE"`	$FILE existe. Si se indica un enlace simbólico, será cierta sólo si existe el enlace simbólico y el fichero apuntado. Es conveniente que esté entre comillas por el mismo motivo anterior.
	`-f "$FILE"`	$FILE existe y es regular. Si se indica un enlace simbólico, el tipo es el del fichero apuntado.
	`-h "$FILE"`	$FILE existe y es un enlace simbólico
	`-d "$DIR"`	$DIR existe y es un fichero de tipo directorio
	`-p "$FILE"`	$FILE existe y es un fichero especial tubería (pipe)
	`-b "$FILE"`	$FILE existe y es un fichero especial de bloques
	`-c "$FILE"`	$FILE existe y es un fichero especial de caracteres
	`-r "$FILE"`	$FILE existe y puede leerse
	`-w "$FILE"`	$FILE existe y puede modificarse

Tipo	Expresión	Verdadera sí (devuelve 0)
	`-x "$FILE"`	$FILE existe y puede ejecutarse
	`-s "$FILE"`	$FILE existe y su tamaño es mayor de cero bytes

Cualquiera de las condiciones anteriores puede ser precedida por el operador negación `!`, en cuyo caso la condición será cierta si no se satisface la comparación indicada. Por ejemplo, `! -d $DIR` se cumplirá si `$DIR` NO es un directorio.

Asimismo, se permite crear condiciones múltiples mediante los operadores:

`condicion1 -a condicion2`	AND: Verdadero si ambas condiciones son verdaderas
`condicion1 -o condicion2`	OR: Verdadero si se cumple alguna de las dos condiciones

Recuerde las restricciones de sintaxis del shell en lo que respecta a los espacios, especialmente importantes en la escritura de las condiciones. Por ejemplo, la siguiente entrada dará error de sintaxis (el espacio tras `;` sí puede omitirse):

```
if[ condicion ]; then
```

Y la siguiente dará error porque buscaría el comando `[comando]` (incluyendo los corchetes como parte del nombre del comando), que en general no encontrará (mostrando un mensaje de orden no encontrada).

```
if [comando]; then
```

TAREAS

1. El comando test se puede ejecutar sin necesidad de utilizarlo en una estructura de control. Escriba la siguiente instruccion y analice su comportamiento (pruebe asignando también 1 a la variable V y una cadena de texto):

   ```
   V=0; [ $V -eq 0 ] && { echo ES; echo 0; } || echo ES 1
   ```

2. El siguiente comando imprime por pantalla si el usuario actual es root. Ejecútelo como root y como un usuario normal:

`if [ "`id -u`" -eq 0 ]; then echo ROOT; fi`

3. Mire el contenido del siguiente script en su sistema y compruebe que tiene el permiso de ejecución:

script_if.sh

```
#!/bin/sh

FILE=/tmp/archivo
if [ -r $FILE -a ! -w $FILE ]; then
   echo Fichero $FILE existe y no es modificable
else
   echo Fichero no encontrado o es modificable
fi

VAR1=1; VAR2=1
if [ $(($VAR1)) -ne $(($VAR2)) ]; then
   echo Distintos
elif ls /; then
   :
fi
```

4. Ejecute los comandos siguientes y analice el resultado:

`rm –f /tmp/archivo`

`./script_if.sh`

5. Ejecute ahora los comandos siguientes y vuelva a analizar el resultado:

`touch /tmp/archivo`

`chmod –w /tmp/archivo`

`./script_if.sh`

3.5.6.3 Condicional: case

Presenta la siguiente sintaxis:

```
case cadena_texto in
  patron1) lista-compuesta1;;
  patron2) lista-compuesta2;;
  ...
  * ) lista-defecto [;;] #coincide con todo
esac
```

`cadena_texto` debe aparecer obligatoriamente en la misma línea que la palabra reservada `case` (la palabra reservada `in` puede estar en la siguiente línea). En esta estructura, primero se expande `cadena_texto` (si es necesario) y busca el primer patrón que encaje con dicho valor. Cuando lo encuentre, ejecuta la lista-compuesta correspondiente y finaliza la estructura. Los `patronN` se interpretan como cadenas de caracteres y si es necesario se expanden (por ejemplo, pueden contener variables). Admite los mismos caracteres que los usados para la expansión de ruta (`*` , `?` y `[]`). Asimismo, pueden usarse patrones múltiples mediante el operador `|` y opcionalmente pueden comenzar con el paréntesis:

```
patronA  | patronB)
```

```
(patronA  | patronB)
```

```
(patronC)
```

El doble punto y coma `;;` permite determinar el final de los elementos a interpretar cuando se cumpla su patrón asociado. Por ese motivo, el `;;` del último patrón puede omitirse. Es posible añadir espacios entre los patrones y los paréntesis abiertos `)` que marcan el final del patrón. Conforme a esto, serían sintaxis válidas alternativas las siguientes:

```
case cadena_texto in
  patron1)
       cmd1;
       cmd2;;
esac
```

```
case cadena_texto in
  patron1 )
       cmd1
       cmd2
esac
```

```
case cadena_texto in patron1) cmd1; cmd2;; esac
```

```
case cadena_texto in (patron1) cmd1; cmd2; esac
```

TAREAS

1. Mire el contenido del siguiente script en su sistema y compruebe que tiene el permiso de ejecución:

`script_case.sh`

```
#!/bin/sh

case $1 in
    archivo | file)
        echo Archivo ;;
    *.c)
        echo Fichero C ;;
    *)
        echo Error
        echo Pruebe otro ;;
esac
```

2. Ejecute los comandos siguientes y analice el resultado:

`./script_case.sh archivo`

`./script_case.sh file`

`./script_case.sh file.c`

`./script_case.sh file.c++`

3.5.6.4 Bucles incondicionales: for

Presenta la siguiente sintaxis:

```
for VAR in lista_valores; do
  lista-compuesta
done
```

El nombre de la variable `VAR` debe aparecer obligatoriamente junto con la palabra reservada `for` en la misma línea. `lista_valores` debe estar obligatoriamente en la misma línea que la palabra reservada `in`. El punto y coma `;` puede sustituirse por un salto de línea, y viceversa. Así, por ejemplo, serían sintaxis válidas las siguientes:

```
for VAR in lista_valores; do lista-compuesta done
```

```
for VAR
in lista_valores
do
  lista-compuesta
done
```

```
for VAR in lista_valores
do
  lista-compuesta
done
```

`lista_valores` se corresponde con un conjunto de valores (tomándose cada valor como una cadena de caracteres que puede ser objeto de expansión y como caracteres de separación los caracteres definidos en la variable `IFS`). La estructura `for` define la variable `VAR` (si no ha sido previamente definida). Para cada uno de los valores del resultado de expandir `lista_valores`, la estructura inicializa la variable `VAR` con dicho valor y realiza una iteración (ejecutando lista-compuesta, en la cual suele ser habitual acceder al valor de la variable `VAR`).

