

# UNIX, *Shell* y *Scripts*

Francisco Rosales García  
Ángel Herranz

Universidad Politécnica de Madrid

Otoño 2012



# Índice general

<b>1. El Entorno UNIX</b>	<b>1</b>
1.1. ¿Qué es UNIX? . . . . .	1
1.2. Usuarios y Grupos . . . . .	3
1.3. Sesión . . . . .	3
1.4. Mandatos . . . . .	4
1.5. Procesos . . . . .	7
1.6. Árbol de Ficheros . . . . .	7
1.7. Descriptores de fichero . . . . .	11
1.8. Intérprete de mandatos . . . . .	12
1.9. Variables de entorno . . . . .	13
1.10. Ficheros de texto ejecutables . . . . .	14
<b>2. Uso del <i>Shell</i></b>	<b>15</b>
2.1. El <i>prompt</i> . . . . .	15
2.2. Estado de terminación . . . . .	15
2.3. Primer y segundo plano . . . . .	16
2.4. Redirección . . . . .	16
2.5. Secuencia de mandatos . . . . .	17
2.6. Metacaracteres . . . . .	18
2.7. Uso interactivo . . . . .	19
2.8. Algunos mandatos útiles . . . . .	22
2.9. Configuración . . . . .	25
2.10. Explorando el mandato <b>ln</b> . . . . .	27
2.11. Explorando el mandato <b>find</b> . . . . .	27
2.12. Explorando el mandato <b>sort</b> . . . . .	28
2.13. Explorando secuencias de mandatos . . . . .	28
<b>3. Programación de <i>Scripts</i></b>	<b>29</b>
3.1. Edición . . . . .	30
3.2. Comentarios . . . . .	30
3.3. Variables y Entorno . . . . .	30
3.3.1. Variable <b>PATH</b> . . . . .	31
3.3.2. Variables especiales . . . . .	31
3.4. El mandato <b>test</b> . . . . .	32
3.5. Funciones . . . . .	34
3.6. Entrada, salida y error estándar . . . . .	35
3.7. Manipulación de tiras de caracteres . . . . .	36

<b>4. Interfaz de usuario del Sistema Operativo</b>	<b>39</b>
4.1. Introducción . . . . .	39
4.2. Evolución de la interfaz de usuario . . . . .	40
4.3. Funciones de la interfaz de usuario . . . . .	41
4.4. Interfaces alfanuméricas . . . . .	42
4.4.1. Funcionalidad del intérprete . . . . .	45
4.5. Interfaces gráficas . . . . .	47
4.6. Caso de estudio: UNIX . . . . .	49
4.6.1. El shell de UNIX . . . . .	49
4.6.2. Mandatos de UNIX . . . . .	65
<b>5. Herramientas de desarrollo</b>	<b>71</b>
5.1. Editor . . . . .	71
5.2. Compilador . . . . .	73
5.3. Depurador . . . . .	75
5.4. Bibliotecas . . . . .	76
5.5. Constructor . . . . .	78
5.6. Otras herramientas . . . . .	79

# Capítulo 1

## El Entorno UNIX

En este apartado intentaremos hacernos una idea de cuáles son las características y los conceptos básicos que se manejan en el entorno UNIX.

### 1.1. ¿Qué es UNIX?

Es sin lugar a dudas uno de los sistemas operativos más extensos y potentes que hay. Sus principales virtudes son:

**Multiusuario** En una máquina UNIX pueden estar trabajando simultáneamente muchos usuarios. Cada usuario tendrá la impresión de que el sistema es suyo en exclusiva.

**Multiproceso** Cada usuario puede estar ejecutando simultáneamente muchos procesos. Cada proceso se ejecuta independientemente de los demás como si fuese el único en el sistema. Todos los procesos de todos los usuarios son ejecutados concurrentemente sobre la misma máquina, de forma que sus ejecuciones avanzan de forma independiente. Pero también podrán comunicarse entre si.

**Multiplataforma** El núcleo del sistema operativo UNIX está escrito en más de un 95 % en lenguaje C. Gracias a ello ha sido portado a máquinas de todos los tamaños y arquitecturas. Los programas desarrollados para una máquina UNIX pueden ser llevados a cualquier otra fácilmente.

### ¿Cómo es UNIX?

Desde el punto de vista del usuario, hay que resaltar las siguientes características:

**Sensible al tipo de letra.** Distingue entre mayúsculas y minúsculas. No es lo mismo `unix` que `Unix` que `UNIX`.



**Ficheros de configuración textuales.** Para facilitar su edición sin necesidad de herramientas sofisticadas.

**Para usuarios NO torpes.** Otros Sistemas Operativos someten cada acción “peligrosa” (Ej. reescribir un fichero) al consentimiento del usuario.

En UNIX, por defecto, se entiende que el usuario sabe lo que quiere, y lo que el usuario pide, se hace, sea peligroso o no.

**Lo borrado es irrecuperable.** Es un ejemplo del punto anterior. Hay que tener cuidado. Como veremos más adelante existe la posibilidad de que el usuario se proteja de sí mismo en este tipo de mandatos.

## ¿Cómo es LINUX?

**POSIX.** Es una versión de UNIX que cumple el estándar *Portable Operating System Interface*.

**Libre.** Se distribuye bajo licencia GNU, lo que implica que se debe distribuir con todo su código fuente, para que si alguien lo desea, lo pueda modificar a su antojo o necesidad.

**Evoluciona.** Por ser libre, está en permanente desarrollo y mejora, por programadores voluntarios de todo el mundo.

En este capítulo se presenta el entorno UNIX desde el punto de vista de un usuario del mismo. El objetivo es que el usuario se sienta mínimamente confortable en UNIX. Veremos, en este orden:

1. Usuarios y Grupos
2. Sesión
3. Mandatos
4. Procesos
5. Árbol de Ficheros
6. Descriptores de fichero
7. Intérprete de mandatos
8. Configuración

## 1.2. Usuarios y Grupos

Para que una persona pueda utilizar un determinado sistema UNIX debe haber sido previamente dado de alta como usuario del mismo, es decir, debe tener abierta una cuenta en el sistema.

### Usuario

Todo usuario registrado en el sistema se identifica ante él con un nombre de usuario (*login name*) que es único. Cada cuenta está protegida por una contraseña (*password*) que el usuario ha de introducir para acceder a su cuenta. Internamente el sistema identifica a cada usuario con un número único denominado UID (*User Identifier*).

### Grupo

Para que la administración de los usuarios sea más cómoda estos son organizados en grupos. Al igual que los usuarios, los grupos están identificados en el sistema por un nombre y por un número (GID) únicos. Cada usuario pertenece al menos a un grupo.

### Privilegios

Las operaciones que un usuario podrá realizar en el sistema estarán delimitadas en función de su identidad, esto es por la pareja UID-GID, así como por los permisos de acceso al fichero o recurso al que desee acceder.

### Superusuario

De entre todos los usuarios, existe uno, denominado “superusuario” o *root* que es el encargado de administrar y configurar el sistema.

Es aquel que tiene como UID el número 0 (cero). El superusuario no tiene restricciones de ningún tipo. Puede hacer y deshacer lo que quiera, ver modificar o borrar a su antojo. Ser superusuario es una labor delicada y de responsabilidad.

La administración de un sistema UNIX es compleja y exige amplios conocimientos del mismo.

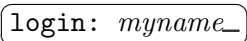
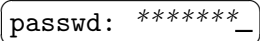

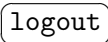

## 1.3. Sesión

Una *sesión* es el periodo de tiempo que un usuario está utilizando el sistema. Como sabemos, cada usuario tiene una cuenta que estará protegida por una contraseña o *password*. Para poder entrar en el sistema necesitamos un *terminal* que nos permita conectarnos a la máquina UNIX en cuestión.

A veces es tan sencillo como acceder a través del interfaz gráfico del sistema operativo si usted ya dispone de una máquina UNIX y poner en marcha un programa que haga de *terminal* (por ejemplo `gnome-terminal`, `xterm`, etc.). Si es así, acceda al sistema, ponga en marcha uno de estos programas y salte a la tarea 1.3.

En general, para poder acceder a una máquina UNIX desde otra máquina (probablemente Windows), necesitará un programa de SSH (por ejemplo `Putty`, `openssh`, etc.) al

que le pedirá que se conecte a la máquina en cuestión. Deberá entonces seguir los siguientes pasos para entrar y salir del sistema:

- T 1.1  *Para entrar al sistema introduzca su nombre de usuario o login name.*
- T 1.2  *Si ya ha puesto una contraseña deberá usted introducirla ahora, para que el sistema verifique que usted es quien dice ser.*
- T 1.3  *Al entrar en el sistema se arranca un programa que nos servirá de **intérprete de mandatos**, un shell en general y Bash en los sistemas más actuales (aunque es configurable). Nos presenta un mensaje de “apremio” o prompt. Algunos prompts clásicos son:*
- ```
$ _  
> _  
k0174@triqui3:~ $ _
```
- T 1.4  *Para terminar una sesión deberá usar este mandato o bien .*




## 1.4. Mandatos

El interprete de mandatos (podrá ver esto en más detalle en la sección 1.8) de todos los sistemas operativos (Bash, Csh, Ksh, CMD, etc.) admite mandatos que facilitan el uso del sistema. Su forma general es:

mandato [opciones] [argumentos...]

Los campos están separados por uno o más espacios. Algo entre corchetes [ ] es opcional. La notación ... indica uno o varios. Las opciones generalmente comenzarán por un - (menos).

A lo largo de esta presentación se irá solicitando que usted realice numerosas prácticas, observe el comportamiento del sistema o conteste preguntas...

- T 1.5  *Ahora debe usted entrar en el sistema como se indicó en el apartado anterior.*
- T 1.6  *Observe que UNIX distingue entre mayúsculas y minúsculas.*
- T 1.7  *Conteste, ¿sabría usted terminar la sesión?*

...en muchos casos se sugerirá que ejecute ciertos mandatos...



T 1.8 `who am i`  
*Obtenga información sobre usted mismo.*

T 1.9 `date`  
*Conozca la fecha y hora.*

`cal` T 1.10  
*Visualice un calendario del mes actual.*

...se darán más explicaciones para ampliar sus conocimientos...

La mayoría de los mandatos reconocen las opciones `-h` o `--help` como petición de ayuda sobre él mismo.

...y se solicitará que realice más practicas...

`date --help` T 1.11  
*Aprenda qué opciones admite este mandato.*

`?` T 1.12  
*¿Qué día de la semana fue el día en que usted nació?*

...así mismo, durante esta presentación iremos introduciendo un resumen de algunos mandatos que se consideran más importantes.

Si desea información completa sobre algo, deberá usar el mandato `man`.

`man [what]` \_\_\_\_\_ *Manual Pages*

Visualiza una copia electrónica de los manuales del sistema. Es seguro que nos asaltará a menudo la duda de como usar correctamente algún mandato o cualquier otra cosa. Deberemos consultar el manual.

Los manuales se dividen en secciones:

- 1 Mandatos (Ej. `sh`, `man`, `cat`).
- 2 Llamadas al sistema (Ej. `open`, `umask`).
- 3 Funciones de librería (Ej. `printf`, `fopen`).
- 4 Dispositivos (Ej. `null`).
- 5 Formato de ficheros (Ej. `passwd`).
- 6 Juegos.
- 7 Miscelánea.
- 8 Mandatos de Administración del Sistema.

`man -h` T 1.13  
*Busque ayuda breve sobre el mandato `man`.*

`man man`

T 1.14

*Lea la documentación completa del mandato `man`.*

`man [1-8] intro`

T 1.15

*En caso de conflicto, si ejemplo existe en diferentes secciones la misma hoja de manual, deberá explicitar la sección en la que buscar.*

*La hoja `intro` de cada sección informa de los contenidos de la misma.*

### Algunos mandatos útiles

A continuación se citan algunos de los muchísimos mandatos disponibles en UNIX. Si desea más información sobre alguno de ellos, consulte el manual.

`pwd` Visualización del directorio actual de trabajo.

`cd` Cambio del directorio actual de trabajo.

`mkdir` Creación de directorios.

`rmdir` Eliminación de directorios.

`echo` Muestra el texto que se indique por su salida.

`env` Muestra el valor de las variables de entorno.

`ls` Listado del contenido de un directorio.

`cat` Visualización del contenido de un fichero.

`pico` Un sencillo editor de ficheros.

`vi` El editor de ficheros más extendido en entorno UNIX.

`emacs` El editor de ficheros más potente disponible.

`rm` Eliminación de fichero.

`cp` Copiar ficheros.

`mv` Mover o renombrar ficheros.

`wc` Contar las palabras y líneas de un fichero.

`grep` Búsqueda de una palabra en un texto.

`sort` Ordenación de ficheros.

`find` Búsqueda de ficheros que cumplen determinadas características.

`chmod` Cambio de los permisos de un fichero.

`sleep` Simplemente espera que transcurra los segundos indicados.

`id` Muestra la identidad del usuario que lo ejecuta.

T 1.16



*¿Cuántas “palabras” contiene el fichero `/etc/password` del sistema?*

## 1.5. Procesos

Cando usted invoca un mandato, el **fichero ejecutable** del mismo nombre es ejecutado. Todo programa en ejecución es un *proceso*. Todo proceso se identifica por un número único en el sistema, denominado PID (*Process Identifier*).

### Concurrencia

En todo momento, cada proceso activo de cada usuario del sistema, esta compitiendo con los demás por ejecutar. El sistema operativo es el que marca a quién le toca el turno en cada momento.

