

Programación y Administración de Sistemas

Práctica 1. Programación de la *shell*.

Víctor Manuel Vargas

Asignatura "Programación y Administración de Sistemas"
2º Curso Grado en Ingeniería Informática
Escuela Politécnica Superior
(Universidad de Córdoba)
vvargas@uco.es

22 de febrero de 2021



Objetivos del aprendizaje I

- Saber cuándo utilizar *scripts* para resolver tareas de programación, identificando las ventajas e inconvenientes de los lenguajes de *scripting* y su aplicabilidad en administración de sistemas.
- Conocer los distintos intérpretes de órdenes en GNU/Linux y justificar el uso de `bash` para la programación de *scripts* de administración de sistemas.
- Escribir *scripts* de `bash` de la mejor forma posible y ejecutarlos correctamente.
- Declarar y utilizar correctamente variables en `bash`.
- Conocer la diferencia entre el uso de comillas dobles y comillas simples en *scripts* de `bash`.
- Diferenciar las variables locales de un *script* de las variables de entorno.
- Utilizar correctamente el comando `export`.

Objetivos del aprendizaje II

- Conocer las variables de entorno más habituales en `bash`.
- Utilizar las variables intrínsecas de `bash` para interactuar de forma más efectiva con la terminal de comandos.
- Utilizar correctamente el comando `exit`.
- Utilizar correctamente el comando `read`.
- Aplicar correctamente la sustitución de comandos en `bash`.
- Conocer y utilizar distintas alternativas para realizar operaciones aritméticas en `bash`.
- Utilizar estructuras condicionales `if`.
- Comparar correctamente cadenas y números, chequear el estado de ficheros y aplicar operadores lógicos.
- Utilizar estructuras condicionales `case`.
- Utilizar estructuras iterativas `for`.
- Utilizar *arrays* en `bash`.

Objetivos del aprendizaje III

- Utilizar funciones en `bash`.
- Aplicar diversas opciones para la depuración de *scripts* en `bash`.
- Redirigir la entrada y la salida de comandos desde y hacia ficheros.
- Interconectar distintos comandos mediante el uso de tuberías.
- Utilizar correctamente el comando `tee` para la redirección de salida.
- Conocer los *here documents* y utilizarlos para hacer *scripts* más legibles.
- Utilizar correctamente los siguientes comandos adicionales: `cat`, `head`, `tail`, `wc`, `find`, `basename`, `dirname`, `stat` y `tr`.
- Aplicar el mecanismo de expansión de llaves en la creación de *arrays*.

1. Introducción.
 - 1.1 Justificación.
 - 1.2 ¿Programación o *scripting*?
 - 1.3 Primeros programas.
2. Variables.
 - 2.1 Concepto y declaración.
 - 2.2 Comillas simples y dobles.
 - 2.3 Variables locales y de entorno.
 3. Diferencia entre variables locales y variable de entorno.
 3. Comando `export`.
 3. Variables de entorno más importantes.
 3. Variables intrínsecas.
 3. Comando `exit`.
 - 2.4 Dando valor a variables.
 4. Comando `read`.
 4. Sustitución de comandos.
 - 2.5 Operadores aritméticos.
3. Estructuras de control.

3.1 Condicionales if.

- 1. Comparación de cadenas.
- 1. Comparación de números.
- 1. Chequeo de ficheros.
- 1. Operadores lógicos.

3.2 Condicionales case.

3.3 Estructura iterativa for.

3.4 Estructuras iterativas while y until.

4. Otras características.

4.1 Funciones en bash.

4.2 Depuración en bash.

4.3 Redireccionamiento y tuberías.

- 3. Redireccionamiento de salida.
- 3. Redireccionamiento de entrada.
- 3. Tuberías.
- 3. Comando tee.
- 3. *Here documents*.

4.4 Comandos interesantes.

- 4. Comando cat.
- 4. Comandos head, tail y wc.
- 4. Comando find.
- 4. Comandos basename y dirname.
- 4. Comando stat.
- 4. Comando tr.
- 4.5 Expansión de llaves.

- Pruebas de validación de prácticas.

¿Línea de comandos?

- ¿Para qué necesito aprender a utilizar la línea de comandos?
- Historia real¹:
 - Unidad compartida por cuatro servidores que está llenándose → impedía a la gente trabajar.
 - El sistema no soportaba cuotas.
 - Un ingeniero escribe un programa en C++ que navega por los archivos de todos los usuarios, calcula cuanto espacio está ocupando cada uno y genera un informe.
 - Utilizando un entorno GNU/Linux y su *shell*:

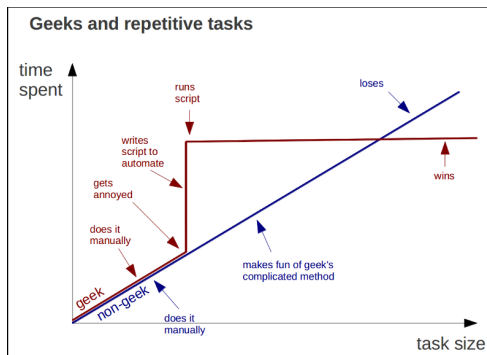
```
1 du -s * | sort -nr > $HOME/user_space_report.txt
```

¹http://www.linuxcommand.org/lc3_learning_the_shell.php



bash

- Las interfaces gráficas de usuario (GUI) son buenas para muchas cosas, pero no para todas, especialmente las más repetitivas.



bash

- ¿Qué es la *shell*?
 - Programa que recoge comandos del ordenador y se los proporciona al SO para que los ejecute.
 - Antiguamente, era la única interfaz disponible para interactuar con SO tipo Unix.
- En casi todos los sistemas GNU/Linux, el programa que actúa como *shell* es *bash*.
 - Bourne Again SHell → versión mejorada del *sh* original de Unix.
 - Escrito por Steve Bourne.



bash

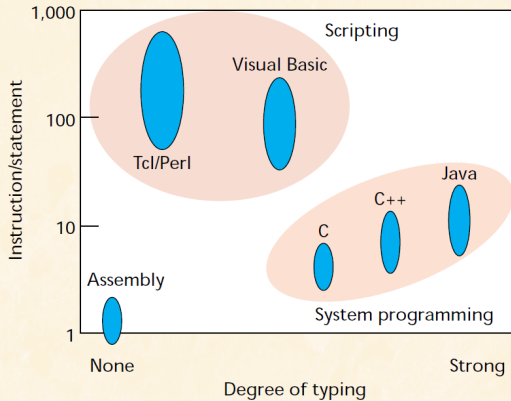
- Alternativas a bash:
 - Bourne shell (`sh`), C shell (`csh`), Korn shell (`ksh`), TC shell (`tcsch`)...
- `bash` incorpora las prestaciones más útiles de `ksh` y `csh`.
 - Es conforme con el estándar IEEE POSIX P1003.2/ISO 9945.2 Shell and Tools.
 - Ofrece mejoras funcionales sobre la shell desde el punto de vista de programación y de su uso interactivo.
- ¿Qué es una terminal?
 - Es un programa que emula la terminal de un computador, iniciando una sesión de *shell* interactiva.
 - `gnome-terminal`, `konsole`, `xterm`, `rxvt`, `kvt`, `nxterm` o `eterm`.



