Programación en SHELL
(Bash)

¿ Que es una shell ?

El **shell** de Unix o también **shell**, es el termino usado en informática para referirse a un interprete de comandos. Los comandos que aportan los intérpretes, pueden usarse a modo de guión si se escriben en ficheros ejecutables denominados **shell-scripts**, de este modo, cuando el usuario necesita hacer uso de varios comandos o combinados de comandos con herramientas, escribe en un fichero de texto marcado como ejecutable, las operaciones que posteriormente, línea por línea, el intérprete traducirá al núcleo para que las realice.
Sin ser un **shell** estrictamente un lenguaje de programación, al proceso de crear **scripts** de **shell** se le denomina programación **shell** o en inglés, ***shell programming*** o ***shell scripting***.
Los usuarios de **Unix** y similares, pueden elegir entre distintos **shells** (programa que se debería ejecutar cuando inician la sesión, bash, ash, csh, zsh, ksh, tcsh).

El término **shell** también hace referencia a un programa particular, tal como el **Bourne shell**, **sh**. El **Bourne shell** fue el **shell** usado en las primeras versiones de **Unix** y se convirtió en un estándar; todos los sistemas similares a **Unix** tienen al menos un **shell** compatible con el **Bourne shell**. El programa **Bourne shell** se encuentra dentro de la jerarquía de archivos de **Unix** en **/bin/sh**. En algunos sistemas, tal como **BSD**, **/bin/sh** es un **Bourne shell** o un equivalente, pero en otros sistemas como muchas distribuciones de **Linux**, **/bin/sh** es un enlace simbólico a un **shell** compatible con más características (como **Bash**). **POSIX** especifica su **shell** estándar como un subconjunto estricto del **Korn shell**.

Primeros pasos

Uno de los primeros pasos es saber que **shell** (interprete de comando estamos usando).

```
$ echo $SHELL
/bin/bash
```

Las variables por convención se utiliza en mayúsculas y las forma de obtener el contenido de la variable o mejor dicho su valor es poniendo el símbolo **$**.

Si queremos saber la versión de nuestro interprete de comandos en este caso **bash** ejecutamos el siguiente comando :

```
$ echo $BASH_VERSION
4.1.5(1)-release
```

Para conocer todos los interpretes de comandos que tenemos veremos que se guardan dentro de un archivo llamado **shells**.

```
$ cat /etc/shells
/bin/csh
/bin/sh
/bin/ksh
/bin/tcsh
/bin/dash
/bin/bash
```

Si queremos cambiar nuestro **shell** por defecto a **bash** ejecutamos el siguiente comando y luego volvemos a loguearno para que tome la nueva configuración.

$ chsh -s /bin/bash

Manejo de comodines

Los comodines de búsqueda son signos o símbolos que nos ayudarán a la hora de encontrar algún archivo en nuestro ordenador. Donde mas lo podemos llegar a utilizar es con el comando **ls**.

$ ls /bin/bash

Acá tenemos una tabla con los distintos comodines y su explicación.

Comodín	**Descripción**
?	Uno o sólo un carácter.
*	Cero o más caracteres.
[conjunto]	Uno los caracteres de *conjunto*.
[!conjunto]	Un carácter que no este en *conjunto*.

Ejemplo de usos :

$ ls *.txt

En el ejemplo anterior me trae cualquier archivo que tenga como extensión **txt**.

$ ls /bin/l?
/bin/ln /bin/ls

El tercer comodín permite indicar un conjunto de caracteres que son válidos. Por ejemplo **l[eno]*** encontrara el fichero **less lesssecho lessfile lesskey lesspipe ln loadkeys login** no traira **liss**. Es decir como primer letra si o si **l** a continuación puede ser **e**, **n** o **o** y luego como bien dijimos el comodín (*) representa cero o más caracteres realiza una distributiva.

$ ls /bin/l[eno]*
/bin/less /bin/lessecho /bin/lessfile /bin/lesskey /bin/lesspipe /bin/ln /bin/loadkeys /bin/login

El comodín tilde

Este comodín es ~ que se utiliza para referirse al **home** de los usuarios, por ejemplo **~pablo** se refiere a **/home/pablo**.

$ ls -a ~pablo
./ ../ .bash_logout .bash_profile .bashrc

El comodín llaves

Este comodín a comparación de los otros, no estudia el nombre de los ficheros existentes en disco para nada, simplemente expande una palabra por cada una de las cadenas de caracteres que contiene, por ejemplo :

$ echo c{ami,ontamina,}on
camion cantaminaon con

Lo que realiza es una distribución **c->ami->on**, **c->ontaminaon** y **c->con**, al poner **,}** lo que hace es poder formar la palabra **con** es decir como luego de la **,** no hay nada puede formar otra palabra con **on**.

Veamos un ejemplo mas complejo que añade mas llaves :

$ echo c{a{mi,nt}a,ose}r
caminar cantar coser

Siempre combina **c->a** y luego con **{mi,nt}** luego con **a** y la **r** y por ultimo combina **c->ose->r**

Se puede usar dos puntos (**..**) para hacer algo similar a lo que hacen los corchetes es decir indicar un rango. Por ejemplo :

$ echo l{a..j}
la lb lc ld le lf lg lh li lj

Las llaves deben tener al menos dos cadenas, el siguiente ejemplo vemos que esto no sucede.

$ echo ca{a}sa
ca{a}sa

Comodines extendidos

Bash permite usar un conjunto de comodines extendidos, pero para poder usarlos debemos de activar la opción ***ext_glob*** de **Bash**.

Para fijar estos valores usamos el comando ***shopt*** (shell option), si ejecutamos el comando ***shopt*** veremos la lista de variables que podemos modificar.

$ shopt
autocd off
cdable_vars off
cdspell off
checkhash off
checkjobs off
checkwinsize on
cmdhist on
compat31 off
compat32 off
compat40 off
dirspell off
dotglob off
execfail off
expand_aliases on
extdebug off
extglob on
extquote on
failglob off
force_fignore on
globstar off
gnu_errfmt off

histappend on
histreedit off
histverify off
hostcomplete off
huponexit off
interactive_comments on
lithist off
login_shell off
mailwarn off
no_empty_cmd_completion off
nocaseglob off
nocasematch off
nullglob off
progcomp on
promptvars on
restricted_shell off
shift_verbose off
sourcepath on
xpg_echo off

Todas estas variables son booleanas que contienen el valor ***on***/***off***, para activarlo usamos el comando ***shopt -s opcion*** y para desactivarlo ***shopt -u opcion***.

Vemos en la tabla algunas de esa opciones con mayor detalle de lo que hace :

Opción	Descripción
cdable_vars	Permite que ***cd*** use los valores de las variables como nombres de directorios.
cdspell	Ignora pequeños errores en los cambios de directorio con ***cd***. Sólo en la ejecución interactiva.
cmdhist	Guarda los comandos que hemos escrito en varias líneas en una sola línea del historial.
dotglob	Incluye en la expansión de comodines los ficheros que empiezan por (**.**).
expand_aliases	Expande un alias cuando lo ejecuta.
extglob	Utiliza extensiones de los comodines.
failglob	Si falla la expansión de un comodín porque no encuentra nada falla el comando (como hace el ***C shell***).
force_fignore	Los sufijos especificados en la variable de entorno ***FIGNORE*** no se usan para completar palabras con tabulador.
hostcomplete	Se intenta completar nombres de host al pulsar tabulador cuando la palabra contiene una **@**.
interactive_comments	Permite que dentro de un comando de sesión interactiva haya comentarios (precedidos por **#**).
login_shell	Variable de sólo lectura que indica si Bash ha sido lanzado como un shell de login.
nocaseglob	Indica si los comodines expanden sin sensibilidad a mayúsculas/minúsculas. No confundir con la variable ***completion-ignore-case*** de ***inputrc*** que lo que hacía era expandir con tabuladores.
nullglob	Hace que cuando un patrón no encuentra ficheros, se expandan por la cadena vacía en vez de por el patrón sin expandir.
sourcepath	Hace que el comando interno ***source*** busque el argumento en los directorios que indique ***PATH***.

Ejemplo :

$ export DOC=/usr/share/doc
$ shopt -s cdable_vars
$ cd DOC

La opción ***cdable_vars*** permite que el comando ***cd*** pueda utilizar variables como nombres de directorios. Observamos que ***DOC*** al utilizarlo con el comando ***cd*** no va precedido por ***$***.

$ shopt -s extglob

Con esta opción podemos utilizar uno de estos cinco nuevos tipos de patrones:

?(pattern-list)	Cero o una ocurrencia de ***pattern-list***.
*(pattern-list)	Cero o más ocurrencias de ***pattern-list***.
+(pattern-list)	Una o más ocurrencias de ***pattern-list***.
@(pattern-list)	Exactamente uno de los patrones de la lista.
!(pattern-list)	Cualquier cosa excepto uno de los patrones de la lista.

$ ls -l /bin/l?
-rwxr-xr-x 1 root root 48920 abr 28 2010 /bin/ln
-rwxr-xr-x 1 root root 108008 abr 28 2010 /bin/ls

Ejemplo donde solo listo que coincida con ***0*** o ***9***.

$ touch carta{0..9}.txt
$ ls carta+([0..9]).txt
carta0.txt carta9.txt

Ejemplo donde listo el rango de ***0*** a ***9***.

$ touch carta{0..9}.txt
$ ls carta+([0-9]).txt
carta0.txt carta1.txt carta2.txt carta3.txt carta4.txt carta5.txt carta6.txt carta7.txt carta8.txt carta9.txt

Si queremos buscar los archivos que solo tengan extensión ***.c***, ***.h*** y ***.o*** realizamos lo siguiente :

$ touch ejemplo{c,h,o,cpp}.txt
$ ls *.[cho]
ejemplo.c ejemplo.h ejemplo.o

Vemos que en el ejemplo anterior solamente me trae las extensiones que contengan solo un carácter (****.c***, ****.h*** y ****.o***), el corchetes ***[]*** sólo permite acepta un carácter. Si quiero traer los que tengan extensión ***.cpp*** no se podrá. En cambio con el siguiente ejemplo si podemos :

$ touch ejemplo{c,h,o,cpp}.txt
$ ls *.@(c|o|h|cpp)
ejemplo.c ejemplo.cpp ejemplo.h ejemplo.o

Con el ***pipe*** (|) realiza un **or** (o) que coincida con algunos de los patrones correspondientes lo que me permite añadir patrones.

El ejemplo anterior es equivalente a :

$ ls @(*.c|*.o|*.h|*.cpp)
ejemplo.c ejemplo.cpp ejemplo.h ejemplo.o

Si quiero listar lo que no sea con extensión ***.c***, ***.o***, ***.h*** y ***.cpp*** realizo la muestra con el patrón ***!***

$ ls !(*.c|*.o|*.h|*.cpp)
carta0.txt carta1.txt carta2.txt carta3.txt carta4.txt carta5.txt carta6.txt carta7.txt carta8.txt carta9.txt

Si quiero borrar todos los archivos excepto los que empiezan con **papel[0-9].txt** realizo lo siguiente :

$ touch carta{0..9}.txt
$ touch papel{0..9}.txt
$ rm -v !(papel+([0-9])).txt
«carta0.txt» borrado
«carta1.txt» borrado
«carta2.txt» borrado
«carta3.txt» borrado
«carta4.txt» borrado
«carta5.txt» borrado
«carta6.txt» borrado
«carta7.txt» borrado
«carta8.txt» borrado
«carta9.txt» borrado

Comandos internos de Bash

Bash siempre busca los programas que ejecutamos en los directorios indicados en la variable de entorno ***$PATH***, pero hay ciertos comandos internos de bash que no corresponde a ningún archivo del disco, algunos de estos comandos son ***cd***, ***chdir***, ***alias***, ***set*** o ***export***, *etc*.

Para saber los comandos que tenemos en bash ejecutamos ***help***.

$ help
$ help alias
alias: alias [-p] [name[=value] ...]
Define o muestra aliases.

`alias' sin argumentos muestra la lista de aliases en la forma reutilizable `alias NOMBRE=VALOR' en la salida estándar.

De otra manera, se define un alias por cada NOMBRE cuyo VALOR se proporcione. Un espacio final en VALOR causa que se revise la siguiente palabra para sustitución de alias cuando se expande el alias.

Opciones:
-p Muestra todos los aliases definidos en un formato reusable

Estado de salida:
alias devuelve verdadero a menos que se de un NOMBRE para el cual no se haya definido ningún alias.

Caracteres especiales y entrecomillado

Los caracteres <,>, |, ***&***, *******, ***?***, ~, ***[,]***, ***{, }*** son ejemplos de caracteres especiales de Bash que ya hemos visto en capítulos anteriores.

Carácter	***Descripción***
~	Directorio home.
`	Sustitución de comando.
#	Comentario.
$	Variable.
&	Proceso en background.
;	Separador de directorios.
*	Comodín **0** a **n** carácter.
?	Comodín de un sólo carácter.
/	Separdor de directorios.
(	Empezar un subshell.
)	Terminar un subshell.
\	Carácter de escape.
<	Redirigir la entrada.
>	Redirigir la salida.
|	Pipe.
[	Empieza conjunto de caracteres comodín.
]	Acaba conjunto de caracteres comodín.
{	Empieza un bloque de comando.
}	Acaba un bloque de comando.
'	Entrecomillado fuerte.
“	Entrecomillado débil.
!	No lógico de código de terminación.

Caracteres de escape

Otra forma de cambiar el significado de un carácter de escape es precederlo por \, que es lo que se llama el ***carácter de escape***.

Por ejemplo :

$ echo 2*\3\>5 es una expresion cierta
*2*3>5 es una expresion cierta*

Donde hemos puesto el carácter de escape a los caracteres especiales para que Bash no los interprete.

También lo usamos para que poder usar los espacios a los nombres de ficheros, ya que el espacio es interpretado por Bash como un separador de argumentos de la línea de comandos.

$ echo “Hola Mundo” > ejemplo\ espacio.txt

$ cat ejemplo\ espacio.txt

Entrecomillar los entrecomillados

Podemos usar el carácter de escape para que no se interpreten los entrecomillados simples o dobles, es decir:

$ echo \"2*\3\>5\" es una expresion cierta
*"2*3>5" es una expresion cierta*

$ echo \'2*\3\>5\' es una expresion cierta
*'2*3>5' es una expresion cierta*

Texto de varias líneas

Bash nos permite utilizar el carácter de escape para ignorar los retornos de carro de la siguiente forma :

**$ echo Esto es una prueba **
**> de varias lineas **
**> linea 1 **
> linea 2
Esto es una prueba de varias lineas linea 1 linea 2

Al poner el carácter \ y darle **enter** en la siguiente linea nos pone >, y nos permite seguir escribiendo.

Otro ejemplo es usando las comillas simples ' al principio y al final, en vez de usar \.

$ echo 'Esto es una prueba
> de varias lineas
> linea 1
> linea 2'
Esto es una prueba de varias lineas linea 1 linea 2

Ver historial de comandos

Para ver el historial de los comandos que ejecutamos habíamos visto que existe el comando ***history***, también encontramos el comando ***fc*** (fix command) es un clásico del ***C Shell*** traído al mundo de Bash, que al igual que el comando ***history*** lee el archivo ***.bash_history***.

Como dijimos anteriormente estos comandos internos de ***bash*** no contienen ***man*** sino la forma de ver ayuda es mediante el comando ***help***.

$ help fc

Podemos obtener un listado de los últimos comandos usados junto con su numero usado :

$ fc -l

Es equivalente al comando ***history*** excepto que solo el comando ***fc*** muestra sólo los últimos comandos del historial y no todo.