Todo esto sucede muy muy deprisa, con el resultado de que cada proceso avanza en su ejecución sin notar la presencia de los demás, como si fuera el único proceso del sistema, pero en realidad todos están avanzando simultáneamente. A esta idea se le denomina *concurrencia*.

|                     |                                                                                                                                |        |
|---------------------|--------------------------------------------------------------------------------------------------------------------------------|--------|
| <code>ps</code>     | Permite ver una instantánea de los procesos en el sistema. Admite multitud de opciones tal y como muestra su página de manual. | T 1.17 |
| <code>man ps</code> |                                                                                                                                | T 1.18 |
| <code>top</code>    | Permite observar la actividad de los procesos en el sistema con cierta periodicidad.                                           | T 1.19 |

### Jerarquía de procesos

Todos los procesos del sistema se organizan en una jerarquía *padre-hijo(s)* que tiene como ancestro último al proceso denominado *init* cuyo PID es 1.

Todo proceso tiene también asociado un número que identifica a su proceso padre, es el PPID.

|                      |                                                                                                     |        |
|----------------------|-----------------------------------------------------------------------------------------------------|--------|
| <code>ps tree</code> | Permite ver una instantánea de la jerarquía de procesos en el sistema. Admite multitud de opciones. | T 1.20 |
|----------------------|-----------------------------------------------------------------------------------------------------|--------|

## 1.6. Árbol de Ficheros

Existe una única estructura jerárquica de nombres de fichero, es decir, **no existe el concepto de “unidad”**. En UNIX los dispositivos se “montan”.

### Jerarquía de directorios

A continuación se presenta cual sería la jerarquía de directorios básica presente en una máquina Linux. Sólo se presentan algunos de los directorios de primer y segundo nivel.

```

/          Directorio raíz.
|-- bin    Ejecutables globales (públicos).
|-- dev    Ficheros especiales (representan dispositivos).
|-- etc    Ficheros de configuración globales.
|-- home   Contiene las cuentas de usuario.
|-- lib    Librerías básicas globales (públicas).
|-- proc   Información instantánea sobre el sistema y sus procesos.
|-- root   Cuenta del superusuario.
|-- sbin   Ejecutables para administración.
|-- tmp    Espacio para ficheros temporales (público).
|-- usr    Segundo nivel. No imprescindible para el arranque.
|   |-- bin
|   |-- doc        Documentación básica.
|   |-- etc
|   |-- include    Ficheros de cabecera (.h) del lenguaje C.
|   |-- lib
|   |-- local      Tercer nivel. Añadidos a la instalación básica.
|   |-- man        Manuales del sistema.
|   |-- sbin
|   |-- share      Ficheros compartidos.
|   '-- tmp
'-- var        Spollers, cerrojos, correo, log, etc.

```

## Directorio raíz

La raíz de la jerárquica de nombres es el denominado directorio raíz y se denota con el carácter / (dividido por) **no por el \ (*backslash*)**.

## Directorio HOME

Nada más entrar en el sistema, el usuario es situado en el denominado directorio HOME de su cuenta.

Cada cuenta tiene su propio HOME, que es creado por el administrador del sistema al abrirle la cuenta al usuario.

Este directorio pertenece al usuario correspondiente. Por debajo de él podrá crear cuantos subdirectorios o ficheros quiera.

## Tipos de objeto

Para averiguar qué es o qué hay contenido bajo un nombre de fichero puede usar el mandato `file`.

Existen (básicamente) 3 tipos de “objetos”, a saber:

**Directorios:** Es un contenedor cuyas entradas se refieren a otros ficheros y/o directorios.

El uso de los directorios da lugar a la jerarquía de nombres. Todo directorio contiene siempre las siguientes dos entradas:

- (*punto*) Se refiere al mismo directorio que la contiene.

.. (*punto punto*) Se refiere al directorio padre de este, es decir, a su padre en la jerarquía.

**Ficheros normales:** Están contenidos en los directorios. Contienen secuencias de bytes que podrían ser códigos de programas, datos, texto, un fuente en C o cualquier otra cosa.

**Ficheros especiales:** Son como ficheros normales, pero no contienen datos, sino que son el interfaz de acceso a los dispositivos periféricos tales como impresoras, discos, el ratón, etc, etc.


Cada dispositivo, existente o posible, se haya representado por un fichero especial que se encuentra bajo el directorio `/dev`.


UNIX trata estos ficheros especiales (y por lo tanto a los dispositivos) exactamente igual que trata a los ficheros normales, de forma y manera que para los programas, los unos y los otros son **indistinguibles**.

`pwd` \_\_\_\_\_ *Print Working Directory*

Permite averiguar cuál es el directorio en que nos encontramos en cada momento, al cuál denominamos directorio actual de trabajo.

`pwd` T 1.21  
*Visualice su directorio actual de trabajo.*

 *Observe que se muestra el camino absoluto (desde el directorio raíz) de su directorio HOME.* T 1.22

 *Observe que las componente de un camino se separan entre sí con el carácter / (dividido por) y no por el \ (backslash).* T 1.23

`cd [dir]` \_\_\_\_\_ *Change Directory*

Con este mandato cambiamos de un directorio a otro. Admite como argumento el nombre del directorio al que queremos cambiar.

Existen tres casos especiales que son de utilidad.

`cd`  
Invocado sin argumento, nos devuelve directamente a nuestro HOME.

`cd .`  
El directorio de nombre `.` (*punto*) hace referencia siempre al directorio actual de trabajo. Realmente no nos va a cambiar de directorio.

`pwd` T 1.24  
*Observe que permanecemos en el mismo directorio.*

Como veremos más adelante, el directorio `.` resulta útil para referirnos a ficheros que “están aquí mismo”.

`cd ..`

Todo directorio contiene siempre una entrada de nombre `..` (punto punto) que hace referencia siempre al directorio padre del directorio actual.

T 1.25

`pwd`

*Observe que ascendemos al directorio padre.*

T 1.26

`cd ..`

*Repita la operación hasta llegar al raíz.*

T 1.27

`pwd`

*Observe que no se puede ir más allá del directorio raíz.*

T 1.28

`cd`

*¿A qué directorio hemos ido?*

El directorio HOME es siempre accesible mediante el símbolo `~`.

T 1.29

`cd /etc`

*Muévase al directorio `/etc`.*

T 1.30

`pwd`

*Compruebe en qué directorio está ejecutando el intérprete.*

T 1.31

`cd ~`

*Vuelva al directorio HOME.*

T 1.32

`pwd`

*Compruebe de nuevo en qué directorio está ejecutando el intérprete.*

`ls [-opt] [dirs...] _____ List Directory Contents`

Este mandato nos presenta información sobre los ficheros y directorios contenidos en el(os) directorio(s) especificado(s).

Si no se indica ningún directorio, informa sobre el directorio actual de trabajo.

Este mandato admite cantidad de opciones que configuran la información obtenida. Entre ellas las más usadas son, por este orden:

`-l` Formato “largo”. Una línea por fichero o directorio con los campos: permisos, propietario, grupo, tamaño, fecha y nombre.

`-a` Informa también sobre los nombres que comienzan por `.` (punto), que normalmente no se muestran.

`-R` Desciende recursivamente por cada subdirectorio encontrado informando sobre sus contenidos.

Si necesita explorar más opciones, consulte el manual.

T 1.33

`ls`

*Observe que su directorio HOME parece estar vacío.*

T 1.34

`ls -a`

*Pero no lo está. Al menos siempre aparecerá las entradas `.` y `...`*

T 1.35



*Es posible que aparezcan otros nombres que con `ls` no vio.*

*Este mandato evita mostrar aquellas entradas que comienzan por el carácter punto. Podríamos decir que los ficheros o directorios cuyo nombre comienza por un punto están "ocultos", pero se pueden ver si usamos la opción `-a`.*

*Suelen ser ficheros (o directorios) de configuración.*

`ls -la`

T 1.36

*Si queremos conocer los detalles del "objeto" asociado a cada entrada de un directorio, usaremos la opción `-l`.*



*¿A quién pertenecen los ficheros que hay en su HOME?*

T 1.37

*¿Qué tamaño tienen?*

*¿Cuándo fueron modificados por última vez?*

`ls /home`

T 1.38

*Observe el HOME de otros usuarios de su sistema.*

*Inspeccione el contenido de sus directorios.*

`ls -la /`

T 1.39

*Muestra el "tronco" del árbol de nombres.*

`man 7 hier`

T 1.40

*Lea una descripción completa de la jerarquía del sistema de ficheros UNIX que está usando.*

`ls -??? /bin`

T 1.41

*Obtenga el contenido del directorio `/bin` ordenado de menor a mayor el tamaño de los ficheros.*



*Observe que aparece un fichero de nombre `ls`. Ese es el fichero que usted está ejecutando cada vez que usa el mandato `ls`.*

T 1.42

*Esos ficheros son sólo una parte de los que usted podría ejecutar, hay muchos más.*



*¿A quién pertenece el fichero `/bin/ls`?*

T 1.43



*Sus permisos permiten que usted lo ejecute.*

T 1.44

## 1.7. Descriptores de fichero

Los procesos manejan ficheros a través de los denominados descriptores de fichero. Por ejemplo, el resultado de abrir un fichero es un descriptor. Las posteriores operaciones de manejo (lectura, escritura, etc.) de este fichero reciben como parámetro el descriptor.

Desde el punto de vista de nuestro programa, un descriptor no es más que un número entero positivo (0, 1, 2, ...).

## Descriptores estándar

Los tres primeros descriptores de fichero son los denominados “estándar”, y se considera que siempre están disponibles para ser usados.

Normalmente estarán asociados al terminal pero podrían estar asociados a un fichero o a cualquier otro objeto del sistema de ficheros, pero este es un detalle que no ha de preocuparnos a la hora de programar.

Los programas que se comporten de forma “estándar” los utilizarán adecuadamente.

**Entrada estándar.** Es la entrada principal al programa. Los programas que se comportan de forma estándar reciben los datos que han de procesar leyendo de su entrada estándar (descriptor de fichero número 0).

**Salida estándar.** Es la salida principal del programa. Los programas que se comportan de forma estándar emiten sus resultados escribiendo en su salida estándar (descriptor de fichero número 1).

**Error estándar.** Es la salida para los mensajes de error del programa. Si un programa que se comporta de forma estándar ha de emitir un mensaje de error deberá hacerlo a través de su error estándar (descriptor de fichero número 2).

## 1.8. Intérprete de mandatos

La interacción del usuario con el sistema UNIX puede ser a través de un interfaz gráfico, pero lo más común es que sea a través de un interfaz textual, esto es, a través del diálogo con un intérprete de mandatos.

En UNIX se denomina *shell* (cáscara) al intérprete de mandatos, esto es, el programa capaz de interpretar y poner en ejecución los mandatos que el usuario teclea. No existe un único *shell* sino muchos.

T 1.45 `ls -l /bin/*sh`

*Los ficheros que aparecen son los diferentes shell disponibles. Quizás aparezca algún otro fichero. Quizás aparezcan enlaces simbólicos (otro tipo de objeto del sistema de ficheros).*

Se distinguen fundamentalmente en la sintaxis que reconocen y en las facilidades que proporcionan para su uso interactivo y para usarlos como lenguaje de programación de “guiones” de mandatos (*scripts*).

**sh** Es el denominado *Bourne Shell*. Es el primer *shell* que se desarrolló para UNIX, y es el preferido para la programación de *scripts*.

**csh** El “C” *shell*. Se llama así porque la sintaxis que reconoce se aproxima a la del lenguaje C. También es uno de los primeros *shell* desarrollados para UNIX, y se ha preferido para uso interactivo.

**bash** *Bourne Again Shell*, es una versión más evolucionada del **sh**, más apta para el uso interactivo.



**tcsh** Versión mejorada del **csch**, con más facilidades para el uso interactivo como completar nombres de fichero y edición de línea de mandatos.

otros Existen otros muchos: **ksh**, **zsh**, **rc**, etc. (para gustos, colores).

Ya hemos estado usando el *shell*, de hecho, observe que si está usted en su cuenta, entonces **está usted ejecutando uno de ellos en este momento**, probablemente Bash.



## 1.9. Variables de entorno

Además de los argumentos que indicamos en su invocación, cada mandato que ejecutemos recibirá una serie de parejas "nombre=valor", que son las denominadas variables de entorno.

Las variables de entorno son pasadas de proceso padre a proceso hijo por un mecanismo de herencia.

Las variables del entorno se utilizan para configurar ciertos comportamientos de los procesos. Por poner algunos ejemplos citaremos las siguientes:

**HOME** Su valor es el nombre del directorio raíz de nuestra cuenta en el sistema.

**PATH** Enumera un conjunto de directorios donde se localizará los ejecutables que invoquemos.

**TERM** Indica el nombre o tipo del terminal que estemos usando.

**USER** Indica el nombre del usuario que somos.

**SHELL** Es el nombre completo del intérprete de mandatos que estemos usando.

Existen otras muchas más variables de entorno, y podríamos situar en el entorno cualquier otra información que sea de nuestro interés.

Podremos visualizar las variables de entorno existentes haciendo uso del mandato **env**.

**env**

T 1.46

*Visualice todas las variables de entorno.*

Para visualizar el valor de una variable de entorno determinada se puede usar el mandato **echo** junto con la expansión de dicha variable.

**echo Hola mundo**

T 1.47

*Logre que el intérprete de mandatos imprima por pantalla **Hola mundo**.*

**echo \$HOME**

T 1.48

*Imprima el valor de la variable **HOME**.*



*¿Qué shell está usted usando?*

T 1.49

## Variable de entorno PATH

Para entender el comportamiento del *shell* es conveniente profundizar en la misión de la variable de entorno PATH.


Cuando le pedimos al *shell* que ejecute un mandato, este lo buscará en los directorios indicados en la variable de entorno PATH y sólo en ellos. Es conveniente que la variable PATH se refiera al menos a los directorios más comunes, esto es, `/bin`, `/usr/bin` y `/usr/local/bin`.