¿Programación o scripting?

- `bash` no es únicamente una excelente *shell* por línea de comandos...
- También es un **lenguaje de *scripting*** en sí mismo.
- El *shell scripting* sirve para automatizar multitud de tareas que, de otra forma, requerirían múltiples comandos introducidos de forma manual.
- Lenguaje de programación (LP) vs. *scripting*:
 - Los LPs son, en general, más potentes y mucho más rápidos que los lenguajes de *scripting*.
 - Los LPs comienzan desde el código fuente, que se compila para crear los ejecutables (lo que no permite que los programas sean fácilmente portables entre diferentes SOs).





(OUSTERHOUT, J., "Scripting: Higher-Level Programming for the 21st Century",
IEEE Computer, Vol. 31, No. 3, March 1998, pp. 23-30.)



¿Programación o scripting?

- Un lenguaje de *scripting* (LS) también comienza por el código fuente, pero no se compila en un ejecutable.
- En su lugar, un **intérprete** lee las instrucciones del fichero fuente y las ejecuta secuencialmente.
 - Programas interpretados → más lentos que los compilados.
 - “Tipado” débil (**¿ventaja o desventaja?**).
- Ventajas:
 - En general, una línea de LS “cunde” más que una de un LP.
 - El fichero de código es fácilmente portable a cualquier SO.
 - Todo lo que yo pueda hacer con mi *shell*, lo puedo automatizar con un *script*.
 - Nivel de abstracción muy superior en cuanto a operaciones con ficheros, procesos...



Primer programa bash: `holaMundo.sh`

- Abrir un editor de textos:

```
1 victor@victor-ayrna:~$ gedit holaMundo.sh &
```

- Escribimos el código:

```
1 #!/bin/bash  
2 echo "Hola Mundo"
```

- Hacemos que el fichero de texto sea `ejecutable`:

```
1 victor@victor-ayrna:~$ chmod u+x holaMundo.sh  
2 victor@victor-ayrna:~$ ls -l holaMundo.sh  
3 -rwxr--r-- 1 victor victor 30 ene  8 20:26 holaMundo.sh
```



Primer programa bash

```
1  #!/bin/bash
2  echo "Hola Mundo"
```

- El carácter **#!** al principio del script se denomina *SheBang/HashBang* y es un comentario para el intérprete *shell*.
- Es utilizado por el cargador de programas del SO (el código que se ejecuta cuando una orden se lanza).
- Le indica **qué intérprete de comandos** se debe utilizar para este fichero, en el caso anterior, `/bin/bash`.



Primer programa bash

- Para ejecutar el programa:

```
1 victor@victor-ayrna:~$ holaMundo.sh
2 holaMundo.sh: orden no encontrada
```

- El directorio \$HOME, donde está el programa, no está dentro del *path* por defecto:

```
1 $ echo $PATH
2 /usr/local/bin:/usr/bin:/bin:/usr/local/games:/usr/games
```

- Por tanto, ¡hay que especificar la ruta completa!:

```
1 victor@victor-ayrna:~$ /home/victor/holaMundo.sh
2 Hola Mundo
3 victor@victor-ayrna:~$ ~/holaMundo.sh
4 Hola Mundo
5 victor@victor-ayrna:~$ ./holaMundo.sh
6 Hola Mundo
```



Primer programa bash

- Orden `echo`:
 - Imprime (manda al stdout) el contenido de lo que se le pasa como argumento.
 - Es un comando del sistema (un ejecutable), no una palabra reservada del lenguaje de programación.
 - Se puede utilizar el `man` para ver sus opciones.

```
1 victor@victor-ayrna:~$ echo "Imprimo una línea con salto de línea"
2 Imprimo una línea con salto de línea
3 victor@victor-ayrna:~$ echo -n "Imprimo una línea sin salto de línea"
4 Imprimo una línea sin salto de líneavictor@victor-ayrna:~$ which echo
5 /bin/echo
6 victor@victor-ayrna:~$ echo "ho\nla"
7 ho\nla
8 victor@victor-ayrna:~$ echo -e "ho\nla"
9 ho
10 la
```



Segundo programa bash: `papelera.sh`

- Especificar los comandos para:
 - Crear una subcarpeta `papelera`.
 - Copiar todos los ficheros que hay en la carpeta `~` a la subcarpeta `papelera`.
 - Posteriormente, borrarlos.



Segundo programa bash

- Comandos:

```
1 victor@victor-ayrna:~$ mkdir papelera
2 victor@victor-ayrna:~$ cp * papelera
3 cp: se omite el directorio «carpeta»
4 ...
5 victor@victor-ayrna:~$ rm -Rf papelera/
```

- El mensaje que nos aparece es un *warning*. Por defecto, el comando `cp` no copia carpetas, las omite y copia únicamente los ficheros.
- En lugar de tener que escribir todo esto de forma interactiva en la *shell*, escribimos un *script*.



Variables: concepto

- Al igual que en los LP, se pueden utilizar **variables**.
- Todos los valores son almacenados como **tipo cadena de texto** (“**tipado**” **débil**).
- ¿No puedo operar?
 - Operadores matemáticos que convierten las variables en número para el cálculo.
- Como no hay tipos, no es necesario declarar variables, sino que al asignarles un valor, es cuando se crean.



Variables: primer ejemplo

- Primer ejemplo: `holaMundoVariable.sh`

```
1  #!/bin/bash
2  STR="Hola Mundo!"
3  echo $STR
```

- Asignación: `VARIABLE="valor"`
- Resolver una variable, es decir, `sustituir` la variable por su valor: `$VARIABLE`
- Probar a poner espacios antes y después del "="
 - ¿Qué sucede?



Variables: precaución

- El lenguaje de programación de la *shell* no hace un *casting* (conversión) de los tipos de las variables.
- Una misma variable puede contener datos numéricos o de texto:

```
1 contador=0
2 contador=Domingo
```

- La conmutación del tipo de una variable puede conllevar a confusión.
- Buena práctica: asociar siempre el mismo tipo de dato a una variable en el contexto de un mismo *script*.



Variables: precaución

- **Carácter de escape:**
 - Un **carácter de escape** es un carácter que permite que los símbolos especiales del LP no se interpreten y se utilice su valor literal.
 - Por ejemplo, en C:

```
1 "Esta cadena contiene el carácter \" en su interior"
```

- En bash:

```
1 victor@victor-ayrna:~$ ls \*
```



Comillas simples y dobles

- Cuando el valor de la variable contenga espacios en blanco o caracteres especiales, se deberá encerrar entre comillas simples o dobles.
- Las comillas simples servirían para que la cadena se represente tal cual → como si cada carácter de la cadena tuviese un “\”.
- Si son dobles, se permitirá especificar variables internas que se resolverán:

```
1 victor@victor-ayrna:~$ var="cadena de prueba"
2 victor@victor-ayrna:~$ nuevavar="Valor de var es $var"
3 victor@victor-ayrna:~$ echo $nuevavar
4 Valor de var es cadena de prueba
```

- ¿Qué hubiera pasado en este caso?

```
1 victor@victor-ayrna:~$ var="cadena de prueba"
2 victor@victor-ayrna:~$ nuevavar='Valor de var es $var'
3 victor@victor-ayrna:~$ echo $nuevavar
```



Comillas simples y dobles

- Hacer un *script* que muestre por pantalla (**comillas.sh**):

```
1 Valor de 'var' es "cadena de prueba"
```



Variables locales y de entorno

- Hay dos tipos de variables:
 - Variables **locales**.
 - Variables **de entorno**:
 - Establecidas por el SO, especifican su configuración.
 - Se pueden listar utilizando el comando `env`.

```
1 victor@victor-ayrna:~$ echo $SHELL
2 /bin/bash
3 victor@victor-ayrna:~$ echo $PATH
4 /usr/local/bin:/usr/bin:/bin:/usr/local/games:/usr/games
```

- Se definen en *scripts* del sistema que se ejecutan al iniciar el proceso `bash`.
`/etc/profile`, `/etc/profile.d/` y `~/.bash_profile`.
- Al salir, se ejecutan los comandos en `~/.bash_logout`.