Para editar uno de ellos indicamos su número :

$ fc 20

Esto nos abrirá un editor que dependerá de la variable ***EDITOR*** cual nos habrá es decir si queremos usar el editor ***vi/vim*** exportamos la variable ***EDITOR***.

$ export EDITOR=vim

El problema de este comando que una vez que salimos del editor ejecuta la sentencia que modificamos.

Ejecutar comandos anteriores

Al igual que ***history*** podemos ejecutar comando anteriores o por el número de comando.

Comando	*Descripción*
!!	Ejecutar el último comando.
!n	Ejecutar el comando número ***n***.
!cadena	Ejecutar el último comando que empiece por ***cadena***.

Con la opción ***!cadena***, ya que con dar las primeras letras de un comando que hayamos ejecutado previamente lo busca en el historial y lo ejecuta, se ejecuta el comando más reciente que empiece por la ***cadena***.

Las teclas de control del terminal

Las teclas de control del terminal son combinaciones de teclas que interpreta el terminal (no Bash).

$ stty --all

Veremos que por ejemplo **^C** esto significa la tecla ***Ctrl+C***, veremos algunas que son útiles:

Combinación de tecla	*Nombre stty*	*Descripción*
Ctrl+c	intr	Para el contro
Ctrl+\	quit	Fuerza la parada del comando actual (usar si ***Ctrl+c*** no responde).
Ctrl+d	eof	Final del flujo de entrada.
Ctrl+u	kill	Borra desde la posición actual al principio de la línea.
Ctrl+w	werase	Borra desde la posición actual al principio de la palabra.

Si queremos usar detener un programa por lo general se usa la combinaciones de las teclas ***Ctrl+C***, esto manda un mensaje al programa para que pueda terminar liberando correctamente los recursos asignados. Si por alguna razón no termina es decir ignora el mensaje enviado de ***Ctrl+C***, siempre podemos utilizar ***Ctrl***+\, el cual termina el programa sin esperar a que este libere recursos.

Modos de edición en la línea de comandos

Por defecto ***Bash*** utiliza las teclas del modo de edición de ***emacs***, pero puede cambiar a las teclas del modo de edición de ***vi*** usando el siguiente comando :

$ set -o vi

Y para volver a la combinación de teclas por defecto o utilizar las teclas de ***emacs***.

$ set -o emacs

Moverse por la línea

Para moverse por la línea de edición, borrar partes de la línea, ir al principio y final de la línea usando la combinaciones de teclas como se muestra en la siguiente tabla.

Combinación de teclas	***Descripción***
Ctrl+a	Ir al principio de la línea.
Ctrl+e	Ir al final de la línea.
Esc+b	Ir una palabra hacia atrás (backward).
Esc+f	Ir una palabra hacia adelante (forward).
Ctrl+b	Ir una letra hacia atrás.
Ctrl+f	Ir una letra hacia adelante.
Ctrl+u	Borra de la posición actual al principio de la línea.
Ctrl+k	Borra de la posición actual al final de la línea.
Ctrl+w	Borra de la posición actual al principio de la palabra.
Esc+d	Borra de la posición actual al final de la palabra.
Ctrl+d	Borra el carácter actual hacia adelante.
Ctrl+y	Deshace el último borrado (Yank).

Buscar y ejecutar comandos del historial

Ya vimos como movernos, y borrar, ahora si queremos buscar un determinado comando podemos usar la combinación de teclas ***Ctrl+R***, que nos permite buscar hacía atrás en el historial un comando que contenga un determinado texto. Al pulsar esta combinación de teclas el prompt cambia de forma y según vamos escribiendo nos va indicando el comando del histórico que cumple el patrón dado.

La siguiente tabla muestra la combinaciones de teclas :

Combinación de teclas	***Descripción***
Ctrl+r	Realiza una búsqueda hacia atrás (Reverse).
Intro	Ejecuta el comando encontrado.
Esc	Pasa a editar el comando encontrado.
Ctrl+g	Cancela la búsqueda y deja limpia la línea de edición.

Autocompletar con el tabulador

La tecla de ***Tab*** (tabulador) nos ayuda a completar el comando que estamos escribiendo o fichero o directorio.

Reglas a tener en cuenta :

1. Si no hay nada que empiece por el texto de la palabra que precede al cursor se produce un pitido que informa del problema.

2. Si hay un comando (en el ***PATH***), una variable (siempre precedida por ***$***), un nombre de fichero o función ***Bash*** que comienza por el texto escrito, ***Bash*** completa la palabra.

3. Si hay un directorio que comienza por el nombre escrito, ***Bash*** completa el nombre de directorio

seguido por una barra de separación de nombres de directorios /.

4. Si hay más de una forma de completar la palabra el shell completa lo más que puede y emite un pitido informado de que no la pudo terminar de completar.

5. Cuando ***Bash*** no puede completar una cadena (por haber varias posibles que empiecen igual), podemos pulsar dos veces el tabulador y se nos mostrará una lista con las posibilidades cadenas candidatas.

La librería readline

Esta librería permite editar líneas de texto usando los modos ***emacs*** y ***vi***. Aunque originalmente creada por ***Bash***, actualmente la utiliza miles de comandos de GNU. Esto permite estandarizar las combinaciones de teclas entre muchos comandos.

Readline se puede personalizar usando ficheros de configuración o bien asignado combinaciones de teclas para la sesión.

El fichero de configuración

Tiene un archivo global que debe ser colocado en ***/etc/inputrc***, y otro para cada usuario que por defecto se llama ***.inputrc*** y que debe ser colocado en el ***home*** de cada usuario.

Cuando ***bash*** arranca lee estos ficheros (si existen) y carga sus valores. Dentro del fichero puede haber comentarios (precedidos por ***#***), combinaciones de teclas y asignaciones a variables de ***readline***.

Combinaciones de teclas

Lo primero que veremos son las combinaciones de teclas que están asignadas para una función determinada. Para obtener información podemos consultar haciendo ***info readline*** o en ***man bash***. Estas funciones realizan multitud de operaciones comunes. Para listar estas variables podemos crear el fichero ***.inputrc*** en nuestro directorio ***home*** y añadir la siguiente entrada :

 # Lista las funciones y variables de readline
“\C-x\C-f”: dump-functions
“\C-x\C-v”: dump-variables

Una vez que realizamos estos cambios podemos o bien logearnos nuevamente para que tome efecto el archivo ***.inputrc*** o bien presionas las teclas ***Ctrl+x Ctrl+r*** para que ***readline*** vuelva a leer nuevamente el archivo ***.inputrc***.

Ahora realizamos la combinación de teclas ***Ctrl+x Ctrl+f*** muestre las funciones de ***readline*** (que ejecuta la función ***dump-functions()*** de ***readline***) y ***Ctrl+x Ctrl+v*** par que muestre las variables de ***readline***.

Secuencia	***Descripción***
\C-	Tecla ***Ctrl***.
\M-	Tecla de ***escape*** (Meta).
\e	Secuencia de escape.
\\	El carácter \ literal.
\”	El carácter **“** literal.
\'	El carácter ' literal.

Un problema que tiene a veces la terminal es que normalmente la teclas ***DEL*** no funciona correctamente, en

este caso necesitamos usar una secuencia de escape para hacer que esta tecla llame a la función ***delete-char()*** de ***readline***.

 # Borra con la tecla DEL
"\e[3~": delete-char

readline tiene una serie de variables, que aparecen documentadas junto con las funciones de ***readline*** (que podemos consultar con la combinaciones de teclas ***Ctrl+x Ctrl+v***), las cuales podemos asignar un valor en el fichero de configuración de ***readline*** con ***set***.

Una variable interesante de ***readline*** es que al completar con tabulador distingan minúsculas de mayúsculas.

$ mkdir Videos
$ cd vid <TAB>

Nos aparecerá :

$ cd Videos/

Para ello podemos añadir la siguiente línea al fichero de configuración de ***readline*** :

 # Ignora diferencias de mayusculas/minusculas al
 # completar con tabulador
set completion-ignore-case on

Asignación de teclas de sesión

Si lo que queremos es probar combinaciones de teclas que no queremos almacenar de forma permanente podemos usar el comando ***bind***, cuya sintaxis es similar a la del fichero de configuración, pero tenemos que encerrar cada asignación entre comillas simples o dobles.

$ bind "\"\C-x\C-g\": dump-functions"

Si queremos ver las asignaciones de teclas que hay hechas a cada función de readline podemos usar el comando :

$ bind -P

Otra forma de sacar las funciones de readline :

$ bind -l

Podemos sacar todas las asignaciones de teclas en formato ***inputrc*** usando el comando :

$ bind -p

Con el comando anterior podemos enviar la salida a un fichero para luego editarlo :

$ bind -p > .inputrc

Podemos asignar combinaciones de teclas que ejecuten comandos con la opción -***x*** :

$ bind -x "\"\C-l\": ls -la"

Al presionar la tecla ***Ctrl+l*** ejecutara ***ls -la***.

Los ficheros de configuración de Bash

Cada vez que nos logeamos se ejecuta el contenido del fichero ***/etc/profile***, y luego se ejecuta el fichero ***.bash_profile*** de nuestro ***home***. Cualquier configuración que añadamos a ***.bash_profile*** no será efectiva hasta que salgamos de la cuenta y volvamos a logarnos, si queremos ejecutar los cambios que realizamos en este archivo sin salir y volvernos a loguear utilizamos el comando ***source***, el cual ejecuta el contenido del fichero que le digamos.

$ source .bash_profile

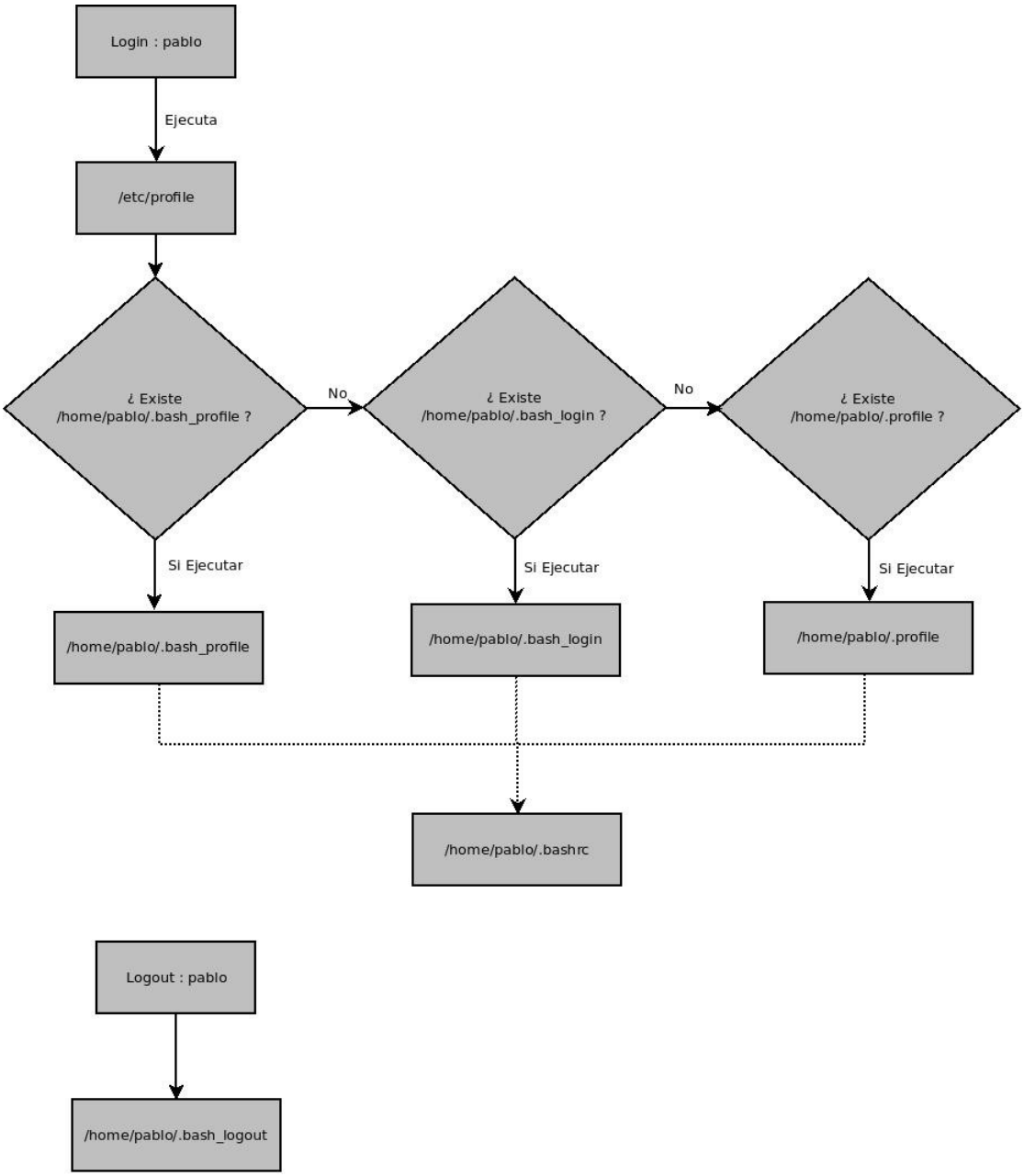
También podemos usar el ***punto*** (**.**), con lo cual haría lo mismo que ***source***.

$ . .bash_profile

Bash permite usar dos nombres alternativos para ***.bash_profile*** por razones de compatibilidad histórica: ***.bash_login***, nombre derivado del fichero ***.login*** del C Shell, y ***.profile*** nombre usado por el Bourne Shell y el Korn Shell. En cualquier caso, sólo uno de estos ficheros será ejecutado: Si ***.bash_profile*** existe los demás serán ignorados, sino Bash comprueba si existe ***.profile***. La razón por la que se eligió este orden de búsqueda es que podemos almacenar en ***.profile*** opciones propias del Bourne Shell, y añadir opciones exclusivas de Bash en el fichero ***.bash_profile*** seguido del comando ***source .profile*** para que Bash también cargue las opciones del fichero ***.profile***.

.bash_profile se ejecuta sólo al logarnos, si abrimos otro shell (ejecutando ***bash*** o ***su***) desde la línea de comandos de Bash lo que se intenta ejecutar es el contenido de ***.bashrc***. Si ***.bashrc*** no existe no se ejecutan configuraciones adicionales al abrir un nuevo shell. Este esquema nos permite separar configuraciones que se hacen una sola vez, al logarnos, de configuraciones que se cambian cada vez que se abre un nuevo shell. Si hay configuraciones en ***.bashrc*** que también queremos ejecutar al logarnos podemos poner ***source .bashrc*** dentro del fichero ***.bash_profile***.

Por último, el fichero ***.bash_logout*** es un fichero que, de existir, contiene ordenes que se ejecutarán al abandonar la cuenta, por ejemplo eliminar ficheros temporales o almacenar datos de actividad de usuarios en un fichero de log.



Las variables de entorno

Nos ayudan a configurar el entorno en el que se ejecutan los programas. Las variables de entorno no pueden tener espacios, y si queremos asignar un valor con espacio debemos de entrecomillarlo, o bien preceder el espacio por un carácter de escape :

$ NOMBRE='Marcos Pablo'
$ echo $NOMBRE
Marcos Pablo

$ NOMBRE=Marcos\ Pablo
$ echo $NOMBRE
Marcos Pablo

Como vemos para obtener el valor de la variable precede del signo ***$***, observamos que las comillas o el carácter de escape no forman parte del valor de la variable una vez que se realiza la asignación.

Para eliminar una variable de entorno usamos el comando ***unset***.

$ unset NOMBRE

Es conveniente utilizar los nombre de las variables en mayúsculas por convicción.