**Por razones de seguridad, es altamente recomendable que el directorio actual (“.”) NO esté incluido en la variable PATH.**

T 1.50  ¿Por qué?


## 1.10. Ficheros de texto ejecutables

Si un fichero de texto tiene permisos de ejecución (`man chmod`) y su primera línea reza `#! ejecutable`, podremos ejecutar directamente este fichero, y su contenido será leído e interpretado por el ejecutable. De esta manera podemos crear nuestros propios ficheros de mandatos ejecutables, que denominamos “guiones” (*scripts*).

T 1.51  Escriba un fichero cuya primera línea sea `#! /bin/cat`, y luego añada las líneas de texto de desee. Dele permisos de ejecución y ejecútelo.

Como podrá observar, un fichero de texto ejecutable no tiene porqué estar necesariamente asociado a un *shell*, sino que puede asociarse a cualquier mandato que haga uso de su entrada estándar.

Además en la primera línea podemos añadir opciones del mandato (una opción como máximo, aunque puede ser compuesta).

T 1.52  Modifique el fichero anterior para que al ser ejecutado cuente las líneas y las palabras de sí mismo, pero **no** los caracteres. Para ello consulte el manual sobre el mandato `wc`.

Este mecanismo, de ficheros de texto ejecutables nos permitirá la realización de ficheros de mandatos, que serán interpretados por el *shell* de nuestra elección.

## Capítulo 2

# Uso del *Shell*

En este apartado exploraremos el uso interactivo del intérprete de mandatos. Vamos a ver algunas características básicas del *shell* que nos serán de mucha utilidad.

### 2.1. El *prompt*

Cuando el usuario entra en su cuenta a través de un terminal, el sistema arranca un nuevo intérprete de mandatos asociado a dicho terminal. Un *shell* arrancado de esta manera está en modo interactivo y nos informará de ello mostrando en el terminal un *prompt* o mensaje de apremio. Un *shell* está a nuestro servicio.

En este documento se muestran los diálogos con el *shell* de la siguiente manera:

```
$ _
```

Se utilizará habitualmente será el símbolo \$ como *prompt* del shell.

### 2.2. Estado de terminación

Todo mandato UNIX termina devolviendo un valor que permite identificar su estado de terminación. Un valor distinto de 0 indica, por convención, que ocurrió un error de algún tipo, mientras que un valor 0 indica que funcionó como se esperaba.

Para conocer el estado de terminación del último mandato utilizado se puede usar la variable de entorno “?”.

```
ls
```

*Ejecute un mandato que no falle.*

T 2.1

```
echo $?
```

*Compruebe el estado de terminación de dicho mandato, debería ser 0.*

T 2.2

```
ls Supercalifragilisticexpialidocious
```

*Ejecute un mandato que falle.*

T 2.3

```
echo $?
```

*Compruebe el estado de terminación de dicho mandato, debería ser distinto de 0.*


T 2.4

Los programadores deben ser consistentes con esta convención por lo que tendrán que asegurarse de devolver un estado de terminación significativo cuando escriben programas.

## 2.3. Primer y segundo plano

Hasta ahora hemos venido arrancando mandatos que “toman el control del terminal” hasta que terminan. Se dice de ellos que ejecutan en primer plano (*foreground*). Podemos solicitar la ejecución de varios mandatos en primer plano invocando cada uno en una línea diferente, pero también podríamos hacerlo separándolos con el carácter ‘;’.

T 2.5 `sleep 3; ls /; sleep 3; who`

T 2.6  *Observe como se ejecuta un mandato después del anterior en estricta secuencia. Observe cómo se respetan los tiempos entre mandatos.*

En UNIX es posible solicitar la ejecución de mandatos en segundo plano (*background*) poniendo el carácter & al final de los mismos. El mandato se ejecutará en concurrencia con otros mandatos que ejecutemos en primer o segundo plano a continuación.

T 2.7 `sleep 3 & ls / & sleep 3 & who &`

T 2.8  *Observe como se ejecutan concurrentemente y la salida de los mandatos se mezcla.*

## 2.4. Redirección

En UNIX podemos redirigir la entrada estándar, salida estándar y/o error estándar de los mandatos. Esta es una característica muy muy útil, ya que, sin necesidad de modificar en absoluto un mandato podemos usarlo para que tome su entrada de un fichero en vez de tener que teclearla (el teclado es habitualmente la entrada estándar de los programas que ejecutemos desde el intérprete).

De igual manera en vez de que la salida del mandato aparezca en la pantalla (salida estándar), podríamos enviarla a un fichero para usarla con posterioridad.

La sintaxis correcta para realizar redirecciones es añadir al final del mandato la redirección con la siguiente notación:

`mandato < fichero`

En su ejecución, el mandato usará **fichero** como entrada, leerá de él y no del teclado.

`mandato > fichero`

En su ejecución, la salida del mandato será enviada a **fichero** y no a la pantalla. Si **fichero** no existiera con anterioridad, será creado automáticamente, pero si existe, será previamente truncado a tamaño cero.

`mandato >> fichero`

Añade la salida del mandato al final del fichero indicado.

```
mandato 2> fichero
```

Los mensajes de error que un mandato pueda generar también pueden ser redirigidos a un fichero, pero normalmente preferimos que se visualicen por la pantalla.

```
mandato 2>&1
```

Redirige la salida de error a donde quiera que esté dirigida la salida estándar.



Obtenga en un fichero de nombre `todos_los_ficheros` un listado en formato largo de todo el árbol de ficheros recorriéndolo recursivamente desde el directorio raíz.

T 2.9

Dado que este mandato puede llevar un rato en ejecutar, lo arrancamos en segundo plano, para poder seguir realizando otros mandatos.

```
ls -l todos_los_ficheros
```

T 2.10

Compruebe que tamaño del fichero está aumentando.



Aparecerán en pantalla sólo los mensajes de error que el mandato produce al no poder acceder a ciertos directorios.

T 2.11

```
grep login_name < todos_los_ficheros > mis_ficheros
```

T 2.12

Use el mandato `grep` para filtrar las líneas del fichero de entrada que contienen el texto indicado, en este caso su nombre de usuario. Deje el resultado en un fichero de nombre `mis_ficheros`.

```
ls Supercalifragilisticexpialidocious > ficheroraro
```

T 2.13



¿Qué ha ocurrido? ¿Cuál es el contenido del fichero `ficheroraro`?

T 2.14

```
ls Supercalifragilisticexpialidocious 2> ficheroraro
```

T 2.15



¿Qué ha ocurrido esta vez? ¿Cuál es el contenido del fichero `ficheroraro`? Explíquelo.

T 2.16

## 2.5. Secuencia de mandatos

### *Pipeline*

En UNIX podemos redirigir la salida estándar de un mandato directamente a la entrada estándar de otro. Con ello construimos secuencias de mandatos que procesan el resultado del anterior y emiten su resultado para que sea procesado por el siguiente mandato de la secuencia.


Esto lo conseguimos separando los mandatos con el carácter “|” (*pipe*).


Esta es una facilidad muy poderosa de UNIX, ya que, sin necesidad de ficheros intermedios de gran tamaño podemos procesar grandes cantidades de información.

```
ls -la /home | grep login_name
```


T 2.17

*Obtiene un listado en formato largo de todos los directorios de cuentas y filtra aquellas líneas que contienen nuestro nombre de usuario.*

T 2.18  *No se crea ningún fichero intermedio.*

T 2.19  *Obtenga el listado “todos sus ficheros” de la sección anterior, conectando el resultado al mandato **grep** para quedarse sólo con sus ficheros. Mande el resultado a un nuevo fichero de nombre **MIS\_ficheros**. Póngalo así, con mayúsculas, para no machacar el fichero nombrado así pero en minúsculas.*

*Recuerde invocar el mandato con **&** para poder seguir haciendo otras cosas.*

T 2.20  *Cuando hayan terminado de ejecutar los mandatos, compruebe las diferencias entre los ficheros de nombre **mis\_ficheros** y **MIS\_ficheros**. Para ello utilice el mandato **diff**, consultando el manual si es necesario.*

## Secuencias condicionales

Es también posible separar mandatos o secuencias con los siguientes símbolos:

**&&** Solicitamos que el mandato o secuencia a la derecha del símbolo sólo se ejecute si el de la izquierda terminó correctamente.

**||** Solicitamos que el mandato o secuencia a la derecha del símbolo sólo se ejecute si el de la izquierda terminó incorrectamente.

T 2.21 

```
true && echo "El anterior terminó bien"
```

T 2.22 

```
false || echo "El anterior terminó mal"
```

## 2.6. Metacaracteres

Los metacaracteres (*wildcards*) son caracteres comodín. Con su uso podemos hacer referencia a conjuntos de ficheros y/o directorios, indicando la expresión regular que casa con sus nombres.

**?** Comodín de carácter individual.

**\*** Comodín de tira de cero o más caracteres.

**~** Abreviatura del directorio *home*.

**~user** Abreviatura del directorio *home* de *user*.

**[abc]** Casa con un carácter del conjunto especificado entre los corchetes.

[*x-y*] Casa con un carácter en el rango especificado entre los corchetes.

[^...] Niega la selección indicada entre los corchetes.

{*str*,...} Agrupación. Casa sucesivamente cada tira.



*Obtenga un listado en formato largo, de los ficheros de los directorios /bin y /usr/bin/ que contengan en su nombre el texto sh.* T 2.23










*Obtenga un listado en formato largo, de los ficheros del directorio /dev que comiencen por tty y no terminen en dígito.* T 2.24

## 2.7. Uso interactivo

Un buen *shell* para uso interactivo ofrece servicios que le facilitan la tarea al usuario. Esto significa que existen múltiples combinaciones de teclas asociadas con alguna funcionalidad especial. Para conocerlas completamente deberá consultar el manual.



### Edición de línea

Podemos mover el cursor () sobre la línea de mandatos que estamos tecleando con  , borrar antes y sobre el cursor con  y , ir al inicio y al final con  y , etc.

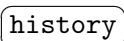


*Observe que pasa si intenta editar una línea más larga que la pantalla.* T 2.25

### Histórico de mandatos

Si hace poco que tecleamos un mandato no es preciso volverlo a teclear. Basta con buscarlo haciendo uso de las flechas del teclado  y .

El mandato interno **history** nos informa de todos los mandatos que hemos usado últimamente.



*Visualice los mandatos que hasta ahora ha realizado en su shell.* T 2.26



*Observe que **history** guarda los mandatos anteriores de una sesión otra.* T 2.27



*Repite el último mandato.* T 2.28



*Repite el mandato que ocupa la posición 5 en la historia.* T 2.29

## Fin de datos

La combinación de teclas **Ctrl D** en una línea vacía indica “fin de datos”.

Si la pulsamos y hay algún programa leyendo del terminal, entenderá que no hay más datos y terminará normalmente.


T 2.30 **cat**

*Es un mandato que lo que lee de su entrada lo escribe en su salida. Escriba varias líneas de datos para observar su comportamiento.*

*Como está leyendo del terminal, para que termine normalmente, deberá indicarle “fin de datos”.*

T 2.31 **\$ Ctrl D**

*El shell también es un programa que lee de su entrada standard. Indíquele “fin de datos”.*

T 2.32 

*El shell entenderá que no hay más datos, terminará normalmente y por lo tanto nos sacará de la cuenta.*

*Si ha sido así y este comportamiento no le agrada, hay una forma de configurar el shell para evitar que suceda. Más adelante la veremos.*

## Control de trabajos

En UNIX podemos arrancar mandatos para que se ejecuten en segundo plano (*background*) poniendo el carácter **&** al final de la línea.

También, desde su *shell* puede suspender el trabajo que está en primer plano (*foreground*), rearrancar trabajos suspendidos, dejarlos en segundo plano, matarlos, etc.

**Ctrl C** Pulsando esta combinación de teclas cuando tenemos un programa corriendo en primer plano, le mandaremos una señal que “lo matará”.


**Ctrl Z** El programa en primer plano quedará suspendido y volveremos a ver el *prompt* del *shell*.

**jobs** Este mandato interno nos informa de los trabajos en curso y de su estado. Podremos referirnos a uno concreto por la posición que ocupa en esta lista.

**fg** Ejecutándolo, el último proceso suspendido volverá a primer plano. Para referirnos al segundo trabajo de la lista de trabajos usaremos **fg%2**.

**bg** Ejecutándolo, el último proceso suspendido volverá a ejecutar, pero en segundo plano! Es como si lo hubiéramos arrancado con un **&** detrás.

**kill** Este mandato interno permite enviarle una señal a un proceso, con lo que, normalmente, lo mataremos.

T 2.33 

*Arranque varios procesos en background, por ejemplo varios **sleep 60 &**, y a continuación consulte los trabajos activos.*





Arranque un proceso en primer plano, por ejemplo `cat` y suspéndalo. Consulte los trabajos activos. Vuelva a ponerlo en primer plano y luego mándele una señal que lo mate.

T 2.34



¿Cómo lo mataría si se encuentra en segundo plano?

T 2.35

## Completar nombres de fichero

Si tenemos un nombre de fichero escrito “a medias” y pulsamos el tabulador (`↩`), el `tcsh` intentará completar dicho nombre. Si lo que pulsamos es la combinación `Ctrl D`, nos mostrará la lista de nombres que empiezan como él.

También funciona para nombres de mandato.

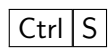


Escriba `ls .cs↩` y luego intente `ls .cs Ctrl D`, observe la diferencia.

T 2.36

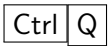
## Control de *scroll* de terminal

Denominamos terminal al conjunto pantalla/teclado o a la ventana a través de la cual dialogamos con el *shell*.



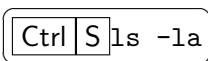
Congela el terminal. No mostrará nada de lo que nuestros programas escriban, ni atenderá a lo que tecleemos.