Comando export

- El comando **export** establece una variable en el entorno, de forma que sea accesible por los procesos hijos.

```
1 victor@victor-ayrna:~$ x=hola
2 victor@victor-ayrna:~$ bash      # Ejecutar una shell hija
3 victor@victor-ayrna:~$ echo $x   # No aparece nada
4
5 victor@victor-ayrna:~$ exit      # Volver al padre
6 exit
7 victor@victor-ayrna:~$ export x  # También se podría export x=hola
8 victor@victor-ayrna:~$ bash
9 victor@victor-ayrna:~$ echo $x   # Ahora si
10 hola
```



Comando export

- Si el proceso hijo modifica la variable, no se modifica la del padre:

```
1 victor@victor-ayrna:~$ x=hola
2 victor@victor-ayrna:~$ export x
3 victor@victor-ayrna:~$ bash
4 victor@victor-ayrna:~$ x=adios
5 victor@victor-ayrna:~$ exit
6 victor@victor-ayrna:~$ echo $x
7 hola
```



Algunas variables importantes

- “Home, sweet *\$HOME*”:
 - *\$HOME*: directorio personal del usuario, donde debería almacenar todos sus archivos.
 - *\$HOME* \equiv *~* \equiv */home/usuario*
 - Argumento por defecto del comando *cd*.
- *\$PATH*: carpetas que contienen los comandos.
 - Es una lista de directorios separados por “:”.
 - Normalmente, ejecutamos *scripts* así:

```
1 $ ./trash.sh
```

- Pero si antes hemos establecido *PATH=\$PATH:~*, podríamos ejecutar los scripts que haya en el *\$HOME* de la siguiente forma:

```
1 $ trash.sh
```

- *\$LOGNAME* o *\$USER*: ambas contienen el nombre de usuario.



Algunas variables importantes

- Si creamos una carpeta:

```
1 $ mkdir ~/bin
```

- Y modificamos el `.bash_profile`:

```
1 PATH=$PATH:$HOME/bin  
2 export PATH
```

- El directorio `/home/usuario/bin` será incluido en la búsqueda de programas binarios a ejecutar.
- ¿Qué hubiera pasado si no hubiese incluido el `export`?



Más variables importantes

- **\$HOSTNAME**: contiene el nombre de la máquina.
- **\$MACHINE**: arquitectura.
- **\$PS1**: cadena que codifica la secuencia de caracteres mostrados antes del *prompt*
 - **\t**: hora.
 - **\d**: fecha.
 - **\w**: directorio actual.
 - **\h**: nombre de la máquina.
 - **\W**: última parte del directorio actual.
 - **\u**: nombre de usuario.
- **\$UID**: contiene el id del usuario que no puede ser modificado.
- **\$SHLVL**: contiene el nivel de anidamiento de la *shell*.
- **\$RANDOM**: número aleatorio.
- **\$SECONDS**: número de segundos que **bash** lleva en marcha.



Más variables importantes

- Ejercicio: haz un *script* que muestre la siguiente información:

```
1 victor@victor-ayrna:~/tmp$ ./informacion.sh
2 Bienvenido victor!, tu identificador es 1000.
3 Esta es la shell número 1, que lleva 107 arrancada.
4 La arquitectura de esta máquina es x86_64-pc-linux-gnu y el cliente de
  terminal es xterm
```

- Ejercicio: personaliza el *prompt* para que adquiriera este aspecto:

```
1 victor-ayrna:~(hola, son las 11:16:30) &
```



Variables intrínsecas

- \$#: número de argumentos de la línea de comandos (`argc`).
- \$**n**: **n**-ésimo argumento de la línea de comandos (`argv[n]`), si **n** es mayor que 9 utilizar `${n}`.
- \$*: todos los argumentos de la línea de comandos (como una sola cadena).
- \$@: todos los argumentos de la línea de comandos (como un *array*).
- \$!: pid del último proceso que se lanzó con `&`.
- \$-: opciones suministradas a la *shell*.
- \$?: valor de salida la última orden ejecutada (ver `exit`).



Variables intrínsecas

- Ejercicio: escribir un *script* ([parametros.sh](#)) que imprima el número de argumentos que se le han pasado por línea de comandos, el nombre del *script*, el primer argumento, el segundo argumento, la lista de argumentos como una cadena, y la lista de argumentos como un *array*.

```
1 victor@victor-ayrna:~/tmp$ ./parametros.sh estudiante1 estudiante2
2 2; ./parametros.sh; estudiante1; estudiante2; estudiante1 estudiante2;
   estudiante1 estudiante2
```



Variables intrínsecas: navegar por comandos anteriores

- `!$`: último argumento del último comando ejecutado.
- `!:n`: `n`-ésimo argumento del último comando ejecutado.

```
1 victor@victor-ayrna ~ $ echo argumentos 2 3
2 argumentos 2 3
3 victor@victor-ayrna ~ $ echo !$
4 echo 3
5 3
6 victor@victor-ayrna ~ $ echo !:0
7 echo echo
8 echo
```

- Comandos interactivos de consola:
 - Buscar un comando en el historial de la consola: `Ctrl+R` (en lugar de pulsar `↑` `n` veces).
 - Navegar por los argumentos del último comando: `Alt+.`



Comando exit

- Se puede utilizar para finalizar la ejecución de un *script* y devolver un valor de salida (0 – 255) que estará disponible para el proceso padre que invocó el *script*.
 - Si lo llamamos sin parámetros, se utilizará el valor de salida del último comando ejecutado (equivalente a `exit $?`).

```
1 victor@victor-ayrna:~$ echo $?
2 0
3 victor@victor-ayrna:~$ bash
4 victor@victor-ayrna:~$ exit 1
5 exit
6 victor@victor-ayrna:~$ echo $?
7 1
8 victor@victor-ayrna:~$ echo $?
9 0
10 victor@victor-ayrna:~$ echo $?
11 0
12 victor@victor-ayrna:~$ bash
13 victor@victor-ayrna:~$ exit
14 exit
15 victor@victor-ayrna:~$ echo $?
16 0
```



Comando read

- El comando `read` permite leer un comando del usuario por teclado y almacenarlo en una variable.
 - Ejemplo:

```
1 #!/bin/bash
2 echo -n "Introduzca nombre de fichero a borrar: "
3 read fichero
4 rm -i $fichero # La opción -i pide confirmación
5 echo "Fichero $fichero borrado!"
```