Es posible omitir `in lista_valores`. Si se omite equivale a haber escrito: `in "$@"`

TAREAS

1. Ejecute el siguiente comando y compruebe como se puede utilizar una expansión de ruta dentro de un bucle for:

   ```
   for i in ~/.*; do echo Fichero oculto: $i; done
   ```

2. Mire el contenido del siguiente script en su sistema, compruebe que tiene el permiso de ejecución, invóquelo y compruebe el contenido del fichero "ficherosalida" que crea:

   script_for1.sh

   ```
   #!/bin/sh

   for i in 1 2 3; do
       echo "Iteracion: $i"
   done > ficherosalida
   ```

   script_for2.sh

   ```
   #!/bin/sh

   for file in `ls /`; do
       echo "Fichero: $file"
   done
   ```

   Mire el contenido del siguiente script en su sistema, compruebe que tiene el permiso de ejecución e invóquelo.

Suele ser habitual el uso del comando externo `seq` para generar una lista de valores. Si bien este comando no está recogido en el estándar POSIX, es habitual su presencia en la mayoría de los sistemas UNIX. El comando `seq` presenta la sintaxis:

```
seq   valor_inicial   valor_final
```

siendo ambos valores números enteros. La salida del comando es la secuencia de números enteros entre ambos valores extremos indicados.

TAREAS

1. Ejecute el comando siguiente y observe su salida:

   ```
   seq 1 10
   ```

2. Mire el contenido del siguiente script en su sistema, compruebe que tiene el permiso de ejecución general e invóquelo:

`script_for_seq.sh`

```
#!/bin/sh

for i in `seq 1 3`; do
    echo "Iteracion: $i"
done
```

Compruebe cómo se obtiene el mismo resultado que se tenía al invocar el fichero `script_for1.sh`.

3.5.6.5 Bucles condicionales: while y until

Presentan la siguiente sintaxis:

```
while lista-comp-condicion do
  lista-compuesta
done
```

```
until lista-comp-condicion do
  lista-compuesta
done
```

La `lista-comp-condicion` es una lista compuesta que se rige por las mismas directrices indicadas en la estructura `if`. La estructura:

- `while` va iterando (interpreta la `lista-compuesta`) mientras se cumpla la condición indicada (`lista-comp-condicion` devuelve el valor 0)
- `until` va iterando (interpreta la `lista-compuesta`) mientras NO se cumpla la condición indicada (`lista-comp-condicion` devuelve un valor distinto de 0).

Así, por ejemplo, serían válidas y equivalentes las sintaxis siguientes, si la condición del `until` es la condición del `while` negada:

`while lista-comp-condW`
`do`
`  cmd1`
`  cmd2`
`done`
	`until lista-comp-condU`
	`do`
	`  cmd1`
	`  cmd2`
	`done`

`while lista-comp-condW ; do cmd1; cmd2; done`	
`while lista-comp-condW ; do { cmd1; cmd2; } done`	

TAREAS

1. Mire el contenido del siguiente script en su sistema, compruebe que tiene el permiso de ejecución e invóquelo:

   `script_while.sh`

   ```
   #!/bin/sh

   CONTADOR=0
   while [ $CONTADOR -lt 3 ]; do
       echo "Contador: $CONTADOR "
       CONTADOR=$(($CONTADOR+1))
   done
   ```

2. Mire el contenido del siguiente script en su sistema, compruebe que tiene el permiso de ejecución e invóquelo:

   `script_until.sh`

   ```
   #!/bin/sh

   CONTADOR=0
   until [ $CONTADOR -ge 3]]; do
       echo El contador es $CONTADOR
       CONTADOR=$(($CONTADOR+1))
   done
   ```

   Podrá comprobar cómo ambos scripts devuelven la misma salida.

3.5.6.6 Ruptura de sentencias de control

Igual que en otros lenguajes de programación, como en el lenguaje C, es posible romper el funcionamiento normal de las estructuras repetitivas (`for` , `while` y `until`). Sin embargo, hacerlo supone hacer código no estructurado. Por coherencia con lo visto en otras asignaturas, no se aconseja su uso.

En shell script esto se realiza con dos comandos internos: `continue` y `break`

Ambos son comandos internos de la shell con la siguiente sintaxis y funcionalidad:

a. `continue` : utilizado en estructuras de control repetitivas para detener la iteración actual y continuar con la siguiente. Su sintaxis es:

   ```
   continue  [n]
   ```

   El parámetro opcional `n` es un número entero positivo que permite especificar la estructura de control en la que se desea detener la iteración. Si se tienen varias estructuras de control anidadas, la estructura actual en la que se encuentra el `continue` corresponde a la estructura `1` ; la estructura superior que engloba a ésta sería la estructura `2` , y así sucesivamente. Así, el valor de `n` referencia a la estructura de control en la que deseamos detener la iteración actual y continuar con la siguiente (por omisión, "n=1").

b. `break` : utilizado en estructuras de control repetitivas para detener todas las iteraciones restantes de la estructura de control actual. Su sintaxis es:

   ```
   break  [n]
   ```

   El parámetro opcional `n` es un número entero positivo que permite indicar si se desean cancelar varias estructuras de control anidadas (por omisión, "n=1", que referencia a la estructura actual en la que se encuentra el `break`).

3.5.7 Funciones

Presentan la siguiente sintaxis:

Definición	`fnombre() comando-compuesto [redir]`
Invocación	`fnombre  [arg1  arg2 … ]`

El paréntesis siempre debe estar vacío (sólo indica que se está definiendo una función). Pueden insertarse espacios antes, entre y después del paréntesis. El comando compuesto puede ser cualquier de los que se han visto (agrupación de comandos, estructuras condicionales, estructuras repetitivas). Opcionalmente pueden aplicarse redirecciones a la función (afecta a los comandos que contiene, salvo que contengan una redirección explícita). A continuación se muestran ejemplos básicos de definición de funciones:

`fnombre(){`
`  comando1`
`  comando2`
`}`
	`fnombre(){`
	`  comando1;`
	`  comando2;`
	`}`

`fnombre() { comando1; comando2; }`	

En lo que se refiere al nombrado de las funciones, se aplican los mismos criterios antes expuestos para el nombrado de las variables.