Variables y entrecomillado

Vemos en este caso como combinar la variable con la salida de pantalla utilizando el comando ***echo***.

$ NOMBRE='Marcos Pablo'
$ EDAD=37
$ echo "Su nombre es : $nombre y la edad es : $edad"
Su nombre es : Marcos Pablo y la edad es : 37

Personalizar el prompt

Podemos personalizar nuestro prompt de Bash usando las variables de entorno ***PS1***, ***PS2***, ***PS3*** y ***PS4***. En la siguiente tabla se muestra las opciones de personalización que podemos usar en cualquiera de los prompt.

Opción	***Descripción***
\a	El carácter ASCII bell (007).
\A	La hora en formato ***HH:MM***.
\d	La fecha en formato ***semana mes día***.
\D {*formato*}	Nos permite personalizar más la fecha.
\e	El carácter de escape ASCII (033).
\H	Hostname.
\h	El nombre de la máquina hasta el primer punto.
\j	Número de jobs hijos del shell.
\l	Nombre del terminal en el que estamos trabajando (p.e. ***tty2***).
\n	Retorno de carro y nueva línea.
\r	Retorno de carro.
\s	Nombre de shell.
\t	Hora en formato ***HH:MM:SS*** con 12 horas.
\T	Hora en formato anterior pero con 24 horas.
\u	Nombre de usuario.
\v	Versión de Bash.
\w	Ruta del directorio actual.
\W	Nombre del directorio actual (sin ruta).
\#	Número de comandos ejecutados.
\!	Número histórico del comando.
\$	**#** si somos root, o ***$*** en caso contrario.
\nnn	Código de carácter a mostrar en octal.
\\	La barra hacía atrás.
\[	Empieza una secuencia de caracteres no imprimibles, como los caracteres de control de secuencias del terminal.

\]	Acaba una secuencia de caracteres no imprimibles.

Un ejemplo sencillo es de poner ***\w*** que es el directorio donde estamos parado y esto nos ayuda siempre para saber donde estamos parado.

```
$ PS1='\w->'
~->
```

Donde ~ significa que estamos en el directorio ***home***.

Es muy típico querer saber si somos ***root*** o cualquier otro usuario de la máquina, para lo cual se usa la opción ***\$*** que muestra una ***#*** si somos ***root***, o un ***$*** en cualquier otro caso.

```
$ PS1='\w\$'
~$
```

Es muy común que accedamos a servidores remotos por medio de ***ssh*** o ***telnet*** y no sabes por medio del prompt en que servidor estamos, para eso podemos incorporar en la variables ***PS1*** la opción ***\H*** que nos da el nombre de la máquina donde estamos situados, o la opción ***\h*** que nos da el nombre de la máquina sólo hasta el primer punto :

```
$ PS1='\h\w\$'
centrux~$

$ PS1='\H\w\$'
centrux.org~$
```

Cuando nos movemos en distintas tty de la consola como vimos con la teclas ***Alt+F1*** hasta ***Alt+F6*** (desde la tty1 hasta la tty6), es conveniente que en nuestro prompt tengamos también que ***tty*** es donde estamos.

```
$ PS1='[\u@\h@\l] \w\$'
[pablo@centrux@1] ~$
```

Con la opción ***\u*** es la que indica el nombre del usuario, ***\l*** nos indica la ***tty*** que estamos.

También podemos poner color a nuestro prompt, poniendo las secuencias de escape de caracteres no imprimibles tienen que encerrarse ente ***\[033[*** y ***\]***. Para las secuencias de escape de color, también deben aparecer, seguidos además de una ***m*** minúscula.

Por ejemplo para incluir texto azul en el prompt :

```
$ PS1='\[\033[34m\][\u@\h:\w]$ '
```

En este caso el color azul esta dado por el código ***34m*** y en ningún momento se retorna el color habitual, es decir todo lo que escribamos sera de color azul para arreglar esto y que solo nuestro prompt sea de color azul al final tenemos que poner ***[\033[0m\]***

```
$ PS1='\[\033[34m\][\u@\h:\w]$ \[\033[0m\]'
```

En la siguiente tabla tenemos los colores :

Color	*Código*	*Color*	*Código*
Negro	0;30	Gris oscuro	1;30
Azul	0;34	Azul claro	1;34
Verde	0;32	Verde claro	1;32

Cyan	0;36	Cyan claro	1;36
Rojo	0;31	Rojo claro	1;31
Púrpura	0;35	Púrpura claro	1;35
Marrón	0;33	Amarillo	1;33
Gris claro	0;37	Blanco	1;37

También podemos poner color de fondo, usando ***44*** para fondo azul, ***41*** para fondo rojo, etc, como vemos le sumamos ***10 + al colo***r.

Color	***Código***
Negro	0;40
Azul	0;44
Verde	0;42
Cyan	0;46
Rojo	0;41
Púrpura	0;45
Marrón	0;43
Gris claro	0;47

Por ejemplo podemos combinar color de fondo azul y color de las letras blancas :

$ PS1="\[\033[44m\] \[\033[1;37m\][\u@\h:\w]\$ "

Otros códigos que disponibles incluyen subrayado, parpadeante, inverso y oculto.

Tipo	***Código***
Subrayado	4
Parpadeante	5
Inverso	7
Oculto	8

$ PS1="\[\033[44m\] \[\033[1;37m\] \[\033[4m\][\u@\h:\w]\$ "

PS2 es el prompt secundario y por defecto vale >. Se usa cuando escribimos un comando inacabado, por ejemplo :

**$ echo Esto es una prueba de escribir **
> en dos lineas.

Por último, ***PS3*** y ***PS4*** son prompts de programación y depuración que veremos mas adelante.

Variables de entorno internas

Existen una serie de variables de entorno, las cuales se resumen en la siguiente tabla, estas variables las fija el shell no nosotros.

Variable	*Descripción*
SHELL	***Path*** del fichero que estamos usando.
LINES	Líneas del terminal en caracteres.
COLUMNS	Columnas del terminal en caracteres.
HOME	Directorio ***home*** del usuario.
PWD	Directorio actual.
OLDPWD	Anterior directorio actual.
USER	Nombre de usuario en la cuenta donde estamos logados.

Algunas variables se actualizan dinámicamente, por ejemplo ***LINES*** y ***COLUMNS*** en todo momento almacenan en número de filas y columnas de la terminal, y las usan algunos programas como el editor ***vi/vim*** o ***emacs*** para modificar su apariencia.

Exportar variables

Las variables de entorno que definimos dentro de una sesión de ***Bash*** no están disponibles para los subprocesos que lanza ***Bash*** (soló las puede usar ***Bash*** y nuestro comandos interactivos y ***scripts***) a no ser que las exportemos con el comando ***export***.

$ EDITOR=vi
$ export EDITOR

O directamente podemos exportarlo :

$ export EDITOR=vi

Podemos obtener un listado de las variables exportadas usando el comando ***env*** o el comando ***export*** sin argumentos. Para obtener las variables tanto exportadas como no exportadas debemos usar el comando ***set*** sin argumentos.

Programación básica del shell

Un script es un fichero que contiene comandos Bash ejecutables. Una vez que creamos un script con nuestro editor de texto favorito hay dos forma de ejecutarlo.

- Darle permiso de ejecución tanto para el dueño, grupo y otros (***chmod +x archivo_script)***.Una vez puesto este permiso, podremos ejecutarlo siempre que esté en alguno de los directorios indicados en la variable de entorno ***PATH***. Es muy típico dentro de los script poner como primer linea ***!#/bin/bash***, de esta forma se indica que el script debe ser ejecutado con ***Bash***, aunque ***bash*** no sea el ***shell*** por defecto, otros scripts para otros lenguajes como ***awk***, ***ksh*** o ***perl*** también se escriben empezando la primera línea.

- Cuando ejecutamos un script con ***source***, éste se ejecuta dentro del proceso del ***shell***, mientras que si lo ejecutamos como un fichero ejecutable se ejecuta en un ***subshell***.

Ejemplo :

$ vi ejemplo.sh

```
!#/bin/bash
A=10
echo $A
```

$ chmod +x ejemplo.sh

Al llamarlo de la siguiente forma, lo estamos ejecutando en un ***subshell***.

$ ./ejemplo.sh
10

Veremos que si queremos imprimir el valor de ***A*** no mostrara nada :

$ echo $A

Ahora lo llamaremos con el comando ***source*** o **.** (punto) y veremos que este se ejecuta en el ***shell*** y no el ***subshell*** como el ejemplo anterior.

$ source ejemplo.sh
10

$ echo $A
10

o

$ . ./ejemplo.sh
10

$ echo $A
10

Funciones

A diferencia de los scripts, se ejecutan dentro de la memoria del propio proceso de Bash, con lo que son más eficientes que ejecutar aparte, pero tienen el inconveniente de que tienen que estar siempre cargadas en la memoria del proceso Bash para poder usarse.

Existe dos formas para definir una función :

El estilo del Bourne Shell :

function nombre_funcion
{
…
comandos bash
…
}

O el estilo del C Shell :

nombre_funcion()
{
…
comandos bash
…
}

No existe diferencia entre ellos, para borrar una función podemos utilizar el comando ***unset -f nombre_función.*** Cuando definimos una función se almacena como una variable de entorno (con su valor almacenado la implementación de la función). Para ejecutar la función escribimos el nombre de la función y los parámetros necesarios.

$ vi funcion.sh

```
!#/bin/bash
imprimir_valor() {
        A=20
        echo $A
}
```

$ chmod +x funcion.sh
$ source funcion.sh

$ imprimir_valor
20

Si queremos borrar una función como explicamos anteriormente con el comando ***unset -f nombre_función.***

$ unset -f imprimir_valor
$ imprimir_valor
bash: imprimir_valor: no se encontró la orden

Para ver todas las funciones que tenemos declaradas simplemente ejecutamos **declare -f**, por ejemplo creamos la siguiente función.

$ vi hola.sh

```
!#/bin/bash
imprimir_saludo() {
        echo “Hola : ${USER}”
}
```

$ chmod +x hola.sh
$ source hola.sh

$ imprimir_saludo
Hola : pablo

$ declare -f
…
```
imprimir_saludo ()
{
   echo "Hola : ${USER}"
}
```
…

Si solo queremos obtener el nombre de la función sin el código :

$ declare -f
…
declare -f imprimir_saludo
…

Si una función tiene el mismo nombre que un script o ejecutable, la función tiene preferencia.

Orden de preferencia de los símbolos de Bash

A continuación se enumera el orden de preferencia que sigue Bash a la hora de resolver un símbolo:

Orden	***Descripción***
1	Alias.
2	Palabras clave (keywords) como ***function***, ***if*** *o* ***for***.
3	Funciones.
4	Comandos internos (***cd***, ***type***,***etc).***

5	Scripts y programas ejecutables, para los cuales se sigue el orden en que se dan sus directorios en la variable de entorno ***PATH***.

Si nos interesa conocer la procedencia de un comando podemos usar el comando interno ***type***, si le damos un comando nos indica cual es su fuente.

Ejemplo :

$ type ls
ls es un alias de `ls –color=auto'

$ type -path cat
/bin/cat

Con la opción *-**path*** me devuelve en que ruta se encuentra el comando.

Podemos también pasarle la función de un script como la función que realizamos anteriormente :

$ type imprimir_saludo
imprimir_valor: es una función
imprimir_valor ()
{
A=20;
echo ${A}
}

Como vimos anteriormente los alias tiene mayor prioridad que las funciones, si declaramos un alias con el mismo nombre de la función y luego realizamos un type veremos que nos muestra lo siguiente :

$ alias imprimir_saludo=”ls -a”
$ type imprimir_saludo
imprimir_saludo es un alias de `ls -a'

Pero si queremos ver todas las definiciones ejecutamos ***type -a*** :

$ type -a imprimir_saludo
imprimir_saludo es un alias de `ls -a '
imprimir_saludo: es una función
imprimir_saludo() {
echo “Hola : ${USER}”
}

Si ejecutamos ***imprimir_saludo*** como tiene mayor prioridad el ***alias*** solo ejecutara el ***ls*** y no la función.

$ imprimir_saludo
./ ../ .bash_history .vim/ .viminfo

Si queremos conocer el tipo de un símbolo con la opción *-**t***, es decir esto nos devolverá si es un ***alias***, ***keyword***, ***function***, ***builtin*** (comando internos) o ***file***.

Ejemplo :

$ type -t ls
alias

$ type -t rm
file

$ type -t cd
builtin

$ type -t imprimir_saludo
alias

Borramos el ***alias*** anteriormente definido con el comando ***unalias***.

$ unalias imprimir_saludo
$ type -t imprimir_saludo
function

Variables de Shell

Por convenio las variables de entorno exportadas se escriben en mayúsculas, acordemonos que GNU/Linux y UNIX deferencia entre minúsculas y mayúsculas.

Los parámetros posicionales

Son los en encargados de recibir los argumentos de un script y los parámetros de una función. Sus nombres son ***1***, ***2***, ***3***, etc., con lo que para acceder a ellos utilizaremos, como normalmente, el símbolo ***$*** de la siguiente forma ***$1***, ***$2***, ***$3***, etc. Además tenemos el parámetro posicional ***0*** que contiene el nombre del script que se esta ejecutando.

Ejemplo :

$ vi parametros.sh

```
#!/bin/bash
# Esto es un ejemplo de script utilizando parametros.

echo "El nombre del script es : $0"
echo "Se pasaron los siguientes parametros : $1 $2 $3 $4"
```

$ chmod +x parametros.sh
$ ./parametros.sh hola mundo
El nombre del script es : parametros.sh
Se pasaron los siguientes parametros : hola mundo

Como los parámetros ***$3*** y ***$4*** no se pasaron son nulos dan a una cadena vacía que no se imprime.

Podemos definir una función y llamarla con el comando **source** y pasar los parámetros necesarios

$ vi funcion_01.sh

```
function recibir
{
 echo "La funcion $0"
 echo "Recibe como parametros $1 $2 $3 $4"
}
```

$ chmod +x funcion_01.sh
$ ./recibir
La funcion recibir
Recibe como parametros buenos dias Marcos

Variables locales y globales

Los parámetros son locales al script o función y no se puede acceder o modificar desde otra función. Por ejemplo el siguiente script.

$ vi funcion_01.sh

```
function recibir
{
 echo “La funcion $0”
 echo “Recibe como parametros $1 $2 $3 $4”
}
```

Ejemplo de variable global.

$ vi funcion_01.sh

```
function recibir
{
 nombre='Dentro de la funcion'
}

 nombre='Nombre del programa funcion_01.sh'
echo $nombre

recibir
echo $nombre
```

$ chmod +x funcion_01.sh
$ ./funcion_01.sh
Nombre del programa funcion_01.sh
Dentro de la funcion

Si queremos que una variable sea ***local*** debemos poner el modificador ***local***, el cual sólo se puede usar dentro de las funciones (no en los scripts). Veremos en el ejemplo que se crea una variable ***local nombre*** y no modifica a la global del script.

$ vi funcion_01.sh

```
function recibir
{
 local nombre='Dentro de la funcion'
}

 nombre='Nombre del programa funcion_01.sh'
echo $nombre

recibir
echo $nombre
```

$ chmod +x funcion_01.sh
$ ./funcion_01.sh
Nombre del programa funcion_01.sh
Nombre del programa funcion_01.sh

Este ejemplo combina tanto la variable local como global. Como vemos por defecto las variables son

globales.

$ vi funcion_02.sh

```
function soy
{
 local donde='Dentro de la funcion'
 quiensoy=Marcos
}

donde='En el script'
echo $donde

soy
echo $donde
echo "Soy $quiensoy"
```

$ chmod +x funcion_02.sh
$ ./funcion_02.sh
En el script
En el script
Soy Marcos

Las variables $*, $@ y $#

La variable ***$#*** contiene la cantidad de parámetros que se pasan, este valor es de tipo cadenas de caracteres, más adelante veremos como podemos convertir este valor a número para poder operar con el.