Comando read

- Opciones del comando `read`:
 - `read -s`: no hace *echo* de la entrada.
 - `read -nN`: solo acepta `N` caracteres de entrada.
 - `read -p "mensaje"`: muestra el mensaje `mensaje` al pedir la información al usuario.
 - `read -tT`: acepta la entrada durante un tiempo máximo de `T` segundos.

```
1 victor@victor-ayrna:~$ read -s -t5 -n1 -p "si (S) o no (N)?" respuesta
2 si (S) o no (N)?S
3 victor@victor-ayrna:~$ echo $respuesta
4 S
```



Sustitución de comandos (IMPORTANTE)

- El acento hacia atrás (`) es distinto que la comilla simple (').
- ``comando`` se utiliza para sustitución de comandos, es decir, se ejecuta comando, se recoge lo que devuelve por consola y se sustituye por el comando:

```
1 victor@victor-ayrna:~/tmp$ LISTA=`ls`  
2 victor@victor-ayrna:~/tmp$ echo $LISTA  
3 d1 ecj-read-only simbolico tar
```

- También se puede utilizar `$(comando)`:

```
1 victor@victor-ayrna:~/tmp$ LISTA=$(ls)  
2 victor@victor-ayrna:~/tmp$ echo $LISTA  
3 d1 ecj-read-only simbolico tar  
4 victor@victor-ayrna:~/tmp$ ls $(pwd)  
5 d1 ecj-read-only simbolico tar  
6 victor@victor-ayrna:~/tmp$ ls $(echo /carpeta)  
7 a.out imagen.png
```

- Antes de ejecutar una instrucción, bash sustituye las variables de la línea (empiezan por \$) y los comandos (`$()` o `` ``).



Operadores aritméticos

- Bash permite realizar operaciones aritméticas

Operador	Significado
+	Suma
-	Resta
*	Multiplicación
/	División
**	Exponenciación
%	Módulo

```
1 victor@victor-ayrna:~/tmp$ a=(5+2)*3
2 victor@victor-ayrna:~/tmp$ echo $a
3 (5+2)*3
4 victor@victor-ayrna:~/tmp$ b=2**3
5 victor@victor-ayrna:~/tmp$ echo $a+$b
6 (5+2)*3+2**3
```



Operadores aritméticos

- Hay que utilizar la instrucción `let`:

```
1 victor@victor-ayrna:~/tmp$ let X=10+2*7
2 victor@victor-ayrna:~/tmp$ echo $X
3 24
4 victor@victor-ayrna:~/tmp$ let Y=X+2*4
5 victor@victor-ayrna:~/tmp$ echo $Y
6 32
```

- Alternativamente, las expresiones aritméticas también se pueden evaluar con `$(expresión)` o `$((expresión))`:

```
1 victor@victor-ayrna:~/tmp$ echo "$((123+20))"
2 143
3 victor@victor-ayrna:~/tmp$ VALOR=$((123+20))
4 victor@victor-ayrna:~/tmp$ echo "$[123*$VALOR]"
5 17589
6 victor@victor-ayrna:~/tmp$ echo "$[123*$VALOR]"
7 17589
8 victor@victor-ayrna:~/tmp$ echo '$[123*$VALOR]'
9 $[123*$VALOR]
```



Operadores aritméticos

- Ejercicio:
 - Implementar un *script* (*operaciones.sh*) que lea dos números y aplique todas las operaciones posibles sobre los mismos.

```
1 victor@victor-ayrna:~/tmp$ ./operaciones.sh
2 Introduzca un primer número: 2
3 Introduzca un segundo número : 9
4 Suma: 11
5 Resta: -7
6 Multiplicación: 18
7 División: 0
8 Módulo: 2
```



Condicionales if

- La forma más básica es:

```
1  if [ expresión ];  
2  then  
3      instrucciones  
4  elif [ expresión ];  
5  then  
6      instrucciones  
7  else  
8      instrucciones  
9  fi
```

- Las secciones elif (else if) y else son opcionales.
- **IMPORTANTE:** espacios antes y después [y].



Expresiones lógicas

- Expresiones lógicas pueden ser:
 - Comparación de cadenas.
 - Comparación de números.
 - Chequeo de ficheros.
 - Combinación de los anteriores mediante **operadores lógicos**.
- Las expresiones se encierran con corchetes [expresion].
- En realidad, se está llamando al programa /usr/bin/.

```
1 victor@victor-ayrna:~$ /usr/bin/[ 3 = 4 ]
2 victor@victor-ayrna:~$ echo $?
3 1
4 victor@victor-ayrna:~$ /usr/bin/[ 4 = 4 ]
5 victor@victor-ayrna:~$ echo $?
6 0
7 victor@victor-ayrna:~$ /usr/bin/[ 'asa' == 'asa' ]
8 victor@victor-ayrna:~$ echo $?
9 0
10 victor@victor-ayrna:~$ /usr/bin/[ 'asa' == 'asaa' ]
11 victor@victor-ayrna:~$ echo $?
12 1
```



Comparación de cadenas

Operador	Significado
<code>s1 == s2</code>	Igual a
<code>s1 != s2</code>	Distinto a
<code>-n s</code>	Longitud mayor que cero
<code>-z s</code>	Longitud igual a cero

- Ejemplos:
 - `[s1 == s2]`: true si s1 es igual a s2, sino false.
 - `[s1 != s2]`: true si s1 no es igual a s2, sino false.
 - `[s1]`: true si s1 no está vacía, sino false.
 - `[-n s1]`: true si s1 tiene longitud > 0, sino false.
 - `[-z s2]`: true si s2 tiene longitud 0, sino false.
- Los dobles corchetes permiten usar expresiones regulares:
 - `[[s1 == s2*]]`: true si s1 empieza por s2, sino false.



Comparación de cadenas

- Implementar un *script* que pregunte el nombre de usuario y devuelva un error si el nombre no es correcto:

```
1 victor@victor-ayrna:~/tmp$ ./saludaUsuario.sh
2 Introduzca su nombre de usuario: victor
3 Bienvenido "victor"
4 victor@victor-ayrna:~/tmp$ ./saludaUsuario.sh
5 Introduzca su nombre de usuario: pagutierrez
6 Eso es mentira!
```



Comparación de números

Operador	Significado
n1 -lt n2	Menor que
n1 -gt n2	Mayor que
n1 -le n2	Menor o igual que
n1 -ge n2	Mayor o igual que
n1 -eq n2	Igual
n1 -ne n2	Distinto



Comparación de números

- Implementar un *script* que pida un número en el rango [1, 10) y compruebe si el número introducido está o no fuera de rango:

```
1 victor@victor-ayrna:~/tmp$ ./numeroRango.sh
2 Introduzca un número (1 <= x < 10): 1
3 El número 1 es correcto!
4 victor@victor-ayrna:~/tmp$ ./numeroRango.sh
5 Introduzca un número (1 <= x < 10): 0
6 Fuera de rango!
7 victor@victor-ayrna:~/tmp$ ./numeroRango.sh
8 Introduzca un número (1 <= x < 10): 10
9 Fuera de rango!
```



Chequeo de ficheros

Operador	Significado
-e f1	¿Existe el fichero f1?
-s f1	¿f1 tiene tamaño mayor que cero?
-f f1	¿Es f1 un fichero normal?
-d f1	¿Es f1 un directorio?
-l f1	¿Es f1 un enlace simbólico?
-r f1	¿Tienes permiso de lectura sobre f1?
-w f1	¿Tienes permiso de escritura sobre f1?
-x f1	¿Tienes permiso de ejecución sobre f1?



