Parte 5.pdf

QuesoViejo_

Sistemas Operativos

2º Grado en Ingeniería Informática

Facultad de Informática
Universidad de A Coruña

Capítulo 5

Expresiones regulares

5.1. Expresiones regulares

Una expresión regular no es más que un conjunto de caracteres que definen un "patrón" de texto, con el que debe concordar.

Las expresiones regulares son utilizadas por utilidades del sistema, tales como los editores `ed`, `vi`, `ex`, y `emacs`, y los filtros `grep`, `egrep`, `sed` y `awk`, entre otros. La mayoría de estas órdenes buscan en un fichero o en la entrada para encontrar una cadena o secuencia de caracteres, que se ajusten a un patrón. Cuando se ha encontrado se dice que la expresión regular concuerda, equivale a, o coincide con la cadena. El uso de las expresiones regulares depende de la orden.

Una expresión regular no es más que una cadena en la que distinguimos dos tipos de caracteres:

Caracteres ordinarios Sólo concuerdan consigo mismos.

Caracteres especiales Tienen un significado especial para el programa que los interpreta.

Hay caracteres especiales que tienen un significado especial independientemente del lugar de la cadena donde aparezcan, estos son: `. * [ \`. Sin embargo hay otros que sólo tienen un significado especial cuando se sitúan en una posición determinada; el carácter `^` sólo tiene un significado especial al comienzo de una cadena y el carácter `$` sólo lo tiene cuando va al final.

Si queremos usar un carácter especial dentro de una cadena como carácter ordinario deberemos protegerlo.

A continuación veremos los distintos tipos de expresiones regulares que podemos encontrar y el significado de los caracteres especiales.

5.1.1. Expresiones regulares básicas

Las expresiones regulares básicas sólo hacen uso de los caracteres especiales que se describen en el cuadro 5.1.

Símbolo	Significado
`\`	Anula el significado especial del carácter que le sigue.
`.`	Concuerda con cualquier carácter simple.
`*`	Concuerda con cero o más ocurrencias de la expresión regular precedente.
`$`	Sólo tiene un significado especial cuando es el último carácter de la expresión regular. Concuerda con el final de la línea.
`^`	Sólo tiene significado especial cuando es el primer carácter de la expresión regular. Concuerda con el comienzo de la línea.
`[`*conjunto*`]`	Un conjunto de caracteres entre corchetes, concuerda con un carácter simple del *conjunto*. Dentro de un conjunto sólo tienen significado especial los caracteres `-`, `]`, `^`. La notación [*c1*-*c2*] indica el conjunto de caracteres ASCII comprendidos en el rango de *c1* a *c2*. La secuencia `[^`*conjunto*`]` concuerda con cualquier carácter simple que no esté en el *conjunto*.

Cuadro 5.1: Expresiones regulares básicas

Ejemplos:

1. La expresión regular `$9\.98` concuerda con `$9.98` ya que hemos anulado el significado especial del carácter punto. Observe, además, que el carácter `$` no tiene significado especial, ya que no va al final de la expresión regular.
2. `.ora` concuerda con `hora`, `ahora`, `espora`, no concuerda con `ora`.
3. `Pr*` concuerda con `P`, `Pr`, `Prr`, `Pato`, `Perro`.
4. `.*` concuerda con cualquier conjunto de caracteres.

MENÚ

BURRITO SAN DIEGO

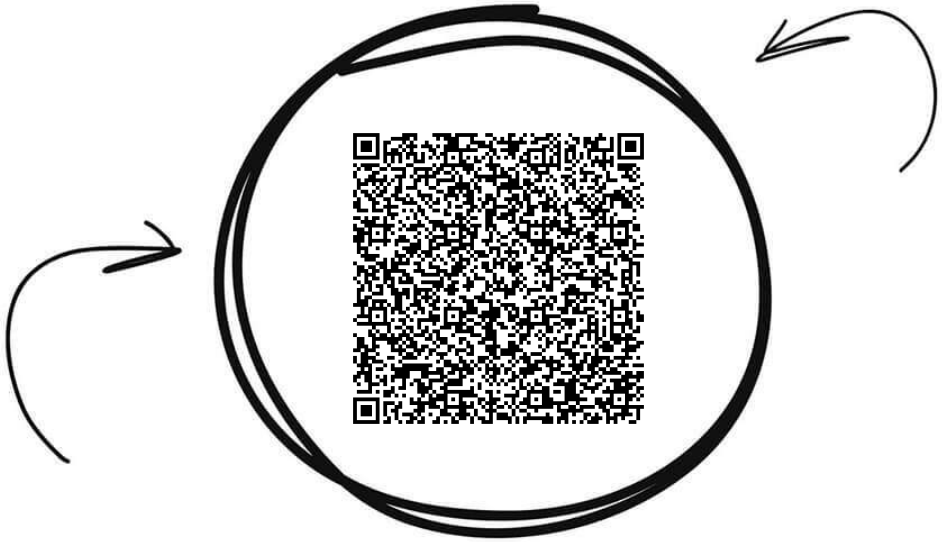
4 ‘95€

Burrito relleno de jugosa carne picada, arroz, lechuga, pico de gallo e irresistible salsa de queso que #EstáDeLoocos

Promoción válida hasta el 13/11/2023 en restaurantes participantes. Promoción válida para menú mediano: Burrito San Diego, patatas Mexican y bebida. No acumulable a otras promociones. Consultar el precio del servicio a domicilio. El cambio a menú grande solo afecta a la bebida y al complemento del menú. Para más información consultar bases legales en tacobell.es. El precio de algunos productos de la carta se verá incrementado por el uso de plásticos de un solo uso, en aplicación al art. 55.2 de la Ley 7/2022 de residuos del 8 de abril.

TACO BELL.

Sistemas Operativos



Banco de apuntes de la

WUOLAH

Comparte estos flyers en tu clase y consigue más dinero y recompensas

1. Imprime esta hoja
2. Recorta por la mitad
3. Coloca en un lugar visible para que tus compis puedan escanar y acceder a apuntes
4. Llévate dinero por cada descarga de los documentos descargados a través de tu QR

5. `g.*d` concuerda con cualquier cadena que comience por `g` y termine en `d`.
6. `\.$` concuerda con cualquier línea que termine en punto.
7. `^[A-Z]` concuerda con cualquier línea que comience por letra mayúscula.
8. `^abc$` concuerda con las líneas que únicamente contienen `abc`.
9. `[AEIOUa-m]` concuerda con una vocal en mayúsculas o bien una letra comprendida en el rango de la `a` a la `m`.
10. `[^0-9]` concuerda con cualquier carácter que no sea un dígito.

5.1.2. Expresiones regulares extendidas

Las expresiones regulares extendidas pueden hacer uso de los caracteres especiales que forman parte de las expresiones regulares básicas más los que aparecen en el cuadro 5.2. Los paréntesis `( )` tienen la máxima prioridad, seguidos del grupo `*`, `?`, `+` y `|`.

Símbolo	Significado
`+`	Concuerda con una o más ocurrencias de la expresión regular precedente.
`?`	Concuerda con cero o una ocurrencia de la expresión regular precedente.
expr1 `|` *expr2*	Concuerda con cualquiera de las dos expresiones regulares separadas por `|`.
`(`*expr*`)`	Se utiliza para agrupar una expresión regular, por lo que `*`, `?`, `+`, `|`, podrán aplicarse a la expresión completa.

Cuadro 5.2: Expresiones regulares extendidas

Ejemplos:

1. La expresión `per+o` concuerda con `perro`, `perrro`, `perrrro`.
2. La expresión `aab?cc` concuerda con `aacc`, `aabcc`.
3. La expresión `hola|adios` concuerda con `hola` o con `adios`.
4. La expresión `a(bc)?` concuerda con `a`, o con `abc`.

Ejercicios:

1. Dado el siguiente fragmento de texto:

   > *Desde enero de 1993 ha estado vigente en España el índice de Precios de Consumo (IPC) base 92. Desde enero de 2002 la metodología del IPC ha sido completamente renovada y se ha introducido un nuevo sistema de cálculo, IPC-2001.*

   Indique las concordancias que existen con las expresiones regulares que se dan continuación:

   a) `[Ii][Pp].`
   b) `s[aeiou]*t`
   c) `\.$`
   d) `^[Dd].*\.$`
   e) `[a-z]*002`

5.2. La familia grep

La familia `grep` está formada por tres filtros: `grep`, `egrep`, y `fgrep`. En general, esta familia de programas selecciona aquellas líneas de la entrada, que contienen los caracteres que concuerdan con la cadena de caracteres o la expresión regular especificada, y las copia normalmente en la salida estándar.

Como ya sabemos, una expresión regular es una cadena que contiene caracteres especiales que son interpretados por los programas correspondientes.

La familia `grep` está formada por tres programas cuyos formatos son:

- `grep` [*opciones*] *expresión_regular* [*fichero* ...]
- `egrep` [*opciones*] *expresión_regular* [*fichero* ...]
- `fgrep` [*opciones*] *cadenas* [*fichero* ...]

Las diferencias fundamentales entre las distintas órdenes son:

`fgrep` Sólo admite como patrón de búsqueda cadenas literales.

`grep` Admite expresiones regulares básicas.

`egrep` Admite expresiones regulares básicas y extendidas.

Todas las órdenes anteriores admiten las opciones que se citan a continuación.

Opciones:

`-n` Las líneas seleccionadas van precedidas del número de línea que le corresponde dentro del fichero.

Ejemplo:
```
$ grep -n man fichero
```

`-v` Muestra las líneas que no concuerdan con la expresión regular o con la cadena.

Ejemplo:
```
$ grep -v /bin/sh /etc/passwd
```

`-c` Visualiza sólo el número de líneas que contienen o concuerdan con la expresión regular o la cadena especificada.

Ejemplo:
```
$ grep -c /bin/sh /etc/passwd
```

`-i` Ignora la distinción entre mayúsculas y minúsculas en la búsqueda.

Ejemplo:
```
$ grep -i Man fichero
```

`-w` Busca palabras completas.

`-f` *fichero* Toma la cadena a buscar o la expresión regular del *fichero* especificado. Permite buscar simultáneamente varias cadenas o expresiones regulares. Previamente tendremos que crearnos el fichero con las cadenas o expresiones regulares que queramos buscar.

Ejemplo:
```
$ egrep -f patrones /etc/group
```

`-l` Muestra sólo el nombre de los ficheros con líneas coincidentes con la expresión; no muestra las líneas.

Ejemplo:
```
$ grep -il division ~/cobol/*
```

`-L` Muestra sólo el nombre de aquellos ficheros en los que no encuentre ninguna concordancia con la expresión regular.

`-b` Antepone a cada línea que se muestra en pantalla el número de caracteres que la preceden en el fichero de entrada.

5.2.1. La orden grep

Su formato es:

grep [*opciones*] *expresión_regular* [*fichero* ...]

Busca en la entrada estándar o en el *fichero* indicado las líneas que contienen la cadena de caracteres que concuerda con la expresión regular especificada.

Admite expresiones regulares básicas. Se pueden utilizar los caracteres comodines en los nombres de ficheros.

Ejemplos:

1. `$ grep hola *`
   Busca la expresión `hola` *en todos los ficheros del directorio de trabajo.*
2. `$ grep "la pera en dulce" frutas`
3. `$ grep 'sh$' /etc/passwd`
   Busca en el fichero `/etc/passwd` *todas aquellas líneas que terminen en* `sh` *y las copia en la salida estándar.*
4. `$ grep ^From $MAIL | grep -v pepe`
   Del fichero especificado en la variable `MAIL` *extrae las líneas que empiecen por* `From`*; la salida es filtrada por* `grep` *y se seleccionan las líneas que no contengan* `pepe`.
5. `$ ls | grep -v '~$'`
   Listado de todos los ficheros que no terminen en `~`.
6. `$ ls -l | grep '^.......rw'`
   Lista todos los ficheros que tengan permiso de lectura y escritura para los otros.

5.2.2. La orden egrep

Busca aquellas líneas de la entrada que contengan una o varias de las expresiones regulares dadas. Es la orden más potente de la familia. Admite expresiones regulares básicas y extendidas. El algoritmo de búsqueda de `egrep` consume gran cantidad de memoria. Su formato es:

egrep [*opciones*] *expresion_regular* [*ficheros* ...]

Ejemplo:

```
$ egrep 'mail|ftp' /etc/passwd
```

Busca en el fichero `/etc/passwd` *todas las líneas que contengan las expresiones* `mail` *o* `ftp` *y las copia en la salida estándar.*

5.2.3. La orden fgrep

Busca en la entrada aquellas líneas que contienen las cadenas especificadas. No admite expresiones regulares, pero sí más de una cadena. El algoritmo de búsqueda de **fgrep** es más rápido que el de **grep**. El formato de **fgrep** es:

fgrep [*opciones*] *cadena(s)* [*fichero* ...]

Para indicarle más de una cadena hay que separarlas por un **<intro>**.

Ejemplo:

```
$ fgrep 'Susana
  Samuel' telefonos
```

La existencia de tres programas: **grep**, **egrep** y **fgrep**, con funciones tan parecidas, es debida a razones históricas. En la actualidad se tiende a integrar los tres programas en uno solo (**grep**) que puede ser llamado con distintas opciones para que se comporte como uno de los tres anteriores. En nuestro sistema disponemos de la orden **grep** de GNU; ésta admite tres opciones: **-G**, **-F** y **-E**, aparte de las citadas anteriormente. Si llamamos a **grep** con la opción **-G** se comportará como el **grep** descrito anteriormente; éste es el comportamiento predeterminado. Si lo llamamos con la opción **-F** se comportará como **fgrep**, y si lo llamamos con la opción **-E** se comportará como **egrep**. Además podemos utilizar las órdenes **fgrep** y **egrep**, que son enlaces al programa **grep**.