Tanto $* como $@ nos devuelven los parámetros pasados al script o función.

Ejemplo :

$ vi parametros_01.sh

```
echo "Nombre del script : $0 recibe : $# parametros : " $*
echo "Nombre del script : $0 recibe : $# parametros : " $@
```

$ chmod +x parametros_01.sh
$ ./parametros_01.sh 1 2 3 4 5
Nombre del script : ./parametros_01sh recibe : 6 parametros : 1 2 3 4 5 6
Nombre del script : ./parametros_01sh recibe : 6 parametros : 1 2 3 4 5 6

La diferencia que existe entre ***$**** y ***$@*** :

- Podemos cambiar el símbolo que usa ***$**** para separar los argumentos indicándolo en la variable de entorno ***IFS*** (Internal Field Separator), mientras que ***$@*** siempre usa como separador un espacio.

Ejemplo:

$ vi parametros_01.sh

```
IFS=','
echo "Nombre del script : $0 recibe : $# parametros : " $*
echo "Nombre del script : $0 recibe : $# parametros : " $@
```

$ ./parametros_01.sh 1 2 3 4 5
Nombre del script : ./parametros_01sh recibe : 6 parametros : 1,2,3,4,5,6
Nombre del script : ./parametros_01sh recibe : 6 parametros : 1 2 3 4 5 6

Expansión de variables usando llaves

Realmente la forma que usamos para expandir una variable ***$variable*** es una simplificación de la forma más general ***${variable}***. La simplificación se puede usar siempre que no existan ambigüedades. La forma ***${variable}*** se usa siempre que la variable vaya seguida por una letra, dígito o guión bajo (_), en caso contrario podemos usar la forma simplificada ***$variable***.

Ejemplo :

$ nombre=Marcos
$ apellido=Russo
$ echo $nombre_$apelldio
Russo

Se produce esto porque ***$nombre_*** piensa que es una variable hasta el otro símbolo ***$***.

$ nombre=Marcos
$ apellido=Russo
$ echo ${nombre}_${apelldio}
Marcos_Russo

También lo tenemos que usar las llaves cuando vamos a sacar el décimo parámetro posicional. Es decir si usamos ***$10***, bash lo expandirá por ***$1*** seguido de un ***0***, entonces la forma de usarlo para que lo interprete como el décimo parámetro es usando ***${10}***.

Operadores de cadena

Nos permiten manipular cadenas sin necesidad de escribir complicadas rutinas de procesamiento de texto. Los operadores de cadenas nos permiten realizar las siguientes operaciones :

- Asegurarnos de una variable existe (que está definida y que no es nula).
- Asignar valores por defecto a las variables.
- Tratar errores debidos a que una variable no tiene un valor asignado.
- Tomar o eliminar partes de la cadena que cumplen un patrón.

Operadores de sustitución

Operador	***Descripción***
${variable:-valor}	Si ***variable*** existe y no es nula retorna el valor de ***variable***, sino retorna valor.
${variable:+valor}	Si ***variable*** existe y no es nula retorna ***valor***, sino retorna una cadena nula.
${variable:=valor}	Si ***variable*** existe y no es nula retorna el valor de ***variable***, sino asigna ***valor*** a ***variable*** y retorno su valor.
${variable:?mensaje}	Si ***variable*** existe y no es nula retorna el valor de ***variable***, sino imprime *variable:mensaje* y aborta el script que se esté ejecutando (sólo en shells no interactivos). Si omitimos ***mensaje*** imprime el mensaje por defecto ***parameter null or not set***.

${variable:offset:longitud}	Retorna una subcadena de ***variable*** que empieza en ***offset*** y tiene ***longitud*** caracteres. El primer carácter de ***variable*** empieza en la posición ***0***. Si se omite ***longitud*** la subcadena empieza en ***offset*** y continua hasta el final de ***variable***.

Los dos puntos (**:**) en este operador son opcionales. Si se omiten en vez de comprobar ***si existe y no es nulo, sólo comprueba que exista*** (aunque sea nulo).

Ejemplo :

${variable:-valor} se utiliza para retornar un valor por defecto cuando el valor de la variable ***variable*** está indefinido. Esto no modifica el contenido de ***variable***.

$ echo ${variable:-10}
10

${variable:+valor} por contra se utiliza para comprobar que una variable tenga asignado un valor no nulo.

$ echo ${variable:+1}

Si ***variable*** contiene datos entonces mostrara el valor ***1*** esto se puede tomar como un valor verdadero. Esto no modifica el contenido de ***variable***.

${variable:=valor} que asigna un valor a la variable si ésta está indefinida. En el siguiente ejemplo asigna el valor ***1*** a ***variable*** sino tiene valor.

$ echo ${variable:=1}
1

${variable:offset:longitud} podemos tomar una parte de la cadena.

$ variable=”Marcos Pablo”
$ echo ${variable:0:6}
Marcos

$ echo ${variable:7:5}
Pablo

Operadores de búsqueda de patrones

Un uso frecuente de los operadores de búsqueda de patrones es eliminar partes de la ruta de un fichero, como pueda ser el directorio o el nombre del fichero.

Un uso frecuente de los operadores de búsqueda de patrones es eliminar partes de la ruta un fichero, como pueda ser el directorio o el nombre del fichero.

Ejemplo :

$ ruta=/usr/share/doc/apt/examples/sources.list

Si ejecutamos la siguiente de operadores de búsqueda de patrones, obtendremos los siguientes resultados:

Operador	***Resultado***
${ruta##/*/}	sources.list
${ruta#/*/}	share/doc/apt/examples/sources.list

${ruta%.*}	/usr/share/doc/apt/examples/sources
${ruta%%.*}	/usr/share/doc/apt/examples/sources

En la búsqueda de patrones se pueden usar tanto los comodines tradicionales, el siguiente cuadro son los comodines extendidos:

Operador	***Descripción***
${variable#patron}	Si ***patron*** coincide con la primera parte del valor de ***variable***, borra la parte más pequeña que coincide y retorna el resto.
${variable##patron}	Si ***patron*** coincide con la primera parte del valor de ***variable***, borra la parte más grande que coincide y retorna el resto.
${variable%patron}	Si ***patron*** coincide con el final del valor de ***variable***, borra la parte más pequeña que coincide y retorna el resto.
${variable/patron/cadena}	La parte más grande de ***patron*** que coincide en ***variable*** es reemplazada por cadena. La primera forma sólo reemplaza la primera ocurrencia, y la segunda forma reemplaza todas las ocurrencias. Si ***patron*** empieza por **#** debe producirse la coincidencia al principio de ***variable***, y si empieza por **%** debe producirse la coincidencia al final de ***variable***. Si ***cadena*** es nula se borran las ocurrencias. En ningún caso ***variable*** se modifica, sólo se retorna su valor con modificaciones.

Ejemplo :

$ ruta=/usr/share/doc/apt/examples/sources.list
$ echo ${ruta#/usr/*/doc}
/apt/examples/sources.list

$ echo ${ruta%*.list}
/usr/share/doc/apt/examples/sources

$ echo ${ruta%%/examples/sources.list/}
/usr/share/doc/apt

$ echo ${ruta/usr/var}
*/**var**/share/doc/apt/examples/sources*

Un operador muy interesante es ***?*** que es una condición si no existe tal cosa entonces hacer esto.

Ejemplo :

$ vi entrada.sh

#!/bin/bash
fichero_entrada=${1:?'Falta parametros'}

$ ./entrada.sh
./entrada.sh: línea 1: 1: Falta parametros

Si queremos ir mostrando linea por linea el ***PATH*** tomando como separador los (**:**), realizamos lo siguiente :

$ echo -e ${PATH//:/'\n'}
/usr/local/bin
/usr/bin
/bin

/usr/local/games
/usr/games

El *'\n'* significa salto de linea, es decir reemplaza los ***:*** por *'\n'* (salto de linea).

Operadores longitud

Tiene la forma ***${#variable}*** donde ***variable*** es la variable cuyo valor queremos medir.

Ejemplo :

nombre='Marcos Pablo Russo'
$ echo ${#nombre}
18

Sustitución de comandos

Nos permite usar la salida de un comando como si fuera el valor de una variable. La sintaxis es la siguiente :

$(comando)

Las comilla ***`comando`*** sirve para realizar la ejecución de un comando. Aunque en Bash mantiene esta sintaxis por compatibilidad hacía atrás, la forma recomendada es mediante el uso de paréntesis, que permite anidar sustituciones de comandos.

Ejemplo :

$ midir=$(ls $HOME)
$ echo $midir

También podemos cargar el contenido de un archivo a una variable usando ***$(<nombre_del_fichero)***.

$ miarchivo=$(</etc/passwd)
$ echo $miarchivo

Bash dispone de los comandos internos ***pushd*** y ***popd*** que nos permiten movernos por los directorios de la siguiente forma:

- Al ejecutar ***pushd*** directorio nos movemos a ese directorio (como un ***cd***) y el directorio anterior se guarda en una pila de forma que al hacer ***popd*** regresamos al directorio que está en la cima de la pila.

- El comando ***dirs*** nos permite ver la pila de directorios que tenemos en cada momento. En parte estos comandos son parecidos al ***cd*** -, que nos lleva al anterior directorio en el que estamos (al guardado en la variable de entorno ***OLDPWD***), sólo que ***pushd*** y ***popd*** nos permite almacenar cualquier número de directorios en la pila.

Ejemplo :

$ pushd /usr/share/doc
/usr/share/doc ~
$ pushd /etc
/etc /usr/share/doc ~
$ pushd /usr/include
/usr/include /etc /usr/share/doc ~

$ dirs -v

0 /usr/include
1 /etc
2 /usr/share/doc
3 ~
$ popd
/etc /usr/share/doc ~
$ popd
/usr/share/doc ~
$ popd
~

Control de flujo

Las sentencias de control de flujo es un denominador común de la mayoría de los lenguajes de programación, incluido Bash.

En este caso veremos las sentencias de control de flujo (***if***, ***else***, ***for***, ***while***, ***until***, ***case*** y ***select***).

Las sentencias condicionales

La sentencia condicional tiene el siguiente formato :

```
if condicion
then
    sentencias
elif condicion
then
    sentencias
else
    sentencias
fi
```

Este lenguaje nos obliga a que las sentencias estén organizadas con estos retornos de carro, aunque algunas personas prefieren poner los ***then*** en la misma línea que los ***if***, para lo cual debemos de usar el separador de comandos, que en Bash es el punto y coma (***;***) así :

```
if condicion ; then
    sentencias
elif condicion ; then
    sentencias
else
    sentencias
fi
```

Los códigos de terminación

En **UNIX** los comandos terminan devolviendo un código numérico al que se llama ***código de terminación*** (exit status) que indica si el comando tuvo éxito o no.

Aunque no es obligatorio que sea así, normalmente un código de terminación ***0*** significa que el comando terminó correctamente, y un código entre ***1*** y ***255*** corresponde a posibles códigos de error.

La sentencia ***if*** comprueba el código de terminación de un comando en la condición, si éste es ***0*** la condición se evalúa como cierta.

Ejemplo :

$ vi ir.sh

```
#!/bin/bash

if cd ${1:?"Faltan parametros"}; then
  echo $1
else
 echo "Directorio $1 no valido"
fi
```

$ chmod +x ir.sh
$ ./ir.sh
./ir.sh: línea 3: 1: Faltan parametros

$ ./ir.sh xx
./ir.sh: línea 3: cd: xx: No existe el fichero o el directorio
Directorio xx no valido

$ ./ir.sh /usr/share/doc
/usr/share/doc

Las sentencias return y exit

Los comandos UNIX retornan un código de terminación indicando el resultado de una operación, pero qué si estamos haciendo una función. ¿ Cómo retornamos el código de terminación ?.

Podemos usar la variable especial ***?***, cuyo valor es ***$?***, y que indica el código de terminación del último comando ejecutado por el shell.

Ejemplo :

$ cd xx
bash: cd: xx: No existe el fichero o el directorio

$ echo $?
1

$ cd /usr/share/doc

$ echo $?
0

Si estamos en una función nosotros necesitamos devolver dicho valor mediante ***return*** para poder salir de la función, esta sentencia solo puede ser usado dentro de las funciones. Por contra, la sentencia ***exit*** puede ser ejecutada en cualquier sitio, y lo que hace es abandonar el script (aunque se ejecute dentro de una función), suele usarse para detectar situaciones erróneas que hacen que el programa deba detenerse, aunque a veces se utiliza para “cerrar” un programa.

Ejemplo :

$ vi cd.sh

```
cd()
{
    builtin cd "$@"
    local ct=$?
    echo "$OLDPWD -> $PWD"
    return $cd
```

```
}
```

Como dijimos anteriormente las funciones tiene preferencia sobre los comando internos, como es en este caso la función que declaramos anteriormente ***cd()*** tiene mayor prioridad que el comando interno ***cd***.

Para ejecutar el comando interno y no la función podemos usar el comando interno ***builtin***, sino quedaría en un bucle infinito llamándose a si mismo.

```
builtin cd "$@"
```

Observamos también que tenemos que guardar el resultado devuelto de la anterior linea en una variable y al final retornamos el valor, si no hacemos así y retornamos sin guardarla en una variable, estaríamos devolviendo el valor devuelto por el ***echo "$OLDPWD -> $PWD"***.

```
local ct=$?
```

Operadores lógicos y códigos de terminación

Podemos combinar varios códigos de terminación de comandos mediante los operadores lógicos ***and*** (representado con ***&&***) ***or*** (representado con ||) y ***not*** (representado con ***!***).

Ejemplo :

```
if cd /tmp && cp archivo.tmp $HOME; then
  …
fi
```

Primero ejecuta el comando ***cd /tmp*** si este tiene éxito (es decir el código de retorno es ***0*** verdadero), ejecuta el siguiente comando ***cp archivo.tmp $HOME***, pero si el primer comando falla no verificara el segundo comando, para que se cumpla la condición ambos comandos deberían de tener éxito (si un operando falla ya no tiene sentido evaluar el otro).

Muchas veces se utiliza el operador ***&&*** para implementar un ***if***. Para ello se aprovecha el hecho de que si el primer operando no se cumple, no se evalúa el segundo. Si por ejemplo queremos que se imprima el mensaje ***"No quedan elementos"*** cuando ***elemento*** sea ***0*** podemos realizar el siguiente ejemplo:

```
[ $elemento = 0 ] && echo "No quedan elemento"
```

El operador || (or) por contra ejecuta el primer comando, y sólo si éste falla se ejecuta el segundo.

Ejemplo :

```
if cp archivo_1.tmp $HOME ||  cp archivo_2.tmp $HOME; then
  …
fi
```

En este ejemplo vemos que si se cumple una condición no comprueba el resto.

Por último veremos el operador ***!*** (not), que niega un código de terminación.

Ejemplo :

```
if ! cp archivo_1.tmp $HOME; then
  Procesa el error
fi
```

Test condicionales

La sentencia ***if*** lo único que sabe es evaluar códigos de terminación. Pero no podemos verificar si un comando se ha ejecutado bien.
El comando interno ***test*** nos permite comprobar otras condiciones, que le pasamos como argumento, para acabar devolviendo un código de terminación. Una forma alternativa al comando ***test*** es el operador ***[ ]*** dentro del cual metemos la condición a evaluar, es decir, ***test cadena1 = cadena2*** es equivalente a ***[cadena1 = cadena2]***. Los espacios entre corchetes y los operados, o entre los operandos y el operador de comparación = son obligatorios. Con lo que es coherente con la sintaxis de ***if*** que hemos visto en el apartado anterior.