Si por descuido pulsamos `Ctrl S` podríamos pensar que el sistema se ha quedado colgado, pero no es así.



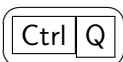
Descongela el terminal. Lo que los programas hubiesen escrito mientras la pantalla estuvo congelada, así como lo que hubiésemos tecleado, aparecerá en este momento.

El uso común de la pareja `Ctrl S` `Ctrl Q` es para controlar la salida de programas que emiten muchas líneas.



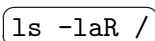
Observe que, de momento, nada de lo que hemos tecleado aparece en el terminal.

T 2.37



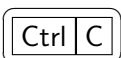
Verá como el mandato que tecleamos aparece y se ejecuta con normalidad.

T 2.38



El resultado de este mandato son demasiadas líneas de texto. Juegue con `Ctrl S` `Ctrl Q` a parar y dejar correr la pantalla.

T 2.39



Cuando se canse, mate el trabajo.

T 2.40

## 2.8. Algunos mandatos útiles

`less [files...]` \_\_\_\_\_ *On Screen File Browser*

Filtro para la visualización por pantalla de ficheros. Es una versión mejorada del clásico mandato `more`.

Si lo usamos indicando un conjunto de ficheros, podremos pasar al siguiente y al anterior con `:n` y `:p` respectivamente.

Con este mandato sólo podremos ver los ficheros, pero no modificarlos, más adelante veremos algún editor.

T 2.41 `less *_ficheros`

*Visualice el contenido de los ficheros que existen ya en su HOME que terminan por “\_ficheros”.*

T 2.42 `ls *`

*Observe que el metacaracter `*` no alude a los ficheros cuyo nombre empieza por punto.*

T 2.43 `ls -aLR | less`

*Liste todos los ficheros del sistema pero visualícelos por páginas.*

`passwd` \_\_\_\_\_ *Change User Password*

UNIX protege el acceso a las cuentas de los usuarios mediante una palabra clave.

Las cuentas del sistema, sus contraseñas y resto de información, están registradas en el fichero `/etc/passwd`.

T 2.44 `less /etc/passwd`

*Visualice el contenido del fichero de contraseñas.*

*Este fichero contiene la información sobre las cuentas de usuario que hay en su sistema.*

*Compruebe que existe una cuenta con su nombre de usuario.*

T 2.45 `man 5 passwd`

*Estudie cuál es el contenido del fichero de contraseñas.*

Un usuario puede cambiar su palabra clave haciendo uso del mandato `passwd`. Es muy importante que la contraseña no sea trivial, ni una fecha, ni una matrícula de coche, teléfono, nombre propio, etc.

T 2.46 `passwd`

*¡Si no tiene un buen password, cámbielo ahora mismo!*

`cp orig dest` \_\_\_\_\_ *Copy Files*

Obtiene una copia de un fichero o bien copia un conjunto de ficheros al directorio indicado.

Admite variedad de opciones, que no vamos a describir en este documento (si tiene dudas, consulte el manual).

Interesa destacar que este **es un mandato peligroso**, dado que (por defecto) hace lo que le pedimos sin consultar, lo cuál puede suponer que machaquemos ficheros que ya existan. Para que nos consulte deberemos usarlo con la opción `-i`.

```
cp /etc/passwd ~
```

T 2.47

*Copie en su HOME el fichero /etc/passwd.*



*Visualice que su contenido es realmente una copia del original.*

T 2.48



*Copie el fichero .cshrc sobre el que ahora tiene en su cuenta con el nombre passwd.*

T 2.49



*¿Le ha consultado cp antes de reescribir? ¿Se ha modificado el fichero?*

T 2.50



*Intente las mismas operaciones usando la opción -i.*

T 2.51

`rm files...` \_\_\_\_\_ *Remove Files*

Borrar ficheros. Para ver sus opciones consulte el manual.

También **es un mandato peligroso**. En este caso, `rm` no sólo hace lo que le pedimos sin consultar, sino que en UNIX **no existe manera de recuperar** un fichero (o directorio) que haya sido borrado. Por lo tanto se hace imprescindible usarlo con la opción `-i`.



*Intente borrar el fichero passwd de su cuenta con `rm -i passwd`. Conteste que no.*

T 2.52



*Borre el fichero passwd que ahora tiene en su cuenta.*

T 2.53



*Observe que efectivamente ha sido eliminado.*

T 2.54



*Intente borrar el fichero /etc/passwd.*

T 2.55



*¿Puede hacerlo? ¿Porqué?*

T 2.56

`mv orig dest` \_\_\_\_\_ *Move (Rename) Files*

Cambia el nombre de un fichero o bien mueve un fichero o conjunto de ficheros a un directorio.

También **es un mandato peligroso**, porque podría “machacar” el(os) fichero de destino sin pedirnos confirmación. Se debe usar con la opción `-i`.



*Intente mover el fichero /etc/passwd a su HOME.*

T 2.57



*Intente mover algún fichero de su cuenta (uno que no le valga) al directorio `/etc`.* T 2.58



T 2.59 *¿Puede hacer estas operaciones? ¿Porqué?*



T 2.60 *Cambie de nombre al fichero `mis_ficheros` a `MIS_ficheros`.*



T 2.61 *¿Pudo hacerlo? ¿Le consultó?*

`mkdir dir _____` *Make Directory*

Crea un nuevo directorio. Por supuesto, si ya existe un fichero o directorio con dicho nombre dará un error.

Los directorios que usted cree serán suyos. Sólo podrá crear ficheros o directorios en aquellos directorios donde usted tenga permiso de escritura.



T 2.62 *Cree en su cuenta un subdirectorio de nombre `subarbol`.*



T 2.63 *Visualice su contenido con `ls -la`.*



T 2.64 *Cree otros subdirectorios más profundos, pero sin hacer uso del `cd`.*

T 2.65 `ls -laR ~`

*Visualice el contenido actual de su HOME recursivamente.*

`rmdir dir _____` *Remove Directory*

Borra un directorio. Por supuesto el directorio debe estar vacío.



T 2.66 *Intente borrar un directorio **no** vacío, Ej. `subarbol`.*



T 2.67 *¿Puede hacerlo?*



T 2.68 *Visualice el contenido de `subarbol`.*



T 2.69 *Elimine los directorios más profundos, y vaya ascendiendo, hasta conseguir eliminar el directorio `subarbol`.*

## 2.9. Configuración

Cada usuario puede configurar el comportamiento de su cuenta editando convenientemente ciertos ficheros presentes en su HOME que dependen de qué *shell* use usted.

El fichero de configuración que es de nuestro interés en este momento es:

**.bashrc** Se ejecuta siempre que el usuario arranca un nuevo **bash**.

**.cshrc** Se ejecuta siempre que el usuario arranca un nuevo **csh** o **tcsh**.

Dado que estos ficheros sólo se leen al arrancar el *shell*, para que los cambios que hagamos en ellos tengan efecto, deberemos bien salir de la cuenta y volver a entrar o bien hacer uso del mandato interno **source** para que su *shell* lea e interprete el contenido del fichero.

### Mandatos peligrosos

En UNIX, **si se borra un fichero o un directorio no hay forma de recuperarlo**. Además, los mandatos obedecen ciegamente a quien los usa, y no consultan. Para evitar meter la pata es conveniente usar ciertos mandatos de forma que consulten antes de actuar. Para ello, vamos a usar el mandato interno **alias**, para protegernos de los mandatos problemáticos.

### Fin de datos

Evitaremos que el *shell* termine a la pulsación **Ctrl****D** pidiéndole que lo ignore.

#### **alias**

T 2.70

*Ejecutando este mandato observará que alias están en funcionamiento.*

*Los alias modifican “al vuelo” los mandatos que queremos ejecutar, justo antes de ser finalmente ejecutados.*

#### **pico file**

T 2.71

*Es un editor de uso muy sencillo. Úselo para realizar la siguiente modificación en el fichero de configuración de su shell.*

*Escoja entre la versión de la izquierda, si usted usa el **tcsh** o el **csh**, o la de la derecha si usa el **bash**. Añada al final del fichero de configuración indicado las siguientes líneas:*

**.cshrc**

```
alias cp 'cp -i'
alias mv 'mv -i'
alias rm 'rm -i'
set ignoreeof
```

**.bashrc**

```
alias cp='cp -i'
alias mv='mv -i'
alias rm='rm -i'
IGNOREEOF=10
```

#### **source**

T 2.72

*Recuerde que debe pedirle a su shell que lea e interprete el contenido del fichero de configuración, para que tenga constancia de la modificación que acaba de hacer.*

`alias`

T 2.73

*Observará si los alias que ha establecido están en funcionamiento.*

T 2.74

`Ctrl` `D`

*Compruebe si esta secuencia hace terminar su shell.*

## 2.10. Explorando el mandato ln

Con este mandato podremos crear “enlaces”, esto es, crear otros nombres que se refieran a un fichero ( o directorio) ya existente.

```
ln ...
```

T 2.75

*Cree un enlace físico (hard link) del fichero /etc/passwd en su cuenta.*

```
ln -s ...
```

T 2.76

*Cree un enlace simbólico (symbolic link) del fichero /etc/passwd en su cuenta.*

```
ls -l
```

T 2.77

*Observe los atributos (permisos, propietario, número de enlaces, etc.) de los enlaces que acaba de crear.*



*¿Tiene usted permiso para escribir sobre dichos enlaces? Intente escribir en ellos, por ejemplo redirigiendo sobre ellos la salida de otro mandato.*

T 2.78

```
rm ...
```

T 2.79

*Ahora borre los enlaces que creó. ¿Puede hacerlo? ¿Cómo es posible? De una explicación razonable.*

## 2.11. Explorando el mandato find

Explore en profundidad las opciones de este mandato. Para ello consulte el manual y realice los siguientes ejercicios.

```
find ... -name ...
```

T 2.80

*Localice todos los ficheros del sistema cuyo nombre termina en sh. Guarde el resultado en un fichero de nombre significativo.*

*NOTA: use comillas simples para evitar la expansión de los metacaracteres.*

```
find ... -ls
```

T 2.81

*Repita la operación anterior, pero ahora obtenga un listado que detalle los atributos de los ficheros encontrados.*

```
find ... -mtime ...
```

T 2.82

*Localice todos los ficheros del sistema que han sido modificados en los 3 últimos días. Guarde el resultado en un fichero de nombre significativo.*

```
find ... -user ...
```

T 2.83

*Localice todos los ficheros del sistema que le pertenecen. Guarde el resultado en un fichero de nombre significativo.*

```
find ... -and ...
```

T 2.84

*Localice todos los ficheros del sistema que han sido modificados en los 3 últimos días y le pertenecen. Guarde el resultado en un fichero de nombre significativo.*

`find ... -type`

T 2.85

*Localice todos los ficheros del sistema que no son fichero ni directorio ni fichero especial (orientados a caracter o a bloque). Guarde el resultado en un fichero de nombre significativo.*

## 2.12. Explorando el mandato sort

Explore en profundidad las opciones de este mandato. Para ello consulte el manual y realice los siguientes ejercicios.

T 2.86 `ls -lRa`

*Obtenga un listado detallado de todos los ficheros de su cuenta. Guarde el resultado en un fichero de nombre MIOS.*

T 2.87 `sort ...`

*Ordene las líneas del fichero MIOS, alfabéticamente.*

T 2.88 `sort ...`

*Ordene las líneas del fichero MIOS, por tamaño del fichero, de menor a mayor.*

T 2.89 `ls -S ...`

*Intente obtener el mismo resultado usando directamente el mandato `ls`.*

T 2.90 `sort ...`

*Ordene las líneas del fichero MIOS, por fecha del fichero, del más antiguo al más moderno.*


## 2.13. Explorando secuencias de mandatos

T 2.91 `sort ... | head ...`


*Combine el mandato anterior con el mandato `head` en una secuencia para quedarnos sólo con las 4 primeras líneas.*

T 2.92 `sort ... | tail ...`


*Combine el mandato anterior con el mandato `tail` en una secuencia para quedarnos sólo con las 4 últimas líneas.*

T 2.93 

*Combine los mandatos `find`, `sort` y `head` para componer una secuencia de mandatos cuyo resultado sea un listado que detalle los atributos de los 10 ficheros (no directorios) del sistema con mayor número de enlaces.*

T 2.94 

*De forma similar, y combinando también el mandato `cut` componga una secuencia de mandatos cuyo resultado sea sólo en nombre de los 10 ficheros de tamaño mayor en el sistema, uno por línea.*

T 2.95 

*De forma similar, componga una secuencia de mandatos cuyo resultado sea sólo en nombre de los 10 ficheros de más antiguos del sistema.*



## Capítulo 3

# Programación de *Scripts*

En este apartado usaremos el *Bash* para programar ficheros de mandatos (*scripts*), pero en ocasiones, para probar un mandato complejo o un *script* sencillo, lo usaremos de forma interactiva.

Simplemente ejecutaremos el mandato `sh` y este nos devolverá el carácter `$` que es *prompt* característico del `sh`. Para salir de nuevo al `cs`h pulsaremos *Ctrl-D* o bien usaremos el mandato interno `exit`.

### Ejemplo

```
user@hostname $
user@hostname $ ps
  PID TTY STAT  TIME COMMAND
 1154  p1 S    0:02 -csh
 2556  p1 R    0:00 ps
user@hostname $ echo Mi PID es $$
Mi PID es 1154
user@hostname $ date
Thu Jul 30 18:20:37 CEST 1998
user@hostname> who am i
hostname.domain!user  tty1      Jul 30 11:23 (:0.0)
user@hostname $ sh
$ ps
  PID TTY STAT  TIME COMMAND
 1154  p1 S    0:02 -csh
 2561  p1 S    0:00 sh
 2562  p1 R    0:00 ps
$ echo $$
2561
$ ^D
user@hostname $
```

### 3.1. Edición

Para poder realizar los ejercicios de este capítulo será necesario utilizar un editor de texto plano. Le recomendamos que utilice uno de los editores clásicos de UNIX (**vi**, **emacs**, **pico**, **nano**, etc.). El uso de todos los editores mencionados se basan en la activación de comandos (borrar, avanzar línea, seleccionar un texto, etc.) mediante la combinación de teclas. Usted tendrá que adaptarse al uso de este tipo de editores antes de continuar con los ejercicios.