Chequeo de ficheros

- Ejemplo: *script* que comprueba si el archivo `/etc/fstab` existe y si existe, lo copia a la carpeta actual.

```
1  #!/bin/bash
2  if [ -f /etc/fstab ];
3  then
4      cp /etc/fstab .
5      echo "Hecho."
6  else
7      echo "Archivo /etc/fstab no existe."
8      exit 1
9  fi
```



Chequeo de ficheros

- Ejercicio: escribir un *script* bash que haga lo siguiente:
 - Acepta un nombre de fichero.
 - Comprueba si el fichero existe.
 - Si existe, hace una copia del mismo poniéndole como nombre `nombreOriginal.bak_Fecha`, donde *Fecha* la podéis conseguir a partir del comando "`date +%d-%m-%y`"².

```
1 victor@victor-ayrna:~/tmp$ date +%d-%m-%y
2 17-02-21
3 victor@victor-ayrna:~/tmp$ ./backup.sh
4 El uso del programa es ./backup.sh nombreFichero
5 victor@victor-ayrna:~/tmp$ ./backup.sh copiaFstab.sh
6 victor@victor-ayrna:~/tmp$ ls copiaFstab* -l
7 -rwxr--r-- 1 victor victor 130 feb 17 15:02 copiaFstab.sh
8 -rwxr--r-- 1 victor victor 130 feb 17 15:14 copiaFstab.sh.bak_17-02-21
```

²Consulta [man date](#) para más información



Operadores lógicos

Operador	Significado
!	No
&& o -a	Y
o -o	O

- **Ojo:** uso distinto de las dos versiones de los operadores:

```
1  if [ $n1 -ge $n2 ] && [ $s1 = $s2 ];  
2  ...  
3  if [ $n1 -ge $n2 -a $s1 = $s2 ];  
4  ...
```

- Ejercicio: implementar el *script* `numeroRango.sh` utilizando un solo `if`.



Condicionales case

- Evitar escribir muchos if seguidos:

```
1  case $var in
2      val1)
3          instrucciones;;
4      val2)
5          instrucciones;;
6      *)
7          instrucciones;;
8  esac
```

- El * agrupa a las instrucciones por defecto.
- Se pueden evaluar dos valores a la vez `val1 | val2`).



Condicionales case

● Ejemplo:

```
1  #!/bin/bash
2  echo -n "Introduzca un número t.q. 1 <= x < 10: "
3  read x
4  case $x in
5      1) echo "Valor de x es 1.>";;
6      2) echo "Valor de x es 2.>";;
7      3) echo "Valor de x es 3.>";;
8      4) echo "Valor de x es 4.>";;
9      5) echo "Valor de x es 5.>";;
10     6) echo "Valor de x es 6.>";;
11     7) echo "Valor de x es 7.>";;
12     8) echo "Valor de x es 8.>";;
13     9) echo "Valor de x es 9.>";;
14     0 | 10) echo "Número incorrecto.>";;
15     *) echo "Valor no reconocido.>";;
16  esac
```



Estructuras iterativas for

- Se utiliza para iterar a lo largo de una lista de valores de una variable:

```
1  for var in lista
2  do
3      instrucciones;
4  done
```

- Las instrucciones se ejecutan con todos los valores que hay en `lista` para la variable `var`.
- `ejemploFor1.sh`:

```
1  #!/bin/bash
2  let sum=0
3  for num in 1 2 3 4 5
4  do
5      let "sum = $sum + $num"
6  done
7  echo $sum
```



Estructuras iterativas for

● ejemploFor2.sh:

```
1 #!/bin/bash
2 for x in papel lapiz boligrafo
3 do
4     echo "El valor de la variable es $x"
5     sleep 5
6 done
```

¿y si queremos esta salida?:

```
1 victor@victor-ayrna:~$ ./ejemploFor2Bis.sh
2 El valor de la variable es papel dorado
3 El valor de la variable es lapiz caro
4 El valor de la variable es boligrafo barato
```



Estructuras iterativas for

- Si eliminamos la parte de `in lista`, la lista sobre la que se itera es la lista de argumentos (\$1, \$2, \$3...),
`ejemploForArg.sh`:

```
1  #!/bin/bash
2  for x
3  do
4      echo "El valor de la variable es $x"
5      sleep 5
6  done
```

produce la salida:

```
1  victor@victor-ayrna:~$ ./ejemploForArg.sh estudiante1 estudiante2
2  El valor de la variable es estudiante1
3  El valor de la variable es estudiante2
```



Estructuras iterativas for

- Iterando sobre listas de ficheros
(**ejemploForListarFicheros.sh**):

```
1  #!/bin/bash
2
3  # Listar todos los ficheros del directorio actual
4  # incluyendo información del número de nodo
5  for x in *
6  do
7      ls -i $x
8  done
9
10 # Listar todos los ficheros del directorio /bin
11 for x in /bin
12 do
13     ls -i $x
14 done
```



Estructuras iterativas for

- Comando find:

```
1 victor@victor-ayrna:~$ find -name "*.sh"
2 ./ejemplos/ejemploForArg.sh
3 ./ejemplos/holaMundoVariable.sh
4 ...
```

- Listar ficheros que tengan extensión .sh
(**ejemploForImpFichScripts.sh**):

```
1 #!/bin/bash
2
3 # Imprimir todos los ficheros que se encuentren
4 # con extensión .sh
5 for x in $(find -name "*.sh")
6 do
7     echo $x
8 done
```



Estructuras iterativas for

- Comando útil: **seq**.

```
1  #!/bin/bash
2  for i in $(seq 8)
3  do
4      echo $i
5  done
```



Estructuras iterativas for

- for tipo C:

```
1  for (( EXPR1; EXPR2; EXPR3 ))
2  do
3      instrucciones;
4  done
```

- Ejemplo (**ejemploForTipoC.sh**):

```
1  #!/bin/bash
2
3  echo -n "Introduzca un número: "; read x;
4  let sum=0
5  for (( i=1; $i<$x; i=$i+1 ))
6  do
7      let "sum=$sum + $i"
8  done
9  echo "La suma de los primeros $x números naturales es: $sum"
```



Arrays

- Para crear arrays: `miNuevoArray[i]=Valor`.
- Para crear arrays: `miNuevoArray=(Valor1 Valor2 Valor3)`.
- Para acceder a un valor: `${miNuevoArray[i]}`.
- Para acceder a todos los valores: `${miNuevoArray[*]}`.
- Para longitud: `${#miNuevoArray[@]}`.

```
1 victor@victor-ayrna:~$ miNuevoArray[0]="Gran"
2 victor@victor-ayrna:~$ miNuevoArray[1]="Array"
3 victor@victor-ayrna:~$ miNuevoArray[2]="Triunfador"
4 victor@victor-ayrna:~$ echo ${miNuevoArray[2]}
5 Triunfador
6 victor@victor-ayrna:~$ miNuevoArray=( "Gran" "Array" "Triunfador" )
7 victor@victor-ayrna:~$ echo ${miNuevoArray[1]}
8 Array
9 victor@victor-ayrna:~$ echo ${miNuevoArray[*]}
10 Gran Array Triunfador
```



