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.