Ejercicios:

1. ¿Qué hacen las siguientes órdenes?

   a) `$ ypcat passwd | fgrep ".*"`

   b)
   ```
   $ ls -l | grep '^-' | cut -d" " -f1,1 | \
     grep 'rwx$' | wc -l
   ```

   c) `$ grep -cw '....' /usr/share/dict/words`

   d) `$ cat /usr/share/dict/words | egrep '^(l|r)+.*a$`

2. ¿Cuál será el resultado de ejecutar la siguiente orden? ¿Cuál es la primera operación que realiza el *shell* en esta línea de órdenes?

   ```
   sort -o datos > datos
   ```

3. ¿Qué diferencias hay entre estas dos órdenes?

   a) `sort numeros`

   b) `sort -n numeros`

4. Obtenga la información acerca de los usuarios del sistema ordenada según el nombre de *login*.

5. ¿Qué hace la siguiente orden?

   ```
   sort -u equipo1 equipo2 > equipo3
   ```

6. Muestre los números de identificación de los usuarios y los grupos que existen en el sistema.

7. Créese un fichero que contenga las siguientes líneas:

   ```
   8 Miguel
   12 Carlos Severo
   9 Roberto Astola
   6 Julia Fernández
   4 Adolfo Domínguez
   10 Pitita Crucecitas
   ```

   Ordene el fichero y muestre el resultado en la salida estándar. Ordénelo numéricamente. Compare los resultados.

8. Sobre el fichero anterior, escriba una orden que muestre sólo los nombres de las personas en mayúsculas.

9. ¿Cuántos usuarios del sistema tienen un nombre de usuario que empieza por `m`?

10. ¿Cómo comprobaría si el usuario `pepe` está conectado al sistema?

11. Obtenga una lista de todos los subdirectorios de su directorio de trabajo.

12. Obtenga una lista de los ficheros de su directorio de trabajo que tienen permiso de lectura para el grupo.

13. ¿Qué hacen estas órdenes?

    a) `$ egrep A[0-9]+Q numserie`
    b) `$ ypcat passwd | egrep '[Cc]onde|[Mm]ario'`
    c) `$ egrep -c 'DIVISION|SECTION' ~/cobol/*`
    d) `$ who | egrep 'tty?'`

14. Diga con qué concuerdan las siguientes expresiones regulares:

    a) `[a-z]+[0-9]`
    b) `[a-z]+[0-9]?`
    c) `[a-z]*[0-9]`
    d) `Tony|Antony`
    e) `(Tony|Antony) Quinn`
    f) `Tony|(Antony Quinn)`
    g) `(so?)+`

15. Combine dos ficheros, ordénelos y elimine las líneas duplicadas.

16. Créese un fichero con los siguientes valores:

```
-40
14.8
14,8
-12.6
0
0.00
-9
```

    Ordénelos primero sin emplear ninguna opción. Segundo con la opción `-n`. tercero con las opciones `-nu`. Compare los resultados.

17. ¿Qué hacen estas órdenes?

a) `$ who | grep "06"`

b) `$ ls -l | grep ^d`

c) `$ ypcat passwd | grep '^[^:]*::'`

d) `$ grep -c '......' palabras`

18. Obtenga en la salida estándar una lista de todos los usuarios del sistema cuyo nombre de *login* empiece por `n` y que utilicen el *shell* bash.

19. Escriba en una sola línea una orden que busque a partir de su directorio de entrada, todos los ficheros que terminen en `.c` y los imprima ordenados alfabéticamente.

20. A partir del fichero `/usr/share/dict/words`, créese otro llamado `WORDS` donde todas las letras minúsculas se conviertan en mayúsculas.

Capítulo 6

Introducción al *shell* bash

6.1. ¿Qué es un *shell*?

Como se comentó en el capítulo 1, el *shell* es una interfaz entre el sistema GNU/Linux y el usuario. Es decir, es un programa que interpreta las órdenes que introduce el usuario, las traduce en instrucciones que puede entender el sistema operativo, y devuelve al usuario la salida que proporciona el primero.

GNU/Linux dispone de varios *shells*; los más difundidos son: el C, de Korn, Z y bash. Éste último es el que vamos a utilizar nosotros.

6.2. El *shell* bash

El nombre del *shell* bash es un acrónimo de "*Bourne-Again SHell*", el primer *shell* escrito para un sistema Unix obra de Steve Bourne, que apareció en la Séptima Edición de los Laboratorios Bell de Unix.

El *shell* bash se creó para ser usado en el proyecto GNU, que fue iniciado por Richard Stallman de la *Free Software Foundation* con el propósito de crear un sistema operativo compatible con Unix y reemplazar todas las utilidades comerciales de Unix con otras disponibles de forma gratuita.

Bash pretendía ser el *shell* estándar del sistema GNU. Las versiones originales fueron escritas por Brian Fox. Posteriormente se le unió Chet Ramey.

El *shell* bash es compatible con el de Bourne, e incluye las características principales del C y el de Korn, así como aspectos genuinos. Al igual que otros *shells* se ocupa de las tareas siguientes:

- Como intérprete de órdenes es responsable de leer y ejecutar las órdenes que le damos desde nuestro terminal.
- Como lenguaje de programación nos permite escribir programas sofisticados, que reciben el nombre de *scripts*. También nos posibilita personalizar nuestro entorno de trabajo mediante éstos.

En este capítulo estudiaremos las siguientes características del *shell* bash:

- La orden `echo`.
- Manejo de la entrada/salida estándar.
- Protección de caracteres especiales.
- Listas de órdenes.
- Ambientes de ejecución.
- Agrupación de órdenes.
- Órdenes incorporadas.
- Control de trabajos.
- Aspectos de configuración del entorno del *shell* bash, tales como alias, opciones, variables y los ficheros de arranque.

6.3. Cómo obtener ayuda sobre el *shell* bash

Para obtener información sobre el *shell* bash tenemos varias fuentes: la página del manual, la documentación proporcionada por el sistema `info` y la orden `help`. Como ya conocemos cómo se manejan las dos primeras, vamos a describir aquí la orden `help`, que nos permite obtener información sobre las órdenes del *shell* bash. Su formato es:

`help` [*orden*]

Si la damos sin argumentos nos muestra una lista de todas las órdenes y su formato; si le pasamos como argumento el nombre de una orden nos muestra información sobre ella.

La orden `help` es una característica propia del *shell* bash, y nos proporciona información sobre las órdenes incorporadas en él. Las **órdenes incorporadas** (*builtin commands*) se diferencian de las órdenes de GNU/Linux en que no son órdenes independientes, por tanto, no se ejecutan como un proceso separado del *shell*.

Ejercicio:

La orden incorporada `history` nos permite visualizar las últimas líneas de órdenes ejecutadas. ¿Cómo podría obtener información sobre ella? Escriba la orden para mostrar en pantalla las 6 últimas líneas de órdenes que haya dado.

6.4. Escribir en la salida estándar: la orden `echo`

El mandato incorporado `echo` muestra en la salida estándar los argumentos que se le pasan. Su formato es:

`echo` [*opciones*] *argumentos*

Opciones:

`-n` Cuando se especifica se suprime el carácter `Nueva-Línea` al final del argumento mostrado.

`-e` Habilita la interpretación de los caracteres de escape de los que vamos a hablar a continuación.

`-E` Deshabilita la interpretación de los caracteres de escape, incluso en aquellos sistemas en que está habilitada su interpretación por omisión.

Ejemplo:

```
$ echo -n "Introduzca opción:"
Introduzca opción:$
```

Esto es equivalente a:

```
$ echo -e "Introduzca opción:\c"
Introduzca opción:$
```

Hay un cierto número de caracteres especiales de *escape*, al estilo de los del lenguaje C, que permiten formatear los argumentos de `echo` (cuadro 6.1). Por ejemplo, para mostrar los argumentos en líneas separadas, en vez de usar varias veces la orden `echo`, podemos hacer lo siguiente:

```
$ echo -e "X\nY\nZ"
X
Y
Z
```

Carácter	Significado
\a	Campana o pitido (alarma).
\b	Espacio atrás.
\c	La línea no termina con un carácter Nueva-Línea (el resto de los argumentos no son tenido en cuenta).
\n	Nueva-Línea.
\r	Retorno al principio de la línea.
\t	Tabulador horizontal.

Cuadro 6.1: Algunos caracteres de escape para `echo`

\n introduce un carácter Nueva-Línea entre los argumentos.

Los caracteres de escape deben estar encerrados entre comillas, ya que de otra forma no son interpretados correctamente. Pruebe:

```
$ echo -e X\nY\nZ
```

6.5. Protección de caracteres especiales

Una de las características del *shell* bash, estudiada en el capítulo **??**, es la expansión de caracteres con significado especial (*, ?, [], ...). Para que éstos pierdan ese significado y concuerden consigo mismos hay que protegerlos. Esto se puede hacer de las siguientes formas:

1. Se puede proteger un carácter simple precediéndolo del carácter \. Preserva el valor literal del carácter que le sigue, con la excepción del carácter Nueva-Línea. Si aparece la pareja \Nueva-línea, y \ no está protegido, esto se interpreta como una continuación de la línea.

2. Se puede proteger un grupo de caracteres entrecomillándolos:

   Comillas simples '...' retira el significado especial de todos los caracteres especiales excepto el de la comilla simple. No pueden aparecer las comillas simples dentro de otras comillas simples, incluso si van precedidas de \.

   Comillas dobles "..." retira el significado especial de todos los caracteres especiales excepto de $, \ y ‘ (sustitución de órdenes antigua). Permite la sustitución de órdenes y la expansión de parámetros. El carácter \ retiene su significado especial sólo cuando va seguido de uno de los caracteres siguientes: $, ‘, ", \, o

`Nueva-Línea`. Las comillas dobles pueden ser protegidas dentro de otras comillas dobles precediéndolas del caracter \.

Ejercicio:

Explique la salida que producen las siguientes líneas de órdenes:

1. `$ echo \*`
2. `$ echo *`
3. `$ echo dos '\\barras_invertidas'`
4. `$ echo dos "\\barras_invertidas"`
5. `$ echo dos \\barras_invertidas`

6.6. Control de trabajos

Cuando ejecutamos una orden desde una terminal, el *shell* le da a la orden el control de ésta hasta que termina su ejecución; por tanto, no podremos ejecutar otra orden hasta que no termine la anterior. Existe la posibilidad de ejecutar órdenes desde la terminal haciendo que ésta nos quede libre inmediatamente para ejecutar otras. Esto se puede conseguir haciendo que la ejecución se realice en **segundo plano**. Para ello debemos poner al final de la línea de órdenes el símbolo `&`.

Una orden que se ejecuta de esta forma se dice que es un trabajo en segundo plano (*background job*); cuando se ejecutan de la forma normal se habla de trabajo en primer plano (*foreground job*).

La ejecución de órdenes en segundo plano se suele utilizar cuando éstas no necesitan entrada y además tardan mucho tiempo en ejecutarse.

GNU/Linux asigna a cada proceso un identificador, conocido como **identificador del proceso** (PID), que es único en todo el sistema. Asimismo, el *shell* bash asigna a cada trabajo que se ejecuta en segundo plano un número que se conoce como **identificador del trabajo**. Si mandamos a ejecutar una orden en segundo plano, el *shell* bash nos devuelve su número de trabajo y su identificador (PID). El número de trabajo nos va a servir para referirnos a él cuando queramos hacer alguna operación.

Ejemplo:

```
$ find / -name core -exec rm -rf {} \; &
[1] 1435
```

El número entre corchetes, `[1]`*, es el "número de trabajo";* `1435` *es el identificador del proceso (PID).*

Cuando ejecutamos una orden en segundo plano es conveniente redirigir la salida de información (la salida estándar y de errores) a un fichero, para evitar interferencias con el proceso que se ejecuta en primer plano.

El *shell* bash tiene órdenes que nos permiten realizar un control completo de los trabajos. Éstas nos van a permitir parar procesos, rearrancarlos, pasar la ejecución de éstos de primer a segundo plano o viceversa, mostrar su estado, etc.

6.6.1. La orden `jobs`

La orden `jobs` nos muestra todos los trabajos que tenemos y sus identificadores. Su formato es:

`jobs` [*opciones*] [*espec_trabajo*]

Opciones:

`-l` Muestra el identificador de trabajo y el del proceso.

`-p` Sólo nos nuestra el número de identificación del proceso.

Ejemplos:

1. ```
   $ jobs
   [1]-  Running       netscape &
   [2]+  Running       find / -name core > core.file &
   ```
2. ```
   $ jobs -l
   [1]- 1515 Running     netscape &
   [2]+ 2075 Running     find / -name core > core.file &
   ```
3. ```
   $ jobs -p
   1515
   2075
   ```

La descripción de la línea de información que nos proporciona `jobs -l` es la siguiente:

1. El número del trabajo entre corchetes, que puede ir seguido de un signo más, un signo menos o nada.
   ```

- Si es un signo + → Indica que se trata del trabajo actual.
- Si es un signo - → Indica que se trata del trabajo previo.

2. El número de proceso.
3. El estado en que se encuentra el trabajo: ejecución, parado, terminado, etc.
4. La línea de órdenes que hemos dado.