### 3.2. Comentarios

Tanto el programa en su conjunto como cada una de sus partes o funciones deben estar bien documentados, a base de comentarios que clarifiquen qué hace y cómo lo hace.

Los comentarios son cualquier texto, comenzando con el carácter almohadilla (#) y terminando al final de la línea. `Esto se ejecuta. # Esto no.`

### 3.3. Variables y Entorno


El *shell* proporciona variables cuyo valor son tiras de caracteres. El nombre de una variable podrá contener letras, dígitos y el carácter '\_', pero no puede empezar por '\_'. Las minúsculas y las mayúsculas se consideran distintas.


Para dar valor a una variable escribiremos `variable=valor`, sin espacios en blanco! Para obtener el valor de una variable usaremos la notación `$variable` o `${variable}` si a continuación del nombre de la variable viene letra o dígito.

El *entorno* es un subconjunto de las variables del *shell* que son heredadas por todo proceso derivado de este. Una variable puede ser añadida al conjunto de variables de entorno con el mandato `export variable`. Las variables de entorno pueden ser visualizadas con el mandato `env`.

T 3.1  *Escriba el mandato identidad con el siguiente contenido y ejecútelo:*

```
#!/bin/bash
# identidad
# Identifica al usuario usando las variables de entorno.
echo Usted es el usuario \"$USER\".
```

T 3.2  *Como podrá observar el Bash lee, interpreta y ejecuta el contenido del fichero. Este pequeño programa hace uso además de la variable de entorno USER.*

T 3.3  *Amplíe el programa anterior para que visualice una descripción completa del usuario y la máquina basándose en la información contenida en el entorno.*

### 3.3.1. Variable PATH

Cuando le pedimos al *shell* que ejecute un mandato, lo intentará encontrar entre sus mandatos internos y entre las funciones que tengamos definidas. Si no se encuentra, lo buscará en el conjunto de directorios indicados en la variable de entorno PATH y sólo en ellos.

Dado que un *script* puede fallarle a alguna persona si tiene un PATH diferente o incompleto, se recomienda incluir en una correcta especificación de esta variable de entorno.

```
PATH=/usr/bin:/bin
```



*Experimente con el siguiente mandato en PATH:*

T 3.4

```
#!/bin/bash -x
# en_PATH
# Experimento sobre la variable PATH.
# El valor de esta variable afecta a la
# localización de los mandatos que usamos.
PATH=""; ls # No se encuentra.
PATH=""; /bin/ls # SI se encuentra.
PATH=/bin; ls # SI se encuentra.
```



*Observe que la opción -x del shell permite “depurar” nuestro programa.*

T 3.5

### 3.3.2. Variables especiales

La denominadas variables especiales, no pertenecen al entorno. Es el *shell* quien les da valor.

\$0 Nombre del *script*.

\$9 Parámetro del *script* o función en la posición indicada (1 a 9).

\$# Número de parámetros posicionales o argumentos.

\$\* Lista de argumentos.

\$@ Lista de argumentos. Pero "\$@" no es una tira de caracteres, sino copia exacta de la lista de argumentos

\$\$ Identificador del propio proceso *shell*.

\$? Valor devuelto por el último mandato ejecutado.

#! Identificador del último proceso lanzado en segundo plano.



*Codifique el siguiente mandato argumentos que visualiza el valor de todas las variables especiales relativas a los argumentos. Experimente la diferencia entre "\$@", "\$@", "\$\*" y \$\*.*

T 3.6

```

#!/bin/bash
# argumentos args...
# Experimento sobre variables especiales relativas a argumentos.

echo '"$@"' vale "$@", y si lo recorro obtengo:
for ARG in "$@"; do echo __${ARG}__; done

echo '$@' vale $@, y si lo recorro obtengo:
for ARG in $@; do echo __${ARG}__; done

echo '"$*"' vale "$*", y si lo recorro obtengo:
for ARG in "$*"; do echo __${ARG}__; done


echo '$*' vale $*, y si lo recorro obtengo:
for ARG in $*; do echo __${ARG}__; done


```

*Ejecútelo con los siguientes argumentos.*

```
argumentos 1 "Dos 2" "3 III" 'Cuatro ****'
```

T 3.7  ¿Dónde y cuándo se expanden los comodines de 'Cuatro \*\*\*\*'?

T 3.8  ¿Porqué es precisa la notación \${ARG}?

T 3.9 

*El mandato interno **shift** desplaza los parámetros posicionales un lugar a la izquierda, perdiéndose el antiguo valor de \$1, que pasará a valer lo que valía \$2, etc. y decrementa \$#.*

*Añada al final del mandato argumentos un último recorrido del siguiente modo:*

```

echo Y si voy desplazando los parámetros con "shift" obtengo:
while [ $# -ne 0 ]      # Equivalente a...      until test $# -eq 0
do
    echo __${1}__
    shift
done

```

### 3.4. El mandato test

Es uno de los mandatos más usados en *shell scripts*, estúdielo en profundidad puede serle muy útil man test. Puede ser un mandato interno del *shell* o ser un mandato externo.

```
ls -l /usr/bin/[ /usr/bin/test
```

Puede usarse de dos maneras: `test expression` o `[ expression ]`. Observe que sus argumentos (así como el carácter `]`) deben estar convenientemente separados por espacios.

Un buen *shell script* debe verificar que los argumentos con los que ha sido invocado (si es preciso alguno) son correctos, y si no lo son debe indicarle al usuario la forma correcta de utilización. Para verificar el número de argumentos usaremos el mandato `test`.



*Amplíe el siguiente mandato `test_test` para que realice todos los posibles `test` de uno o dos argumentos.* T 3.10

```
#!/bin/bash
# test_test arg1 [arg2]
# Usa el mandato test(1) de todas las formas posibles
# sobre 1 o 2 argumentos.

test_as_file()
{
    for opt in b c d e f g
    do
        test -$opt "$1" && : TRUE || : false
    done
}

if [ $# -eq 2 ]
then
    set -x          # Activamos el modo depuración.
    test_as_file "$1"
    test_as_file "$2"
    ...
elif [ $# -eq 1 ]
then
    ...
else
    ...
fi
```

*Ejécútelo con...* `"", /dev/console, ., 2 02, "1 -lt 3", etc.`




*¿\$1 hace referencia al mismo parámetro cuando se usa en el programa principal que cuando se usa dentro de una función?* T 3.11



*¿Porqué es precisa la notación entrecomillada "\$1"?* T 3.12



*¿Que significa la notación `mandato && mandato || mandato`?* T 3.13

T 3.14  : es un mandato interno. ¿Para qué sirve? (`man bash` y busque :).

T 3.15  `set` es un mandato interno. ¿Para qué sirve?

### 3.5. Funciones

Usando el *Bourne Shell* podemos hacer uso de grupos de mandatos, agrupados entre llaves '`{}`' allí donde podríamos haber usado un mandato simple.


Las funciones en *Bourne Shell* son simplemente el resultado de ponerle un nombre único a un conjunto de mandatos agrupados mediante llaves. Son una de las características más poderosas del *sh* y es necesario acostumbrarse a usarlas.

```
function() {  
    # Conjunto de mandatos  
    # que implementan la función  
    return 0 # terminación correcta  
}
```

- En una función los parámetros posicionales `$1`, `$2`, etc. son los argumentos de la función (no los argumentos del *script*).
- Una función puede ser usada de igual modo que se puede usar un mandato externo.
- Una función devolverá su propio estado de terminación usando la sentencia `return`. Si se usa `exit` dentro de una función, se estará terminando el *script* completo, no solo la función.

```
return 0      # Terminación correcta
```

- Las funciones son ideales para encapsular una determinada labor. Se pueden redirigir su entrada, salida o error, conectarlas por pipes, etc.
- Las funciones del *Bourne Shell* **pueden ser recursivas!**

T 3.16  Escriba el mandato árbol que reciba como argumentos una lista de ficheros y/o directorios e imprima los nombres completos de cada fichero y recorra cada directorio de forma recursiva en profundidad.

- El resultado debe ser un nombre completo de fichero por línea, convenientemente endentado (con tabuladores) para indicar la profundidad de la recursión.
- Utilice una función recursiva de nombre `sub_arbol` que reciba como primer argumento una tira de tabuladores, como segundo el nombre del directorio que se está explorando en este momento y a continuación cualquier número de nombres de fichero o directorio a evaluar.
- Para obtener el contenido de un directorio utilice '`ls -A1 $DIR`'.

### 3.6. Entrada, salida y error estándar

La entrada, salida y error estándar son los descriptores de fichero 0, 1 y 2. Cada uno tiene una misión concreta y deben ser usados convenientemente.

**Error** Los mensajes de error deben aparecer en el estándar error y no en la salida estándar.

```
echo Fichero de configuración inexistente. >&2
```

A veces, por estética, interesará silenciar posibles mensajes de error generados por los mandatos que usamos. Lo conseguiremos dirigiendo la salida de error de dicho mandato a `/dev/null`.

```
expr $NUM 2>/dev/null || reintentar
```

**Salida** La salida estándar es para los mensajes que van dirigidos al usuario o a otros procesos.

```
echo -n "Desea crearlo?[s] "
```

**Entrada** La entrada estándar es para recoger la respuesta de los usuarios o la salida de otros procesos.

```
read RESPUESTA
```

```
CWD='pwd'
```

**Interactivo** Es posible que la entrada estándar no esté asociada al terminal (haya sido redirigida) y no haya un usuario al cual consultar. Si es preciso nuestros programas pueden tener esto en cuenta y comportarse de forma distinta según estén siendo usados de forma interactiva o no `tty -s`.

*in-situ* Podemos incorporar en nuestro propio *script* el texto que queremos que sea la entrada a un mandato.

```
cat <<'Fin_ayuda'
```

Las líneas de texto siguientes, hasta una que literalmente rece `Fin_ayuda` serán la entrada al mandato. Si la marca de “fin de documento *in-situ*” presenta algún *quoting* el texto será literal, si no serán expandidas las variables y los mandatos entre comillas inversas.



*Escriba el mandato **reto** que retará al usuario a que adivine un número (por ejemplo el PID del proceso). Para ello leerá los valores numéricos que el usuario teclee y responderá indicando si el número secreto es mayor, menor o igual, en cuyo caso terminará.*

T 3.17

- Para comparar los valores numéricos haga uso del mandato `test`.
- El programa sólo debe admitir ser usado de forma interactiva.
- El programa debe llamar al usuario por su nombre haciendo uso de la variable `USER`.
- Debe evitar morir si el usuario le manda una señal desde teclado. Para ello capturará las señales `SIGINT` y `SIGQUIT` (`man 7 signal`) haciendo uso del mandato interno `trap`.
- Si el usuario responde al reto con `Ctrl-D` (fin de datos), el programa pedirá una confirmación positiva antes de terminar.

### 3.7. Manipulación de tiras de caracteres

Internamente el *shell* ofrece pocos mecanismos para manipular tiras de caracteres, si bien siempre se puede hacer uso de mandatos externos capaces de manipular ficheros de texto de forma sofisticada.

**case** La sentencia **case** permite localizar patrones sencillos en tiras de caracteres. Se usa por ejemplo para analizar los parámetros de entrada de los *shell scripts* y reconocer opciones.

```
case $1 in
  0) echo Cero ;;
  1|2|3|4) echo Positivo ;;
  -[1234]) echo Negativo ;;
  *.* ) echo Decimal ;;
  *) echo Otra cosa ;;
esac
```

**read** El mandato interno **read** lee una línea de su entrada estándar y asigna valor a las variables que se le pasan como argumentos. La línea se divide por los caracteres que se consideran separadores de campo que normalmente son el tabulador y el espacio.

**set** El mandato interno **set**. Si le pasamos una lista de argumentos, los separa por los caracteres que se consideran separadores de campo y los asigna como valores a los parámetros posicionales **\$1**, **\$2**, etc. y actualiza **\$#**.

Además, el mandato **set** sirve para activar opciones del propio *shell*, como el modo depuración, modo *verbose*, etc. Consulte el manual.

**IFS** El conjunto de caracteres que el *shell* considera separadores de campo son los que contiene la variable de entorno **IFS** (*Input Field Separator*). Cambiando el valor de esta variable podemos dividir líneas por los caracteres que queramos.

```
IFS=: read field_1 field_2 rest <file
```

```
IFS=', ' set "1,2,3,4 5.6"
```

Observe en el ejemplo anterior la forma de asignar valor a una variable en la misma línea en la que se invoca al mandato, sin separarlos por **;**. Usando esta sintaxis la modificación del valor de la variable sólo afecta a dicha invocación, y no es permanente.

T 3.18



Documente adecuadamente el siguiente código. Para ello, consulte el manual hasta entender cómo se consigue el valor adecuado en cada caso.

```
#!/bin/bash -x
# hora
# Experimento manipulación de tiras de caracteres.
TIME='date | cut -c12-19'
TIME='date | cut -d\ -f4'
```



```

TIME='date | sed 's/. * . * \(. *\) . * .*/\1/'
TIME='date | awk '{print $4}''
TIME='set `date`; echo $4'
TIME='date | (read u v w x y z; echo $x)'
TIME='date +%T'

```



Escriba el mandato **adivina** que tendrá que intentar adivinar el valor de un parámetro numérico que sólo el usuario conoce, preguntándole cosas que sólo debe responder con si o no.