El estándar permite que dentro de una función se invoque a otra. Los argumentos pasados a la función en su invocación son accesibles desde el cuerpo de la función mediante los parámetros posicionales `$1`, `$2`,..., `$9`, `${10}`,... Por tanto, dentro de la función, los parámetros posicionales no corresponden a los argumentos usados en la invocación del script.

Al igual que las variables, las funciones son:

- Locales: sólo existen en el proceso shell en que son definidas.
- Sólo son accesibles desde el momento de su definición hacia abajo del script, esto es, siempre deben definirse primero e invocarse después (no puede invocarse a una función que es definida más adelante).
- Dentro de la función son visibles todas las variables y funciones definidas antes de su invocación. Y las variables definidas dentro de la función son visibles fuera tras la invocación de la función.

Dentro del cuerpo de la función suele ser habitual el uso del comando `return`, el cual provoca la salida inmediata de la función con el valor de retorno (número) indicado:

```
return  [n]
```

Si no se indica ningún valor de retorno, la función devuelve el valor del último comando ejecutado. Como siempre, el valor devuelto por la función puede obtenerse con la variable `$?`. `return` también puede utilizarse para terminar un script invocado implícitamente con `.`.

TAREAS

Mire el contenido del siguiente script en su sistema, compruebe que tiene el permiso de ejecución, invóquelo con 2 números enteros como argumentos y analice su funcionamiento :

script_funcion.sh

```
#!/bin/sh

suma () {
    C=$(($1+$2))
    echo "Suma: $C"
    return $C
    echo "No llega"
}

suma 1 2
suma $1 $2 #suma los 2 primeros argumentos
echo "Valor devuelto: " $?
```

3.6 Uso de comandos y aplicaciones

Con objeto de alcanzar una mayor homogeneización entre los sistemas, el estándar POSIX recoge una lista de comandos que deben ser implementados en cualquier sistema, clasificándolos según sean ordenes internas del shell (built-in) o aplicaciones externas.

3.6.1 Comandos internos

Los comandos internos corresponden a órdenes interpretadas por el propio shell (luego no existe ningún fichero ejecutable asociado al comando). Se distinguen dos tipos de comandos internos:

- Especiales: un error producido al ejecutar este comando da lugar a que el shell termine. Por consiguiente, el script termina con un error y un valor de retorno mayor de cero (vea el apartado 2.8.2 del estándar para saber el valor de retorno).
- Regulares: el shell no tiene que terminar cuando se produce un error en estos comandos.

En la siguiente tabla se recogen los comandos especiales definidos en el estándar.

Comando interno especial	Descripción
`break`, `continue`, `export`, `return`, `unset`, `.`	Se han descrito anteriormente.
`:`	Comando nulo. Se suele utilizar en estructuras de control que requieren un comando para ser sintácticamente correctas, pero no se quiere hacer nada.
`eval`	Permite crear comandos a partir de sus argumentos (ver más adelante)
`exec`	Ejecuta comandos (sustituyendo al shell actual) y abre, cierra y copia descriptores de fichero
`exit`	Provoca que el shell termine (ver más adelante)
`readonly`	Permite hacer que una variable sea de sólo lectura (no admite asignaciones)
`set`	Establece opciones del proceso shell actual y modifica los parámetros posicionales.
`shift`	Elimina el número indicado de parámetros posicionales empezando desde el 1, desplazando el resto de parámetros a posiciones inferiores.

Comando interno especial	Descripción
`times`	Muestra los tiempos de procesamiento del shell y sus procesos hijos.
`trap`	Permite atrapar o ignorar señales del sistema.

Puede obtener más información en el estándar.

Como comandos internos regulares, el estándar define los siguientes:

Básicos regulares	`bg`, `cd`, `false`, `fg`, `jobs`, `kill`, `pwd`, `read`, `true`, `wait`
Para Profundizar regulares	`alias`, `command`, `fc`, `getopts`, `newgrp`, `umask`, `unalias`

Consulte el estándar para más información.

Entre dichos comandos (especiales y regulares), por su utilidad en los shell-scripts deben destacarse los siguientes:

3.6.1.1 Salida del proceso shell actual, `exit`

La sintaxis de este comando es:

```
exit [n]
```

`exit` provoca la eliminación inmediata del proceso correspondiente al shell que está leyendo el script. El parámetro opcional es un número entero que corresponde al valor devuelto por el script. Si no se indica ningún parámetro, el valor devuelto por el script será el del último comando ejecutado.

TAREA

1. Abra una consola de comandos. Mediante el comando `su` abra una sesión con el superusuario. Al abrir la sesión con `root`, se ha creado un nuevo proceso en memoria correspondiente al shell encargado del intérprete de comando en la sesión del `root`. Use el comando `ps ax` para localizar dicho proceso.

2. Ejecute el comando `exit`, volviendo al usuario normal. Esto habrá provocado la eliminación del proceso shell encargado de la línea de comandos como en la que trabajaba `root`. Ejecute de nuevo `ps ax` comprobando cómo dicho proceso ha desaparecido
3. Mire el contenido del script `script_exit.sh`, que deberá contener lo siguiente:

   `script_exit.sh`

   ```
   echo Dentro del script
   exit 3
   echo Fuera del script
   ```

4. Compruebe que dispone del permiso de ejecución general. Vuelva a usar el comando `su` para abrir una sesión con el superusuario. Desde dicha sesión ejecute el comando anterior mediante las siguientes invocaciones:

   `/bin/bash script_exit.sh`

   `./script_exit.sh`

   Ambas invocaciones provocan la creación de un subshell (un nuevo proceso shell) encargado de leer el script. Por ello, cuando se llega al comando `exit` se para la interpretación del script y dicho subshell es eliminado, volviendo al proceso shell padre correspondiente a la línea de comandos desde la que estábamos trabajando.

   Ejecute el comando `echo $?` para comprobar cómo el script ha devuelto el código de error "3".

5. Ejecute los siguientes comandos:
   - `script_exit.sh && echo Hola`, comprobando que no se imprime la cadena `Hola` debido a que el script devuelve un código de error (>0).
   - `script_exit.sh || echo Hola`, comprobando que ahora sí se imprime.
   - Edite el script para que el comando `exit` devuelva cero (`exit 0`), y vuelva a invocar el comando `script_exit.sh && echo Hola`, comprobando que ahora también se imprime la cadena `Hola`.
6. Vuelva a invocar el script pero ahora mediante:

   `. script_exit.sh`

   Verá cómo la sesión del `root` se cierra automáticamente y vuelve a la sesión del usuario normal. Esto se debe a que en este caso no se está creando ningún nuevo subshell, sino que el propio shell de la línea de comandos del usuario `root` es el encargado de interpretar el script. Al llegar al comando `exit`, éste provoca la eliminación del proceso shell que lee el script, esto es, la eliminación de la sesión del usuario `root`.