6.6.2. La orden `fg`

Un trabajo que se está ejecutando en segundo plano puede pasar a ejecutarse en primer plano mediante la orden `fg`. Su formato es:

`fg` [*identificador*]

Si sólo tenemos un trabajo en segundo plano, dando `fg` pasará a primer plano. Si tenemos varios, habrá que indicarle al *shell* mediante su *identificador* cuál de ellos es el que queremos cambiar, para esto tenemos las siguientes posibilidades:

`%`*n* ó *n*	Hace referencia al trabajo *n*.
`%`*cadena*	Hace referencia al trabajo cuyo nombre empieza con la *cadena* especificada.

Ejemplo:

```
$ fg %2
```

Nos devuelve a primer plano el trabajo número 2.

6.6.3. La orden `bg`

El *shell* bash también nos permite pasar a segundo plano un proceso que se está ejecutando en primer plano. Para ello debemos, en primer lugar, parar su ejecución; esto lo podemos hacer pulsando la secuencia de teclas CTRL-Z. A continuación utilizando la orden `bg` podemos pasarlo a segundo plano. Su formato es:

`bg` [*identificador*]

Si damos la orden `bg` sin argumentos pasa a segundo plano el trabajo parado más recientemente o el único existente. Si hay varios y queremos que actúe sobre uno específico deberemos indicarlo mediante su *identificador*, para ello podemos utilizar cualquiera de las formas vistas anteriormente.

6.6.4. La orden `ps`

La orden `jobs` sólo nos muestra los trabajos que tenemos activos en nuestra sesión del *shell* bash. Sin embargo, a veces a un usuario le interesa conocer todos los procesos que hay en el sistema, independientemente de si han sido ejecutados por él o no. Esta información se puede obtener mediante la orden `ps`, que no es una orden incorporada en el *shell*. Ésta nos da una instantánea de qué procesos se están ejecutando en el sistema. Su formato es el siguiente:

`ps` [*opciones*]

Opciones:

`a` Muestra información de todos los procesos asociados a una terminal (modo texto o consola en X Window), independientemente del usuario que da la orden.

`x` Da información sobre los procesos que no están asociados a una terminal.

`-A` Muestra información sobre todos los procesos del sistema

`-u` ***login*** Muestra los procesos del usuario indicado.

Si no se especifican opciones, muestra los procesos que tiene el usuario en la sesión iniciada.

6.6.5. La orden `kill`

La orden `kill` manda una señal a un proceso. Una señal es un mensaje que un proceso envía a otro cuando ocurre algún suceso anormal o cuando quiere que el otro proceso haga algo. El formato de esta orden es:

`kill` [*-señal*] *proceso* | *trabajo* ...

donde *proceso* identifica al proceso mediante su PID, y *trabajo* es el identificador de trabajo empleado en las órdenes `fg` y `bg`.

Las señales se identifican mediante números o por su nombre. Para ver todas las señales que admite el sistema podemos dar `kill -l`, y nos muestra una lista con los nombres de las señales y los números que les corresponde. Si no se le especifica a `kill` la señal a enviar, manda por omisión la señal `SIGTERM`, que mata los procesos.

Ejemplos:

1. `$ kill -9 %1`
   Manda la señal `9 SIGKILL` *al trabajo 1. Fuerza la orden de eliminación del proceso.*
2. `$ kill -l`
   Lista el nombre de todas las señales.
3. `$ kill 8506`
   Mata al proceso cuyo identificador es 8506.

Ejercicio:

¿Cómo se puede ejecutar la aplicación `kclock` en segundo plano? ¿Cómo podemos saber si tenemos ejecutando algún trabajo cuyo nombre comienza con la cadena `kclock`? ¿Cómo podemos terminar su ejecución?

6.7. Configuración del *shell* bash

6.7.1. Opciones del *shell*

El *shell* bash permite establecer opciones que modifican su comportamiento. Para establecer estas opciones se utiliza la orden:

`set -o` *opción*

Si queremos desactivar una opción utilizaremos:

`set +o` *opción*

El cuadro 6.2 muestra algunas de las opciones con las que cuenta el *shell* bash. Para ver las que están establecidas se pueder dar `set -o` o `set +o`.

Opción	Descripción
allexport	Todos los parámetros que se definan a continuación serán exportados automáticamente.
emacs	En la edición de la línea de órdenes se utilizan las teclas correspondientes del editor `emacs`.
ignoreeof	El *shell* no finalizará cuando se le dé un carácter de fin de fichero (la combinación de teclas CTRL-D). Para terminar la sesión hay que dar la orden `exit`.
monitor	Los trabajos en segundo plano se ejecutarán en un grupo de procesos separado.
noclobber	No se reescribirá un fichero existente con un operador de redirección. Para reescribir el fichero habrá que especificar `>|`.
noexec	Lee órdenes y comprueba si existen errores de sintaxis, pero no las ejecuta. Se ignora cuando se trabaja en interactivo.
noglob	Desactiva la expansión de nombres de ficheros.
privileged	Se activa siempre que el uid(gid) real no sea igual que el uid(gid) efectivo.
vi	Usa teclas como las del editor `vi` para editar la línea de órdenes.

Cuadro 6.2: Opciones del *shell* bash

6.7.2. Variables del *shell*

El *shell* bash utiliza una serie de variables que pueden dividirse en dos grupos:

Variables establecidas por el *shell* Son variables a las que el *shell* les da un valor determinado y no es conveniente cambiarlo.

Variables usadas por el *shell* Son variables que el usuario puede establecer para modificar el comportamiento del *shell*. En algunos casos, el *shell* asigna un valor por omisión a estas variables.

Para establecer una variable teclearemos:

`VARIABLE=`*valor*

Si lo que queremos es ver todas las variables que hay y su valor, daremos la orden `set`. Para ver el valor de una variable usamos la expresión `$VARIABLE`.

Ejemplo:

```
$ echo $HOME
```

Obtendremos el valor de la variable `HOME`.

Las variables establecidas por el *shell* bash aparecen reflejadas en el cuadro 6.3. En el cuadro 6.4 podemos ver cuales son las variables usadas por el *shell*.

variable	Descripción
PPID	Número de identificación del proceso que ha llamado al *shell* (PID del proceso padre).
PWD	Directorio de trabajo actual.
OLDPWD	Directorio de trabajo previo.
UID	Su valor es el identificador del usuario.
EUID	Su valor es el identificador efectivo del usuario.
BASH	Se expande al camino completo utilizado para llamar al *shell*.
BASH_VERSION	Número de versión del *shell* bash.
RANDOM	El *shell* le asigna un número entero al azar, entre 0 y 32767, cada vez que es referenciada.
SECONDS	Su valor es el número de segundos que han pasado desde que se llamó al *shell*. Si se le asigna un valor a esta variable, más tarde su valor será el asignado más el número de segundos que han pasado desde que se le asignó el valor.
HOSTTYPE	Es establecida automáticamente a una cadena que describe el tipo de máquina en la que se está ejecutando el *shell* bash.
OSTYPE	Es establecida automáticamente a una cadena que describe el sistema operativo en el que el *shell* se está ejecutando.
SHELLOPTS	Contiene las opciones establecidas en el *shell* bash.

Cuadro 6.3: Variables establecidas por el *shell* bash

Variable	Descripción
PATH	Lista de directorios donde el *shell* va a buscar las órdenes que le demos cuando no empiecen por `/`. En esta lista los directorios van separados por el carácter ':'.
CDPATH	Su valor es una lista de directorios separados por `:` que es usada por la orden `cd` cuando se le especifica un directorio que no empieza por `/`; en este caso el *shell* bash busca cada uno de los directorios especificados en CDPATH.
HOME	Contiene el camino del directorio de entrada del usuario. El valor de HOME es el argumento que usa `cd` por omisión.
BASH_ENV	Cada vez que se llama al *shell* bash, expande esta variable para generar el camino completo de un *script*; si éste existe será ejecutado. Cuando la opción **privileged** está activada, el *shell* no expande esta variable, y no ejecuta el *script* resultante.
HISTFILE	Nombre del fichero donde se van guardando las órdenes que hemos dado previamente.
HISTFILESIZE	Número máximo de líneas que puede tener el fichero donde se guardan las órdenes previas. Si no se le da un valor, toma 500.
MAIL	Nombre del buzón de correo entrante.
MAILCHECK	Frecuencia con que se comprueba si hay correo (en segundos). Por omisión toma 60 segundos.
MAILPATH	Lista de buzones donde mirar si hay correo.
PS1	Indicador primario. Por omisión toma el valor `bash-n:$`, siendo `n` el número de la versión del *shell* bash.
PS2	Indicador secundario. Si presionamos RETURN antes de introducir una línea de órdenes completa, nos aparece el indicador secundario. Por omisión: `>`.
PS3	Indicador de selección. Se utiliza en conjunción con la orden `select`, que será estudiada más adelante.
PS4	El *shell* utiliza este indicador cuando muestra la traza de ejecución de una orden. Por omisión: `+`.
TMOUT	Si se le da un valor mayor que cero, el *shell* finaliza si no se introduce una orden dentro del número de segundos especificado.

Cuadro 6.4: Variables usadas por el *shell* bash

6.7.3. Alias

El *shell* bash permite dar nombres alternativos a las órdenes; esto se consigue estableciendo los llamados **alias**. Para establecer nuevos alias o ver los que tenemos definidos se usa la orden `alias`.

Dando la orden `alias` sin argumentos, bash mostrará la lista de alias definidos, en la salida estándar, en la forma *nombre=valor*. Si queremos ver el valor que tiene un alias concreto, daremos:

`alias` *nombre*

Si queremos definir un nuevo alias, teclearemos:

`alias` *nombre=valor*

donde *valor* debe contener un texto válido para el *shell*.

Ejemplo:

```
$ alias ll='ls -l'
```

Crea el alias `ll` *que va a ser equivalente a la orden* `ls -l`.

Para desestablecer un alias previamente definido, se usa la orden `unalias` seguida del nombre del alias. Ésta admite la opción `-a` que nos permite borrar todos los alias definidos previamente.

6.7.4. Ficheros de arranque del *shell*

Muchos programas ejecutan ficheros de arranque cuando se les llama. Estos ficheros de arranque sirven generalmente para asegurarnos de que ciertas cosas van a ocurrir siempre que llamemos al programa.

Se suele denominar *script* a un fichero de texto que contiene órdenes que son interpretadas por GNU/Linux y por un *shell*. Los ficheros de arranque de los que hemos hablado anteriormente son *scripts*.

Cuando cualquier usuario inicia su sesión el sistema sabe qué *shell* va a utilizar porque se le indica en el fichero `/etc/passwd`. Si el *shell* de ese usuario es el bash, éste busca el fichero `/etc/profile` y si existe lo lee. A continuación busca los ficheros `~/.bash_profile`, `~/.bash_login` y `~/.profile`, en el

orden indicado y el primero de ellos que encuentra es el que lee, sin seguir buscando más.

El *script* `/etc/profile` se ejecuta primero y es compartido por todos los usuarios del sistema que utilizan el *shell* bash; por tanto, es un buen sitio para que el administrador del sistema ponga información que deberían tener todos los usuarios de éste. Cualquiera de los tres restantes puede ser configurado según las necesidades de cada usuario ya que se encuentran en su directorio de entrada.

¿Qué tipo de información suelen contener los ficheros de arranque? En general, todo lo que deseemos que se haga cuando iniciemos una sesión en el sistema. Entre otras cosas, establecer variables, opciones y alias, así como ejecución de otras órdenes útiles.

Al terminar la sesión el *shell* bash busca el fichero `~/.bash_logout` y si existe lo lee.

Cuando ejecutamos al *shell* bash de forma interactiva (ejecutamos `bash` en la línea de órdenes), éste busca el fichero `~/.bashrc` y si existe lo lee.

Cuando se llama al *shell* de forma no interactiva (ejecución de un *script*) hace lo siguiente: si la variable `BASH_ENV` está definida, la expande y lee el fichero que indica su valor.

6.8. Ambientes de ejecución

Suponga que está en el *shell* bash y ejecuta un *script* escribiendo su nombre. Por ejemplo:

```
$ miscript
```

El *script* que se está ejecutando es un proceso hijo del *shell* bash. El *shell* bash es el padre del *script*. Cada proceso padre o hijo tiene su propio ambiente de ejecución. Este ambiente es realmente un conjunto de valores de variables, privilegios y recursos.

¿Qué cosas pasa un proceso padre a su hijo? En el cuadro 6.5 se resumen todas las características del ambiente y si éstas las heredan o no los procesos hijos. Por omisión, un proceso padre pasa a su hijo sus derechos y privilegios. Por ejemplo, si el padre tiene permiso para leer un fichero particular, también lo tiene el hijo. Sin embargo, el padre no pasa al hijo variables, alias ni funciones. Para que el hijo pueda *ver* una variable o función del padre, éste deberá *exportarla* al hijo. La forma de exportar una variable es:

No heredadas	Alias. Opciones.
Heredadas	Derechos de acceso a los ficheros, directorios, etc. Ficheros abiertos. Límites de recursos. Ej.: la cantidad de memoria principal que el proceso puede usar. La respuesta del padre a señales.
Heredadas, si se exportan	Funciones. Variables.

Cuadro 6.5: Características que forman parte del ambiente de un proceso

`export` *variable*

Los ficheros de arranque se utilizan para definir y exportar, en su caso, variables, opciones, que queramos tener disponibles en todas las sesiones o cuando ejecutemos un *script*.

6.8.1. *Dot Scripts*

Un *dot script* es un *script* que se ejecuta en el ambiente del proceso padre. En otras palabras, a diferencia de los *scripts* normales, un *dot script* no es un proceso hijo del proceso que lo llama. El *dot script* hereda todo el ambiente del llamador, incluyendo las variables que no han sido exportadas.