T 3.19

- El programa sólo debe admitir ser usado de forma interactiva.
- El programa debe presentar un menú de los conceptos que sabe adivinar (edad, altura, etc.) y leer la opción que el usuario seleccione. Para identificar la opción utilice la sentencia **case**.
- Escriba la función auxiliar *pregunta\_si* que reciba como argumentos el texto de una pregunta, se lo presente al usuario, lea su respuesta (usando el mandato interno **read**) y con un **case** compruebe si la respuesta es afirmativa. Si lo es terminará bien, sino terminará mal.
- Escriba la función auxiliar *dicotomía* que controle dos valores numéricos menor y mayor que representan un rango, y pregunte al usuario sobre el valor medio del rango, de forma que se pueda reducir el rango a la mitad. Para calcular el valor medio utilice el mandato externo **expr**.



Codifique el programa **ckpw** que realice ciertas verificaciones de seguridad relativas al fichero **/etc/passwd**.

T 3.20

- Debe leer cada línea del fichero **/etc/passwd** separándola en sus componentes (login, password, uid, gid, name, home y shell).
- Comprobará que el password no está vacío.
- Que el uid y el gid son numéricos.
- Que el home es un directorio que no tiene permisos de escritura para el usuario del programa.
- Que el shell existe como fichero ejecutable y que está catalogado como shell válido en el fichero **/etc/shells**.



## Capítulo 4

# Interfaz de usuario del Sistema Operativo

### 4.1. Introducción

Cuando un usuario trabaja con un computador necesita poder interactuar con el sistema operativo para poder llevar a cabo operaciones tales como ejecutar un programa o borrar un archivo. El sistema operativo, por lo tanto, además de dotar de servicios (llamadas al sistema) a las aplicaciones, debe de proporcionar una interfaz a los usuarios que les permita dar instrucciones al sistema para realizar diversas operaciones. Sin esta interfaz, aunque el sistema operativo estuviese listo para dar servicio a las aplicaciones, el usuario no podría arrancar ninguna.

La interfaz de usuario de los sistemas operativos, al igual que la de cualquier otro tipo de aplicación, ha sufrido una gran evolución. Esta evolución ha venido condicionada, en gran parte, por la enorme difusión del uso de los computadores, que ha tenido como consecuencia que un gran número de usuarios sin conocimientos informáticos trabajen cotidianamente con ellos. Se ha pasado de interfaces alfanuméricas que requerían un conocimiento bastante profundo del funcionamiento del computador a interfaces gráficas que ocultan al usuario la complejidad del sistema proporcionándole una visión intuitiva del mismo.

Ha existido también una evolución en cómo se integra la interfaz de usuario con el resto del sistema operativo. En la actualidad el módulo que maneja la interfaz de usuario no está generalmente dentro del sistema operativo, sino que se trata de un conjunto de programas externos que usan los servicios del mismo como cualquier otro programa. Esta estrategia de diseño proporciona una gran flexibilidad pudiendo permitir que cada usuario utilice un programa de interfaz que se ajuste a sus preferencias o que, incluso, cree uno propio. El sistema operativo se caracteriza, por lo tanto, por los servicios que proporciona y no por su interfaz que, al fin y al cabo, puede ser diferente para los distintos usuarios.

De forma estricta, por lo tanto, la interfaz del sistema operativo no es parte del mismo y por ello muchos libros de texto apenas tratan este tema. Sin embargo, consideramos que un curso de introducción a los sistemas operativos debe tratar este tema con más profundidad puesto que es la parte con la que se tiene el primer contacto y, además, es un buen ejemplo de un programa que usa casi todos los servicios del sistema operativo.

Este capítulo se va a centrar, por lo tanto, en la interfaz de usuario del sistema operativo, o sea, la parte del sistema encargada de atender y llevar a cabo las peticiones de

los usuarios del computador. En primer lugar se presentará la evolución de la interfaz de usuario: desde los sistemas primitivos que carecían de ella hasta los sistemas actuales que proporcionan interfaces muy sofisticadas. A continuación se expondrá cuál es la funcionalidad típica de una interfaz de usuario. En las dos siguientes secciones se analizarán las características de las interfaces alfanuméricas y gráficas respectivamente. Por último, como caso de estudio, se presentará de forma más detallada la interfaz de usuario de UNIX.

## 4.2. Evolución de la interfaz de usuario

La evolución de la interfaz de usuario ha ido muy ligada a la del propio sistema operativo que, a su vez, ha estado condicionada por el desarrollo de la tecnología.

En los primeros sistemas informáticos, en los cuales prácticamente no existía sistema operativo, los usuarios manipulaban directamente interruptores y botones de la máquina para poder dar instrucciones al computador. Por lo tanto, se podría considerar que la interfaz de usuario de estos sistemas era puramente hardware.

Las siguientes generaciones de computadores, hasta aproximadamente mediados de los sesenta, fueron sistemas por lotes. En estos sistemas los usuarios no trabajaban interactivamente con el computador. Por motivos de rendimiento, un operador del sistema agrupaba los programas de distintos usuarios, típicamente escritos en tarjetas perforadas, y el sistema operativo se encargaba de ir ejecutándolos. El usuario tenía que incluir una serie de tarjetas de control al principio y al final de cada trabajo para especificar todas las características del mismo. Por ejemplo, si se trataba de un programa FORTRAN, el usuario debía especificarlo en la tarjeta de control correspondiente para que el sistema operativo usara el compilador de ese lenguaje para procesar el programa. El contenido de estas tarjetas de control se fue sofisticando sucesivamente llegando a constituir un lenguaje que, típicamente, se denominó *Lenguaje de control de trabajos*. Estas tarjetas de control, que estaban identificadas por un símbolo especial en su primera columna (por ejemplo, un \$), eran la manera que usaban los usuarios para dar instrucciones al sistema operativo. El sistema operativo incluía un intérprete de las tarjetas de control que se encargaba de leerlas y llevar a cabo las órdenes especificadas en ellas. Este mecanismo constituía, por lo tanto, una interfaz de usuario rudimentaria de carácter no interactivo.

Con el desarrollo de los sistemas de tiempo compartido a finales de la década de los sesenta, los usuarios podían trabajar interactivamente con el computador. Cada usuario trabajaba en un terminal alfanumérico del computador y desde el mismo podía introducir directamente instrucciones al sistema escribiendo *líneas de mandatos*. La parte del sistema operativo que se encargaba de atender a los usuarios se denominaba típicamente *intérprete de mandatos* y su función era ir leyendo cada mandato introducido por un usuario y, a continuación, ejecutarlo. Inicialmente se trataba de una extensión de los lenguajes de control de trabajos pero de carácter interactivo. Progresivamente se fueron sofisticando hasta convertirse en una especie de lenguaje de programación con variables y estructuras de control. La interfaz de este tipo de sistemas era muy potente y estaba orientada a usuarios con un conocimiento bastante profundo del sistema. Otro aspecto importante es que en la mayoría de los sistemas el intérprete de mandatos dejó de estar incluido en el sistema operativo pasando a ser un conjunto de programas sin ninguna característica diferente del resto de los programas. De este forma, se facilitaba que cada usuario pudiese usar el que se ajustase mejor a sus preferencias o que, incluso, construyese uno propio.

El principio de los ochenta se caracterizó por la aparición de los computadores personales. Se trataba de máquinas dedicadas normalmente a un único usuario que trabajaba de forma interactiva. Las primeras interfaces de estos sistemas eran similares a las que existían en los sistemas de tiempo compartido aunque algo simplificadas. Sin embargo, dados la enorme difusión del uso de estos sistemas entre usuarios sin conocimientos informáticos y el desarrollo de máquinas con una gran capacidad gráfica, se comenzaron a desarrollar interfaces gráficas basadas en ventanas que ocultaban al usuario la complejidad del sistema proporcionándole una visión intuitiva del mismo. El éxito de este tipo de interfaces hizo que se extendieran a toda la gama de computadores.

El resultado final de esta evolución es la convivencia de ambos tipos de interfaces, gráficas y alfanuméricas, en un entorno de ventanas para, de esta forma, satisfacer los requisitos de un amplio rango de usuarios: desde los especializados que precisan introducir complejas líneas de mandatos para expresar sus necesidades hasta los no especializados que requieren una visión más *amigable* del sistema basada en iconos gráficos que el usuario manipula mediante un ratón.

En la actualidad hay una proliferación de sistemas formados por un conjunto de máquinas, posiblemente heterogéneas en hardware y software, conectadas por una red. El objetivo principal de este tipo de sistemas y de sus interfaces de usuario va a ser ofrecer al usuario la capacidad de manejar los recursos del sistema en red de forma transparente a la localización de los mismos ocultando lo más posible el carácter distribuido del sistema.

### 4.3. Funciones de la interfaz de usuario

La principal misión de la interfaz, sea del tipo que sea, es permitir al usuario acceder y manipular los objetos y recursos del sistema. Aunque las funciones de cada interfaz tendrán características específicas, se puede intentar clasificarlas en las siguientes categorías genéricas:

- Manipulación de archivos y directorios. La interfaz debe proporcionar operaciones para crear, borrar, renombrar y, en general, procesar archivos y directorios.
- Ejecución de programas. El usuario tiene que poder ejecutar programas y controlar la ejecución de los mismos (por ejemplo, parar temporalmente su ejecución o terminarla incondicionalmente).
- Herramientas para el desarrollo de las aplicaciones. El usuario debe de disponer de utilidades, tales como editores, compiladores, ensambladores, enlazadores y depuradores, para construir sus propias aplicaciones.
- Comunicación con otros sistemas. Existirán herramientas para acceder a recursos localizados en otros sistemas accesibles a través de una red de conexión.
- Información de estado del sistema. El usuario dispondrá de utilidades para obtener informaciones tales como la fecha, la hora, el número de usuarios que están trabajando en el sistema o la cantidad de memoria disponible.
- Configuración de la propia interfaz y del entorno. Cada usuario tiene que poder configurar el modo de operación de la interfaz de acuerdo a sus preferencias. Un ejemplo sería la configuración de los aspectos relacionados con las características específicas

del entorno geográfico del usuario (lenguaje, formato de fechas y de cantidades de dinero, etc.). La flexibilidad de configuración de la interfaz será una de las medidas que exprese su calidad.

- Intercambio de datos entre aplicaciones. El usuario va disponer de mecanismos que le permitan especificar que, por ejemplo, una aplicación utilice los datos que genera otra.
- Control de acceso. En sistemas multiusuario la interfaz debe encargarse de controlar el acceso de los usuarios al sistema para mantener la seguridad del mismo. Normalmente el mecanismo de control estará basado en que cada usuario autorizado tenga una contraseña que deba introducir para acceder al sistema.
- Otras utilidades y herramientas. En aras de no extender innecesariamente la clasificación, agrupamos en este apartado utilidades misceláneas tales como calculadoras o agendas.
- Sistema de ayuda interactivo. La interfaz debe incluir un completo entorno de ayuda que ponga a disposición del usuario toda la documentación del sistema.

Para concluir esta sección, es importante resaltar que en un sistema, además de las interfaces disponibles para los usuarios normales, pueden existir otras específicas destinadas a los administradores del sistema. Más aún, el propio programa (residente normalmente en ROM) que se encarga de la carga del sistema operativo proporciona generalmente una interfaz de usuario muy simplificada y rígida que permite al administrador realizar operaciones tales como pruebas y diagnósticos del hardware o la modificación de los parámetros almacenados en la memoria RAM no volátil de la máquina que controlan características de bajo nivel del sistema.

#### 4.4. Interfaces alfanuméricas

La característica principal de este tipo de interfaces es su modo de trabajo basado en líneas de texto. El usuario, para dar instrucciones al sistema, escribe en su terminal un mandato terminado con un carácter de final de línea. Cada mandato está normalmente estructurado como un nombre de mandato (por ejemplo, **borrar**) y unos argumentos (por ejemplo, el nombre del archivo que se quiere borrar). Nótese que algunos intérpretes permiten que se introduzcan varios mandatos en una línea. El *intérprete de mandatos*, que es como se denomina típicamente al módulo encargado de la interfaz, lee la línea escrita por el usuario y lleva a cabo las acciones especificadas por la misma. Una vez realizadas, el intérprete escribe una indicación (*prompt*) en el terminal para notificar al usuario que está listo para recibir otro mandato. Este ciclo repetitivo define el modo de operación de este tipo de interfaces. El usuario tendrá disponibles un conjunto de mandatos que le permitirán realizar actividades tales como manipular archivos y directorios, controlar la ejecución de los programas, desarrollar aplicaciones, comunicarse con otros sistemas, obtener información del estado del sistema o acceder al sistema de ayuda interactivo.

Esta forma de operar basada en líneas de texto viene condicionada en parte por el tipo de dispositivo que se usaba como terminal en los primeros sistemas de tiempo compartido. Se trataba de teletipos que imprimían la salida en papel y que, por lo tanto, tenían

intrínsecamente un funcionamiento basado en líneas. La disponibilidad posterior de terminales más sofisticados que usaban una pantalla para mostrar la información y ofrecían, por lo tanto, la posibilidad de trabajar con toda la pantalla, no cambió, sin embargo, la forma de trabajo de la interfaz que continuó siendo en modo línea. Como reflejo de esta herencia obsérvese que en el mundo UNIX se usa el término **tty** (abreviatura de *teletype*) para referirse a un terminal aunque no tenga nada que ver con los primitivos teletipos.