   Ejecute el comando `echo $?` para comprobar cómo el script ha devuelto el código de error `0` (sin error) correctamente.

7. Ejecute y analice el funcionamiento de los siguientes comandos:

```
sh -c "exit 0" && echo "Sin error1A" || echo "Con error1B"
```

```
sh -c "exit 1" && echo "Sin error2A" || echo "Con error2B"
```

3.6.1.2 Entrada estándar a un shell-script, `read`

El comando `read` read lee una línea de la entrada estándar (teclado) y la guarda en variables. Solo funciona en shell interactivos (leyendo la entrada del teclado), de lo contrario no hace nada. En su forma más básica, presenta la siguiente sintaxis:

```
read VAR1 [VAR2 …]
```

Este comando espera a que el usuario introduzca una línea de texto incluyendo espacios (la entrada termina cuando el usuario pulsa la tecla "Intro"; la pulsación "Intro" no forma parte del valor asignado a la cadena). Esta línea se divide en campos (según la variable `IFS`). Tras ello, el comando define las variables dadas como argumentos, inicializándolas con los campos obtenidos en la división. Si hay más campos que variables, los campos restantes se asignan a la última variable. Si hay más variables que campos, las variables sobrantes reciben como valor la cadena vacía `""` .

Consulte la página del estándar para obtener más información.

Algunos intérpretes de comandos como `bash` añaden otras opciones a este comando, como la posibilidad de imprimir un mensaje usando la opción `-p` (vea la ayuda de `read` en `bash` con el comando `man bash`).

TAREAS

Cree el script `script_read.sh` ; (éste, como es breve, no se le proporciona) que contenga lo siguiente:

`script_read.sh`

```
echo " Introduzca una cadena "
read CAD
echo " Cadena introducida: $CAD "
```

Asígnele el permiso de ejecución general, invóquelo y analice su funcionamiento.

El comando `read` también puede ser útil, por ejemplo, para detener la interpretación del script hasta que el usuario pulse una tecla:

TAREAS

Cree el script `script_read_pause.sh` que contenga lo siguiente:

`script_read_pause.sh`

```
echo "Pulse intro para continuar..."
read CAD
echo " Continuamos... "
```

Asígnele el permiso de ejecución general, invóquelo y analice su funcionamiento.

Resulta habitual el uso de estructuras `while`, combinadas con `case` y `read`, para crear menús interactivos, permitiendo mantenerse dentro del menú.

1. Edite el script `script_case_menu.sh` para que tenga el siguiente contenido:

TAREAS

`script_case_menu.sh`

```
#!/bin/sh
clear
SALIR=0
OPCION=0
while [ $SALIR -eq 0 ]; do
   echo "Menu:"
   echo "1) Opcion 1"
   echo "2) Opcion 2"
   echo "3) Salir"
   echo "Opcion seleccionada: "
   read OPCION
   case $OPCION in
       1)
           echo "Opcion 1 seleccionada" ;;
       2)
           echo "Opcion 2 seleccionada" ;;
       3)
           SALIR=1 ;;
       *)
         echo "Opcion erronea";;
   esac
done
```

   El comando `clear`, tampoco recogido en el estándar POSIX, también suele encontrarse habitualmente en la mayoría de los sistemas UNIX. Su funcionalidad es, simplemente, limpiar la información impresa en la consola de comandos.

2. Ejecute el comando siguiente y seleccione las opciones del menú:

   `script_case_menu.sh`

3.6.1.3 Construcción de comandos en tiempo de ejecución: `eval`

El comando `eval` construye un comando mediante la concatenación de sus argumentos (pueden ser variables, etc.) separados por espacios. Dicho comando

construido es leído por el shell e interpretado. La sintaxis del comando es:

```
eval [argumentos …]
```

Un posible uso es la creación de referencias indirectas a variables (parecido a usar punteros en lenguaje de programación C). En la tarea siguiente se muestra esto.

TAREAS

Cree el script `script_eval.sh` que contenga lo siguiente:

script_eval.sh

```
#ejemplo de referencia indirecta con eval
VAR="Texto"
REF=VAR              #REF es una variable que vale VAR
eval OTRA='$'$REF    #equivale a ejecutar OTRA=$VAR
echo $OTRA           #se ha accedido al contenido de VAR a través de REF
```

Asígnele el permiso de ejecución, invóquelo y analice su funcionamiento.

3.6.2 Comandos externos

Los comandos externos corresponden a ficheros ejecutables externos al shell. Cualquier posible aplicación pertenecería a esta categoría de comandos (`ps` , `firefox` , `emacs` ,...). El estándar POSIX recoge una lista de comandos externos que aconsejablemente deberían estar en un sistema UNIX, clasificándolos en obligatorios u opcionales según se exija o no su implementación para satisfacer el estándar. Entre los comandos externos obligatorios, el estándar define los siguientes:

Básicos	`cat` , `chmod` , `chown` , `cmp` , `cp` , `date` , `dirname` , `echo` , `expr` , `printf`
Para Profundizar	`awk` , `basename` , `chgrp`

TAREAS

Busque información sobre el comando `echo` y `printf` . Ejecute los siguientes comandos y analice su funcionamiento:

```
echo Uno
```

```
echo -n Uno; echo Dos
```

```
echo -e "Uno\nDos"
```

```
NOMBRE=Ana
```

```
printf "Hola %s. Adios %s\n" $NOMBRE $NOMBRE
```

4 Depuración de shell-scripts

Si bien la programación de shell-scripts no se puede depurar fácilmente, los shells suelen ofrecer algunos mecanismos en este aspecto. En concreto, tanto "bash" como "dash" ofrecen los siguientes argumentos, que pueden usarse simultáneamente:

`-x`	Traza: expande cada orden simple, e imprime por pantalla la orden con sus argumentos, y a continuación su salida.
`-v`	Verbose: Imprime en pantalla cada elemento completo del script (estructura de control, ...) y a continuación su salida.

También es posible depurar sólo parte del script insertando en él los siguientes comandos (pueden usarse igualmente en la propia línea de comandos):

`set -x` `set –xv`	Activa las trazas/verbose. Ubicarlo justo antes del trozo del script que se desea depurar.
`set +x` `set +xv`	Desactiva las trazas/verbose. Ubicarlo justo después del trozo del script que se desea depurar.