Usando los ***test condicionales*** podemos evaluar atributos de un fichero (si existe, que tipo de fichero es, que permisos tiene, etc), comparar dos ficheros para ver cual de ellos es más reciente, comparar cadenas, e incluso comparar los números que almacenan las cadenas (comparación numérica).

Comparación de cadenas

Operador	***Verdadero si ...***
str1 = str2	Las cadenas son iguales.
str1 != str2	Las cadenas son distintas.
str1 < str2	***str1*** es menor lexicográficamente a ***str2***.
str1 > str2	***str1*** es mayor lexicográficamente a ***str2***.
-n str1	***str1*** es no nula y tiene longitud mayor a cero.
-z str1	***str1*** es nula (tiene longitud cero).

Ejemplo :

$ vi cadenas.sh

```
#!/bin/bash

if [ -z $1 ]; then
  echo "Cadena nula."
  exit 1
fi

if [ -z $2 ]; then
  echo "Cadena nula."
  exit 1
fi

if [[ $1 = $2 ]]; then
  echo "Son iguales las cadenas."
elif [[ $1 > $2 ]]; then
  echo "La primer cadena es mas grande que la segunda cadena."
else
  echo "La segunda cadena es mas grande que la primer cadena."
fi
```

$ chmod +x cadenas.sh
$ ./cadenas.sh
Cadena nula.

$ ./cadenas.sh a a
Son iguales las cadenas.

$ ./cadenas.sh a b
La segunda cadena es mas grande que la primer cadena.

$ ./cadenas.sh b a
La primer cadena es mas grande que la segunda cadena.

Comparación numérica de enteros

También permite comparar variables que almacenan cadenas interpretando estas cadenas como números, para ello se deben de utilizar los siguiente operadores que se muestran en la siguiente tabla :

Operador	***Descripción***
-lt	Menor que.
-le	Menor o igual que.
-eq	Igual.
-ge	Mayor o igual que.
-gt	Mayor que.
-ne	No son iguales.

Los test de condición (las que van entre corchetes ***[ ]***) también se pueden combinar usando los operadores lógicos ***&&***, || y ***!***.

if [condicion] && [condicion]; then

También es posible combinar comandos del shell con test de condición:

if comando && [condicion]; then

Además a nivel de test de condición (dentro de los ***[ ]***) también se pueden usar operadores lógicos, pero en este caso debemos de usar los operadores -***a*** (para and) y -***o*** (para or).

Ejemplo :

$ vi comparacion_valores.sh

```
#!/bin/bash

valor_producto_1=$1
valor_producto_2=$2

if [ ${valor_producto_1} -le ${valor_producto_2}  -a  ${valor_producto_1} -le 10 ]
then
  echo "El producto ingresado es menor o igual al producto2 y menor o igual a 10"
fi
```

$ chmod +x comparacion_valores.sh
$ ./comparacion_valores.sh 10 20
El producto ingresado es menor o igual al producto2 y menor o igual a 10

Aunque el operador lógico -***a*** tiene menor precedencia que el operador de comparación -***le***, en expresiones complejas conviene usar paréntesis que ayuden a entender la expresión, pero si los usamos dentro de un test

de condición conviene recordar dos reglas :

- Los paréntesis dentro de expresiones condicionales deben ir precedidos por el carácter de escape \ (para evitar que se interpreten como una sustitución de comandos).

- Los paréntesis, al igual que los corchetes, deben de estar separados por un espacio.

El ejemplo anterior se puede también escribir de la siguiente manera :

```
if [ \( ${valor_producto_1} -le ${valor_producto_2}  \) -a  \( ${valor_producto_1} -le 10 \) ]
then
  echo "El producto ingresado es menor o igual al producto2 y menor o igual a 10"
fi
```

Comprobar atributos de ficheros

El tercer tipo de operadores de comparación nos permiten comparar atributos de fichero. Existen 22 operadores de este tipo.

Operador	***Verdadero si ...***
-a ***archivo***	***archivo*** existe.
-b ***archivo***	***archivo*** existe y es un dispositivo de bloque.
-c ***archivo***	***archivo*** existe y es un dispositivo de carácter.
-d ***archivo***	***archivo*** existe y es un directorio.
-e ***archivo***	***archivo*** existe (equivalente a -***a***).
-f ***archivo***	***archivo*** existe y en un archivo regular.
-g ***archivo***	***archivo*** existe y tiene activo el bit de ***setgid***.
-G ***archivo***	***archivo*** existe y es poseído por el grupo ***ID efectivo***.
-h ***archivo***	***archivo*** existe y es un enlace simbólico.
-k ***archivo***	***archivo*** existe y tiene el ***sticky bit*** activado.
-L ***archivo***	***archivo*** existe y es un enlace simbólico.
-N ***archivo***	***archivo*** existe y fue modificado desde la última lectura.
-O ***archivo***	***archivo*** existe y es poseído por el user ID efectivo.
-p ***archivo***	***archivo*** existe y es un pipe o ***named pipe***.
-r ***archivo***	***archivo*** existe y podemos leerlo.
-s ***archivo***	***archivo*** existe y no está vacío.
-S ***archivo***	***archivo*** existe y es un ***socket***.
-w ***archivo***	***archivo*** existe y tenemos permiso de escritura.
-x ***archivo***	***archivo*** existe y tenemos permiso de escritura, o de búsqueda si es un directorio.
archivo1 -nt ***archivo2***	La fecha de modificación de ***archivo1*** más moderna que (Newer Than) la de ***archivo2***.
archivo1 -ot ***archivo2***	La fecha de modificación de ***archivo1*** más antigua que (Older Than) la de ***archivo2***.
archivo1 -ef ***archivo2***	***archivo1*** y ***archivo2*** son el mismo archivo (Equal File).

Ejemplo :

$ vi archivo_01.sh

```
#!/bin/bash

if [ -z "$1" ]
then
  echo "Forma de uso :"
  echo ""
  echo "$0 nombre_archivo|directorio"
  echo ""
  exit 1
fi

if [ -d "$1" ]
then
   echo "Es un directorio"
fi

if [ -L "$1" ]
then
  echo "Es un enlace simbólico"
fi

if [ -w "$1" ]
then
  echo "Tiene permiso de escritura"
fi

if [ -x "$1" ]
then
  echo "Tiene permiso de ejecucion"
fi
```

$ chmod +x archivo_01.sh
$ ./archivo_01.sh
archivo_01.sh nombre_archivo|directorio

$ ./archivo_01.sh /usr/share/doc
Es un directorio
Tiene permiso de ejecucion

El bucle for

El bucle ***for*** en Bash es un poco distinto a los bucles de otros lenguajes como ***Java*** o ***C***. Aquí no se repite un número fijo de veces, sino que se procesan las palabras de na frase una a una.

La sintaxis es la siguiente :

```
for variable [in lista]
do
  …
  Sentencias que usan $variable
  ...
done
```

Si se omite ***in lista***, se recorre el contenido de ***$@***, pero aunque vayamos a recorrer esta variable, es conveniente declararla explícitamente.

Ejemplo :

```
for planeta in Mercury Venus Terra Marte Jupiter Saturno
do
  echo $planeta
done
```

La lista del bucle ***for*** puede contener comodines. Por ejemplo, el siguiente bucle muestra información detallada de todos los archivos que contiene un directorio actual :

```
for archivos in *
do
  ls -l "$archivos"
done
```

Para recorrer los argumentos recibidos por el script, lo correcto es utilizar "***$@***" entrecomillado, tanto ***$**** como ***$@*** sin entrecomillar interpretan mal los argumentos con espacios, y "***$****" entrecomillado considera un sólo elemento a todos los argumentos.

Ejemplo :

$ vi argumentos.sh

```
for arg in "$*"
do
  echo "Elemento: $arg"
done

for arg in "$@"
do
  echo "Elemento: $arg"
done
```

$ chmod +x argumentos.sh
$ ./argumentos.sh 1 2 3 4
Elemento: 1 2 3 4
Elemento: 1
Elemento: 2
Elemento: 3
Elemento: 4

Como vemos con "***$****" los parámetros pasados los mantiene en una misma linea, mientras que con "***$@***" lo tenemos cada parámetros en distintas lineas.

La sentencia select

Nos permite generar fácilmente un menú simple. Su sintaxis es la siguiente :

```
select variable [in lista]
do
  Sentencias que usan $variable
done
```

Observamos que tiene la misma sintaxis que el bucle ***for***, excepto por la keyword ***select*** en vez de ***for***. De

hecho si omitimos ***in lista*** también se usa por defecto ***$@***. Lo que genera la sentencia es un menú con los elementos de ***lista***, donde asigna un número. El valor elegido se almacena en variable, y el número elegido en la variable ***REPLY***. Una vez elegida una opción por parte del usuario, se ejecuta el cuerpo de la sentencia y el proceso se repite en un bucle infinito.

El bucle se puede abandonar usando la sentencia ***break***, se usa para abandonar un bucle, se puede usar tanto en el caso de ***select***, como también en los bucles ***for***, ***while*** y ***until***.

$ vi select.sh

```
options1="option1 option2 quit"
echo opciones : $options1
select opt in $options1 ; do
  echo $opt
  if [ "$opt"="quit" ]; then
    exit
  fi
done
```

$ chmod +x select.sh
$ ./select.sh

```
opciones : option1 option2 quit
1) option1
2) option2
3) quit
#?
```

Como vemos el prompt que nos muestra es ***#?*** este lo podemos cambiar ya que es la variable ***PS3*** este mismo lo podríamos modificar de la siguiente manera :

$ vi select.sh

```
PS3='Elija las siguientes opciones : '
options1="option1 option2 quit"
echo opciones : $options1
select opt in $options1 ; do
  echo $opt
  if [ "$opt"="quit" ]; then
    exit
  fi
done
```

$ ./select.sh

```
opciones : option1 option2 quit
1) option1
2) option2
3) quit
Elija las siguientes opciones :
```

Opciones de la línea de comandos, expresiones aritméticas y arrays

A lo largo de este tema aprenderemos a realizar dos operaciones frecuentes en la programación de scripts: La primera es recibir opciones (precedidas por un guión) en la línea de comandos. La segunda es aprender a realizar operaciones aritméticas con las variables, de esta forma superaremos la limitación de que estábamos teniendo, de que todas nuestras variables sólo contenían cadenas de caracteres.

Opciones de la línea de comandos

Es muy típico que los comandos UNIX tengan el formato :

comando [-opciones] argumentos

Las opciones suelen preceder a los argumentos y tienen un guión delante y una abreviatura del significado de la opcion, mientras que se ponen dos guiones cuando indicamos la opción completa es decir :

$ ls -R

o

$ ls --recursive

Supongamos el siguiente ejemplo donde le pasamos argumentos :

./hacer	**-o**	**hola**	**mundo.txt**
$0	$1	$2	$3

El problema está en que normalmente las opciones son “opcionales”, es decir, pueden darse o no, con lo que el script que procesa la opción anterior debería tener la forma :

```
if [ $1 = -o ]; then
  Ejecuta la operación con $2 y $3
else
 Ejecuta la operación con $1 y $2
fi
```

En consecuencia, cuando el número de opciones crece, la programación de script se vuelve engorrosa.

La sentencia shift

Esta sentencia nos permite solucionar este problema.

shift [n]

Donde ***n*** es el número de desplazamiento a la izquierda que queremos hacer con los argumentos. Si se omite ***n*** por defecto vale 1. En el ejemplo anterior si ejecutamos el comando ***shift 1***, ***$1*** pasará a ser ***hola.txt*** y ***$2*** pasa a ser ***mundo.txt***, y la opción se pierde.

```
if [ $1 = -o ]; then
  Procesa -o
  shift
fi
Ejecuta la operación con $1 y $2
```

El siguiente ejemplo muestra mejor el uso de ***shift***, donde veremos que recibe tres parámetros posibles .

```
while [ -n "$(echo $1 | grep '^-')" ]
do
  case $1 in
        -a) echo "Valor : -a"
           ;;
        -b) echo "Valor : -b"
           ;;
        -c) echo "Valor : -c"
           ;;
        *)  echo "forma de uso : $0 [-a] [-b] [-c] args..."
           exit 1
           ;;
       esac
       shift
done

echo "Argumentos : "
```

La condición del bucle busca la expresión regular '^-' significa cualquier cosa que empiece (^) por un guión (-), cada vez que procesamos una opción ejecutamos ***shift*** para desplazar una vez los argumentos.

También puede ser que cada parámetro tenga su propio argumento, por ejemplo que la opción -b recibe a continuación un argumento. En ese caso deberíamos modificar el bucle anterior para leer este argumento :

```
while [ -n "$(echo $1 | grep '^-')" ]
do
  case $1 in
        -a) echo "Valor : -a"
           ;;
        -b) echo "Valor : -b $2"
           shift
           ;;
        -c) echo "Valor : -c"
           ;;
        *)  echo "forma de uso : $0 [-a] [-b argumento] [-c] args..."
           exit 1
           ;;
       esac
       shift
done

echo "Argumentos : "
```

En este caso ***$2*** es el argumento de la opción -***b*** y realizamos un ***shift*** para que pase a la próxima opción, esto lo hace siempre y cuando pasemos como opción -***b***.

<u>El comando interno getopts</u>

getopts nos permite procesar opciones de línea de comandos más cómodamente, el problema de esto es que solo permite una sola letra, es decir -***a*** y no ***--activo***. Además ***getopts*** nos permite procesar opciones cuando están agrupadas (p.e. -***abc*** en vez de -***a*** -***b*** -***c***) y sus argumentos de opciones cuando estos no están separados de la opción por un espacio (-***barg*** en vez de -***b argumento/s***) no es conveniente utilizarlo de esta forma siempre conviene tener un espacio es decir -***b argumento/s***. Generalmente se usa con un bucle ***while***. El comando ***getopts*** recibe dos argumentos:

- Es una cadena con las letras de las opciones que vamos a reconocer. Si la opción tiene un

argumento se precede por dos puntos (:). El argumento de la opción lo podemos leer consultando la variable de entorno ***OPTARG***. El comando ***getopts*** coge una de las opciones de la línea de comandos y la almacena en una variable cuyo nombre se da como ***segundo argumento***. Mientras que el comando encuentra opciones devuelve el código de terminación 0, cuando no encuentra más opciones devuelve el código de terminación 1.

OPTARG = son los argumentos que se pasan.
OPTIND = es el numero del primer argumento a procesar.

```
while getopts ":ab:c" opt
do
  case $opt in
        -a) echo "Valor : -a"
           ;;
        -b) echo "Valor : -b $OPTARG"
           ;;
        -c) echo "Valor : -c"
           ;;
        *)  echo "forma de uso : $0 [-a] [-b argumento] [-c] args..."
           exit 1
           ;;
      esac
done
shift $(($OPTIND -1))
echo "Argumentos : "
```

Observamos que no hace falta que ponemos ***shift*** dentro del bucle ya que ***getopts*** es lo suficientemente inteligente como para devolvernos cada vez una opción distinta. Lo que sí hemos hecho al final del bucle es desplazar tantas veces como argumentos con opciones hayamos encontrado. Para eso ***getopts*** almacena en la variable de entorno ***OPTIND*** el número del primer argumento a procesar.

VERRRRRRRRRRRRRRR

Variables con tipo

Hasta ahora las variables sólo podían contener cadenas de caracteres, después se introdujo la posibilidad de asignar atributos a las variables que indican, por ejemplo que son enteras o de sólo lectura. Para fijar los atributos de las variables tenemos el comando interno ***declare***, el cual tiene la siguiente sintaxis:

declare [-afFirx] [-p] name[=value] …

La siguiente tabla muestra las opciones dada por este comando, para activar un atributo se precede la opción por un guión -, con lo que para desactivar un atributo decidieron preceder la opción por un +.