Un *dot script* puede contener el mismo código que un *script* regular. Las diferencias entre un *script* y un *dot script* no están en su contenido sino en la forma de llamarlos. Para llamar a un *script* regular, normalmente se usa su nombre; por ejemplo:

```
$ ficus.bash
```

Para llamar un *dot script*, se debe preceder el nombre del *script* con la orden '`.`'; por ejemplo:

```
$ . ficus.bash
```

En otras palabras, un *script* regular se convierte en un *dot script* cuando se le llama de la forma anterior. Un sinónimo de esta orden es `source`.

6.8.2. *Subshell*

El *shell* bash permite la agrupación de órdenes mediante los caracteres `( )` y `{ }`. La diferencia entre ambos agrupamientos es el ambiente de ejecución.

Si utilizamos `( )` el agrupamiento se ejecuta en un *subshell*, es decir, en un ambiente de ejecución distinto al del proceso padre. Sin embargo, si se emplea `{ }` se ejecutan en el mismo ambiente que el proceso padre.

Un *subshell* es una copia separada del *shell* padre, de forma que las variables, funciones, y alias del *shell* padre están disponibles para el *subshell*. Sin embargo, los cambios que se realicen en ellas en el ambiente del *subshell* no afectan al ambiente del *shell* padre. Al ser el *subshell* una copia, no cambiamos los valores de los datos del *shell* padre, sino sólo los de la copia. Por tanto, se ejecuta en un ambiente distinto.

Ejemplos:

1. ```
   $ (A=3)
   $ echo $A
   ```

   Con los caracteres `( )` *hemos creado un subshell, de forma que la variable* `A`*, sólo está definida en él. De este modo, cuando termine la ejecución del subshell no existe la variable.*

2. ```
   $ { A=3; }
   $ echo $A
   3
   ```

   En este caso, el agrupamiento se ejecuta en el ambiente del proceso padre, por lo que una vez finalizado la variable mantiene su valor.

Ejercicio:

Indique de forma razonada qué tiene que hacer para que al entrar al sistema su sesión del *shell* bash posea las siguientes características:

- Los ficheros que cree deben tener permiso de lectura y escritura para el propietario y el grupo, y de sólo lectura para el resto. Los directorios además deben tener permiso de ejecución para el propietario y el grupo.
- Cuando dé la orden `ls` se realizará un listado largo en color donde además se muestra el nodo-i.
- El indicador de entrada será: "`Hola` *login* `$`", siendo *login* su nombre de usuario.
- Cuando la orden introducida esté incompleta, el indicador secundario avisará de tal circunstancia, mediante el mensaje: "`Orden incompleta` *login* `>`".
- Se protegerán los ficheros contra borrados accidentales a la hora de redireccionar la salida de una orden.
- El *shell* bash finalizará si el usuario no interacciona con el sistema durante 60 segundos.
- Cuando se utilice una variable que no ha sido declarada previamente debe mostrar un mensaje de error.

6.9. Mandatos incorporados (*builtin commands*)

Algunas de las órdenes que usamos habitualmente y que pensamos son básicas en GNU/Linux no son programas en el sentido usual, sino que están incorporadas en el *shell*. ¿En qué se diferencian de los programas normales? Al estar incorporadas en el *shell*, a la hora de ejecutarlas no se crea un proceso hijo.

Originalmente, el *shell* proporcionaba sólo aquellas órdenes incorporadas que eran indispensables; es decir, aquéllas que no se podían construir de otra forma. Éste es el caso de la orden `cd`. Si la orden `cd` estuviera implementada de la forma usual, no valdría para nada, ya que el cambio de directorio sólo sería efectivo durante el tiempo que durara su ejecución, pero el directorio de trabajo volvería a su valor original cuando acabara la ejecución de la orden y éste no es evidentemente el resultado deseado. Por tanto, la única forma práctica de implementar la orden `cd` y otras como ella es haciendo que el *shell* cambie su propio ambiente; en este caso, todos los procesos creados por el *shell* heredarán ese ambiente.

Puesto que las órdenes incorporadas pueden ser ejecutadas sin la sobrecarga de tiempo que supone la carga de un programa, su ejecución suele ser más rápida que la de una orden equivalente implementada como un programa. Por esta razón, en la actualidad se han añadido otras órdenes de este tipo, aunque no son indispensables.

Algunas de las órdenes incorporadas que nos brinda el *shell* bash son: `.`, `alias`, `bg`, `cd`, `echo`, `exec`, `exit`, `export`, `fg`, `jobs`, `kill`, `enable`, `pwd`, `umask`, `unalias`, etc.

Otro mandato incorporado que proporciona el *shell* bash es `help`, éste nos permite obtener información sobre otros mandatos incorporados sin tener que ver la página del manual correspondiente a bash completa. Si damos la orden `help` sola obtendremos una lista de todas las órdenes sobre las que podemos conseguir información. Si damos `help` seguido del nombre de una orden, obtendremos información sobre ésta.

Capítulo 6

Programación en el *shell* bash Parte II

6.10. ¿Qué es un *script*?

El *shell* bash, al igual que otros, proporciona un lenguaje de programación. Los programas que se realizan utilizando el lenguaje del *shell* se denominan *shell scripts*, o simplemente *scripts* (aunque a veces se puede encontrar la traducción literal "guiones").

Un *script* es un fichero de texto en el que se introducen órdenes que son interpretadas por el *shell* y por el sistema operativo. Se suelen usar para automatizar procesos de administración de sistemas, tales como realización de copias de seguridad, mantenimiento de servidores, migraciones, etc.

El fichero del *script* puede crearse con cualquier editor de textos, escribiendo una orden en cada línea. Una vez creado, podemos ejecutarlo de cualquiera de las dos formas siguientes:

1. `$ bash` *script* [*parámetro* ...]
2. `$` *script* [*parámetro* ...]

En el primer caso, no es necesario activar previamente el permiso de ejecución del fichero. En el segundo caso, sí lo tenemos que activar. Además la mayoría de sistemas LINUX no incorporan el directorio punto (.) en el PATH, por lo que es necesario indicar la dirección absoluta (o relativa comenzando por punto) del fichero para ejecutar el *script*. Por otro lado, hay que tener en cuenta que para poder ejecutar un *script*, además del permiso de ejecución,

se necesita el permiso de lectura (porque es un lenguaje interpretado). Los permisos SUID y SGID no tienen efecto sobre los ellos.

La forma en que se ejecuta un *script* es similar a cualquier otro programa: el shell ejecutará la primera orden. Cuando termine pasará a ejecutar la segunda y así sucesivamente. Esta sucesión se rompe con las órdenes de control de flujo (`if`, `while`, etc) que veremos más adelante.

A la hora de ejecutar un *script* podemos pasarle argumentos que van a ser utilizados durante su ejecución, son los denominados **parámetros posicionales**.

Si ejecutamos un *script* de cualquiera de las dos formas anteriores, se estará creando un proceso hijo y por tanto se estará ejecutando en un ambiente separado. Esto significa que las variables del ambiente actual no están disponibles para los *scripts* a menos que se hayan exportado explícitamente, asimismo las variables definidas dentro del *script* no serán heredadas por el *shell* padre.

Es bastante usual que el *script* dé errores al ejecutarlo; para averiguar dónde están es conveniente depurarlo, y para ello lo ejecutaremos de la siguiente forma:

`$ bash -xv` *script* [*parámetro ...*]

Cuando se llama al *shell* bash con la opción `-x`, se van mostrando las órdenes que se ejecutan junto a sus argumentos. La opción `-v` muestra las líneas de órdenes leídas por el *shell* bash. De esta forma, podemos ver lo que va interpretando el *shell*, así como los resultados tras ejecutar cada una de las órdenes del *script*.

6.11. Parámetros en el *shell* bash

Se denominan parámetros todas aquellas entidades que almacenan valores en el *shell* bash. Podemos distinguir diversos tipos de parámetros:

Variables Los parámetros que tienen un nombre se denominan variables. Éstas pueden tener atributos que son establecidos por el usuario.

Parámetros posicionales Estos parámetros se denotan por uno o más dígitos. Mantienen los argumentos que se le pasan al *script* o a una función.

Parámetros especiales El *shell* establece automáticamente su valor y se denotan por los caracteres

`* @ # ? - $ !`

6.11.1. Expansión básica de parámetros

Expansión de parámetros es el término que se usa para designar a la capacidad de acceder y manipular los valores de las variables y parámetros. La expansión básica se hace precediendo el nombre de la variable o parámetro con el carácter `$`. En algunos casos es necesario usar la forma `${`*parámetro*`}`. ¿Cuándo es necesario usar esta segunda forma?

- Cuando tenemos más de nueve parámetros posicionales, para acceder al décimo y siguientes, tendremos que encerrarlos entre llaves. Ejemplo: `echo ${12}`.
- Cuando queremos extraer el valor de un elemento de una variable de tipo vector. Esto se verá más adelante.
- Cuando queremos concatenar el valor de una variable a una cadena.

Se pueden usar otros tipos de expansión para devolver la longitud del parámetro, porciones del mismo, etc. Esto se verá más adelante.

6.11.2. Parámetros posicionales y especiales

Los parámetros posicionales sirven para almacenar los argumentos que se le pasan al *script* cuando se ejecuta. Los parámetros posicionales tienen los nombres `1`, `2`, `3`, etc. Esto significa que sus valores son denotados como `$1`, `$2`, `$3`, etc. Hay también un parámetro posicional `0`, cuyo valor es el nombre del *script*.

El parámetro `#` contiene el número de los posicionales (como una cadena de caracteres). Los parámetros especiales `*` y `@` contienen todos los parámetros posicionales, excepto el `0`. ¿Cuál es la diferencia entre ellos? Más adelante la contaremos.

Otros parámetros especiales son `?` y `$`. El parámetro `?` contiene el status de salida de la última orden ejecutada, mientras que `$` contiene el identificador del proceso (PID) correspondiente al *shell* actual.

Todos estos parámetros son de sólo lectura, es decir, no se les pueden asignar nuevos valores dentro de un *script*.

Ejemplo:

Escriba el *script* 6.1 en un fichero. Utilice la orden `chmod` para dotarle de los permisos adecuados. Ejecute el *script* pasándole tres parámetros posicionales cualesquiera.

***Script* 6.1** Parámetros posicionales

```
#!/bin/bash
# ejemplo1: Ejemplo de manejo de parámetros posicionales

echo "El nombre de este script es $0"
echo "El primer parámetro posicional es $1"
echo "El segundo parámetro posicional es $2"
echo "El tercer parámetro posicional es $3"
echo "El numero de parámetros pasados es $#"
echo "Todos los argumentos pasados al script son: $*"
```

Observaciones:

- Se ha puesto como primera línea del *script* la siguiente:

  `#!/bin/bash`

  Esto se hace para indicar qué *shell* es el que va a interpretar nuestro *script*. Si no se pone esa línea, el *shell* que se use depende del que estemos usando en interactivo, y del sistema. Los caracteres `#!` deben ser los dos primeros del fichero.
- La línea que comienza por `#` es una línea de comentario.
- La orden `echo` muestra en la salida estándar los parámetros que se le pasan. Más adelante la estudiaremos en mayor profundidad.

Desplazamiento de los parámetros posicionales

Los parámetros posicionales pueden ser reasignados con la orden `shift`. Ésta desplaza el valor de los parámetros posicionales el número de posiciones que se le indica mediante el parámetro *n*. Si no se le indica ningún parámetro se desplaza una posición.

Formato:

`shift` [*n*]

Ejemplo:

El *script* 6.2 es una modificación del 6.1. Ejecútelo pasándole los mismos argumentos que antes.

Sustituya en el *script* 6.2, la línea que contiene la orden `shift` por `shift 2`, y vuelva a ejecutarlo pasándole los mismos argumentos.

***Script* 6.2** Desplazamiento de parámetros posicionales

```
#!/bin/bash

# ejemplo3: Ejemplo de manejo de los parámetros
# posicionales junto con la orden shift.

echo "El nombre de este script es $0"
echo "El primer parámetro posicional es $1"
echo "El segundo parámetro posicional es $2"
echo "El tercer parámetro posicional es $3"
echo "El numero de parámetros pasados es $#"
echo "Todos los argumentos pasados al script son: $*"
shift
echo "El número de argumentos que quedan es $#"
echo "Los argumentos que quedan son $*"
echo "Arg1=$1, Arg2=$2, Arg3=$3"
```

6.11.3. Variables

En la elaboración de un *script* el programador puede hacer uso de las variables que considere necesarias. Los nombres de éstas pueden contener cualquier combinación de:

- Letras (mayúsculas o minúsculas).
- Dígitos.
- Subrayados (_).

Una limitación que se impone es que no pueden comenzar por un dígito. Sin embargo no se impone ninguna limitación a la longitud del nombre de la variable. Hay que tener en cuenta que los caracteres en mayúsculas y en minúsculas son distintos en los nombres de las variables. Así, `LORO`, `Loro` y `loro` son tres variables distintas.

El *shell* bash reconoce tres tipos de datos:

Cadenas Por omisión, todas las variables son de tipo cadena, a no ser que se indique otra cosa. No es necesario especificar su longitud cuando se declaran.

Enteros La declaración de una variable como entero permite realizar operaciones aritméticas.

Vectores Por omisión, los elementos del vector son declarados de tipo cadena. Cada elemento se referencia mediante un índice, correspondiéndole al primero el índice cero. Los vectores no tienen ningún límite en su tamaño, y sus elementos no tienen que estar de forma contigua.

Se pueden especificar atributos para todos los tipos de variables.