TAREAS

1. Mire el contenido del siguiente script en su sistema y compruebe que tiene el permiso de ejecución general:

script_depuracion.sh

```
#!/bin/sh
echo Hola
if true; then
     echo hola2
     ls /
fi
```

2. Invoque dicho script con las siguientes opciones de depuración y analice la salida:

```
/bin/bash       script_depuracion.sh
/bin/bash  -x   script_depuracion.sh
/bin/bash  -v   script_depuracion.sh
/bin/bash  -xv  script_depuracion.sh
/bin/dash  -x   script_depuracion.sh
/bin/dash  -v   script_depuracion.sh
/bin/dash  -xv  script_depuracion.sh
```

3. Modifique el script para que tenga el siguiente contenido:

script_depuracion2.sh

```
#!/bin/sh
echo Hola
set -xv
if true; then
     echo hola2
     ls /
fi
set +xv
```

4. Invoque dicho script con los siguientes comandos y analice la salida:

```
script_depuracion2.sh
/bin/dash  script_depuracion2.sh
/bin/bash  script_depuracion2.sh
```

5 Propuesta de ejercicios

A continuación se le propone la creación de una serie de scripts. Para ello, será necesario tanto usar los conocimientos expuestos en este documento, como otros relativos a la administración de Linux:

a. Los ficheros `/etc/profile` (leído en el arranque del sistema y común para todos los usuarios), `~/.bashrc`, `~/.bash_profile` (leídos al abrir una sesión con el usuario correspondiente, no es necesario que existan ambos), usados para cargar las variables de entorno, son en realidad shell-scripts. Analice el contenido de dichos scripts y modifíquelos convenientemente para definir los alias:
   - `alias red='ifconfig -a'` : para todos los usuarios del sistema.
   - `alias casa='echo $PWD'` : sólo para el usuario normal.
   - `alias sockets='netstat -l'` : sólo para el usuario `root`.

Repita lo anterior pero utilizando funciones en vez de alias.

b. Cree los siguientes scripts:
   - `variables.sh`: debe inicializar las variables `CONFIG='-a'`, `ARP='-n'` y `ROUTE`.
   - `funciones.sh`: debe definir las funciones `arranque_red()` (encargada de activar la tarjeta "eth0"), `parada_red()` (encargada de desactivar la tarjeta `eth0`) y `estado_red()` (encargada de ejecutar los comandos `ifconfig`, `arp` y `route` aplicándoles los parámetros pasados a la función en los argumentos 1, 2 y 3, respectivamente).
   - `script1.sh`: debe importar las variables y funciones definidas por los dos scripts anteriores. Mostrará al usuario un menú en el que le pregunte si desea arrancar la red (invocando la función `arranque_red()`), pararla (función `parada_red()`) o mostrar su estado (función `estado_red()`). Debe comprobar si alguna de las variables `CONFIG`, `ARP` y `ROUTE` es nula y, en caso de serlo, permitirle al usuario escribir interactivamente (mediante la función `read`) los parámetros con los que invocar el comando correspondiente.

c. `script_copy.sh`: script que copie un fichero regular en otro, ambos pasados como argumentos. Si no se le pasan los argumentos, lo comprobará, y solicitará al usuario que los introduzca interactivamente.

d. `script_print.sh`: script que imprima en pantalla el contenido de un fichero de datos, o el contenido de todos los ficheros de un directorio, según se le pase como argumento un fichero regular o un directorio.

e. `script_borrar.sh`: script que borre con confirmación todos los ficheros pasados como argumentos.

f. `script_sesiones.sh`: script al que, pasándole el login de un usuario, devuelva cuántas sesiones tiene abiertas en el sistema.

g. `script_mostrar.sh`: script que para cada argumento que reciba (puede recibir una lista de argumentos) realice una de las siguientes operaciones:
   - Si es un directorio, ha de listar los ficheros que contiene.
   - Si es un fichero regular, tiene que imprimir su contenido por pantalla.
   - En otro caso, debe indicar que no es ni un fichero regular ni un directorio (por ejemplo, un fichero de bloques o de caracteres del directorio `/dev`).

h. `script_ejecucion.sh`: script que asigne el permiso de ejecución a los ficheros regulares o directorios pasados como argumento (puede admitir una lista de ficheros).

i. `script_doble.sh`: script que pida un número por teclado y calcule su doble. Comprobará que el número introducido es válido y, antes de terminar, preguntará si deseamos calcular otro doble, en cuyo caso no terminará.

j. `script_tipos.sh`: script que devuelva el número de ficheros de cada tipo (ficheros regulares o directorios) que hay en una determinada carpeta, así como sus nombres. Tendrá un único argumento (opcional) que será la carpeta a explorar. Si se omite dicho argumento, se asumirá el directorio actual. Devolverá `0` (éxito) si se ha invocado de forma correcta o `1` (error) en caso contrario.

k. `script_user.sh` : script que reciba como argumento el login de un usuario e imprima por pantalla la siguiente información: login, nombre completo del usuario, directorio home, shell que utiliza, número de sesiones actualmente abiertas y procesos pertenecientes a dicho usuario. El script debe permitir las siguientes opciones:

   - `-p` : sólo muestra la información de los procesos.
   - `-u` : muestra toda la información excepto la referente a los procesos.
   - `--help` : muestra información de ayuda (uso del script).

   El script debe comprobar si los argumentos pasados son correctos, así como la existencia del usuario indicado. Como código de error podrá devolver `0` (éxito), `1` (sintaxis de invocación errónea), `2` (usuario no existe).

l. `script_menu.sh` : script que ofrezca al usuario un menú (con una opción para salir) desde el que el usuario pueda seleccionar cual de los scripts anteriores (apartados "b)" a "k)") quiere utilizar.

m. `script_puerto.sh` : script que reciba como argumento un número de puerto TCP (PUERTO) y comprobará si el valor es un número entero positivo en el rango "[1, 1023]", de modo que:

   - Si es así: el script analizará si los puertos TCP "PUERTO" y "PUERTO+1" del equipo local se encuentran a la escucha. Tras ello, imprimirá un mensaje indicando el estado obtenido. Para ello, puede analizar la salida del comando `netstat` .
   - En otro caso: imprimirá por pantalla un mensaje indicando que el valor indicado no es un puerto de sistema.

n. `script_arp.sh` : script que reciba como argumento un número entero, guardándolo en la variable `POS_ARP` . El script comprobará si el valor de la variable `POS_ARP` es un número entero positivo, de modo que:

   - Si es así: el script calculará el número de entradas que actualmente tiene la caché ARP del equipo, guardándolo en la variable `NENTRADAS` . Si el valor de `POS_ARP` es mayor que `NENTRADAS` , imprimirá por pantalla un mensaje tal como: `Ninguna entrada en la posición POS_ARP` . En caso de que `POS_ARP` sea menor o igual que `NENTRADAS` , imprimirá por pantalla un mensaje con la posición de la entrada ARP `POS_ARP` , seguido del contenido de dicha entrada ARP.
   - En otro caso: Imprimirá por pantalla un mensaje indicando que el valor indicado no es un número entero: `Posición de entrada ARP no válida`

   Se solicitan dos posibles soluciones:

   - `script_arp1.sh` : Basada en el comando `read` y estructuras de control.
   - `script_arp2.sh` : Analizando la salida del comando `arp` mediante comandos de análisis de texto.

o. `script_param.sh` : Considere que en el shell actual se dispone de la variable:

   ```
   VAR="nombre=v1&edad=v2&tlf=v3"
   ```

   Escriba un shell-script que analice el valor de dicha variable y para cada uno de los parámetros extraiga su valor y lo imprima por pantalla. Por ejemplo, que la

salida sea:

```
Cadena analizada: nombre=v1&edad=v2&tlf=v3
nombre: v1, edad: v2, tlf: v3.
```

Se solicitan dos posibles soluciones:

- `script_param1.sh`: Usando la variable de entorno `IFS`.
- `script_param2.sh`: Sin usar `IFS`.

6 Anexo: Descriptores de ficheros y redirecciones

Para facilitar el acceso a los distintos recursos del sistema (zona de almacenamiento en disco, teclado, pantalla, socket,...), el kernel de Linux vincula cada recurso con un fichero, real (espacio en disco) o virtual (en memoria, como /dev y /proc). Ejemplos habituales son:

Recurso	Fichero
Directorio	/carpeta
Pantalla	/dev/ttyXX
Sockets TCP	/proc/net/tcp
Fichero regular	/dir/file
Teclado (según USB...)	/dev/uinput
Proceso con PID xxxx	/proc/xxxx
Disco duro	/dev/sda
Ratón	/dev/psaux
Caché ARP	/proc/net

Cuando un proceso necesita acceder a un recurso, debe realizar las siguientes operaciones:

1. Abrir el fichero correspondiente a dicho recurso: el proceso indica al sistema la ruta del fichero, especificando su intención de abrirlo para lectura (obtener información del recurso), escritura (enviar información al recurso) o lectura/escritura; el sistema concederá el tipo de acceso solicitado según las características del recurso (teclado, pantalla, ...) y los permisos que el usuario efectivo del proceso posea sobre el fichero. Como respuesta, el sistema devolverá al proceso el **descriptor** (o descriptor de fichero) con el que el proceso podrá acceder al fichero.
2. Acceder al fichero/recurso usando el descriptor suministrado por el sistema: el proceso indicará el descriptor al kernel, éste localiza el fichero y accede al recurso asociado.

kernel proceso

Recurso ⇔ Fichero ⇔ Descriptor

Un descriptor no es más que un número entero "n", con las siguientes características:

- Cada **proceso** tiene su propia tabla de **descriptores** para acceder a los **recursos/ficheros**; el fichero que un proceso tenga asociado al descriptor "n" es independiente del que puedan tener asociado para ese mismo descriptor los demás procesos del sistema. El conjunto de descriptores que tiene asociados un determinado proceso puede obtenerse con `ls /proc/PID/fd/`, cambiando `PID` por el identificador de dicho proceso.

Proceso con PID xxxx	
Fichero	Descriptor
Fichero_A	0
Fichero_B	1
Fichero_C	2
...	...

- Se dice que un descriptor es de ***entrada***, ***salida*** o ***entrada/salida*** según el fichero al que esté asociado en ese instante (es posible cambiar el fichero al que está asociado un descriptor) haya sido abierto por el proceso que posee dicho descriptor para lectura, escritura o lectura/escritura, respectivamente.

Al trabajar con un intérprete de comandos encontramos que, de manera habitual, los comandos suelen imprimir su información en pantalla y obtenerla del teclado. Este funcionamiento se debe a los dos motivos siguientes:

a. Muchas aplicaciones, especialmente pensadas para ser usadas en modo consola, son programadas para que, por defecto (sus **procesos**):

- *Obtengan* datos del (**recurso** asociado al) **descriptor** "**0**": dado que éste es el descriptor del que convencionalmente las aplicaciones obtienen información, este descriptor suele denominarse "**entrada estándar** (a pesar de este nombre, dado su uso especial, internamente los descriptores "0, 1, 2" son de entrada/salida).
- *Envíen* su información al (**recurso** asociado al) **descriptor** "**1**": dado que éste es el descriptor al que convencionalmente las aplicaciones envían su información, este descriptor suele denominarse "**salida estándar**".
- *Envíen* su información de errores al (**recurso** asociado al) **descriptor** "**2**": dado que éste es el descriptor al que convencionalmente las aplicaciones envían su información de errores, este descriptor suele denominarse "**salida de error estándar**".

Esta configuración "estándar" es la empleada por los comandos POSIX. Así, por ejemplo, los comandos "echo cadena" o "cat fichero" están preparados para imprimir (enviar la información) en el recurso asociado al descriptor `1` (salida estándar), o el comando `cat` (sin argumentos) está diseñado para obtener la información del recurso asociado al descriptor `0` (entrada estándar).

b. De forma habitual, el **proceso** (shell) de cualquier consola en modo comandos asocia el **descriptor**:

- `0` : al **recurso** "teclado (siempre, a través del fichero asociado a esos dispositivos (/dev/uinput,...), que puede variar según su tipo).
- `1` : al **recurso** "pantalla".
- `2` : al **recurso** "pantalla".

Por omisión, cuando un proceso padre crea un proceso hijo, el proceso hijo sólo dispone de los descriptores "0, 1, 2", cada uno asociado al mismo recurso que en el proceso padre (el proceso hijo hereda la asociación descriptor-fichero para esos tres descriptores, el resto de descriptores no se "heredan"). Todo ello lleva a que en una consola de comandos, las órdenes "echo cadena" o "cat fichero" suelan imprimir por pantalla (recurso del descriptor `1`), o el comando `cat` (sin argumentos) obtenga la información del teclado (recurso del descriptor `0`), al heredar esa asociación descriptor-fichero del shell (proceso padre).