Arrays

- Combinar *arrays* y `for` (`arrayFor.sh`).

```
1  #!/bin/bash
2  elArray=("pelo" "pico" "pata")
3  for x in ${elArray[*]}
4  do
5      echo "--> $x"
6  done
```



Estructura iterativa while

```
1 while expresion_evalua_a_true
2 do
3     instrucciones
4 done
```

Ejemplo (`while.sh`):

```
1 #!/bin/bash
2 echo -n "Introduzca un número: "; read x
3 let sum=0; let i=1
4 while [ $i -le $x ]; do
5     let "sum = $sum + $i"
6     let "i = $i + 1"
7 done
8 echo "La suma de los primeros $x números es: $sum"
```



Estructura iterativa until

```
1 until expresion_evalua_a_true
2 do
3     instrucciones
4 done
```

Ejemplo (until.sh):

```
1 #!/bin/bash
2 echo -n "Introduzca un número: "; read x
3 until [ "$x" -le 0 ]; do
4     echo $x
5     x=$((x-1))
6     sleep 1
7 done
8 echo "TERMINADO"
```



Funciones en bash

- Las funciones hacen que los *scripts* sean más fáciles de mantener.
- El programa se divide en piezas de código más pequeñas.
- Función simple ([functionHola.sh](#)):

```
1  #!/bin/bash
2  hola()
3  {
4  echo "Estás dentro de la función hola() y te saludo."
5  }
6
7  echo "La próxima línea llama a la función hola()"
8  hola
9  echo "Ahora ya has salido de la funcion"
```



Funciones en bash

- Los argumentos **NO** se especifican, sino que se usan las variables intrínsecas (`functionCheck.sh`):

```
1  #!/bin/bash
2  function chequea() {
3      if [ -e "$1" ]
4      then
5          return 0
6      else
7          return 1
8      fi
9  }
10
11  echo -n "Introduzca el nombre del archivo: "
12  read x
13  if chequea $x
14  then
15      echo "El archivo $x existe !"
16  else
17      echo "El archivo $x no existe !"
18  fi
```



Depuración en bash

- Antes de ejecutar una instrucción, `bash` sustituye las variables de la línea (empiezan por `$`) y los comandos (`$()` o `` ``).
- Para depurar los *scripts*, `bash` ofrece la posibilidad de:
 - Argumento `-x`: muestra cada línea completa del *script* antes de ser ejecutada, con sustitución de variables/comandos.
 - Argumento `-v`: muestra cada línea completa del *script* antes de ser ejecutada, tal y como se escribe.
- Introducir el argumento en la línea del *SheBang*.
- Ejemplo (`bashDepuracion.sh`):

```
1  #!/bin/bash -x
2  echo -n "Introduzca un número: "
3  read x
4  let sum=0
5  for (( i=1 ; $i<=$x ; i=$((i+1)) )) ; do
6      let "sum = $sum + $i"
7  done
8  echo "La suma de los $x primeros números es: $sum"
```



Depuración en bash

```
1  victor@victor-ayrna:~$ ./bashDepuracion.sh
2  + echo -n 'Introduzca un número: '
3  Introduzca un número: + read x
4  5
5  + let sum=0
6  + (( i=1 ))
7  + (( 1<5 ))
8  + let 'sum = 0 + 1'
9  + (( i=1+1 ))
10 + (( 2<5 ))
11 + let 'sum = 1 + 2'
12 + (( i=2+1 ))
13 + (( 3<5 ))
14 + let 'sum = 3 + 3'
15 + (( i=3+1 ))
16 + (( 4<5 ))
17 + let 'sum = 6 + 4'
18 + (( i=4+1 ))
19 + (( 5<5 ))
20 + echo 'La suma de los 5 primeros números es: 10'
21 La suma de los 5 primeros números es: 10
```



Redireccionamiento de entrada/salida

- Existen diferentes descriptores de **ficheros**:
 - **stdin**: entrada estándar (descriptor número 0) \Rightarrow Por defecto, teclado.
 - **stdout**: salida estándar (descriptor número 1) \Rightarrow Por defecto, consola.
 - **stderr**: salida de error (descriptor número 2) \Rightarrow Por defecto, consola.



Redireccionamiento de salida

- Operadores (cambiar los **por defecto**):
 - comando **>** salida.txt: la salida estándar de comando se escribirá en salida.txt y no por pantalla. Sobrescribe el contenido del fichero.
 - comando **»** salida.txt: igual que **>**, pero añade el contenido al fichero sin sobrescribir.
 - comando **2>** error.txt: la salida de error de comando se escribirá en error.txt y no por pantalla. Sobrescribe el contenido del fichero.
 - comando **2»** error.txt: igual que **2>**, pero añade el contenido al fichero sin sobrescribir.

```
1 ls -la > directorioactual.txt
2 date >> fechasespeciales.txt
3 ls /root 2> ~/quefalloocurrio.txt
4 cp ~/archivo.txt /root 2>> ~/logdefallos.txt
```



Redireccionamiento de salida

- comando `2>&1`: redirecciona la salida de error de comando a la salida estándar.
- comando `1>&2`: redirecciona la salida estándar de comando a la salida de error.
- comando `&> todo.txt`: redirecciona tanto la salida estándar como la de error hacia el fichero `todo.txt`, sobrescribiendo su contenido, y no se muestra por pantalla.
- comando `&» todo.txt`: redirecciona tanto la salida estándar como la de error, lo añade al contenido de `todo.txt` y no se muestra por pantalla.



Redireccionamiento de entrada

- Es posible redireccionar la entrada estándar (stdin):
comando `< ficheroConDatos.txt`.
- comando tomará como datos de entrada el contenido del
fichero `ficheroConDatos.txt`
- Esto incluye los saltos de líneas, por lo que, por cada salto de
línea se alimentará un `read`.



Tuberías

- Hasta ahora, redireccionamos entrada/salida comandos a partir de ficheros.
- **Tuberías**: redireccionar entrada/salida comandos entre si, sin usar ficheros.
- Sintaxis: `comando1 | comando2`
la entrada de comando2 será tomada de la salida de comando1 (salida estándar o de error)
- Se pueden encadenar más de dos comandos.
- Mismo resultado:
 - `cat archivoConDatos.txt | grep -i prueba`
 - `grep -i prueba < archivoConDatos.txt`



Redireccionamiento de salida: tee

- A veces queremos redirigir la salida de forma que aparezca por consola y al mismo tiempo se vuelque a fichero.
- Para esto, podemos usar el comando tee:

```
1 victor@victor-ayrna:~$ echo "Esto es una prueba"
2 Esto es una prueba
3 victor@victor-ayrna:~$ echo "Esto es una prueba" > f1
4 victor@victor-ayrna:~$ cat f1
5 Esto es una prueba
6 victor@victor-ayrna:~$ echo "Esto es una prueba" | tee f1
7 Esto es una prueba
8 victor@victor-ayrna:~$ cat f1
9 Esto es una prueba
10 victor@victor-ayrna:~$ echo "Esto es una prueba" | tee -a f1
11 Esto es una prueba
12 victor@victor-ayrna:~$ cat f1
13 Esto es una prueba
14 Esto es una prueba
```



Redireccionamiento de entrada: *Here documents*

- Los denominados *Here documents* son una manera de pasar datos a un programa de forma que el usuario pueda introducir más de una línea de texto. La sintaxis es la siguiente:

```
1 victor@victor-ayrna:~$ cat << secuenciaSalida
2 > hola
3 > que
4 > tal
5 > secuenciaSalida
6 hola
7 que
8 tal
```

- Características:
 - La entrada se va almacenando. Se van creando nuevas líneas pulsando la tecla *Intro*.
 - Se acaban de recibir datos cuando se detecta la cadena de texto que se seleccionó para indicar la salida, en este caso `secuenciaSalida`.