Opción	***Descripción***
-a	La variable es de tipo array.
-f	Mostrar el nombre e implementación de las funciones.
-F	Mostrar sólo el nombre de las funciones.
-i	La variable es de tipo entero.
-r	La variable es de sólo lectura.
-x	Exporta la variable (equivalente a ***export***).

Si solo escribimos el comando ***declare*** sin ningún argumento veremos que nos muestra todas las variables de

entorno.

```
$ declare
BASH=/bin/bash
BASHOPTS=checkwinsize:cmdhist:expand_aliases:extglob:extquote:force_fignore:histappend:inter
active_comments:progcomp:promptvars:source
path
BASH_ALIASES=()
BASH_ARGC=()
BASH_ARGV=()
BASH_CMDS=()
BASH_COMPLETION=/etc/bash_completion
BASH_COMPLETION_COMPAT_DIR=/etc/bash_completion.d
BASH_COMPLETION_DIR=/etc/bash_completion.d
BASH_LINENO=()
…
```

*Si usamos la opción -**f** nos muestra sólo los nombres de funciones y su implementación*

```
$ declare -f
_ImageMagick ()
{
    local cur prev;
    _get_comp_words_by_ref cur prev;
    case $prev in
        -channel)
...
```

*Y si usamos la opción -**F** nos muestra sólo los nombres de las funciones existentes.*

```
$ declare -F
declare -f _ImageMagick
declare -f __expand_tilde_by_ref
declare -f __get_cword_at_cursor_by_ref
declare -f __git_aliased_command
declare -f __git_aliases
declare -f __git_complete_file
…
```

Las variables que se declaran con ***declare*** dentro de una función son variables locales a la función, de la misma forma que si hubiésemos usado el modificador ***local***.

Por ejemplo el siguiente script muestra como declaramos anteriormente las variables que en definitiva eran de tipo cadenas de caracteres :

```
$ var1=5
$ var2=4
$ resultado=$var1*$var2
$ echo $resultado
5*4
```

Ahora realizamos lo mismo pero utilizando el comando declare -i para indicar que las variables que declaramos son de tipo entero :

```
$ declare -i var1=5
$ declare -i var2=4
$ declare -i resultado
```

$ resultado=$var1*$var2
$ echo $resultado
20

Para saber que tipo es la variable que esta declarada con el comando ***declare*** utilizamos la opción ***-p***.

$ declare -p resultado
declare -i resultado="20"

La opción ***-x*** es equivalente a usar el comando ***export*** sobre la variable, ambas son formas de exportar una variable de entorno.

La opción ***-r*** declara a la variable como de sólo lectura, con lo que a partir de ese momento no podremos modificar ni ejecutar ***unset*** sobre ella.

$ declare -r resultado
$ resultado=$var1*20
bash: resultado: variable de sólo lectura

Existe otro comando interno llamado ***readonly*** que nos permite declarar variables de sólo lectura, pero que tiene más opciones que ***declare -r***. Con ***readonly -p*** nos muestra todas las variables de sólo lectura :

$ readonly -p
declare -r
BASHOPTS="checkwinsize:cmdhist:expand_aliases:extglob:extquote:force_fignore:histappend:interactive_comments:progcomp:promptvars:sourcepath"
declare -ir BASHPID=""
declare -r BASH_COMPLETION="/etc/bash_completion"
declare -r BASH_COMPLETION_COMPAT_DIR="/etc/bash_completion.d"
declare -r BASH_COMPLETION_DIR="/etc/bash_completion.d"
declare -ar BASH_VERSINFO='([0]="4" [1]="1" [2]="5" [3]="1" [4]="release" [5]="x86_64-pc-linux-gnu")'
declare -ir EUID="1000"
declare -ir PPID="3118"
declare -r SHELLOPTS="braceexpand:emacs:hashall:histexpand:history:interactive-comments:monitor"
declare -ir UID="1000"
declare -ir resultado="20"

Además se nos indica si la variable es de tipo ***array*** (***-a***) o ***entero*** (***-i***).

Usando la opción ***-f*** podemos hacer una función de sólo lectura (que el usuario no pueda modificar la función). Por ejemplo :

$ readonly -f mi_funcion

Muchas veces se ataca un sistema modificando una función que se sabe que va a ejecutar un script en modo superusuario (haciendo que la función haga algo distinto, o algo más de lo que hacía originalmente). Para evitar este ataque las funciones que van a ejecutarse en modo superusuario se deben de definir sólo dentro del script que las usa, aunque a veces se necesitan llamar desde fuera y es recomendable protegerlas con la opción ***-r***, ya que una vez que una función se marca como de sólo lectura ya no se puede quitar ese permiso. Es decir si usamos la opción ***-n*** para quitar este atributo, en principio el comando parece funcionar :

$ readonly -n mi_funcion

Pero si luego intentamos redefinir la función se producirá un error indicando que la función sigue siendo de sólo lectura.

Expresiones aritméticas

Las expresiones aritméticas van encerradas entre ***$ ((*** y ***))***, también podemos encerrarlas ente ***$ [*** y ***]***, pero está forma está desestimada por Bash con lo que no se recomienda usarla.

Ejemplo :

$ echo "$((365 - $(date +%j))) dias para el 31 de Diciembre"

Nos permite usar los operadores relacionales (<, >, <=, >=, ==, !=) y lógicos de C (!, &&, ||) interpretando, al igual que C, el 1 como cierto y el 0 como falso.

El comando interno let

Nos permite asignar el resultado de una expresón aritmética a una variable. Tiene la siguiente sintaxis :

let var=expresion

expresion es cualquier expresión aritmética y no necesita estar encerrada entre $((...)). La diferencia con el comando ***declare -i***, no crea una variable de tipo entero, sino una variable de tipo cadena de caracteres normal.

Ejemplo :

$ let a=4*3
$ declare -p a
declare – a=”12”
$ echo $a
12

Sentencias de control de flujo aritméticas

Pueden usarse en las distintas sentencias de control de flujo, en cuyo caso la expresión va entre doble paréntesis, pero sin el ***$***.

if ((expresión aritmética)); then
* cuerpo*
fi

O el ***while*** aritmético tendría la forma :

while ((expresión aritmética))
do
* cuerpo*
done

Ejemplo :

Este programa genera un número aleatorio de 0 a 99, y nos pregunta hasta que acertemos, o fallemos 10 veces. Se utiliza la variable ***$RANDOM*** *la cual cada vez que se lee devuelve un número distinto.*

$ vi obtener.sh

let intentos=10
let solucion=$RANDOM%100
while ((intentos-->0))
do

```
  read -p “Indique un numero: “ numero
  if ((numero==solucion)); then
    echo “Excelente has ganado...”
    exit
  elif ((numero<solucion)); then
    echo “Es mayor”
  else
    echo “Es menor”
  fi
done
echo “Perdiste por superar los 10 intentos.”
```

$ chmod +x obtener.sh
$ ./obtener.sh

Observamos que al ser una expresión aritmética en la condición del ***if*** se usa == y no =. La expresión ***intentos--*>0** primero comprueba que intentos sea mayor a 0, y luego le aplica el operador -- que decrementa en uno ***intentos***.

Arrays

Bash introdujo los ***arrays*** en la versión 2.0. Actualmente Bash sólo soporta trabajar con ***arrays*** unidimensionales.

Para declarar un ***array*** podemos usar como vimos anteriormente ***declare -a*** de la siguiente forma :

$ declare -a A
$ declare -p A
declare -a A='()'

No hace falta declarar un ***array*** con ***declare -a*** también podemos crearlo de la siguiente forma :

$ DIAS=(Lunes Martes Miercoles Jueves Viernes Sabado Domingo)
$ declare -p DIAS
declare -a DIAS='([0]="Lunes" [1]="Martes" [2]="Miercoles" [3]="Jueves" [4]="Viernes" [5] ="Sabado" [6]="Domingo")'

Los elementos del ***array*** empiezan desde ***0*** como vemos ***[0]=”Lunes”***, aunque podemos cambiar los índices de los elementos indicándolos explícitamente:

$ DIAS=([1]=Lunes [2]=Martes [3]=Miercoles [5]=Jueves [4]=Viernes [6]=Sabado [7]=Domingo)
$ declare -p DIAS
declare -a DIAS='([1]="Lunes" [2]="Martes" [3]="Miercoles" [4]="Viernes" [5]="Jueves" [6] ="Sabado" [7]="Domingo")'

No hace falta suministrar los elementos en orden, ni suministrarlos todos, los índices donde no colocamos un valor, simplemente valdrán “cadena nula”.

$ COLORES=([5]=Rojo Azul Blanco Negro Marron)
$ declare -p COLORES
declare -a COLORES='([5]="Rojo" [6]="Azul" [7]="Blanco" [8]="Negro" [9]="Marron")'

Otra forma de rellenar un ***array*** es la siguiente :

$ LISTADO=($(ls -1))
$ declare -p LISTADO
declare -a LISTADO='([0]="archivo_01.sh" [1]="archivo_1.tmp" [2]="archivo_2.tmp" [3]

="argumentos.sh" [4]="clientes.txt" [5]="cond2.sh" [6]="cond.sh" [7]="config.conf" [8] ="desplaza_parametros.sh" [9]="du-.sh" [10]="eje2.sh" [11]="eje.sh" [12]="entrada.sh" [13] ="estado_archivo.sh" [14]="funcion_01.sh" [15]="funcion_02.sh" [16]="funcion.sh" [17] ="get_modulos.sh" [18]="imprimir.sh" [19]="ir.sh" [20]="ocupa.sh" [21]="parametros_01.sh" [22]="piladir.sh" [23]="select.sh" [24]="shift.sh" [25]="solucion.sh" [26]="tojpg.sh" [27] ="tomar_archivo.sh" [28]="ver_cd.sh" [29]="ver.sh" [30]="x.sh")'

Para acceder a los elementos usamos el operador corchete ***[ ]*** para indicar el índice del elemento a acceder, y en este caso es obligatorio encerrar entre llaves ***{ }*** la variable :

$ echo ${LISTADO[5]}
cond2.sh

Si no indicamos el índice de elemento, por defecto nos trae el elemento de índice ***0***.

$ echo $LISTADO
archivo_01.sh

Ahora veremos la forma que no hay que hacer para indicar un elemento determinado, por esa misma razón hay que ponerlo entre llaves :

$ echo $LISTADO[5]
LISTADO[5]

Otra forma de declarar el ***array*** es introducir valores independientemente :

$ A[2]=Casa
$ A[0]=Avion
$ A[1]=Tren
$ declare -p A
declare -a A='([0]="Avion" [1]="Tren" [2]="Casa")'

Podemos usar los índices especiales ***** y **@**, los cuales retornan todos los elementos del ***array*** de la misma forma que lo hacen los parámetros posicionales. Cuando no están encerrados entre comillas dobles ambos devuelven una cadena con los elementos separados por espacio, pero cuando se encierran entre comillas dobles **@** devuelve una cadena con los elementos separados por espacio, y ***** devuelve una cadena con los elementos separados por el valor de ***IFS***.

$ IFS=,
$ echo ${A[*]}
Avion Tren Casa

$ echo ${A[@]}
Avion Tren Casa

$ echo "${A[@]}"
Avion Tren Casa

$ echo "${A[*]}"
Avion,Tren,Casa

También podemos mostrar su contenido con un bucle ***for***.

```
for datos in ${A[*]}
do
  echo $datos
done
```

Los elementos de un ***array*** a lo que no se asigna valor tienen una cadena nula, con lo que el bucle anterior sólo imprimiría los elementos que existan en el ***array A***.

Podemos consultar la longitud de un ***array*** usando la forma ***${#array[@]}***.

$ echo ${#A[@]}
3

nos devuelve sólo el número de posiciones ocupadas.

De la siguiente forma si le indicamos un índice nos dará la longitud del contenido del índice.

$ echo ${A[1]}
Tren

$ echo ${#A[1]}
4

Si queremos saber que elementos no son nulos en un ***array*** podemos usar la forma ***${!array[@]}***, esto esta permitido a partir de la versión 3.0 de Bash.

$ echo ${!A[1]}
0 1 2

Si asignamos un ***array*** compuesto a otro ***array***, se pierden los valores existentes, y se asignan los nuevos valores.

$ echo ${A[@]}
Avion Tren Casa

$ A=(Hola “Que tal” Adios)
$ declare -p A
declare -a A='([0]="Hola" [1]="Que tal" [2]="Adios")'

Podemos borrar un elemento del array :

$ unset A[2]

$ echo ${A[@]}
declare -a A='([0]="Hola" [1]="Que tal")'

O bien podemos borrar todo el array :

$ unset A
$ declare -p A
bash: declare: A: no se encontró

Por último vamos a comentar que algunas de las variables de entorno de Bash son ***arrays***. Por ejemplo ***BASH_VERSINFO*** es un ***array*** de sólo lectura con información sobre la instancia actual de Bash :

$ declare -p BAS_VERSINFO
declare -ar BASH_VERSINFO='([0]="4" [1]="1" [2]="5" [3]="1" [4]="release" [5]="x86_64-pc-linux-gnu")'

Otro muy interesante es ***PIPESTATUS*** es un ***array*** que contiene el código de terminación del ultimo comando ejecutado :

$ ls -l
$ declare -p PIPESTATUS
declare -a PIPESTATUS='([0]="0")'

$ lsx
$ declare -p PIPESTATUS
declare -a PIPESTATUS='([0]="127")'

$ ls -l | more
$ declare -p PIPESTATUS
declare -a PIPESTATUS='([0]="0" [1]="0")'

Redirecciones

Nosotros anteriormente ya vimos este tema los principales operadores de redirección que son : >, <, >>, ***2***> y |, pero esta parte abarca otro grupo de comandos que muchas veces aparecen relacionados con problemas complejos de redirección.

En la siguiente tabla explicamos todos estos operadores :

Operador	***Descripción***
cmd1 | cmd2	Pipe; Toma la salida estándar de ***cmd1*** y la envía a la entrada estándar de ***cmd2***.
>fichero	Redirige la salida estándar del programa al ***fichero***.
<fichero	Redirige el contenido de ***fichero*** a la entrada estándar del programa.
>>fichero	Redirige la salida estándar del programa al ***fichero***. Añade esta a ***fichero*** si éste ya existe.
>|fichero	Redirige la salida estándar del programa al ***fichero***. Sobrescribe a éste incluso si la opción ***noclobber*** está activada.
<>fichero	Usa a ***fichero*** tanto para la entrada estándar como para la salida estándar.
n<>fichero	Usa a ***fichero*** tanto para la entrada estándar como para la salida del descriptor de fichero ***n***.
<<etiqueta	Fuerza a que la entrada a un comando sea la ***stdin*** hasta encontrar ***etiqueta***. Esto se llama ***here document***.
<<<texto	Envían textoa la entrada estándar del comando, es una variante de los ***here document*** son los ***here string***.
n>fichero	Envía el valor del descriptor de fichero ***n*** al ***fichero***.
n>>fichero	Envía el valor del descriptor de fichero ***n*** al ***fichero***. Añade al final del fichero si éste existe.
n>|fichero	Envía el valor del descriptor de fichero ***n*** al ***fichero***. Sobrescribe a éste incluso si la opción ***noclobber*** está activada.
n>&	Enlaza la salida estándar en el descriptor de ***fichero n***.
n<&	Enlaza la entrada estándar en el descriptor de ***fichero n***.
n>&m	Enlaza el descriptor de salida ***m*** en el descriptor de ***fichero n***.
n<&m	Enlaza el descriptor de entrada ***m*** en el descriptor de ***fichero n***.
&>fichero	Redirige la salida estándar y la salida de error estándar al ***fichero***.
<&-	Cierra la entrada estándar.
>&-	Cierra la salida estándar.
n>&-	Cierra el descriptor de salida ***n***.

n<&-	Cierra el descriptor de entrada ***n***.