Sobre este comportamiento habitual, los intérpretes de comandos de Linux permiten:

- Modificar el fichero (recurso) asociado a cada descriptor estándar "0, 1, 2".
- Asociar ficheros a los descriptores no estándar (3 y posteriores).

Para ello, la sintaxis a aplicar varía según el proceso sobre el que queramos establecer la nueva asociación fichero-descriptor sea el propio proceso shell que estamos utilizando, o sobre un comando (proceso hijo) invocado desde este shell (en programación C, por ejemplo, se realizaría con la función "open", entre otras).

6.1 Asociación para el Proceso Shell actual

Para ello se emplea el comando "exec", mediante las siguientes sintaxis:

Sintaxis	Funcionalidad
`exec n< fichero`	Asocia el descriptor de entrada "n" con "fichero" (archivo regular). Para abreviar, en todo el texto se usará el calificativo "descriptor de entrada" para indicar que en ese instante (puede modificarse) el descriptor está asociado con un fichero abierto para "lectura". Al igual para salida (escritura) y entrada/salida.
`exec n> fichero`	Asocia el descriptor de salida "n" con "fichero"
`exec n<> fichero`	Asocia el descriptor de entrada y salida "n" con "fichero"
`exec n<&m`	Asocia el descriptor "n" con el mismo fichero al que actualmente (en el momento de ejecutar este comando) está asociado el descriptor "m" (duplicado de descriptor).
`exec n<&-`	Cierra el descriptor "n" (elimina su asociación con el fichero al que actualmente esté vinculado)

Por ejemplo, el siguiente comando permite asociar el archivo `/tmp/fichero` con el descriptor "4":

```
exec 4< /tmp/fichero
```

A partir de ello, cada vez que queramos que un comando realice operaciones de lectura/escritura sobre ese archivo, podremos aplicar los operadores de redirección (detallados más abajo) con dicho descriptor (en lugar de usar el nombre del archivo). Por ejemplo, para que el comando `ls` ; envíe su información a ese archivo:

```
ls >&4
```

6.2 Asociación para un comando (proceso hijo) invocado desde el shell

Ello se consigue mediante la técnica de "**redirección**", basada en la siguiente sintaxis (es importante que no haya espacios entre `n` y `op`) :

```
comando    [n]op    fichero/descriptor
```

donde:

- `comando`: comando sobre el que aplicar la redirección. Recuerde que, conforme a la sintaxis de los comandos simples, la redirección puede escribirse tanto después como antes del comando.
- `n` (número entero, opcional): descriptor (asociado actualmente o no a algún fichero en el shell actual). POSIX exige que se soporten, al menos, los valores `0, 1,..., 9`. El valor por omisión depende del operador empleado.
- `op`: operador de redirección. Si se "escapa" el descriptor `n` se usará el descriptor por omisión (e.g. `echo \2>fichero` asumirá el descriptor `1`). Si se escapa el operador de redirección, no se aplicará redirección alguna (e.g., `echo 2\>fichero` imprime `2\>fichero` al recurso del descriptor `1`)
- `fichero/descriptor`; a asociar (según indique el operador) con el descriptor `n` descriptor.

Al aplicar esa redirección sobre el comando, para el descriptor `n`, el comando (el proceso hijo que se crea) no heredará (aún en el caso de ser `n=0, 1 o 2`) la asociación descriptor-fichero del proceso shell padre, sino que asociará a dicho descriptor `n` el "fichero" explícitamente indicado.

A continuación se resumen los operadores de redirección definidos por el estándar POSIX, agrupadas según operen sobre descriptores para entrada (lectura de fichero) o salida (escritura en fichero):

a. Redirecciones de **entrada**: por omisión, toman `n=0` (entrada estándar).

Redirección	Funcionalidad: El shell invoca `comando` como proceso hijo, configurándolo para que:
`cmd [n]< fich`	Asocie el descriptor de entrada (recuérde que, para abreviar, se está usando el calificativo "descriptor de entrada" para indicar que en ese instante el descriptor está asociado con un fichero abierto para lectura, pero puede cambiar) `n`; con `fich` (archivo regular). Por omisión `n=0`, esto es, `cmd < fich` hace que para el proceso `cmd`, el descriptor `0` quede asociado a `fich` para lectura.
`cmd [n]<&m`	Asocie el descriptor de entrada `n` con el mismo fichero al que actualmente está asociado el descriptor `m`. Dicho fichero estará así referenciado desde dos descriptores (duplicado de descriptores). `m` es obligatorio.
`cmd [n]<&-`	Cierre el descriptor de entrada `n` (elimina su asociación con el fichero al que actualmente esté vinculado). Si el descriptor no está asociado con ningún fichero, dará error.
`cmd [n]<< delim`	Asocie el descriptor de entrada `n` con el recurso especial `Here-Document` (texto empotrado). Tras ejecutar la orden, aparecerá en consola el carácter `>`; el texto introducido a partir de entonces será guardado en el recurso `Here-Document`, terminando cuando se introduzca una línea que sólo contenga (incluidos espacios en blanco) la cadena `delim` (delimitador) y se pulse nueva línea; en dicho momento, `comando` será ejecutado (pudiendo acceder al contenido del recurso `Here-Document` a través del descriptor `n`). Si al escribir la orden, la cadena `delim` se introduce entre comillas, el shell las eliminará, de modo que para terminar de escribir en el recurso `Here-Document`, habrá que escribir la cadena `delim` sin comillas.
`cmd <<- delim`	Ídem `<<`, pero las tabulaciones añadidas al principio de línea será omitidas, no insertándose en el recurso `Here-Document`.

b. Redirecciones de **salida**: por omisión, toman `n=1` (salida estándar).

Redirección	Funcionalidad: El shell invoca `comando` como proceso hijo, configurándolo para que:
`cmd [n]> fich`	Asocie el descriptor de salida `n` con `fich` (archivo regular). Por omisión `n=1`, esto es, `cmd > fich` hace que para el proceso `cmd`, el descriptor `1` quede asociado a `fich`. Si `fich` no existe, será creado; si existe, será limpiado previamente. Salvo que se haya activado en el shell la opción "no sobrescribir" con el comando `set -C` (recuerde que el comando `set` permite modificar el comportamiento del shell, incluyendo opciones y el valor de sus variables) en cuyo caso dará error.
`cmd [n]>| fich`	Ídem `>`, sin depender de la opción "no sobrescribir" (`set -C`).
`cmd [n]>> fich`	Ídem `>`, sin limpiar previamente el fichero si existe (insertando al final del contenido existente).
`cmd [n]>& m`	Asocie el descriptor de salida `n` con el mismo fichero al que está asociado el descriptor `m`. Dicho fichero estará así referenciado desde dos descriptores (duplicado de descriptores). `m` es obligatorio.
`cmd [n]>& -`	Cierre el descriptor de salida `n` (elimina su asociación con el fichero al que actualmente esté vinculado). Si el descriptor no

Redirección	Funcionalidad: El shell invoca `comando` como proceso hijo, configurándolo para que:
	está asociado con ningún fichero, dará error.

c. Redirecciones de **entrada/salida**: por omisión, toman `n=1` (salida estándar).