Redireccionamiento de entrada: *Here documents*

● ejemploHereDocument.sh:

```
1  #!/bin/bash
2
3  # Sin here documents
4  echo "*****"
5  echo "* Mi script V1 *"
6  echo "*****"
7  echo "Introduzca su nombre"
8
9  # Usando here documents
10 cat << EOF
11 *****
12 * Mi script V1 *
13 *****
14 Introduzca su nombre
15 EOF
```



Comando cat

- **cat:**

- Visualiza el contenido de uno o más ficheros de texto.

```
1 victor@victor-ayrna:~$ cat informacion.sh
2 #!/bin/sh
3 echo "Bienvenido $USER!, tu identificador es $UID."
4 echo "Esta es la shell número $SHLVL, que lleva $SECONDS arrancada."
5 echo "La arquitectura de esta máquina es $MACHTYPE y el cliente de
   terminal es $TERM"
6 victor@victor-ayrna:~$ cat informacion.sh parametros.sh
7 #!/bin/sh
8 echo "Bienvenido $USER!, tu identificador es $UID."
9 echo "Esta es la shell número $SHLVL, que lleva $SECONDS arrancada."
10 echo "La arquitectura de esta máquina es $MACHTYPE y el cliente de
    terminal es $TERM"
11 #!/bin/bash
12 echo "$#; $0; $1; $2; $*; $@"
13 victor@victor-ayrna:~$ cat < f1 > f2
14 # ¿Qué hacemos?
```



Comandos head, tail y wc

- **head y tail:**

- Muestran las primeras o las últimas **n** líneas de un fichero.

```
1 victor@victor-ayrna:~$ head -2 informacion.sh
2 #!/bin/sh
3 echo "Bienvenido $USER!, tu identificador es $UID."
4 victor@victor-ayrna:~$ tail -1 informacion.sh
5 echo "La arquitectura de esta máquina es $MACHTYPE y el cliente de
   terminal es $TERM"
```

- **wc:** muestra el número de líneas, palabras o caracteres de uno o varios ficheros:

```
1 victor@victor-ayrna:~$ wc -l informacion.sh
2 4 informacion.sh
3 victor@victor-ayrna:~$ wc -m informacion.sh
4 219 informacion.sh
5 victor@victor-ayrna:~$ wc -w informacion.sh
6 34 informacion.sh
7 victor@victor-ayrna:~$ wc -w numero*.sh
8 37 numeroRango11f.sh
9 46 numeroRango.sh
10 83 total
```



Comandos more, cmp y sort

- **more** fichero: muestra ficheros grandes, pantalla a pantalla.
- **cmp** f1 f2: compara dos ficheros y dice a partir de qué carácter son distintos.

```
1 victor@victor-ayrna:~$ cmp numeroRango.sh numeroRangoIf.sh
2 numeroRango.sh numeroRangoIf.sh son distintos: byte 95, línea 5
```

- **sort** [fichero]: ordena la entrada estándar o un fichero.
 - **sort**: ordena entrada estándar por orden alfabético.
 - **sort -r**: ordena entrada estándar por orden inverso.
 - **sort -n**: ordena entrada estándar por orden numérico.
 - **sort -t c**: cambia el caracter separador al caracter c.
 - **sort -k 3**: cambia la clave de ordenación a la tercera columna (por defecto, primera columna).



Comando sort

```
1 victor@victor-ayrna:~$ echo -e "18\n017\n9" | sort
2 017
3 18
4 9
5 victor@victor-ayrna:~$ echo -e "18\n017\n9" | sort -r
6 9
7 18
8 017
9 victor@victor-ayrna:~$ echo -e "18\n017\n9" | sort -n
10 9
11 017
12 18
13 victor@victor-ayrna:~$ echo -e "18\n017\n9" | sort -nr
14 18
15 017
16 9
17 victor@victor-ayrna:~$ echo -e "18 1\n017 2\n9 3" | sort -n -k 2
18 18 1
19 017 2
20 9 3
21 victor@victor-ayrna:~$ echo -e "18 1\n017 2\n9 3" | sort -n -k 1
22 9 3
23 017 2
24 18 1
25 victor@victor-ayrna ~ $ echo -e "1\t2\n2\t-1" | sort -t '$\t' -nk1
26 1      2
27 2      -1
28 victor@victor-ayrna ~ $ echo -e "1\t2\n2\t-1" | sort -t '$\t' -nk2
29 2      -1
30 1      2
```

Comando grep

- **grep [opciones] patron [fichero(s)]**: filtra el texto de un(os) fichero(s), mostrando únicamente las líneas que cumplen un determinado patrón.
 - -c: cuenta el número de líneas con el patrón.
 - -l: muestra el nombre de los ficheros que contienen el patrón.
 - -i: *case insensitive* (no sensible a mayúsculas).
 - También admite la entrada estándar (stdin).

```
1 victor@victor-ayrna:~$ grep ^c *
2 case.sh:case $x in
3 victor@victor-ayrna:~$ grep -l ^c *
4 case.sh
5 victor@victor-ayrna:~$ grep -c ^c *
6 arrayFor.sh:0
7 backup.sh:0
8 case.sh:1
9 ...
10 victor@victor-ayrna:~$ ls * | grep ^c
11 case.sh
12 comillas.sh
13 copiaFstab.sh
```



Comando grep

- **grep [opciones] patron [fichero(s)]:**
 - patron: “^” significa comienzo de la línea, “\$” significa fin de la línea, “.” significa cualquier carácter.

```
1 victor@victor-ayrna:~$ ls * | grep s\.sh$
2 comillas.sh
3 ejemploFor2Bis.sh
4 ejemploForImpFichScripts.sh
5 ejemploForListarFicheros.sh
6 operaciones.sh
7 parametros.sh
8 victor@victor-ayrna:~$ ls * | grep ^ejemplo.or
9 ejemploFor1.sh
10 ejemploFor2Bis.sh
11 ejemploFor2.sh
12 ejemploForArg.sh
13 ejemploForImpFichScripts.sh
14 ejemploForListarFicheros.sh
15 ejemploForTipoC.sh
```