Asignación de valores

Para asignar un valor a una variable podemos hacerlo de varias formas:

1. *variable*=*valor*
2. `declare` [±*atributo*] *variable*=*valor*
3. `local` *variable*=*valor*

Si lo hacemos mediante la orden `declare` podremos además especificar al mismo tiempo atributos para las variables. La orden `local` sólo se puede utilizar dentro de una función, en este caso la variable será local a la función donde se encuentre. De esto hablaremos más adelante.

En el caso de una variable de tipo vector, cuando queramos asignarle un valor a un elemento hemos de indicar su índice correspondiente:

variable`[`*índice*`]=`*valor*

Si no se especifica *índice* se utilizará el empleado en la última asignación más uno, y si no se hubiera asignado ningún elemento se tomará el índice cero. Igualmente se pueden realizar asignaciones a distintos elementos. Los diferentes modos de asignar valores a este tipo de variable se pueden ver en el siguiente ejemplo.

Ejemplos:

1. `nombres[42]=Alicia`

`nombres[96]=Antonio`
Se asignan los valores `Alicia` *y* `Antonio` *a los elementos 42 y 96 del vector* `nombres`.

2. `asignaturas=([10]=SSOO [8]=MTPII [15]=BD)`
Asigna los valores `SSOO`, `MTPII` *y* `BD` *a los elementos 10, 8 y 15 del vector* `asignaturas`, *respectivamente.*
3. `profesores=([3]=Fernando Eugenio Antonia Juan)`
Asigna los valores `Fernando`, `Eugenio`, `Antonia` *y* `Juan`, *a los elementos 3, 4, 5 y 6 del vector* `profesores`, *respectivamente.*

El índice de un elemento de un vector debe ser entero, y puede ser el resultado de una operación aritmética. No obstante, si el índice es `*` o `@`, el resultado equivale a todos los elementos del vector, separados por espacio. Pero, si el vector está encerrado entre comillas dobles, el resultado es diferente, pues el `*` se expande a cada uno de los valores de cada elemento separados por el primer carácter de la variable IFS.

Ejemplo:

Ejecute el *script* 6.3 y observe las diferencias.

***Script* 6.3** Utilización de una variable vector

```
#!/bin/bash

declare -a dos=(pepe antonio juan)
IFS=:
echo ${dos[*]} con \*
echo ${dos[@]} con \@
echo "${dos[*]}" con \* y comilla
echo "${dos[@]}" con \@ y comilla
```

Una característica de este tipo de dato es que los elementos no tienen por qué estar en posiciones contiguas, y pueden existir elementos vacíos.

Atributos de las variables

Se pueden establecer atributos para las variables con la orden `declare`,

`declare -`*atributo variable*

En la tabla 6.1 se muestran los atributos que se pueden especificar para las variables.

Atributo	Significado
`i`	Tipo entero.
`r`	Sólo lectura.
`x`	Se exportará.
`a`	Será un vector de tipo cadena.

Cuadro 6.1: Atributos para las variables

Ejemplo:

```
$ declare -ai notas
```

La variable `notas` *se declara como un vector de enteros.*

Excepto en el caso del atributo de sólo lectura, los demás pueden ser eliminados con la orden:

`declare` +*atributo variable*

Otras operaciones con variables

- Se les puede asignar el valor de otra variable:

  Ejemplo:

  ```
  $ X=$HOME
  $ Y=$X
  $ echo $Y
  ```

- Se pueden concatenar.

  Ejemplos:

  1. Concatenación de dos variables

     ```
     $ X=hola
     $ Y=adios
     $ Z=$X$Y
     $ echo $Z
     ```

  2. Concatenación de una variable y una cadena

```
$ M=man
$ M=${M}zana
$ echo $M
```

Observe que hemos encerrado el nombre de la variable entre `{}`. ¿Qué obtendríamos si hubiéramos dado `M=$Mzana`?

- Se les puede asignar la salida de una orden. ¿De qué forma?

  variable=‘*orden*‘ (compatible con el *shell* de Bourne)

  o bien

  variable=`$(`*orden*`)`

  Ejemplo:

```
$ quien=$(who)
$ echo $quien
```

- Se pueden desestablecer:

  `unset` *variable*

  Ejemplo:

```
$ unset quien
$ echo $quien
```

- Se puede calcular su longitud.

  `${#`*variable*`}`

  Ejemplo:

```
$ X=abcd
$ echo ${#X}
4
```

6.12. Funciones

La noción de función en un *script* es similar a la que tienen otros lenguajes de programación: una serie de instrucciones que solucionan un problema concreto. Pueden recibir parámetros que modifiquen su comportamiento y devolver un valor al programa que las invocó.

Las funciones son ideales para organizar *scripts* largos en bloques de código modulares, que son más fáciles de desarrollar y mantener.

Para definir una función podemos usar una de las dos formas siguientes:

```
function nombre-función () {
  órdenes
}
```

o bien:

```
nombre-función () {
  órdenes
}
```

Cuando definimos una función, le decimos al *shell* que almacene su nombre y definición (es decir, las órdenes que contiene) en memoria. Si queremos ejecutar la función más tarde, sólo tendremos que teclear su nombre seguido de los argumentos que requiera, como si se tratara de un *script*.

Para pasar el control de una función al *script* que la ha llamado y pasarle un valor de salida se utiliza la orden `return`. La sintaxis de `return` es:

```
return [n]
```

donde *n* es el valor de retorno que se devuelve al *script* (este valor debe ser un número entero). Para poder utilizar este valor en el programa que llamó a la función se emplea el parámetro especial `?`.

6.13. Ámbito de los parámetros

Cualquier variable definida dentro de un *script* tiene ámbito global. Sin embargo, es posible hacer que una variable sea local a una función utilizando las órdenes `declare` o `local` dentro de su definición. Si éstas no se emplean también serán globales.

Al igual que un *script* utiliza los parámetros posicionales para manejar los argumentos que se le pasan, cada función también maneja los suyos propios mediante éstos. Es decir, los parámetros posicionales son locales al *script* y a cada una de las funciones. Igual ocurre con los parámetros especiales `@`, `*` y `#`.

***Script* 6.4** Manejo de parámetros posicionales y funciones

```
#!/bin/bash
# ejemplo2: Manejo de parámetros posicionales dentro y
# fuera de funciones.

function fun1(){
  echo "En la función: $1 $2"
  var1="dentro de la función"
}

var1="fuera de la función"
echo "El valor de var1 es: $var1"
echo $0: $1 $2
fun1 hola adios
echo "El valor de var1 es: $var1"
echo $0: $1 $2
```

Ejemplo:

Si ejecutamos el *script* 6.4 como se indica, los resultados serán:

```
$ ejemplo2 rosa clavel
El valor de var1 es: fuera de la función
ejemplo2: rosa clavel
En la función: hola adios
El valor de de var1 es: dentro de la función
ejemplo2: rosa clavel
```

Como se puede observar, la función fun1 cambia el valor de la variable var1, y este cambio es conocido fuera de la función, mientras que `$1` *y* `$2` *tienen valores diferentes en la función y en el script. Modifique el ejemplo anterior definiendo* var1 *como una variable local a fun1.*

Cuando hablamos de los parámetros posicionales hicimos referencia a algunos parámetros especiales como `*` y `@`. Vamos ahora a explicar la diferencia

entre `$*` y `$@`. `"$*"` es una cadena simple que consta de todos los parámetros posicionales, separados por el primer carácter de la variable de ambiente IFS, que por omisión es el carácter espacio, TAB o `Nueva-Línea`. `"$@"` es igual a `"$1" "$2" ... "$`N`"`, donde N es el número de parámetros posicionales.

¿Por qué los elementos de `"$*"` están separados por el primer carácter de IFS en vez de sólo por espacios? Para dar flexibilidad a la salida. Veamos un ejemplo simple: supongamos que queremos imprimir la lista de todos los parámetros posicionales separados por comas. Esto lo podríamos conseguir introduciendo en nuestro *script*:

```
IFS=,
echo "$*"
```

¿Por qué `"$@"` es equivalente a los N parámetros por separado? Para permitirnos usarlos otra vez como valores separados. El script 6.5 muestra un ejemplo para diferenciarlos, junto con la utilización del parámetro especial `?`.

***Script* 6.5** Parámetros especiales

```
#!/bin/bash

function cuenta-argumentos() {
  return $#
}

cuenta-argumentos "$@"
echo "Con \@ hay $? argumentos"
cuenta-argumentos "$*"
echo "Con \* hay $? argumentos"
```

Si al ejecutar el *script* 6.5 le pasamos los argumentos `manzana`, `pera` y `melón`, el resultado que obtendremos será:

```
Con \@ hay 3 argumentos
Con \* hay 1 argumentos
```

6.14. Leer de la entrada estándar: la orden `read`

El mandato incorporado `read` se usa para leer la entrada de la terminal o de un fichero.

Formato:

`read` *variable* ...

La orden `read` lee una línea de la entrada estándar y la parte en palabras delimitadas por cualquiera de los caracteres del valor de la variable de ambiente IFS (por omisión espacio, tabulador y `Nueva-Línea`) asignándole cada palabra de la entrada a cada una de las variables. Si no hay suficientes variables para todas las palabras de la entrada, la última contendrá las palabras restantes.

Ejemplos:

1. ```
   $ read X Y Z
   manzana naranja pomelo pera
   $ echo $X
   manzana
   $ echo $Y
   naranja
   $ echo $Z
   pomelo pera
   ```
2. ```
   $ IFS=:
   $ read PAL1 PAL2 PAL3 PAL4
   manzana:naranja:pomelo:pera
   $ echo $PAL1 $PAL2 $PAL3 $PAL4
   manzana naranja pomelo pera
   ```
   Es útil cuando queremos leer datos que no están separados por espacios.

Si a `read` sólo se le pasa como argumento una variable se le asignará a ésta la línea completa.

Ejemplo:

```
$ read LINEA
naranja manzana pera
$ echo $LINEA
naranja manzana pera
```

Si se omiten todos los nombres de variables, se asigna la línea completa a la variable REPLY.

Ejemplo:

```
$ read
Cualquier cosa
$ echo $REPLY
Cualquier cosa
```

Leyendo de un fichero

La orden `read` también puede leer de un fichero. Por sí mismo, `read` sólo lee una línea de la entrada; por tanto, para leer el fichero completo habrá que introducir la orden de lectura dentro de un bucle.

Ejemplo:

En el script 6.6 tenemos un ejemplo de cómo leer las líneas de un fichero mediante la orden `read`*. En él se han introducido algunas sentencias que no se han estudiado todavía, tales como* `case` *y* `while`*. La sentencia* `case` *nos permite realizar distintas acciones dependiendo del valor que tenga una determinada variable (*`$#` *en nuestro caso). El bucle* `while` *nos permitirá leer todas las líneas del fichero. Más adelante veremos más detenidamente cómo utilizar ambas. También se ha realizado una operación aritmética con la variable* `LNUM`*.*

Script 6.6 Lectura de un fichero

```
#!/bin/bash

# ejemplo4: Muestra el uso de la orden read.
# Este script admite un parámetro posicional, que
# debería ser el nombre de un fichero. Si no se le da,
# lo preguntará. Una vez le hemos dado el nombre del
# fichero, nos lo muestra con las líneas numeradas.

case $# in
0)    echo -n "Introduzca un nombre de fichero: "
      read FICHERO;;
*)    FICHERO=$1;;
esac

declare -i LNUM=1

while read  LINEA
do
      echo "$LNUM: $LINEA"
      LNUM=$((LNUM+1))
done < $FICHERO
```

6.15. Operaciones aritméticas

El *shell* bash proporciona varias formas de realizar operaciones aritméticas:

1. La orden `let`, cuya forma de uso es:

   `let "`*expresión_aritmética*`"`

   la *expresión_aritmética* puede contener constantes, operadores y variables del *shell* bash. Se deben usar comillas para encerrar la expresión aritmética siempre que ésta contenga espacios u operadores que tengan un significado especial para el *shell*.

2. Encerrando la expresión aritmética de la siguiente forma:

   `$((`*expresión*`))`

3. Indicando la expresión aritmética directamente, sin poner ningún espacio entre los operadores y operandos.

Las siguientes órdenes son equivalentes:

```
let "X=X + 1"
X=$((X+1))
X=X+1
```

Hay que hacer notar que en este caso no hay que preceder las variables del carácter `$`. Las siguientes órdenes son equivalentes:

```
let "X=X + 1"
let "X=$X + 1"
```

Antes de asignar a una variable el resultado de una operación aritmética debemos declarar dicha variable de tipo entero; si no, podemos llevarnos una sorpresa con el resultado obtenido (especialmente usando la tercera nomenclatura (`X=X+1`).

Ejemplos:

1. ```
   $ declare -i numero
   $ numero=12*2
   $ echo $numero
   24
   ```
2. ```
   $ producto=12*2
   $ echo $producto
   ```
   ¿Qué resultado obtendremos? Haga la prueba.

En la tabla 6.2 pueden verse los operadores aritméticos del *shell* en orden de precedencia.

`-`	Menos unario.
`!`	Negación lógica.
`* / %`	Multiplicación, división, módulo.
`+ -`	Adición, sustracción.
`<= <`	Menor o igual, menor que.
`>= >`	Mayor o igual, mayor que.
`==`	Igual a.
`!=`	Distinto a.
`&&`	Y lógico.
`||`	O lógico.
`=`	Asignación.
`+= -= *= /= %=`	Operadores con asignación.

Cuadro 6.2: Operadores aritméticos en orden de precedencia

Hay que comentar que los operadores `+= -= *= /= %=` realizan una operación aritmética y almecenan el resultado en el operando a su izquierda. Por ejemplo:

Ejemplo:

La siguiente sentencia

```
A=A+8;
```

Equivale a

```
A+=8;
```

Los operadores tienen el comportamiento esperado en cualquier lenguaje de programación.