Redirección	Funcionalidad: El shell invoca `comando` como proceso hijo, configurándolo para que:
`cmd [n]<>fichero`	Asocie el descriptor `n` de entrada y salida con `fichero` (archivo regular, será creado si no existe).

Además de las anteriores, existe un tipo especial de redirección denominado **tubería** o "pipeline", en la cual el "recurso" sería esa "tubería" que conecta dos descriptores. Esta redirección se basa en la siguiente sintaxis:

```
comando1 | comando2
```

bajo la cual, el descriptor `0` de `comando2` se asociaría con el descriptor `1` de `comando1` (esto es, lo que `comando1` envíe a su salida estándar será redirigido a la entrada estándar de `comando2`). Pueden usarse dos o más comandos separados por `|`.

A continuación se proponen varios ejemplos de redirección y su explicación:

`cat < fichero`	Por omisión `cat` tiene el recurso "teclado" asociado al descriptor `0` (POSIX). Con este comando, `cat` es invocado para que su descriptor `0` quede asociado al recurso `fichero`, de modo que `cat` toma la entrada de dicho fichero.
`ls / > fichero`	`ls` es invocado para que su descriptor `1` esté asociado a `fichero` (y no a la "pantalla" a que está asociado en el shell desde el que es invocado), esto es, `ls` envía su salida a `fichero`.
`cat << fin`	Toda la información escrita tras invocar el comando es guardada en el recurso especial `Here-Document`, pasándose al comando `cat` cuando se introduzca la línea `fin` y se pulse nueva línea.
`ls | more`	El descriptor de salida de `ls` se conecta con el de entrada de `more`, de modo que toda la salida `ls` de entrega a `more` como entrada.

Adicionalmente a lo anterior, deben realizarse las siguientes aclaraciones sobre las redirecciones:

a. Las redirecciones son aplicables a cualquier comando, incluso a subshells (shell abierto desde otro shell), de modo que todos los comandos abiertos desde dicho subshell heredarán por defecto su asociación de los descriptores `0, 1, 2`. Por ejemplo, si en un shell ejecutamos:

   ```
   echo "Hola" > /tmp/fichero
   ```

   ```
   /bin/bash < /tmp/fichero
   ```

   con el segundo comando se estará abriendo un nuevo proceso shell (subshell), el cual tendrá asociado el descriptor `0` con el archivo `/tmp/fichero`. Si en dicho subshell abierto ejecutamos ahora:

   ```
   cat
   ```

   este comando hereda la configuración del subshell, teniendo igualmente asociado el descriptor `0` con el archivo `/tmp/fichero`. El comando `cat`, al ser invocado sin argumentos, sigue el estándar POSIX, imprimiendo en el recurso asociado al descriptor `2` (la pantalla en este caso) la información obtenida del recurso asociado al descriptor `0`, que en este caso será el archivo `fichero`; (y no el teclado). Consecuentemente, el resultado es que dicho comando `cat` imprime directamente por pantalla el contenido de `fichero`. El orden de lectura de los comandos es importante. Por ejemplo, `(cat << fin) < fichero` no hará que `fichero` se pase a `cat`, dado que `< fichero` no se aplicará hasta que no haya terminado el comando `cat`. Por ejemplo:

`cat` ["Ctrl-D" para terminar, que equivale a enviar EOF]

`cat << fin` [Teclear "fin" y Enter para terminar]

b. El orden en el que se escriban las redirecciones es importante, pudiendo cambiar el resultado. Por ejemplo, en el comando:
   - `ls /tmp 2> file 1>&2` : sucedería lo siguiente (en este orden):
     - El descriptor `2` se asocia con `file`.
     - El descriptor `1` se asocia con el mismo fichero al que está asociado el descriptor `2`, luego con `file` también.
     - Se ejecuta el comando `ls`, que enviará sus salidas estándar (1) y de errores (2) a `file`. En un comando simple, el comando siempre se ejecuta tras aplicar las posibles expansiones/sustituciones e interpretar las redirecciones (leyéndose en orden de izquierda a derecha).
   - `1>&2 ls /tmp 2> file` : sucederá lo siguiente (en este orden):
     - El descriptor `1` se asocia con el mismo fichero (o recurso, e.g. pantalla) al que actualmente esté asociado el descriptor `2`.
     - El descriptor `2` se asocia con `file`.
     - Se ejecuta el comando `ls`, que enviará su salida de errores a `file`, y su salida estándar (1) con ese recurso (posiblemente pantalla) al que inicialmente estuviese asociado el descriptor `2`.

c. Para redireccionar una información a "ninguna parte" se usa el fichero nulo `/dev/null` (fichero que podría considerarse asociado al recurso virtual "destructor de información") . Por ejemplo, el siguiente comando haría toda la información que `ls` envíe a la salida estándar (descriptor `1`) y a la salida de errores (descriptor `2`) sea enviada a `/dev/null` (se elimina):

```
ls -l /usr > /dev/null 2>&1
```

TAREAS

Ejecute y analice el funcionamiento de las redirecciones empleadas en los siguientes comandos:

- `cat << END > fichero` : lee información del teclado, hasta que se introduce una línea con END. Entonces copia toda la información tecleada al archivo `fichero`.
- `ls -l /bin/bash ./script > fichero 2> error` : redirige la salida estándar al archivo `fichero` y la salida de error al fichero `error`.
- `ls -l /bin/bash ./script > fichero 2>&1` : redirige las salidas estándar y de error al archivo `fichero`.
- `ls 2> /dev/null 1>&2` : redirige las salidas estándar y de error al archivo nulo.

Por último, advertir que la explicación anterior corresponde al estándar POSIX. Algunos intérpretes de comandos como Bash soportan otros operadores (además de los POSIX). A continuación se resumen algunos de los ofrecidos por Bash:

Redirección Bash (**NO POSIX!!**)	**Descripción**	**Equivalente POSIX**
`cmd &> fich`	Asocia los descriptores `1` y `2` con el recurso "fichero"	`cmd > fich 2>&1`
`cmd >& fich`		

Descargar versión antigua en PDF