A pesar de que el modo de operación básico apenas ha cambiado, su estructura e implementación han evolucionado notablemente desde la aparición de los primeros sistemas de tiempo compartido hasta la actualidad. En primer lugar, como ya se ha comentado anteriormente, se pasó de tener el intérprete incluido en el sistema operativo a ser un módulo externo que usa los servicios del mismo, lo cual proporciona una mayor flexibilidad facilitando su modificación o incluso su reemplazo. Dentro de esta opción existen básicamente dos formas de estructurar el módulo que maneja la interfaz de usuario:

- El intérprete de mandatos es un sólo programa que contiene el código para ejecutar todos los mandatos. El intérprete, después de leer la línea tecleada por el usuario, determina de qué mandato se trata y salta a la parte de su código que lleva a cabo la acción especificada por el mandato. Si no se trata de ningún mandato, se interpreta que el usuario quiere arrancar una determinada aplicación, en cuyo caso, el intérprete iniciará la ejecución del programa correspondiente en el contexto de un nuevo proceso y esperará hasta que termine. Con esta estrategia, mostrada en la siguiente figura, los mandatos son internos al intérprete. Obsérvese que en esta sección se está suponiendo que hay un único mandato en cada línea.

#### Repetir Bucle

```

    escribir_indicacion_de_preparado;
    leer_e_interpretar_linea(operacion, argumentos);
    Si (igual(operacion, "fin"))
        Salir del bucle y terminar ejecucion de interprete
    Si (igual(operacion, "renombrar"))
        Renombrar los archivos segun especifican los argumentos
    Si (igual(operacion, "borrar"))
        Borrar los archivos especificados por los argumentos
    .....
    Si no se trata de un mandato
        Arrancar el programa "operacion" pasandole "argumentos"
        Esperar a que termine el programa

```

#### Fin Bucle

- Existe un programa por cada mandato. El intérprete de mandatos no analiza la línea tecleada por el usuario sino que directamente inicia la ejecución del programa correspondiente en el contexto de un nuevo proceso y espera que éste termine. Se realiza el mismo tratamiento ya se trate de un mandato o de cualquier otra aplicación. Con esta estrategia, mostrada en la siguiente figura, los mandatos son externos al intérprete y la interfaz de usuario está compuesta por un conjunto de programas del sistema: un programa por cada mandato más el propio intérprete.

```

Repetir Bucle
    escribir_indicacion_de_preparado;
    leer_e_interpretar_linea(operacion, argumentos);
    Si (igual(operacion, "fin"))
        Salir del bucle y terminar ejecucion de interprete
    Arrancar el programa "operacion" pasandole "argumentos"
    Esperar a que termine el programa
Fin Bucle

```

La principal ventaja de la primera estrategia es la eficiencia, ya que los mandatos los lleva a cabo el propio intérprete sin necesidad de ejecutar programas adicionales. Sin embargo, el intérprete puede llegar a ser muy grande y la inclusión de un nuevo mandato o la modificación de uno existente exige cambiar el código del intérprete y recompilarlo. La segunda solución es más recomendable ya que proporciona un tratamiento y visión uniforme de los mandatos del sistema y las restantes aplicaciones. El intérprete no se ve afectado por la inclusión o la modificación de un mandato.

En los sistemas reales puede existir una mezcla de las dos estrategias. El intérprete de mandatos de MS-DOS (COMMAND.COM) se enmarca dentro de la primera categoría, esto es, intérprete con mandatos internos. El motivo de esta estrategia se debe a que este sistema operativo se diseñó para poder usarse en computadores sin disco duro y, en este tipo de sistemas, el uso de un intérprete con mandatos externos exigiría que el diskette correspondiente estuviese insertado para ejecutar un determinado mandato. Sin embargo, dadas las limitaciones de memoria de MS-DOS, para mantener el tamaño del intérprete dentro de un valor razonable, algunos mandatos de uso poco frecuente, como por ejemplo DISKCOPY, están implementados como externos.

Los intérpretes de mandatos de UNIX, denominados *shells*, se engloban en la categoría de intérpretes con mandatos externos. Sin embargo, algunos mandatos se tienen que implementar como internos debido a que su efecto sólo puede lograrse si es el propio intérprete el que ejecuta el mandato. Así, por ejemplo, el mandato `cd`, que cambia el directorio actual de trabajo del usuario usando la llamada `chdir`, requiere cambiar a su vez el directorio actual de trabajo del proceso que ejecuta el intérprete, lo cual sólo puede conseguirse si el mandato lo ejecuta directamente el intérprete.

En la siguiente figura se muestra cómo sería el esqueleto del código de un *shell* de UNIX básico. Nótese el uso de `FORK` y `EXEC` para crear un proceso hijo que ejecute un mandato externo, mientras que en el caso de un mandato interno (por ejemplo `cd`) es el propio intérprete el que realiza las acciones asociadas al mismo.

```

void interpreta_linea(char *linea, char *args[]) {
/*
    Realiza el análisis sintáctico de la línea rellenando el vector args
    de la forma requerida por EXECVP, almacenando las palabras especificadas
    en la línea en las sucesivas posiciones:
        arg[0]: contendrá el nombre del mandato
        arg[1]: primer argumento
        .....
        arg[N-1]: último argumento
        arg[N]: NULO.

```



```

*/
}
int do_cd(char *args[]) {
    .....
    chdir(args[1]);
    .....
}
int main() {
    /* Escribe prompt */
    fprintf(stderr, "> ");
    /* Ciclo repetitivo del intérprete */
    while (fgets(linea, MAX_LINEA, stdin)) {
        interpreta_linea(linea, &args);
        if (strcmp(args[0], "exit")==0)
            exit(0);
        else if (strcmp(args[0], "cd")==0)
            do_cd(args); /* mandato interno */
        else
            /* Crea un proceso hijo */
            if (fork()==0) {
                /* El hijo ejecuta el mandato especificado */
                execvp (args[0], args);
                /* Si falla ejecución termina proceso hijo */
                exit(1);
            }
            else
                /* padre espera fin del mandato (modo foreground) */
                wait(NULL);
        fprintf(stderr, "> ");
    }
}

```

#### 4.4.1. Funcionalidad del intérprete

Centrándonos en el intérprete de mandatos, sea cuál sea su tipo, se van a exponer a continuación algunas de las características generalmente presentes en dicho módulo:

- Control de la entrada y salida de los programas. El intérprete puede proporcionar al usuario la posibilidad de especificar, a la hora de ejecutar un programa o mandato, de dónde obtendrá su entrada de datos y dónde dejará sus resultados. Con este mecanismo un usuario puede conseguir, por ejemplo, que la salida de un mandato se almacene en un determinado archivo o que la entrada de un programa esté asociada a la salida de otro.
- Tratamiento de los argumentos. Como se pudo ver en el esquema general del funcionamiento de un intérprete, cuando un usuario solicita ejecutar un programa o mandato, el intérprete se encarga de pasarle los argumentos introducidos por el usuario. En el caso general, estos argumentos no son procesados por el intérprete y el programa los recibe tal como los tecleó el usuario. Sin embargo, la mayoría de los

intérpretes reconocen ciertos caracteres como especiales, generalmente denominados *metacaracteres*, y realizan un determinado procesamiento de los mismos. Cuando el intérprete detecta un metacarácter en la entrada de datos, realiza su tratamiento pasándole al programa el resultado del mismo. El realizar el procesamiento de los metacaracteres dentro del intérprete, en lugar de que sea cada mandato o programa el responsable del mismo, proporciona un tratamiento uniforme de los argumentos. Un ejemplo de metacarácter existente en prácticamente todos los sistemas es el símbolo `*` que representa a cualquier cadena de cero o más caracteres dentro del nombre de un archivo.

- Capacidad de programación. Como se comentó antes, la progresiva sofisticación de los intérpretes de mandatos ha hecho que se conviertan en una herramienta de programación proporcionando al usuario mecanismos similares a los disponibles en un lenguaje de programación, tales como sentencias condicionales, bucles, variables y definición de funciones. Con esta funcionalidad, un usuario podría, por ejemplo, solicitar que un mandato o programa se ejecutase sólo si se cumple una determinada condición o que se repitiese su ejecución hasta que deje de cumplirse.
- Dado que con estos mecanismos el usuario puede construir programas de mandatos, parece bastante lógico permitir que, además de su uso en modo interactivo, estos programas se puedan almacenar en archivos y dotar al intérprete de la capacidad de ejecutar estos archivos de mandatos. En este caso el intérprete, en vez de leer los mandatos del terminal, los obtendrá del archivo. Con la inclusión de los archivos de mandatos, las órdenes del usuario que recibe el intérprete podrán corresponder con mandatos del sistema, con archivos de mandatos o con otras aplicaciones. El intérprete debería tratar los tres casos de una manera uniforme.
- Funciones de edición de mandatos. Uno de los objetivos del intérprete es facilitar al usuario el proceso de introducir los mandatos. A continuación se presentan algunas de las funciones que puede incluir un intérprete para lograr ese objetivo:
  - Facilidades avanzadas de edición de la línea en curso. Como ejemplo negativo, muchos intérpretes no permiten que el usuario pueda retroceder al principio de la línea para insertar un carácter.
  - Capacidad de completar mandatos. El usuario teclea sólo la parte inicial del mandato y el intérprete se encarga de completarlo.
  - Corrección ortográfica. El intérprete analiza lo tecleado y si encuentra algún error, escribe en el terminal su propuesta pidiendo la confirmación del usuario.
  - Posibilidad de recuperar mandatos previamente ejecutados. El intérprete permite que el usuario acceda a mandatos previos para editarlos y solicitar su ejecución.
- Control de trabajos. Como se vio antes, el intérprete después de iniciar la ejecución de un programa o mandato espera a que éste termine. En sistemas con multiprogramación (o multitarea), en los cuales múltiples procesos pueden estar activos simultáneamente, existe la posibilidad de que, si así lo desea el usuario, una vez iniciado el programa, el intérprete no espere por su finalización y pase inmediatamente a atender nuevas peticiones del usuario. A este modo de operación se le denomina

ejecución en *background* mientras que al modo normal se le denomina ejecución en *foreground*. Un intérprete que ofrezca esta funcionalidad va a dar al usuario la posibilidad de especificar que un determinado trabajo se ejecute en background y la capacidad para pasar trabajos de foreground a background y viceversa. Asimismo, el usuario va a poder parar temporalmente la ejecución de un trabajo o terminarla incondicionalmente. Nótese que sólo podrá haber en un determinado momento un trabajo en foreground por cada terminal y que normalmente dicho trabajo debería ser el único que lee y escribe del mismo.

- Configuración de la interfaz. Deben existir mecanismos para poder configurar el comportamiento del intérprete según las preferencias del usuario. Además de poder modificar la configuración interactivamente, existe normalmente alguna forma de establecer la configuración inicial. En la mayoría de los sistemas, el intérprete consulta al arrancar un determinado archivo donde el usuario puede especificar qué configuración inicial prefiere.
- Arranque de la interfaz. En sistemas monousuario cuando se carga el sistema operativo, éste arranca el intérprete que, a partir de entonces, se encargará de atender las peticiones del usuario. En sistemas multiusuario, sin embargo, el sistema operativo arranca la parte de la interfaz de usuario que se encarga de controlar el acceso al sistema que, a partir de ese momento, se queda esperando que en cualquier terminal un usuario autorizado quiera entrar al sistema. Cuando esto ocurra, arrancará un intérprete para ese terminal, que se dedicará a atender las peticiones del usuario, y se mantendrá supervisando el conjunto de terminales que no estén siendo usados en ese momento. Cuando ese usuario deje el sistema, terminará la ejecución del intérprete asociado y, al detectarlo, la parte de la interfaz encargada del control de acceso volverá a supervisar ese terminal.

## 4.5. Interfaces gráficas

El auge de las interfaces de usuario gráficas (GUI: *Graphical User Interface*) se debe principalmente a la necesidad de proporcionar a los usuarios no especializados una visión sencilla e intuitiva del sistema que oculte toda su complejidad. Esta necesidad ha surgido por la enorme difusión de los computadores en todos los ámbitos de la vida cotidiana. Sin embargo, el desarrollo de este tipo de interfaces más amigables ha requerido un avance considerable en la potencia y capacidad gráfica de los computadores dado la gran cantidad de recursos que consumen durante su operación.

Las primeras experiencias con este tipo de interfaces se remontan a los primeros años de la década de los setenta. En Xerox PARC (un centro de investigación de Xerox) se desarrolló lo que actualmente se considera la primera estación de trabajo a la que se denominó *Alto*. Además de otros muchos avances, esta investigación estableció los primeros pasos en el campo de los GUI.

Con la aparición, al principio de los ochenta, de los computadores personales dirigidos precisamente a usuarios no especializados, se acentuó la necesidad de proporcionar este tipo de interfaces. Así, la compañía Apple adoptó muchas de las ideas de la investigación de Xerox PARC para lanzar su computador personal Macintosh (1984) con un interfaz gráfico que simplificaba enormemente el manejo del computador. El otro gran competidor

en este campo, el sistema operativo MS-DOS, tardó bastante más en dar este paso. En sus primeras versiones proporcionaba una interfaz alfanumérica similar a la de UNIX pero muy simplificada. Como paso intermedio, hacia 1988, incluyó una interfaz denominada DOS-SHELL que, aunque seguía siendo alfanumérica, no estaba basada en líneas sino que estaba orientada al uso de toda la pantalla y permitía realizar operaciones mediante menús. Por fin, ya en los noventa, lanzó una interfaz gráfica, denominada Windows, que tomaba prestado muchas de las ideas del Macintosh.

En el mundo UNIX, se produjo una evolución similar. Cada fabricante incluía en su sistema una interfaz gráfica además de la convencional. La aparición del sistema de ventanas X a mediados de los 80 y su aceptación generalizada, que le ha convertido en un estándar de facto, ha permitido que la mayoría de los sistemas UNIX incluyan una interfaz gráfica común. Como resultado de este proceso, prácticamente todos los computadores de propósito general existentes actualmente poseen una interfaz de usuario gráfica.