Ejemplo:

Créese un fichero que contenga el *script* 6.7 y otro con valores numéricos, uno por línea. Ejecute el *script*.

```
#!/bin/bash

# ejemplo5: Ejemplo de uso de expresiones aritméticas.
# Lee valores enteros de un fichero y calcula la suma,
# el producto y la media de todos los valores.

case $# in
0)  echo -n "Nombre del fichero de datos: "
    read FICH;;
*)  FICH=$1;;
esac

declare -i CONT=0 SUMA=0 PROD=1 MEDIA=0

while read X
do
   CONT=$((CONT + 1))
   SUMA=$((SUMA + X))
   PROD=$((PROD * X))
done < $FICH

MEDIA=$((SUMA / CONT))

echo "La suma de todos los valores es $SUMA"
echo "El producto de todos los valores es $PROD"
echo "La media de los valores es $MEDIA"
```

6.16. Ejercicios

1. Cree un *script* llamado `busca`, que acepte exactamente el mismo tipo de argumentos que `find`. La diferencia entre `busca` y `find` debe ser que la salida de `busca` esté ordenada alfabéticamente. Nota: no hace falta tener en cuenta la protección de parámetros.

2. Cree un *script* llamado `fichero`, que acepte exactamente tres parámetros. El primero es el nombre de un fichero y el segundo y el tercero son nombres de directorios. El *script* buscará en los directorios indicados (y subdirectorios) ficheros con dicho nombre, y si los encuentra, mostrará por pantalla la información larga de `ls`. Si es posible, evite que el usuario reciba mensajes de error en pantalla.

3. Cree tres ficheros de *script* llamados `respaldo1`, `respaldo2` y `respaldo3`, que realicen una copia de seguridad (usando `tar`) del directorio `datos` (que se encuentra en su directorio casa). Para los siguientes casos, respectivamente:

   a) Sólo se almacena la última copia de seguridad.

   b) Sólo se almacena las dos últimas copias de seguridad de los dos días anteriores.

   c) Se almacenan una copia de seguridad de cada día (puede ayudarse de la orden `date`).

   El *script* se ejecuta una sola vez al día, al final de la jornada laboral.

4. Cree un *script* llamado `recuperacion1`, que restaure los datos que se almacenaron en el ejercicio anterior. El *script* copiará el contenido del directorio `datos` de su directorio de entrada en `datos_anteriores` y restaurará el contenido del fichero copia de seguridad que reciba como parámetro.

5. Realice un *script* llamado `recuperacion2` similar a `recuperacion1`, pero que reciba como segundo parámetro el directorio donde se copiará el contenido del directorio `datos`.

6. Realice un *script* llamado `alta`, que cree el directorio `public_html` dentro del directorio casa del usuario que reciba como parámetro. Además asignará como propietario del directorio a dicho usuario y le pondrá unos permisos que permitan total libertad al propietario y sólo lectura al resto de usuarios.

7. Cree un *script* llamado `segundo` que, simplemente, visualice el segundo parámetro que se le pase, independientemente del número de argumentos que le hayamos pasado. Nota no puede usarse la expresión $2.

8. Haga un *script* llamado `igrep`, que funcione de la misma forma que `grep` pero deberá pedir los parámetros interactivamente.

9. Escriba un *script* llamado `nwho` que le permita saber si un usuario está conectado o no al sistema. El nombre del usuario debe ser aceptado por `nwho` como argumento. Si el usuario está conectado al sistema el procedimiento debe darnos la línea correspondiente al usuario en cuestión que muestra la orden `who` y la cantidad de consolas que tiene abiertas. Si no está conectado no mostrará nada.

10. Realice un *script* llamado `alta`, que cree el directorio `public_html` dentro del directorio casa del usuario que reciba como parámetro. Además asignará como propietario del directorio a dicho usuario, le pondrá unos permisos que permitan total libertad al usuario y lectura y ejecución al resto de usuarios. Por último cree un fichero dentro de dicho directorio llamado `index.html` con el siguiente contenido (siendo <usuario> el nombre del usuario):

```
<HTML>
 <HEAD><TITLE>Suse Linux</TITLE></HEAD>
 <BODY>
  Bienvenido a la web de <usuario>
 </BODY>
</HTML>
```

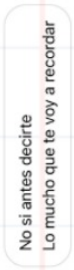
11. Escriba un *script* que permita que le pasen cuatro parámetros posicionales como máximo. Los parámetros que se le pasan deben ser nombres de ficheros y se debe comprobar si estos ficheros existen y si el usuario que ejecuta el *script* puede leerlos. Al final debe indicar el total de ficheros que se pueden leer. Asegúrese de que la órdenes del *script* no genera mensajes de error que lleguen al usuario.

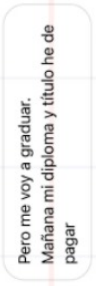
12. Créese un fichero llamado `agenda` donde va colocar una serie de nombres y los números de teléfono correspondientes, separados por el carácter `:`. Escriba un *script* llamado `busca` que reciba como argumento un nombre y responda dando el número de teléfono correspondiente a esa persona. Cuando se le dé un nombre que no esté en la agenda dé el mensaje siguiente:

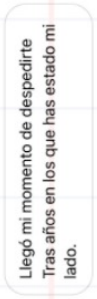
    nombre `no está registrado en la agenda`

    Si no se le pasa ningún nombre debe mostrar la lista completa.

13. Cree un fichero llamado `correos`, que almacenará una dirección de correo electrónico por línea. Escriba un *scripts* llamado `dominios`, que muestre por pantalla los diferentes dominios que aparecen en el fichero sin repeticiones y ordenados alfabéticamente.







14. Cree un fichero llamado `publicidad`, pondrá un anuncio en la página personal del usuario que se le indique como parámetro. Para ello insertará como penúltima línea del fichero `public_html/index.html` de su directorio casa la siguiente expresion:

    <A HREF='http://barrapunto.com'>Visite barrapunto</A>

    Si lo desea puede usar un fichero temporal para realizar este ejercicio.

15. Cree un *script* llamado `ultimo` que, simplemente, visualice el último argumento que se le pase, independientemente del número de argumentos que le hayamos pasado.

QuesoViejo_

si lees esto me debes un besito

Capítulo 6

Programación en el *shell* bash Parte III

6.17. Operaciones con cadenas

Los operadores de cadenas permiten la manipulación de los valores de las variables de una forma sencilla. Permiten hacer las siguientes operaciones:

- Comprobar que una variable está definida.
- Establecer valores por omisión.
- Mostrar mensajes de error si una variable no está establecida.
- Eliminar subcadenas que coincidan con patrones.

6.17.1. Concordancia con patrones

Los operadores empleados para eliminar subcadenas son:

- `${`*parámetro*`#`*patrón*`}`

  Esta construcción se expande al valor del *parámetro*, borrándole la porción más pequeña que concuerde con el *patrón* por la izquierda.

  Ejemplo:

```
$ X=datos.tar.gz
$ echo ${X#*.}
tar.gz
```

- **${*parámetro*##*patrón*}**

  Se expande al valor del *parámetro*, borrándole la porción más grande que concuerde con el *patrón* por la izquierda.

  Ejemplo:

  ```
  $ X=$(pwd)
  $ echo $X
  /home/usuario/pepe
  $ echo ${X##*/}
  pepe
  ```

  Compare este resultado con el obtenido al ejecutar la orden `basename`.

  Estos dos tipos de expansión por la izquierda son útiles para extraer los distintos directorios que componen el nombre de un fichero, para identificar sus permisos, etc.

- **${*parámetro*%*patrón*}**

  Se expande al valor del *parámetro*, borrándole la porción más pequeña que concuerde con el *patrón* por la derecha.

  Ejemplos:

  ```
  $ X=hola.c
  $ echo ${X%.*}
  hola
  ```

- **${*parámetro*%%*patrón*}**

  Esta forma es expandida al valor del *parámetro*, borrándole la porción más grande que concuerde con el *patrón* por la derecha.

  Ejemplo:

  ```
  $ X=usr/spool/mail
  $ echo ${X%%/*}
  usr
  ```

Estos dos tipos de expansiones por la derecha permite eliminar las diferentes extensiones del nombre de un fichero, etc.

6.17.2. Operadores de sustitución

Uso de valores por omisión

La construcción `${`*parámetro*`:-`*palabra*`}` se expande al valor del *parámetro* si éste está establecido y no es nulo; en caso contrario, se expande a *palabra*.

Ejemplo:

```
$ DIRECTORIO=~/temporal
$ echo ${DIRECTORIO:-/tmp}
/home/pepe/temporal
$ unset DIRECTORIO
$ echo ${DIRECTORIO:-/tmp}
/tmp
$ echo $DIRECTORIO

$
```

En este ejemplo, si la variable `DIRECTORIO` *no se establece se utilizará como directorio predeterminado el directorio* `/tmp`.

Este tipo de expansión se suele utilizar para utilizar una serie de valores predeterminados si alguna variable no se declara.

Asignación de valores por omisión

`${`*parámetro*`:=`*palabra*`}` se expande al valor del *parámetro* si éste está establecido y no es nulo; en caso contrario, se establece a *palabra* y se expande. Este tipo de expansión se emplea igual que el anterior, pero en este caso nos interesa que la variable disponga de tal valor.

Ejemplo:

```
$ unset A
$ echo ${A:=abc}
abc
$ echo $A
abc
```

Mostrar un mensaje de error

`${`*parámetro*`:?`*palabra*`}` es expandido al valor del *parámetro* si éste está establecido y no es nulo; en caso contrario, muestra *palabra* en la salida de errores estándar y hace que termine la ejecución del *script*.

Ejemplo:

```
$ unset A
$ echo ${A:?Variable no establecida}
/bin/bash: A: Variable no establecida
```

Se emplea para comprobar que las variables necesarias para el funcionamiento de nuestro *script* están definidas.

Uso de un valor alternativo

`${`*parámetro*`:+`*palabra*`}` se expande a *palabra* si el parámetro está establecido y no es nulo; en caso contrario, no se sustituye nada.

Ejemplo:

```
$ Y=abc
$ echo ${Y:+def}
def
$ unset Y
$ echo ${Y:+def}

$
```

Este tipo se expansión se suele utilizar cuando queremos mostrar mensajes de qué acciones va realizando el *script*.

6.18. La orden [...]

Nos permite evaluar expresiones condicionales con atributos de ficheros, cadenas, enteros, etc.

Formato:

`[` *expresión* `]`

donde *expresión* es la condición que se va a evaluar. Deben dejarse blancos justo después de `[` y antes de `]`. Los argumentos de la expresión deben separarse de los operadores también mediante blancos. Si la expresión que se evalúa es verdadera, se devuelve un status de salida 0, y si es falsa, distinto de 0.

6.18.1. Comprobación de atributos de ficheros

En la tabla 6.1 se dan algunos de los operadores que se pueden utilizar para comprobar atributos de ficheros.

`-r` *fichero*	*fichero* tiene permiso de lectura.
`-w` *fichero*	*fichero* tiene permiso de escritura.
`-x` *fichero*	*fichero* tiene permiso de ejecución.
`-f` *fichero*	*fichero* es un fichero regular.
`-d` *fichero*	*fichero* es un directorio.
`-b` *fichero*	*fichero* es un fichero especial de bloques.
`-c` *fichero*	*fichero* es un fichero especial de caracteres.
`-s` *fichero*	El *fichero* no está vacío.
`-O` *fichero*	El *fichero* existe y es propiedad del UID efectivo del proceso actual.
fichero1 `-nt` *fichero2*	*fichero1* es más nuevo (ha sido modificado más recientemente) que *fichero2*.
fichero1 `-ot` *fichero2*	*fichero1* es más viejo que *fichero2*.

Cuadro 6.1: Operadores de ficheros que se pueden usar con `[...]`

Ejemplo:

```
[ -f tmp ] && echo "tmp existe y es regular"
```

Si `tmp` *es un fichero regular imprime el mensaje, si no existe no imprime nada*

6.18.2. Comprobación de cadenas

Podemos usar la orden `[...]` para comprobar si una variable está establecida a un determinado valor y si su longitud es o no cero.

En la tabla 6.2 se muestran los operadores de cadena usados más frecuentemente con `[...]`.

`-n` *cadena*	Verdad si la longitud de la cadena no es 0.
`-z` *cadena*	Verdad si la longitud de *cadena* es 0.
cadena1 `=` *cadena2*	Verdad si *cadena1* es igual a *cadena2*.
cadena1 `!=` *cadena2*	Verdad si *cadena1* es distinta de *cadena2*.

Cuadro 6.2: Operadores de cadena de la orden `[...]`

Ejemplo:

```
[ $X = abcd ] && echo "X vale abcd"
```

Si la variable `X` *vale* `abcd` *se imprimirá el mensaje; si no es así, no se imprimirá nada*

6.18.3. Comparación de enteros

La orden `[...]` proporciona una serie de operadores que nos permiten comparar enteros.

La tabla 6.3 muestra los operadores de enteros más comúnmente usados de la orden `[...]`. *exp1* y *exp2* pueden ser enteros positivos o negativos.

exp1 `-eq` *exp2*	Verdadero si *exp1* es igual a *exp2*.
exp1 `-ne` *exp2*	Verdadero si *exp1* no es igual a *exp2*.
exp1 `-le` *exp2*	Verdadero si *exp1* es menor o igual que *exp2*.
exp1 `-lt` *exp2*	Verdadero si *exp1* es menor que *exp2*.
exp1 `-ge` *exp2*	Verdadero si *exp1* es mayor o igual que *exp2*.
exp1 `-gt` *exp2*	Verdadero si *exp1* es mayor que *exp2*.