Ejemplos :

En este ejemplo si no existe el archivo lo crea dejándolo en 0 bytes y si existe lo pisa dejándolo también en 0 bytes.

$ > nuevo.txt

En bash existe la opción ***noclobber*** que por defecto no está activa (y se puede activar con el comando ***set -o noclobber***) que hace que si intentamos sobrescribir un fichero con el operador >(redirigir a un fichero existente) el shell produzca un error y no nos deje ejecutar esta operación. Podemos forzar que se sobrescriban los ficheros, incluso con esta opción activada, con el operador >|.

Los descriptores de fichero

Hemos visto en capítulos anteriores que tenemos tres descriptores principales que el ***0*** para la entrada estándar (***stderr***), el ***1*** para la salida estándar (***stdout***), y el ***2*** para la salida de errores estándar (***stderr***).

Nos quedan los números del ***3*** al ***9*** para abrir descriptores adicionales, el descriptor ***5*** puede dar problemas ya que cuando el shell ejecuta un subshell el subproceso hereda este descriptor. Esto se hace así porque a veces es útil asignar un número de descriptor adicional a los descriptores estándar como si fueran una copia adicional de este enlace. Otras veces resulta útil asignar un número de descriptor adicional a un fichero al que luego nos vamos a referir por su descriptor.

El operador ***n>m&*** enlaza el descriptor ***n*** en el descriptor de salida ***m***.

$ ls -yz >> resultado.log
ls: invalid option -- 'y'
Pruebe `ls --help' para más información.

$ ls -yz >> resultado.log 2>&1

El resultado de la linea ilegal la envía por ***stdout*** (***1***), en vez de ***stderr*** (***2***). Si lo hubiéramos ejecutado al revés es decir ***ls 1>&2*** lo que habríamos hecho es enviar el listado de directorios (***stdout***) a la salida de errores estándar (***stderr***).

El operador ***n<&m*** enlaza el descriptor ***n*** en el descriptor de entrada ***m***.

$ cat 3< clave.h 0<&3

Enlaza el descriptor de entrada ***3*** con el fichero ***clave.h***, y luego cambia ***stdin*** para que en vez de leer de teclado, lea del descriptor de fichero ***3***, con lo que el contenido de ***clave.h*** actúa como entrada al comando ***cat***.

Otro ejemplo es :

$ ls 3>mis.ficheros 1>&3

Asigna el descriptor de fichero de salida ***3*** al fichero ***mis.ficheros***, y después enlaza la salida estándar con el descriptor ***3***, con lo que la salida de ***ls*** va al fichero ***mis.ficheros***.

También podemos usar el operador ***n<>fichero*** para que ***n*** sea a la vez un descriptor de entrada y de salida para fichero.

$ cat clientes.txt
Juan
Pedro

Ana

$ sort 3<>clientes.txt 0<&3 1>&3
$ cat clientes.txt
Juan
Pedro
Ana
Ana
Juan
Pedro

Observamos que los clientes ordenados no sobreescriben a los existentes, sino que se escriben a continuación. Esto se debe a que al leerlos, el puntero a fichero se ha colocado al final, y cuando después escribimos estamos escribiendo al final de éste.

El operador ***n*<>*fichero*** se puede abreviar como ***<>fichero***, en cuyo caso ***stdin*** se convierte en un descriptor de fichero de entrada/salida con fichero como fuente y destino. Es decir, el comando anterior se podría haber escrito como ***sort <>cleintes.txt 1>&0***. El final del comando ***1>&0*** se usa para que ***stdout*** se envía a ***stdin*** (el fichero ***clientes.txt***).

Hay veces que queremos redirigir ambas salidas : ***gcc *.c 1>errores.log 2>errores.log*** una forma mas fácil de abreviar esto es ***gcc *.c &>errores.log***.

El comando exec

Este comando nos permite cambiar las entradas y salidas de un conjunto de comandos (de todos los que se ejecutan a partir de él).

$ vi exec.sh

```
exec 6<&0                       # Enlaza el descriptor 6 a stdin.
                                # Esto se hace para salvar stdin
exec < data-file                # Reemplaza stdin por el
                                # fichero data-file
read a1                         # Lee la primera línea de data-file
read a2                         #Lee la segunda línea de data-file

echo
echo "Líneas leidas del fichero"
echo "------------------------------"
echo $a1
echo $a2

exec 0<&6 6<&-                  # Restaura stdin
```

Here documents

El operador ***<<etiqueta*** fuerza a que la entrada a un comando sea la ***stdin*** hasta encontrar etiqueta. Este texto que va hasta encontrar etiqueta es lo que se llama ***here document***.

Ejemplo :

$ cat >> texto.txt <<EOF
> Hola.
> Esto es un mensaje para ver como
> funciona here documents.
>EOF

Esto se puede entre otras cosas utilizar en una conexión de un servidor ftp por ejemplo :

$ vi conexion_ftp.sh

```
# Script que sube un fichero a un servidor de FTP
# Recibe los siguientes argumentos
# $1 Fichero a subir
# $2 Servidor FTP
# $3 Nombre de usuario
# $4 Password
# $5 Directorio destino

ftp << FIN
open $1
$2
$3
cd $4
binary
put $5
quit
FIN
```

Una variante de los ***here document*** son los ***here string***, los cuales se llevan a cabo el operador <<<***texto***, y que simplemente envían texto a la entrada estándar del comando.

Por ejemplo, si queremos añadir una línea al principio del fichero ***texto.txt*** y guardarlo en ***nuevo_texto.txt*** podemos usar lo siguiente :

$ titulo="Mi titulo de ejemplo"
$ cat - texto.txt <<<$titulo > nuevo_texto.txt

El guión (-) como argumento de ***cat*** (al igual que en otros comandos como ***grep*** o ***sort***) indica que ahí va lo que reciba por la entrada estándar (el ***here string*** en nuestro ejemplo). Obsérvese que el ***here string*** (al igual que el ***here document***) puede contener variables, en cuyo caso las expanden por su valor antes de pasar el ***here document*** al comando.

Entrada y salida de texto

En esta sección comentaremos detalladamente los comandos ***echo***, ***printf*** y ***read*** que nos permite realizar operaciones de entrada/salida que requiere un script.

El comando interno echo

Simplemente escribe en la salida estándar los argumentos que recibe.

$ echo "Hola\nAdios"
Hola\nAdios

$ echo -e "Hola\nAdios"
Hola
Adios

La tabla siguiente muestra las opciones que podemos utilizar con el comando ***echo***.

Opción	***Descripción***
-e	Activa la interpretación de caracteres precedidos por el carácter de escape.

-E	Desactiva la interpretación de caracteres precedidos por el carácter de escape. Es la opción por defecto.
-n	Omite el carácter \n al final de la línea (es equivalente a la secuencia de escape \c).

La siguiente tabla muestra las secuencias de escape que acepta echo (cuando se le pasa la opción -e). Para que estas tengan éxito deben de encerrarse entre comillas simples o dobles, ya que sino el carácter de escape \ es interpretado por el shell y no por ***echo***.

Secuencia de escape	***Descripción***
\a	Produce un sonido “poom” (alert).
\b	Borra hacía atrás (backspace).
\c	Omite el \n al final (es equivalente a la opción -n). Debe colocarse al final de la línea.
\f	Cambio de página de impresora (formfeed).
\n	Cambio de línea.
\r	Retorno de carro.
\t	Tabulador.

El siguiente ejemplo repite un bucle de 0 a 9, con la opción ***r** lo que hace es retorna el cursor en el terminal sin introducir un cambio de línea (como lo hace el ***n**).

```
$ for ((i=0;i<10;i++))
> do
> echo -n "Procesado $i"
> sleep 1
> echo -ne "\r"
> done
```

El comando interno printf

Con el comando ***printf*** podemos formatear la salida por pantalla mientras que con ***echo*** no nos deja.

```
$ printf “Hola Mundo”
Hola Mundo
```

printf por defecto interpreta las secuencias de escape.

```
$ printf “Hola Mundo\n”
Hola Mundo
```

El formato de este comando es muy parecido al de la función ***printf()*** del lenguaje ***C***.

```
printf cadenaformato [argumentos]
```

En la siguiente tabla describimos los formatos que podemos utilizar los cuales empiezan por ***%***.

Especificador	***Descripción***
%c	Imprime el primer carácter de una variable cadena.
%d	Imprime un número decimal.
%i	Igual que %d (integer).

%e	Formato exponencial ***b.precisione[+-]e***
%E	Formato exponencial ***b.precisione[+-]e***
%f	Número en coma flotante ***b.precision***
%g	El que menos ocupa entre %e y %f
%G	El que menos ocupa entre %E y %f
%o	Octal sin signo
%s	Cadena (string)
%b	Interpreta las secuencias de escape del argumento cadena
%q	Escribe el argumento cadena de forma que pueda ser usado como entrada a otro comando
%u	Número sin signo (unsigned)
%x	Hexadecimal con letras en minúscula
%X	Hexadecimal con letras en mayúscuña
%%	% literal

Ejemplo :

$ n=30
$ nombre=Marcos
$ printf "El cliente %d es %s\n" $n $nombre
El cliente 30 es Marcos

$ printf "|%10s|\n" Hola
| Hola|

Si queremos indicar un máximo para la cadena debemos de usar el campo ***precisión*** :

$ printf "|%-10.10s|\n" "Marcos Pablo"
|Marcos Pab|

El flag **–** se usa para indicar que queremos alienar a la izquierda :

$ printf "|%-10s|\n" Hola
|Hola |

Dos variantes de %s son %b y %q. El especificador de formato %b hace que se interpreten las secuencias de escape de los argumentos. Es decir :

$ printf "%s\n" "Hola \nAdios"
Hola \nAdios

$ printf "%b\n" "Hola \nAdios"
Hola
Adios

La siguiente tabla muestra los flags de los especificadores de formato para números.

Flag	***Descripción***
+	Preceder por + a los números positivos.
Espacio	Preceder por un espacio a los números positivos.

0	Rellenar con ***0***'s a la izquierda.
#	Preceder por ***0*** a los números en octal ***%o*** y por ***0x*** a los números en hexadecimal ***%x*** y ***%X***.

Ejemplo :

$ printf "|%+10d|\n" 56
| +56|

$ printf "|%010d|\n" 56
|0000000056|

$ printf "|%x|\n" 56
|38|

$ printf "|%#x|\n" 56
|0x38|

El comando interno read

La sintaxis de este comando es :

read var1 var2 …

Esta sentencia lee una línea de la entrada estándar y la parte en palabras separadas por el símbolo que indique la variable ***IFS*** (por defecto espacio o tabulador).

$ read var1 var2 var3
Esto es una prueba

$ echo $var1
Esto

$ echo $var2
es

$ echo $var3
una prueba

Si omitimos las variables, la línea entera se asigna a la variable ***REPLY***.

El uso de ***read*** rompe esta forma de programar, con lo que se recomienda usarlo con moderación. A lo mejor resulta más conveniente pedir un dato como argumento que pedir al usuario que lo introduzca con ***read***.

Las principales opciones del comando ***read*** :

Opción	***Descripción***
-a	Permite leer las palabras como elementos de un array.
-d	Permite indicar un delimitador de fin de línea.
-e	Activa las teclas de edición de ***readline***.
-n ***max***	Permite leer como máximo ***max*** caracteres.
-p	Permite indicar un texto de prompt.
-t ***timeout***	Permite indicar un ***timeout*** para la operación de lectura.

Ejemplos :

$ read -a frase
Hola mundo nuevo

$ declare -p frase
declare -a frase='([0]="Hola" [1]="mundo" [2]="nuevo")'

La opción ***-e*** es recomendable usarla en general siempre, ya que permite que se puedan usar todas las combinaciones de teclas de ***readline*** en el prompt de ***read***.

La opción ***-n*** nos permite especificar un número máximo de caracteres a leer. Si se intentan escribir más caracteres que los indicados en esta opción simplemente se acaba la operación de ***read***.

La opción ***-p*** nos permite aportar un texto de prompt al comando que se imprime antes de pedir el dato:

$ read -p "Cual es tu nombre :
Cual es tu nombre: ***Marcos***

La opción ***-t*** nos permite dar un tiempo máximo en segundos para el prompt, momento a partir del cual se continua con el script. Esto es especialmente útil para los scripts de instalación, donde a veces el administrador no está presente esperando a que la instalación acabe.

$ read -t 10 -p "Cual es tu nombre :
Cual es tu nombre:

Los bloques de comandos

En el siguiente script veremos que estamos cambiando la entrada estándar para que ***read*** lea la entrada.

$ vi idgrupo.sh

```
function idgrupo
{
  IFS=:
  while read nombre asterisco ID resto
  do
    if [ $1 = $nombre ]; then
          echo "Nombre Grupo : $nombre"
          echo "ID Grupo     : $ID"
          echo "Resto        : $resto"
          return
        fi
  done
}

idgrupo $1 < /etc/group
```

$ chmod +x idgrupo.sh
$ ./idgrupo.sh pablo
Nombre Grupo : pablo
ID Grupo   : 1000
Resto   :

Para que esta función use como entrada estándar el fichero /etc/group necesitamos redireccionar su entrada de la siguiente forma :

$ vi idgrupo.sh

```
function idgrupo
{
 IFS=:
 while read nombre asterisco ID resto
 do
  if [ $1 = $nombre ]; then
       echo "Nombre Grupo : $nombre"
       echo "ID Grupo     : $ID"
       echo "Resto        : $resto"
       return
      fi
 done
} < /etc/group

idgrupo $1
```

Si el único que va a utilizar esa entrada es el bucle ***while***, podemos redirigir la entrada estándar durante la ejecución del bucle ***while***.

$ vi idgrupo.sh

```
function idgrupo
{
 IFS=:
 while read nombre asterisco ID resto
 do
  if [ $1 = $nombre ]; then
       echo "Nombre Grupo : $nombre"
       echo "ID Grupo     : $ID"
       echo "Resto        : $resto"
       return
      fi
 done < /etc/group
}

idgrupo $1
```

También por ultimo podemos redirigir la entrada de un conjunto de comandos creando un llamado ***bloque de comandos***, los cuales encierran un conjunto de comandos entre llaves.

$ vi idgrupo.sh

```
{
 IFS=:
 while read nombre asterisco ID resto
 do
  if [ $1 = $nombre ]; then
       echo "Nombre Grupo : $nombre"
       echo "ID Grupo     : $ID"
       echo "Resto        : $resto"
      fi
 done
} < /etc/group
```

En el siguiente ejemplo de un bloque de comandos que pasa a su salida a ***cut*** usando un pipe. En este caso, el

bloque de comandos tiene redirigida su entrada estándar para leer de ***/etc/group***, y la salida del bloque de comandos se envía a través de un pipe a ***cut***.

$ vi idgrupo.sh

```
{
 IFS=:
 while read nombre asterisco ID resto
 do
  if [ $1 = $nombre ]; then
       echo "$nombre:$asterisco:$ID:$resto"
       break
      fi
 done < /etc/group
} | cut -d ':' -f 3
```

Los comando comand, builtin y enable

Como vimos en otro capitulo cuando introducimos un comando en bash el orden de preferencia en la búsqueda del símbolo por parte de bash, mediante los comandos internos ***command***, ***builtin*** y ***enable*** nos permiten alterar este orden de preferencia.

command hace que no se busquen alias ni nombres de funciones, sólo comandos internos y comandos de fichero.