Comando find

- `find [carpeta] -name "patrón"`: busca ficheros cuyo nombre cumpla el patrón y que estén guardados a partir de la carpeta carpeta (por defecto `.`).

```
1 victor@victor-ayrna:~$ find ~ -name "*.sh"
2 /home/victor/Escritorio/PAS/practica3-Shell/ejemplos/saludaUsuario.sh
3 /home/victor/Escritorio/PAS/practica3-Shell/ejemplos/operaciones.sh
4 /home/victor/Escritorio/PAS/practica3-Shell/ejemplos/backup.sh
```

- `find [carpeta] -size N`: busca ficheros cuyo tamaño sea N (+N: mayor que N, -N: menor que N).

```
1 victor@victor-ayrna:~$ find ~ -size 1024
2 /home/victor/.thunderbird/79tm6a62.default/blist.sqlite
3 /home/victor/.thunderbird/79tm6a62.default/cookies.sqlite
```



Comando find, basename y dirname

- `find [carpeta] -user usuario`: busca ficheros cuyo nombre usuario propietario sea usuario.

```
1 victor@victor-ayrna:~$ find ~ -user victor
2 /home/victor
3 /home/victor/.bashrc
```

- `basename fichero [.ext]`: Devuelve el nombre de un fichero sin su carpeta [y sin su extensión].
- `dirname fichero`: Devuelve la carpeta donde se aloja un fichero.

```
1 victor@victor-ayrna:~$ basename "/home/victor/Escritorio/recorrido.sh"
2 recorrido.sh
3 victor@victor-ayrna:~$ basename "/home/victor/Escritorio/recorrido.sh" .sh
4 recorrido
5 victor@victor-ayrna:~$ dirname "/home/victor/Escritorio/recorrido.sh"
6 /home/victor/Escritorio/
```



Comando stat

- `stat fichero`: nos muestra propiedades sobre un determinado fichero.

```
1 victor@victor-ayrna ~ $ stat missfont.log
2   Fichero: «missfont.log»
3   Tamaño: 0                      Bloques: 0                      Bloque E/S: 4096
4   fichero regular vacío
5   Dispositivo: 803h/2051d          Modo-i: 2938624          Enlaces: 1
6   Acceso: (0644/-rw-r--r--) Uid: ( 1000/ victor)  Gid: ( 1000/ victor)
7   Acceso: 2016-02-24 08:50:13.080997730 +0100
8   Modificación: 2016-02-24 08:50:13.080997730 +0100
9   Cambio: 2016-02-24 08:50:13.080997730 +0100
10  Creación: -
```

- `stat -c %a fichero`: nos permite personalizar la salida y obtener diferentes propiedades sobre un fichero³.

```
1 victor@victor-ayrna ~ $ stat -c "Permisos: %a. Tipo fichero: %F" missfont.
2   log
3   Permisos: 644. Tipo fichero: fichero regular vacío
```

³man stat para más información.



Comando tr

- `tr c1 c2`: reemplaza el carácter `c1` por el carácter `c2`.
Trabaja en el `stdin`.

```
1 victor@victor-ayrna ~ $ echo TIERRA | tr 'R' 'L'
2 TIELLA
```

- `tr -d c`: elimina el carácter `c` de la salida.

```
1 victor@victor-ayrna ~ $ echo TIERRA | tr -d R
2 TIEA
3 victor@victor-ayrna ~ $ echo TIERRA | tr -d RT
4 IEA
```



Expansión de llaves

- El operador *brace expansion* o expansión de llaves nos permite generar combinaciones de cadenas de texto de forma simple:

```
1 victor@victor-ayrna ~ $ echo fichero.{pdf,png,jpg}  
2 fichero.pdf fichero.png fichero.jpg
```

- Como se puede observar, la sintaxis es `cadena1{c1,c2,c3,...}`, de forma que se combinará `cadena1` con `c1`, `c2`, `c3...`
- `{c1..c2}` permite especificar todos el rango de caracteres desde `c1` hasta `c2`:

```
1 victor@victor-ayrna ~ $ echo {a..z}  
2 a b c d e f g h i j k l m n o p q r s t u v w x y z  
3 victor@victor-ayrna ~ $ echo {1..8}  
4 1 2 3 4 5 6 7 8  
5 victor@victor-ayrna ~ $ echo {1..3}{a..c}  
6 1a 1b 1c 2a 2b 2c 3a 3b 3c
```



Recorriendo ficheros

- Un ejemplo de redirección de comandos útil para recorrer ficheros:

```
1 find carpeta -name "patron" | while read f
2 do
3   ...
4 done
```

- Explica qué está sucediendo.
- **Cuidado:** la entrada está redirigida durante todo el bucle (no podremos hacer `read` dentro del bucle).
- ¿Cómo lo haríamos con un `for` sin usar tuberías?



Inciso: problemas con espacios en blanco y arrays

- Cuando intentamos construir un *array* a partir de una cadena, `bash` utiliza determinados caracteres para separar cada uno de los elementos del *array*.
- Estos caracteres están en la variable de entorno `IFS` y por defecto son el espacio, el tabulador y el salto de línea.

```
1 victor@Laptop:~$ array=(${echo "1 2 3"})
2 victor@Laptop:~$ echo ${array[0]}
3 1
4 victor@Laptop:~$ echo ${array[1]}
5 2
6 victor@Laptop:~$ echo ${array[2]}
7 3
8 victor@Laptop:~$ array=(${echo -e "1\t2\n3"})
9 victor@Laptop:~$ echo ${array[0]}
10 1
11 victor@Laptop:~$ echo ${array[1]}
12 2
13 victor@Laptop:~$ echo ${array[2]}
14 3
```



Inciso: problemas con espacios en blanco y arrays

- Esto nos puede producir problemas si estamos procesando elementos con espacios (por ejemplo, nombres de ficheros con espacios):

```
1 victor@Laptop:~$ array=(echo -e "El uno\nEl dos\nEl tres"))
2 victor@Laptop:~$ echo ${array[0]}
3 El
4 victor@Laptop:~$ echo ${array[1]}
5 uno
```

- **Solución:** cambiar el IFS para que solo se utilice el `\n`:

```
1 victor@Laptop:~$ OLDFIFS=$IFS
2 victor@Laptop:~$ IFS=$'\n'
3 victor@Laptop:~$ array=(echo -e "El uno\nEl dos\nEl tres"))
4 victor@Laptop:~$ echo ${array[0]}
5 El uno
6 victor@Laptop:~$ echo ${array[1]}
7 El dos
8 victor@Laptop:~$ IFS=$OLDFIFS
```



Referencias



Stephen G. Kochan y Patrick Wood

Unix shell programming.

Sams Publishing. Tercera Edición. 2003.



Evi Nemeth, Garth Snyder, Trent R. Hein y Ben Whaley

Unix and Linux system administration handbook.

Capítulo 2. *Scripting and the shell*.

Prentice Hall. Cuarta edición. 2010.



Aeleen Frisch.

Essential system administration.

Apéndice. *Administrative Shell Programming*.

O'Reilly and Associates. Tercera edición. 2002.



Programación y Administración de Sistemas

Práctica 1. Programación de la *shell*.

Víctor Manuel Vargas

Asignatura "Programación y Administración de Sistemas"
2º Curso Grado en Ingeniería Informática
Escuela Politécnica Superior
(Universidad de Córdoba)
vvargas@uco.es

22 de febrero de 2021