Cuadro 6.3: Operadores de enteros para la orden `[...]`

Ejemplos:

1. `[ $# -gt 2 ] && echo "Demasiados argumentos"`
   Si el número de parámetros posicionales que se han pasado al llamar al script es superior a 2, nos da el mensaje: `Demasiados argumentos`.

2. `[ $(who | wc -l) -gt 10 ] &&`
   `echo "Hay más de 10 usuarios en el sistema"`
   Obtenemos el número de usuarios conectados al sistema y si éste es superior a 10 nos muestra un mensaje.

6.18.4. Otros operadores de la orden `[...]`

La orden `[...]` también admite el uso del operador `!` (negación), y permite combinar expresiones mediante los operadores `-a` (Y lógico) y `-o` (O lógico).

Ejemplo:

```
[ -r $FICH -o -w $FICH ]
```

Mediante esta expresión comprobamos si `$FICH` *tiene activado el permiso de lectura o el de escritura.*

6.19. Órdenes de control de flujo

Al igual que los lenguajes de programación de alto nivel, el *shell* bash proporciona varias órdenes de control de flujo.

6.19.1. La construcción `if`

El tipo más simple de construcción de control de flujo es la condicional. Se usa cuando queremos elegir hacer o no algo, o elegir entre un pequeño número de acciones, dependiendo de que se cumplan o no una serie de condiciones.

La construcción `if` tiene la siguiente sintaxis:

```
if orden1
then
    órdenes
[ elif orden2
then órdenes-1 ... ]
[ else
    órdenes-2 ]
fi
```

Podemos usar la construcción `if` de varias formas:

1. La forma más simple (sin las claúsulas `elif` ni `else`) ejecuta las *órdenes* indicadas si al ejecutar *orden1* obtiene un status de salida 0.

   Ejemplo:

   ```
   if [ ! -f $1 ]
   then
       echo "El fichero $1 no existe"
   fi
   ```

   Si el fichero que se ha pasado como primer parámetro posicional existe, no hace nada; si no existe, muestra un mensaje.

2. Otra forma de `if` se utiliza para ejecutar una serie de órdenes si el status de salida de la condición es 0, y otras órdenes si es distinto de 0.

   Ejemplo:

   ```
   if [ ! -f $1 ]
   then
       echo "El fichero $1 no existe"
   else
       echo "El fichero $1 existe"
   fi
   ```

   En este caso, tanto si el fichero existe como si no, muestra el mensaje correspondiente.

3. También puede comprobar con `if` si una serie de condiciones son verdaderas. Dependiendo de cual sea la verdadera, se ejecutará el conjunto de órdenes asociadas a la misma.

   Ejemplo:

   ```
   if [ ! -f $1 ]
   then
       echo "El fichero $1 no existe"
   elif [ ! -r $1 ]
   then
       echo "El fichero $1 no tiene permiso de lectura"
   elif [ ! -w $1 ]
   then
       echo "El fichero $1 no tiene permiso de escritura"
   elif [ ! -x $1 ]
   then
       echo "El fichero $1 no tiene permiso de ejecución"
   ```

```
else
    echo "El fichero $1 existe y tiene todos los permisos"
fi
```

Se comprueba si no existe el fichero y si no posee algún permiso, mostrando el mensaje correspondiente.

Hay que hacer notar que la percepción de verdad o falsedad es al revés que en el lenguaje de programación C.

6.19.2. La construcción `case`

Se utiliza para comparar un valor simple frente a un cierto número de otros valores. Las órdenes asociadas con ese valor se ejecutan cuando hay una concordancia. La sintaxis de `case` es:

```
case valor in
patrón1)  orden
          ...
          orden;;
patrón2)  orden
          ...
          orden;;
...
patrónN)  orden
          ...
          orden;;
esac
```

donde *valor* se compara con *patrón1*, *patrón2*, ... Cuando concuerda con uno de ellos se ejecutan las órdenes asociadas a ese patrón. La lista de órdenes asociada a cada patrón debe terminar con `;;`. Una vez que el *shell* ha encontrado una concordancia entre *valor* y uno de los patrones, ejecuta las órdenes asociadas y se salta el resto de la sentencia (al contrario del `switch` de C, que necesita la órden `break` para ir al final del bloque).

Ejemplo:

Si ejecutamos el script 6.1, dependiendo del valor introducido como primer parámetro posicional, se realizará la acción correspondiente. Si se introduce algo distinto de `-a`, `-b` *o* `-c`, *nos dirá que la opción introducida no es correcta.*

***Script* 6.1** Utilización de la orden `case`

```
#!/bin/bash

# Ejemplo: Utilización de la orden case

case $1 in
    -a) echo "Has introducido la opción -a";;
    -b) echo "Has introducido la opción -b";;
    -c) echo "Has introducido la opción -c";;
     *) echo "No has introducido una opción correcta";;
esac
```

6.20. Bucles

6.20.1. El bucle `for`

Se utiliza para ejecutar un conjunto de órdenes un número especificado de veces. Su sintaxis es:

```
for variable [in palabra1 palabra2 ...]
do
    órdenes
done
```

donde *variable* va tomando los valores *palabra1*, *palabra2*, etc., y para cada uno de estos valores se ejecutan las *órdenes*.

Ejemplo:

Al ejecutar el *script* 6.2 obtendremos:

```
Ciclo 1: X=rojo
Ciclo 2: X=amarillo
Ciclo 3: X=azul
```

***Script* 6.2** Utilización de la orden `for`

```
#!/bin/bash

# Ejemplo: Utilización de la orden for

declare -i NUMCICLO=1

for X in rojo amarillo azul
do
   echo "Ciclo $NUMCICLO: X=$X"
   let "NUMCICLO+=1"
done
```

También se pueden utilizar la sustitución de nombres de ficheros, la sustitución de órdenes y la sustitución de variables para generar una lista de palabras para la orden `for`.

Ejemplos:

1. `for FICHERO in ejer[1-3]`
2. `for FICHERO in $(ls ejer[1-3])`
3. `EJERCICIOS=$(ls ejer[1-3])`
   `for FICHERO in $EJERCICIOS`

Si existen los ficheros `ejer1`, `ejer2` *y* `ejer3`, *la variable* `FICHERO` *tomará en cada ciclo uno de los tres valores.*

También podemos utilizar la construcción `for` sin especificar una lista de palabras.

```
for variable
do
    órdenes
done
```

en cuyo caso, es equivalente a:

```
for variable in "$@"
do
    órdenes
done
```

6.20.2. Los bucles `while` y `until`

La sintaxis de estas órdenes es la siguiente:

```
while orden1
do
    órdenes
done
```

Si el status de salida obtenido después de ejecutar *orden1* es 0, se ejecutarán las órdenes que aparecen entre **do** y **done**. Esto se hará así hasta que el status de salida de *orden1* deje de ser 0.

```
until orden1
do
    órdenes
done
```

La orden `until` es como `while` excepto en que se ejecuta hasta que la condición sea verdadera.

Ejemplos:

1.
```
while [ $# != 0 ]
do
    echo $1
    shift
done
```
Si al ejecutar el script le pasamos varios parámetros posicionales, nos imprimirá los valores de todos ellos.

2.
```
until [ $# == 0 ]
do
    echo $1
    shift
done
```
Hace lo mismo que el ejemplo anterior. Observe que se ha cambiado la condición a evaluar.

6.21. Ejercicios

1. Escriba un *script* que reciba un fichero (puede ser con una ruta). Si el nombre de dicho fichero no comienza por punto cámbie su nombre para que así sea. Y si lo pueden leerlo, modificarlo o borrarlo intente cambiar los permisos para que no sea posible.

2. Escriba un *script* que se llame `quien`. Si se invoca sin parámetros funcionará como la orden `who`. Si recibe un parámetro mostrará las líneas del comando `who` correspondientes a él. Realice este ejercicio sin el operadores de comparación.

3. Escriba un *script* que permita que le pasen cuatro parámetros posicionales como máximo. Los parámetros que se le pasan deben ser nombres de ficheros y se debe comprobar si estos ficheros existen y si es así, si están vacíos o no. Como resultado debe dar la información correspondiente acerca de cada fichero.

4. Escriba un *script* que se llame `mata` y que mate un proceso por el nombre. Si no se introduce el nombre como argumento, debe pedirlo. Si existen más de un proceso con el mismo nombre debe matar a todos.

5. Escriba un *script* llamado `busca-multi` que admita uno o más parámetros. El script buscará, con `find`, los ficheros que tienen los nombres indicados a partir del directorio donde se invoque. En caso de que no se reciba ningún parámetro informará de su sintaxis y terminará.

6. Escriba un *script* de nombre `permisos` que reciba diez parámetros. Los primeros nueve serán podrán valer r, w, x o - y el décimo será un fichero. La orden podrá los permisos dados al fichero en concreto. Sólo se podrá usar la orden `chmod` una vez, en modo simbólico. En caso de que el número de parámetros no sea el adecuado informe de la sintaxis. Si algún permiso no es correcto o no se puede cambiar los permisos del fichero, informe del error y termine, pero evite que el sistema genere mensajes de error que lleguen al usuario.

7. Escriba un *script* llamado `permisos2` que funcione igual que el anterior pero use `chmod` con números octales.

8. Escriba un *script* que se llame `fecha` y que admita una fecha como parámetro con el formato `dd-mm-aa` y compruebe si es correcta. Si no lo es, debe dar los mensajes correspondientes. Si no se le introduce la fecha como parámetro debe pedirla.

9. Cree un fichero que tenga una columna de palabras. Una vez hecho, escriba un *script* con el siguiente formato:

`palabras -l | -w` *expresion_regular*

- La opción `-l` hace que se genere un fichero que contiene la lista de palabras junto con el número de caracteres de cada una, separados por el carácter `-`.
- La opción `-w` hace que se genere un fichero con aquellas palabras que concuerden con la expresión regular dada.

10. Escriba un *script* que se llame `cuenta` que simule el comportamiento de la orden `wc`. Debe aceptar múltiples ficheros y si no se especifica ninguno debe leer de la entrada estándar.

11. Cree un *script* llamado `thumb` que cree imágenes a tamaño 100 por 100 puntos de todos los parámetros que reciba. En concreto, por cada fichero de nombre `foto.jpg` creará uno llamado `foto_thumb.jpg`. Para ello use el programa `convert` (incluido en el paquete `ImageMagick`) que tiene la siguiente sintaxis (para redimensionar a 100 por 100):

convert -resize 100x100 <origen> <destino>

Capítulo 6

Programación en el *shell* bash Parte IV

6.22. Órdenes de control de flujo 2

6.22.1. La orden `break`

Usted puede querer salir de un bucle antes de que se cumpla la condición; para esto, puede usar la orden **`break`**, que nos permite salir de un bucle sin abandonar el *script*. Una vez hemos salido del bucle, la ejecución continúa en la instrucción que sigue a éste.

Ejemplo:

Al ejecutar el script *6.1 se le dará un argumento, que será el directorio que se desea borrar. Antes de realizar la acción, pide confirmación.*

Si contestamos afirmativamente, se borrará el directorio y nos saldremos del bucle; si contestamos de forma negativa el directorio no se borrará y nos saldremos también; si damos una respuesta incorrecta nos dará un mensaje y nos volverá a preguntar.

La orden `:` *no hace nada, se utiliza para que el bucle se ejecute infinitas veces.*

***Script* 6.1** Salir de un bucle

```
#!/bin/bash

# Ejemplo de la orden break

dir=$1

while :
do
  echo "Quieres borrar el directorio $dir?"
  read respuesta
  case $respuesta in
  [sS]*)  rmdir $dir; break;;
  [nN]*)  echo "No borro $dir"; break;;
  *)      echo "Por favor conteste sí o no" ;;
  esac
done
```

La orden `break` también puede ser usada para salir de un bucle anidado usando el formato:

`break` n

donde n especifica que queremos salir del n-ésimo bucle.

Ejemplo:

Observe el código del *script* 6.2. ¿Qué salida se obtendría al ejecutarlo?

***Script* 6.2** Salir de un bucle

```
#!/bin/bash

# Ejemplo: Utilización de la orden break

for i in 1 2 3
do
  for j in 0 5
  do
    if [ $i -eq 2 -a $j -eq 0 ]
    then
      break 2
    else
      echo "$i$j"
    fi
  done
done
```

6.22.2. La orden `continue`

La orden `continue` hace que el *shell* abandone el resto del bucle actual y empiece inmediatamente la próxima iteración. Es parecida a la orden `break`, salvo que en vez de salir completamente del bucle, sólo se salta las órdenes restantes en el ciclo actual.

Formato:

`continue` [n]

Esta orden admite un argumento opcional n que, si se especifica, debe ser un número entero (o una expresión del *shell* que se evalúe a un entero). El valor de n indica qué bucle es el que debe reiniciarse cuando `continue` está inmerso en bucles anidados. Por omisión, se considera que n vale 1.

Cuando n es mayor que 1, se dan por terminados n-1 bucles como si se hubiera dado un `break` n-1. Sólo el bucle más externo designado por n se reiniciará en la próxima iteración.

Ejemplo:

El script 6.3 puede ser útil para un administrador de sistema, ya que le permite crear los directorios de casa de los usuarios listados

en `/etc/passwd` *que todavía no lo tengan, así como copiar los ficheros de arranque del shell.*

Si el directorio de casa del usuario ya existe, la orden `mkdir` *fallará y se ejecutará la orden* `continue` *haciendo que se ejecute una nueva iteración* `while`.