Ejemplo :

```
cd()
{
      command cd "$@"
      local ct=$?
      echo "$OLDPWD -> $PWD"
      return $ct
}
```

builtin es similar a ***command***, *pero es más restrictivo, sólo busca comandos internos.*

enable nos permite desactivar el nombre de un comando interno, lo cual permite que un comando de fichero pueda ejecutarse sin necesidad de dar toda la ruta del fichero. Podemos usar el comando ***enable -a*** para ver todos los comandos internos y si están habilitados o no

El comando interno eval

Nos permite pasar el valor de una variable al interprete de comandos para que lo ejecute.

$ eval "ls"

Hace que el interprete de comandos ejecute ***ls***.

Aunque en principio puede parecer un poco extraño, el comando ***eval*** resulta muy potente ya que el programa puede construir programas en tiempo de ejecución y luego ejecutarlo. Estrictamente hablando no necesitaríamos disponer del comando ***eval*** para hacer esto con Bash, ya que siempre podemos crear un fichero con los comandos a ejecutar y luego ejecutarlo usando ***source***. Pero ***eval*** nos evita de tener que crear un fichero.

Verrrrrrrrrrrrr

Combnaciones de teclas que envían señales

Existen varias combinaciones de teclas que actúan sobre el proceso que se esté ejecutando en foreground en el terminal.

Con el comando ***stty*** podemos crear nuevas combinaciones de teclas que envíen señales al proceso el foreground usando ***stty señal ^letra***. Donde señal es el nombre de la ***señal*** en minúscula y sin el prefijo ***SIG***. Por ejemplo para que ***Ctrl+Q*** produzca la señal ***SIGQUIT*** podemos usar :

$ stty quit ^Q

Capturar señales desde un script

Comando interno trap

El comando interno trap, el cual tiene el siguiente formano :

trap cmd sig1 sig2 …

cmd es el comando que queremos ejecutar al capturar alguna de las señales ***sig1 sig2*** … Lógicamente ***cmd*** puede ser una función o un script, y las señales se pueden dar por número o por nombre. También se puede ejecutar sin argumentos, en cuyo caso nos da la lista de ***traps*** que están fijados.

Ejemplo :

```
trap “echo 'Pulsastes Ctrl+C'” INT

while true
do
  sleep 60;
  echo “Cambio de minuto”
done
```

Observamos que al recibir la señal ***SIGINT*** bash se la pasa al comando ***sleep***, con lo que éste acaba, pero luego se ejecuta el ***trap*** del script, y por esta razón el script no acaba.

Para pararlo ahora con ***Ctrl+C*** tenemos un problema ya que hemos cambiado la opción por defecto, que es ***Term***, por imprimir un mensaje. Para terminar el script puede pararlo con ***Ctrl+Z*** y luego hacerle un ***kill*** (que envía la ***SIGTERM*** no la ***SIGINT***):

```
Cambio de minuto
^Z
[1]+  Detenido              sleep 60
$ kill %1
$ jobs
[1]+  Terminado             sleep 60
```

Traps y funciones

Como bien sabemos, las funciones se ejecutan en el mismo proceso que el script que las llama, en consecuencia dentro de una función se puede detectar un ***trap*** fijado por el script, y viceversa, un ***trap*** fijado por una función sigue activo cuando ésta termina.

```
Function fijatrap
{
 trap “echo 'Pulsaste Ctrl+C!'” INT
}
```

```
fijatrap
while true
do
  sleep 60;
  echo "Cambio de minuto"
done
```

Ignorar señales

Si lo que queremos es ignorar una señal, simplemente tenemos que pasar una cadena vacía ("" ó '') en el argumento ***cmd*** a ***trap***.

En el próximo ejemplo veremos que queremos ignorar la señal ***SIGHUP*** (***hangup***), la cual recibe un proceso cuando su padre termina (p.e. el shell) y produce que el proceso hijo también termine.

```
function ignorarhup
{
 trap "" HUP
 eval "$@"
 trap – HUP
}
```

$ ignorarhup du -h /usr> ocupa.txt

Actualmente existe un comando en ***UNIX*** que hace esto mismo, que es el script ***nohup*** cuya implementación.

Existe un comando interno, llamado ***disown***, que recibe como argumento un job y elimina el proceso de la lista de jobs controlados por el shell (con lo que no recibiría la señal ***SIGHUP*** cuando el shell que lo lanzó termine).La opcioón -***h*** (hook) de este comando realiza la misma función de ***nohup***, manteniendo al proceso en la lista de jobs, pero no enviándole la señal ***SIGHUP*** cuando el shell termina. También existe la opción -***a*** (all) que libera a todos los procesos en background de la lista de jobs del shell.

Corutinas

Es un conjunto de dos o más procesos ejecutados concurrentemente por el shell, y opcionalmente con la posibilidad de comunicarse entre ellos.

Un ejemplo de esto es cuando realizamos ***ls | more***, el shell llama a un conjunto de primitivas, o llamadas al sistema.

Si no se necesita que dos procesos se comuniquen ente ellos, la forma de ejecutarlos es más sencilla. Por ejemplo, si queremos lanzar los procesos comer y beber como corutinas, podemos hacer el siguiente script:

```
comer &
beber
```

Si ***beber*** es el último proceso en acabar, esta solución funciona, pero si ***comer*** sigue ejecutando después de que acabe de ejecutarse el script, comer se convertiría en un proceso huérfano (también llamado zombie).

En general esto es algo indeseable, y para solucionarlo existe el comando interno ***wait***, el cual para al proceso del script hasta que todos los procesos de background han acabado. Luego la forma correcta de lanzar las corutinas anteriores sería :

```
comer &
beber
wait
```

El comando interno ***wait*** también puede recibir como argumento el ID o el número de job del preoceso al que queremos esperar.

Ejemplo :

```
fuente=$1
shift
for destino in "$@"
do
  cp $fuente $destino &
done
wait
```

Esta solución espera que todos acaben con ***wait***.

Subshells

Otra forma de comunicación entre procesos, mediante ***subshell*** y el ***shell padre***. Los ***subshell*** son parecidos a los bloques de comandos, donde también podemos redirigir su entrada y salida estándar, sólo que ahora se encierran los comandos entre paréntesis y el ***subshell***,a diferencia del bloque de comandos, se ejecuta en un proceso aparte.

```
(
 for ((i=0;i<=9;i++))
do
  echo $i
done
) | sort -r
```

La principal diferencia entre un ***subshell*** y un bloque de comandos es que el primero se ejecuta en un proceso aparte en un proceso aparte, con lo que es menos eficiente, pero a cambio no modifica variables del shell actual, con lo que existe mayor encapsulación. Por ejemplo, si vamos a fijar un ***trap***, lo podemos fijar en un ***subshell*** para no afectar al resto del script.

Cuando ejecutamos un ***subshell*** se hereda del proceso padre, es decir :

- El directorio actual.
- Las variables de entorno exportadas.
- Entrada y salida estándar, como así los errores.

No se hereda :

- Las variables no exportadas.
- Los traps de señales.

La sustitución de procesos

Existen dos operadores de sustitución de procesos: ***<(comando)*** que asigna la salida estándar del comando a un fichero (named pipe) de sólo lectura, y ***>(comando)*** que asigna la salida del comando a un named pipe de sólo escritura.

Ejemplo :

$ grep "passwd" <(ls -l /etc/*)

Esto es equivalente a :

$ ls -l /etc/* | grep "passwd"

Realmente la sustitución de comandos lo que nos devuelve es el nombre del fichero donde se ha depositado la salida estándar del comando :

$ echo <(ls -la)
/dev/fd/63

Depurar scripts

La técnica más básica de depuración, que seguramente ya conozca, es llenar el scripts de ***echo*** que muestran como evoluciona el programa. El objetivo de este tema que pueda usar más y mejores técnicas a la hora de depurar sus scripts.

Vamos a dividirlo en dos partes, la primera veremos como se usan estás técnicas y en la segunda parte veremos paso a paso como construir un depurador de scripts con Bas.

Opciones de Bash para depuración

El shell tiene una serie de opciones para la depuración las cuales, o bien se pasan como argumentos al lanzar bash, o bien se activan con el comando ***set -o opcion***. Si usamos ***set -o opcion*** estamos activando la opción, si usamos ***set +o opcion*** la estamos desactivando. Es decir, al revés de lo que parece.

Opción de set	***Opción de bash***	***Descripción***
noexec	-n	No ejecuta los comandos, sólo comprueba su sintaxis.
verbose	-v	Imprime los comandos antes de ejecutarlos.
xtrace	-x	Imprime los comandos a interpretar y las distintas expansiones que se realizan antes de ejecutarlo.

La opción ***xtrace*** nos muestra tanto el comando a ejecutar, como la expansión de las sustituciones de parámetros, de las sustituciones de comandos, y todas las demás sustituciones que se realicen.

Ejemplo :

$ vi lsfecha.sh
```
function ListaFecha
{
 ls -lad  | grep “$1” | cut -c54-
}

ls -lad $(ListaFecha “$1”)
```

$ bash -x lsfecha '9 Aug'
```
++ ListaFecha '9 Aug'
++ ls -lad Makefile aquello lsfecha
++ grep '9 Aug'
++ cut -c54-
+ ls -lad aquello
-rwxr-xr-x 1 pablo pablo 80 29 Aug 16:40 aquello
```

Cada símbolo + al principio de una línea indica un nivel de expansión, este símbolo es usado por el prompt ***PS4***. Por ejemplo :

$ export PS4='xtrace->'
$ bash -x lsfecha '9 Aug'
```
xxtrace->ListaFecha '9 Aug'
```

xxtrace->ls -lad Makefile aquello lsfecha
xxtrace->grep '9 Aug'
xxtrace->cut -c54-
xtrace->ls -lad aquello
-rwxr-xr-x 1 pablo pablo 80 29 Aug 16:40 aquello

Podemos personalizar aun más el prompt poniendo variables en éste. Por ejemplo, la variable especial ***$LINENO*** nos permite sabe la línea del script que estamos ejecutando.

$ export PS4='$0:$LINENO:'
$ bash -x lsfecha '9 Aug'
llsfecha:12:ListaFecha '9 Aug'
llsfecha:9:ls -lad Makefile aquello lsfecha
llsfecha:9:grep '9 Aug'
llsfecha:9:cut -c54-
lsfecha:12:ls -lad aquello
-rwxr-xr-x 1 pablo pablo 80 29 Aug 16:40 aquello

La opción ***noexec*** sirve para que Bash no ejecute los comandos, sólo lea los comandos y compruebe su sintaxis. Sin embargo una vez que activa esta opción con ***set -o noexec*** ya no podrá volver a desactivarla, ya que el comando ***set +o noexec*** será parseado pero no ejecutado por Bash.

$ bash -n lsfecha '9 Aug'

Fake signals

Las ***fake signals*** (falsas señales) son un mecanismo muy potente de ayuda a la depuración. Se trata de señales producidas por Bash, y no por un programa o suceso externo al shell.

Señal	***Descripción***
SIGEXIT	El script acabó de ejecutarse.
SIGERR	Un comando a retornado un código de terminación distinto de 0.
SIGDEBUG	El shell va a ejecutar una sentencia.
SIGRETURN	Una función o script ejecutado con ***source*** ha acabado.

La señal SIGEXIT

Esta señal se activa justo antes de terminar de ejecutarse un proceso. La señal se debe de solicitar por el proceso (no por el proceso padre que lanza el proceso), es decir, si desde el shell ejecutamos el scrtip ***miscript*** así:

$ trap "echo 'Acabo el script'" EXIT
$ miscript

La señal no se produce, sino que el trap debe de estar dentro del script.

$ vi miscript

trap "echo 'Acabo el script'" EXIT
echo "Empieza el script"

$ chmod +x miscript
$ ./miscript
Empieza el script
Acabo el script

La señal se lanza independientemente de como acabe el script: Por ejecutar la última línea, o por encontrar un ***exit***.

La señal SIGERR

Se lanza siempre que un comando de un script acaba con un código de terminación distinto de 0. La función que la captura puede hacer uso de la variable ? Para obtener su valor.

```
function CapturadoERR
{
 ct=$?
 Echo “El comando devolvio el codigo de terminación $ct”
}
trap CapturadoERR ERR
```

Lo que acabamos obteniendo es el número de línea de la sentencia de la función ***CapturadoERR ERR*.**

```
function CapturadoERR
{
 ct=$?
 echo “Codigo de terminación $ct en la linea $LINENO”
}
trap 'CapturadoERR $LINENO' ERR
```

También existe una forma alternativa de pedir al shell que nos informe si un comando acaba con un código de terminación distinto de 0, que es fijando la opción del ***shell set -o errtrace***, esta opción está disponible sólo a partir de Bash 3.0

La señal SIGDEBUG

Cuando se lanza con ***trap***, se lanza justo antes de ejecutar un comando. Un problema que tiene esta señal es que no se hereda en las funciones que ejecutemos desde el script, con lo que si queremos heredarla tenemos tres opciones :

- Activarla con ***trap*** dentro de cada función.
- Declarar la función con ***declare -t*** que hace que la función si herede el ***trap.***
- O fijar la opción del shell ***set -o functrace*** (o ***set -F***), que hace que todas las funciones hereden el trap de ***SIGDEBUG***.

La señal SIGRETURN

Se lanza cada vez que retornamos de una función, o retornamos de ejecutar un script con ***source***. La señal no se lanza cuando acabamos de ejecutar un comando script (a no ser que lo ejecutemos con ***source***). Si queremos hacer esto último debemos usar ***SIGEXIT***.

Al igual que ***SIGDEBUG***, la señal ***SIGRETURN*** no es heredada por las funciones. De nuevo podemos hacer que una función herede esta señal, declarando a la función con ***declare -t***, o bien activando la opción ***set -o functrace***.

Un depurador Bash

Existe un un depurador en bash llamado bashdb la forma de instalarlo es la siguiente :

$ apt-get install bashdb

Ejemplo :

$ vi ejemplo.sh

```
#!/bin/bash

echo "Esto es un ejemplo"
for i in `seq 1 10`;
do
 echo $i
done
```

$ chmod +x ejemplo.sh
$ bashdb --debugger ejemplo.sh

```
bash Shell Debugger, release 4.0-0.4

Copyright 2002, 2003, 2004, 2006, 2007, 2008, 2009 Rocky Bernstein
This is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.

(/home/pablo/menu/ejemplo.sh:3):
3:      echo "Esto es un ejemplo"

bashdb<0> list
 3:==>echo "Esto es un ejemplo"
 4:   for i in `seq 1 10`;
 5:   do
 6:    echo $i
 7:   done

bashdb<1> help
Available commands:
 /        debug    enable   help     next     show     step+    untrace
 alias    delete   eval     history  print    signal   tbreak   up
 break    disable  examine  info     pwd      skip     trace    watch
 commands display  file     kill     quit     source   tty      where
 condition down    frame    list     restart  step     unalias
 continue edit     handle   load     set      step-    undisplay

Readline command line editing (emacs/vi mode) is available.
Type "help" followed by command name for full documentation.
```

Comando	***Descripción***
help	Muestra la ayuda y los comando que podemos utilizar.
bt	Inicializa.
next	Ejecuta la próxima linea.
list	Muestra el código fuente.
x ***variable***	Me muestra el valor de dicha ***variable*** y el tipo de variable**.**
print variable	Me muestra el valor de dicha ***variable***.
break ***número de linea***	Realiza un break en dicho número de linea.
continue	Continua la ejecución hasta el break.
quit	Salir.

run	Volver a ejecutar.