Si falla la orden `cp` *se reinicia un nuevo ciclo* `while` *ya que se ha especificado un valor para n igual a 2.*

***Script* 6.3** Uso de la orden continue

```
#!/bin/bash

# Ejemplo: Uso de la orden continue

IFS=:
while read login pass uid gid nombre home shell
do
  mkdir $home || continue
  for FICHERO in .profile .bash_logout .bashrc
  do
    if cp /usr/skel/$FICHERO $home
    then :
    else
      echo "No puedo crear $home/$FICHERO ($login)"
      continue 2
  done
done < /etc/passwd
```

6.22.3. La orden `exit`

El formato de esta orden es el siguiente:

Formato:

`exit` [*n*]

Hace que el *shell* o un *script* terminen con un status *n*. Si omitimos *n* el status de salida es el de la última orden ejecutada.

Ejemplo:

El *script* 6.4 posee un ejemplo de utilización del mismo.

***Script* 6.4** Uso de la orden `exit`

```
#!/bin/bash

# Ejemplo de la orden exit

case $# in
0)  echo "Uso: $0 fichero ..."
    exit;;
*)  ;;
esac
```

6.22.4. La orden `select`

Se utiliza para mostrar un menú simple que contiene ítems numerados y un mensaje. Su sintaxis es la siguiente:

Formato:

```
select variable [in palabra1 ... palabraN]
do
  órdenes
done
```

donde *palabra1* hasta *palabraN* se muestran en la salida de errores estándar como elementos de un menú y seguidos de un indicador (por omisión `#?`). Si la respuesta está en el rango de 1 a *N*, entonces *variable* toma como valor la palabra correspondiente y se ejecutan las *órdenes*. La variable REPLY toma como valor la respuesta introducida. ¿Qué ocurre si nuestra respuesta no está dentro del rango válido? Se nos vuelve a mostrar el indicador hasta que introduzcamos una respuesta válida.

Si queremos utilizar un indicador diferente de `#?` tendremos que definir previamente la variable PS3. En el *script* 6.5 tenemos un ejemplo de utilización.

***Script* 6.5** Ejemplo de la orden `select`

```
#!/bin/bash

# Ejemplo de utilización de la orden select

PS3="Introduzca su selección: "

select ORDEN in "Nombre del directorio actual" \
"Listar ficheros" "Usuarios conectados" Salir
do
  case $ORDEN in
  Nombre*)  pwd;;
  Listar*)
          ( PS3="Directorio a listar: "
            select DIR in casa ejercicios actual salir
            do
              case $DIR in
              casa) ls $HOME;;
              ejercicios) ls $HOME/ejercicios;;
              actual) ls;;
              salir) break;;
              *) echo "No ha elegido una opción válida"
                 break;;
              esac
            done );;
  Usuarios*) who;;
  Salir) break;;
  *) echo "No ha elegido una opción válida";;
  esac
done
```

Un uso muy común de `select` es para que el usuario seleccione ficheros, como `select FOTO in *.jpg` o `SELECT FICH in $(find ...)`.

Si no especificamos la lista de *palabras*, los elementos del menú serán los parámetros posicionales.

6.23. La orden `eval`

Formato:

`eval` [*arg1* ... *argN*]

El *shell* trabaja en tres pasos:

1. Expande los argumentos *arg1* hasta *argN*.
2. Concatena los resultados del paso 1 en una orden simple.
3. Ejecuta la orden formada en el paso 2.

Ejemplos:

1. Vamos a suponer que tenemos un *script* llamado `eval_test` al que se le pasan varios parámetros posicionales, y en un momento determinado necesitamos conocer el valor del último parámetro posicional. Un fragmento de dicho *script* podría ser el siguiente:

   ```
   echo "Número total de parámetros posicionales $#"
   echo "Valor del último parámetro posicional $$#"
   ```

   Cuando ejecutamos `eval_test` obtenemos:

   ```
   $ eval_test a b c
   Número total de parámetros posicionales 3
   Valor del último parámetro posicional 224#
   ```

   ¿Qué ha ocurrido? Al parecer no ha funcionado tal como queríamos. Lo que ha pasado es lo siguiente: el shell ha expandido `$$` *en primer lugar, y esto es el identificador del proceso actual, que ha resultado ser 224, y a esto le ha añadido el signo* `#`.

2. Para que hubiera funcionado tal como queríamos tendríamos que haber puesto:

   ```
   echo "Número total de parámetros posicionales $#"
   echo "Valor del último parámetro posicional \
   $(eval echo \$$#)"
   ```

   Si ejecutamos ahora `eval_test`:

   ```
   $ eval_test a b c
   Número total de parámetros posicionales 3
   Valor del último parámetro posicional c
   ```

Se necesita poner el carácter \ para que `$` *sea ignorado en la primera expansión. Después de la primera expansión,* `$(eval echo \$$#)` *se convierte en* `$(echo $3)`. *Y esto a su vez es expandido a* `c`.

3. Supongamos que tenemos un fichero llamado `texto`.

```
$ cat texto
Éste es el contenido del fichero texto
```

Hacemos ahora lo siguiente:

```
$ X="<texto"
$ cat $X
<texto: No such file or directory
```

Sin embargo si usamos la orden `eval`:

```
$ eval cat $X
Éste es el contenido del fichero texto
```

6.24. La orden `getopts`

Una **opción** es un argumento de la línea de órdenes que empieza con `+` o `-` y es seguido por un carácter. La orden `getopts` se utiliza para analizar opciones. Normalmente `getopts` se utiliza como parte de un bucle `while`. El cuerpo del bucle `while` suele contener una sentencia `case`. Es realmente la combinación de las tres sentencias `getopts`, `while`, y `case` la que proporciona la forma de analizar opciones.

Vamos a ir viendo a través de ejemplos las posibilidades de esta orden.

***Script* 6.6** Tratamiento de opciones básicas

```
#!/bin/bash

# Ejemplo de script con dos opciones

while getopts xy argumentos
do
  case $argumentos in
  x)  echo "Has introducido la opción -x";;
  y)  echo "Has introducido la opción -y";;
  esac
done
```

Ejemplos:

1. Tenemos un fichero llamado `getopts1.bash`, cuyo contenido es el que aparece en el *script* 6.6. Vamos a ejecutarlo:

```
$ getopts1.bash -x
Has introducido la opción -x

$ getopts1.bash -y
Has introducido la opción -y

$ getopts1.bash -x -y
Has introducido la opción -x
Has introducido la opción -y
$ getopts1.bash -xy
Has introducido la opción -x
Has introducido la opción -y
```

   Si introducimos una opción errónea:

```
$ getopts1.bash -t -y
getopts1.bash: illegal option -- t
Has introducido la opción -y
```

2. El ejemplo previo trataba de una forma no demasiado elegante el caso de que el usuario introdujera una opción errónea. Vamos a modificar el *script* para que esto no ocurra. Las modificaciones se pueden ver en el *script* 6.7.

***Script* 6.7** Tratamiento de opciones incorrectas

```
#!/bin/bash

# Ejemplo de script con opciones y tratamiento de opciones
# incorrectas

while getopts :xy argumentos
do
  case $argumentos in
  x)  echo "Ha introducido la opción -x";;
  y)  echo "Ha introducido la opción -y";;
  \?)  echo "$OPTARG no es una opción válida";;
  esac
done
```

Hemos introducido el carácter : al principio, lo que dice a

`getopts` *que le dé a* `argumentos` *el valor ? si el usuario especifica una opción distinta de* `x` *o* `y`.
Además establece el valor de la variable del shell OPTARG como el nombre de la opción no definida.
Si ejecutamos el nuevo *script*:

```
$ getopts2.bash -k -x
k no es una opción válida
Has introducido la opción -x

$ getopts2.bash -x -k
Has introducido la opción -x
k no es una opción válida
```

6.24.1. Opciones con argumentos

Un argumento de una opción es una palabra o número que sigue a la opción. Para decirle a `getopts` que una opción requiere un argumento, se coloca el carácter `:` después del nombre de ésta. Cuando se ejecuta el *script*, el *shell* asignará el argumento que sigue a la opción a la variable OPTARG.

***Script* 6.8** Tratamiento de opciones con argumentos

```
#!/bin/bash

# Ejemplo de script con opciones con argumentos

USO="uso: $0 [-x número] [-y número]"
while getopts :x:y: argumentos
do
  case $argumentos in
  x)   echo "Has introducido la opción x"
       arg_de_x=$OPTARG
       echo "argumento de x: $arg_de_x";;
  y)   echo "Has introducido -y como opción"
       arg_de_y=$OPTARG
       echo "argumento de y: $arg_de_y";;
  \?)  echo "$OPTARG no es una opción válida"
       echo "$USO";;
  esac
done
```

Ejemplo:

Al ejecutar el *script* 6.8:

```
$ getopts4.bash -x 1024 -y 800
Has introducido -x como opción
Has introducido 1024 como argumento de x
Has introducido -y como opción
Has introducido 800 como argumento de y
```

¿Qué ocurre si se olvida dar un argumento a una opción? Ya hemos visto que el carácter `:` al principio de la lista de opciones ayuda a tratar con las opciones no válidas; además de esto, hace lo siguiente:

- Establece el valor de `argumentos` a `:` si el usuario olvida especificar el argumento.
- Establece el valor de OPTARG al nombre de la opción a la cual se ha olvidado dar su argumento.

***Script* 6.9** Tratamiento de errores con opciones

```
#!/bin/bash

# Ejemplo de tratamiento de opciones con argumentos
# y tratamiento de posibles errores

USO="uso: $0 [-y número]"

while getopts :y: argumentos
do
  case $argumentos in
  y)    altura=$OPTARG;;
  # Si el usuario olvida especificar el argumento de y,
  # el shell asigna : a argumentos
  :)    echo "No has dado un arg. para $OPTARG";;
  \?)   echo "$OPTARG no es una opción válida"
        echo "$USO";;
  esac
done
```

Ejemplo:

Al ejecutar el *script* 6.9 se obtiene:

```
$ getopts5.bash -y
No has dado un arg. para y
```

6.24.2. Análisis de líneas más complejas

La orden `getopts` también nos permite analizar líneas que contengan opciones, argumentos de opciones, y argumentos no asociados a opciones. Consideremos la siguiente línea de órdenes:

```
$ getopts6.bash -x 1024 -y 800 rojo verde azul
```

La línea de órdenes precedente contiene 2 opciones (`-x` y `-y`) y dos argumentos de opciones (`1024` y `800`). Además contiene tres valores no asociados con ninguna opción (`rojo`, `verde`, y `azul`). En una línea de órdenes mixta como ésta, el usuario debe especificar las opciones y los argumentos de éstas al principio de la línea; es decir, justo detrás del nombre del *script*.

La variable del *shell* bash OPTIND almacena el índice del argumento de la línea de órdenes que `getopts` va a evaluar actualmente. Es decir, cuando `getopts` está evaluando el primer argumento de la línea de órdenes (considerando que es una opción), el valor de OPTIND será 1.

***Script* 6.10** Líneas de órdenes complejas

```
#!/bin/bash

# Ejemplo de script con opciones con argumentos
# y argumentos simples

USO="uso: getopts6.bash [-x ancho] [-y alto] \
[color1 ... colorN]"

while getopts :x:y: argumentos
do
  case $argumentos in
  x)  ancho=$OPTARG;;
  y)  alto=$OPTARG;;
  \?)  echo "$OPTARG no es una opción válida"
       echo "$USO";;
  esac
done
posiciones_ocupadas_por_opciones=$(( OPTIND - 1))

shift $posiciones_ocupadas_por_opciones

echo "Ancho: $ancho"
echo "Alto: $alto"
echo "Colores: $*"
```

Ejemplo:

Al ejecutar el *script* 6.10:

```
$ getopts6.bash -x 1024 -y 800 rojo verde
Ancho: 1024
Alto: 800
Colores: rojo verde
$ getopts6.bash -y 2048 amarillo azul plata
Ancho:
Alto: 2048
Colores: amarillo azul plata
```

6.25. Ejercicios

1. Escriba un *script* que muestre un menú con todos los directorios que hay por debajo de su directorio de entrada (incluidos los que se encuentran en subdirectorios). Al seleccionar uno de ellos, tiene que averiguar si el directorio seleccionado tiene activado el permiso de escritura para el grupo y para los otros. En cualquier caso debe dar el mensaje correspondiente.

2. Escribir un *script* que muestre un menú con los nombres de *login* de los usuarios conectados y al elegir uno se nos muestre el nombre completo de dicho usuario.

3. Escriba un *script* que trabaje sobre un fichero que contenga una relación de nombres con su correspondiente dirección internet, con el siguiente formato:

   ```
   nombre:DireccionInternet1
   ```

   El formato del *script* es el siguiente:

   `agenda-electronica [ -c | -l | -a ] [`*fichero*`]`

   `-c` Pide un nombre y muestra la dirección.

   `-l` Lista todo el fichero.

   `-a` Añade un nuevo registro.

   En caso de no darle ninguna opción, debe mostrar un menú con las operaciones anteriores. Si no se le proporciona un nombre de *fichero*, lo debe pedir al principio. El fichero debe quedar siempre ordenado después de cualquier operación. Se debe controlar e informar de todos los errores posibles.

4. Cree un *script* llamado `fotocopia`. Este *script* facilitará el manejo de fotos con el programa `convert`. Admite las siguientes opciones:

   `-b` hace una copia en blanco y negro de la foto (o fotos) que recibe como parámetro. Por cada foto de nombre `foto.jpg` crea una llamada `foto_BN.jpg`.

   `-n` <num> hace *num* copias de las fotos que reciba como parámetro. Por cada foto de nombre `foto.jpg` crea copias llamadas `foto_1.jpg`, `foto_2.jpg`, ..., `foto_num.jpg`.