A continuación pasaremos a ver cuáles son las características comunes de este tipo de interfaces. En primer lugar, todos ellos están basados en ventanas que permiten al usuario estar trabajando simultáneamente en distintas actividades. Asimismo, se utilizan menús e iconos para representar los recursos del sistema y poder realizar operaciones sobre los mismos. El usuario utiliza un ratón (o dispositivo equivalente) para interactuar con estos elementos. Así, por ejemplo, para arrancar una aplicación el usuario podría tener que apuntar a un icono con el ratón y apretar un botón del mismo, o para copiar un archivo señalar al icono que lo representa y, manteniendo el botón del ratón apretado, moverlo hasta ponerlo encima de un icono que representa el directorio destino. Generalmente, para agilizar el trabajo de los usuarios más avanzados, estos interfaces proporcionan la posibilidad de realizar estas mismas operaciones utilizando ciertas combinaciones de teclas. Dado el carácter intuitivo de estas interfaces y el amplio conocimiento que posee todo el mundo de ellas, no parece necesario entrar en más detalles sobre su forma de trabajo.

En cuanto a su estructura interna, las interfaces gráficas normalmente están formadas por un conjunto de programas que, usando los servicios del sistema, trabajan conjuntamente para llevar a cabo las peticiones del usuario. Así, por ejemplo, existirá un gestor de ventanas para mantener el estado de las mismas y permitir su manipulación, un administrador de programas que permita al usuario arrancar aplicaciones, un gestor de archivos que permita manipular archivos y directorios, o una herramienta que permita configurar la propia interfaz y el entorno. Nótese la diferencia con las interfaces alfanuméricas en las que existía un programa por cada mandato. Además de la funcionalidad comentada, otros aspectos que conviene resaltar son los siguientes:

- Intercambio de datos entre aplicaciones. Generalmente se le proporciona al usuario un mecanismo del tipo *copy-and-paste* para poder transferir información, normalmente alfanumérica, entre dos aplicaciones.
- Sistema de ayuda interactivo. Los sistemas de ayuda suelen ser muy sofisticados basándose muchos de ellos en el hipertexto.
- Oferta de servicios a las aplicaciones (API gráfica). Además de encargarse de atender al usuario, estos entornos gráficos proporcionan a las aplicaciones una biblioteca de primitivas gráficas que permiten que los programas creen y manipulen objetos gráficos.

- Posibilidad de acceso al interfaz alfanumérico. Muchos usuarios se sienten encorsetados dentro de la interfaz gráfica y prefieren usar una interfaz alfanumérica para realizar ciertas operaciones. La posibilidad de acceso a dicha interfaz desde el entorno gráfico ofrece al usuario un sistema con lo mejor de los dos mundos.

## 4.6. Caso de estudio: UNIX

Como caso de estudio se va a presentar la interfaz de usuario de UNIX. Puesto que en UNIX el módulo que maneja la interfaz no está dentro del sistema operativo, cada usuario puede utilizar uno diferente pudiendo tener, por lo tanto, una visión muy distinta del sistema. Sin embargo, esta sección se va a ocupar de lo que es su interfaz típica: una interfaz alfanumérica basada en un intérprete de mandatos, denominado *shell*, con mandatos externos (y algunos mandatos intrínsecamente internos). Es conveniente resaltar que el objetivo de esta sección no es proporcionar un manual de usuario de UNIX, lo cual el lector puede encontrar fácilmente en las referencias bibliográficas recomendadas en la asignatura, sino dar una visión general de la interfaz. Por último, debemos aclarar que la siguiente descripción hace hincapié en los mecanismos del intérprete de UNIX más relacionados con el desarrollo de archivos de mandatos (*shell scripts*), dejando a un lado aquellos aspectos asociados al trabajo interactivo del intérprete. Así, no se tratarán aspectos tales como el control de trabajos, las funciones de edición de mandatos y la configuración de la interfaz.

### 4.6.1. El shell de UNIX

El intérprete de mandatos de UNIX se denomina shell dado que, como una concha, envuelve al núcleo del sistema operativo ocultándolo al usuario. Puesto que el shell es un programa como otro cualquiera, a lo largo de la historia de UNIX ha habido muchos programadores que se han decidido a construir uno de acuerdo con sus preferencias personales. A continuación destacaremos aquéllos que han alcanzado mayor difusión:

- El shell de Bourne (**sh**). El shell original de UNIX. Se trata de un intérprete que ofrece al usuario una herramienta de programación poderosa. Su principal defecto es la falta de ayuda al usuario interactivo para facilitarle la labor de introducir mandatos. No incluye, al menos originalmente, facilidades para el control de trabajos.
- El C-shell (**cs**h). Fue desarrollado en la universidad de Berkeley. Tiene una sintaxis que recuerda al lenguaje C. Incluye facilidades para el usuario interactivo y capacidad de control de trabajos. El **tcsh** es una versión mejorada del mismo con mayor funcionalidad para el usuario interactivo.
- El shell de Korn (**ks**h). Se desarrolló en AT&T. A diferencia del resto de los comentados en esta sección, no es de libre distribución. El diseño de este intérprete tomó como punto de partida el shell de Bourne añadiéndole muchas de las características del C-shell (y de otros sistemas operativos).
- El shell de GNU (**bash**: *Bourne-Again SHell*). Los planteamientos de diseño de este intérprete son similares a los del shell de Korn pero, obviamente, se trata de un programa de libre distribución que ha sido adoptado como el intérprete para Linux.

Dada la gran variedad de shells existente, la exposición se centrará, siempre que sea posible, en el shell de Bourne ya que es el que está disponible en todos los sistemas y dado que sus características son normalmente aplicables a todos los shells derivados del mismo (por ejemplo, ksh y bash). Además, la propuesta de shell especificada en el estándar POSIX 1003.2 está basada en la familia de shells derivados del sh. A continuación se presentan las características más relevantes de los shells de UNIX.

## Estructura de los mandatos

Cada mandato es una secuencia de palabras separadas por espacios tal que la primera palabra es el nombre del mandato y las siguientes son sus argumentos. Así, por ejemplo, para borrar los archivos `arch1` y `arch2` se podría especificar el mandato siguiente:

```
rm arch1 arch2
```

Donde `rm` es el nombre del mandato y `arch1` y `arch2` sus argumentos.

Una vez leído el mandato, el shell iniciará su ejecución en el contexto de un nuevo proceso (*sub-shell*) y esperará la finalización del mismo antes de continuar (modo de ejecución *foreground*).

La ejecución de un mandato devuelve un valor que refleja el estado de terminación del mismo. Por convención, un valor distinto de 0 indica que ha ocurrido algún tipo de error (valor *falso*). Así, en el ejemplo anterior, si alguno de los archivos especificados no existe, `rm` devolverá un valor distinto de 0. Como se verá más adelante, este valor puede ser consultado por el usuario o por posteriores mandatos.

Si una línea contiene el carácter `#`, esto indicará que el resto de los caracteres de la misma que aparecen a partir de dicho carácter se considerarán un comentario y, por lo tanto, el shell los ignorará. Aunque se pueden usar comentarios cuando se trabaja en modo interactivo, su uso más frecuente es dentro de *shell scripts*.

## Agrupamiento de mandatos

Los shells de UNIX permiten que el usuario introduzca varios mandatos en una línea existiendo, por lo tanto, otros caracteres, además del de fin de línea, que delimitan donde termina un mandato o que actúan de separadores entre dos mandatos. Esta posibilidad va a permitir que el usuario pueda expresar operaciones realmente complejas en una única línea. Cada tipo de carácter delimitador o separador tiene asociado un determinado comportamiento que afecta a la ejecución del mandato o de los mandatos correspondientes. El valor devuelto por una lista será el del último mandato ejecutado (excepto en el caso de una lista asíncrona que devuelve siempre un 0). Dependiendo del delimitador o separador utilizado se pueden construir los siguientes tipos de listas de mandatos:

- Lista con tuberías (carácter separador `|`)
- Lista O-lógico (carácter separador `||`)
- Lista Y-lógico (carácter separador `&&`)
- Lista secuencial (carácter delimitador `;`)
- Lista asíncrona (carácter delimitador `&`)

**Lista con tuberías (*pipes*)** Se ejecutan de forma concurrente los mandatos incluidos en la lista de tal forma que la salida estándar de cada mandato queda conectada a la entrada estándar del siguiente en la lista mediante un mecanismo denominado tubería (*pipe*). En el apartado dedicado a las redirecciones se profundizará más sobre el concepto de entrada y salida estándar. La ejecución de la lista terminará cuando terminen todos los mandatos incluidos en la misma (algunos shells relajan esta condición terminando la lista cuando finaliza el mandato que aparece más a la derecha). El siguiente ejemplo muestra una línea de mandatos que imprime (mandato `lpr`) en orden alfabético (mandato `sort`) aquellas líneas que, estando entre las 10 primeras del archivo `carta` (mandato `head`), contengan la cadena de caracteres `Juan` (mandato `grep`).

```
head -10 carta | grep Juan | sort | lpr
```

**Lista O-lógico (OR)** Se ejecutan de forma secuencial (de izquierda a derecha) los mandatos incluidos en la lista hasta que uno de ellos devuelva un valor igual a 0 (por convención, `verdadero`). El siguiente ejemplo imprime el archivo sólo si no es un directorio (mandato `test` con la opción `-d`).

```
test -d archivo || lpr archivo
```

**Lista Y-lógico (AND)** Se ejecutan de forma secuencial (de izquierda a derecha) los mandatos incluidos en la lista hasta que uno de ellos devuelva un valor distinto de 0 (por convención, `falso`). El siguiente ejemplo imprime el archivo sólo si se trata de un fichero ordinario (mandato `test` con la opción `-f`).

```
test -f archivo && lpr archivo
```

**Lista secuencial** Se ejecutan de forma secuencial (de izquierda a derecha) todos los mandatos incluidos en la lista. El resultado es el mismo que si cada mandato se hubiera escrito en una línea diferente. La siguiente línea intercambia el nombre de dos archivos usando repetidamente el mandato `mv`.

```
mv arch1 aux; mv arch2 arch1; mv aux arch1
```

**Lista asíncrona (*background*)** Se inicia la ejecución de cada mandato de la lista, pero el shell no se queda esperando su finalización sino que una vez arrancados continúa con su labor (modo de ejecución *background*). Así, por ejemplo, si se quiere poder seguir trabajando con el shell mientras se compilan (mandato `cc`) dos programas muy largos, se puede especificar la siguiente línea:

```
cc programa_muy_largo_1.c & cc programa_muy_largo_2.c &
```

Los diversos tipos de listas se pueden mezclar para crear líneas más complejas. Cada tipo de delimitador o separador tiene asociado un orden de precedencia que determina cómo interpretará el shell una línea que mezcle distintos tipos de listas. En el caso de la misma precedencia, el análisis de la línea se hace de izquierda a derecha. El orden de precedencia, de mayor a menor, es el siguiente:

- Listas con tuberías
- Listas Y y O (misma preferencia)
- Lista asíncronas y secuenciales (misma preferencia)

Asimismo, existe la posibilidad de agrupar mandatos para alterar las relaciones de precedencia entre los separadores y delimitadores. De manera similar a lo que sucede con las expresiones aritméticas, se usan paréntesis o llaves. Hay algunas diferencias entre el uso de estos símbolos:

- **(lista)**: Los mandatos de la lista no los ejecuta el shell que los leyó sino que se arranca un subshell para ello. Como se verá más adelante esto tiene repercusiones en el uso de variables y mandatos internos.
- **{ lista; }**: Los mandatos de la lista los ejecuta el shell que los leyó. La principal desventaja de esta construcción es que su sintaxis es algo irritante (nótese el espacio que aparece antes de la lista y el punto y coma que hay después).

En el siguiente ejemplo, se muestra el uso combinado de varios tipo de listas. La línea que aparece a continuación imprime dentro de una hora (mandato **sleep**), en orden alfabético, de entre las primeras 100 líneas de **arc** las que contengan la palabra **Figura**, comprobando antes si **arc** es accesible (mandato **test** con la opción **-r**):

```
(sleep 3600; test -r arc && head -100 arc | grep Figura | sort | lp)&
```

Nótese la necesidad de usar paréntesis para lograr que la ejecución de toda la línea se haga de forma asíncrona, esto es, que el shell quede inmediatamente listo para atender más peticiones. Si no se usarán los paréntesis (o llaves), el shell se quedaría bloqueado durante una hora ya que la ejecución en background sólo afectaría a los mandatos que están a la derecha del punto y coma.

Este ejemplo muestra la versatilidad de la interfaz de UNIX que permite combinar mandatos que llevan a cabo labores básicas para realizar operaciones complejas. Obsérvese que las listas de mandatos se consideran a su vez mandatos y que, por lo tanto, cuando a partir de este momento usemos el término mandato, nos estaremos refiriendo tanto a mandatos simples como a listas de mandatos, a no ser que se especifique lo contrario.

## Mandatos compuestos y funciones

Como se comentó previamente, el shell es una herramienta de programación y como tal pone a disposición del usuario casi todos los mecanismos presentes en un lenguaje de programación convencional. Así, proporciona una serie de mandatos compuestos que permiten construir estructuras de ejecución condicionales y bucles. Asimismo, permite definir funciones que facilitan la modularidad de los programas de mandatos. En todas las contrucciones que se presentarán a continuación, el valor devuelto por las mismas será el del último mandato ejecutado.



**El mandato condicional if** Esta construcción ejecutará un mandato o no dependiendo del estado de terminación de otro mandato que ha ejecutado previamente. Su sintaxis es la siguiente:

```
if lista1
then
    lista # si verdadero lista1
elif lista2
then
    lista # si falso lista1 y verdadero lista2
else
    lista # si falso lista1 y lista2
fi
```

Nótese que tanto la rama correspondiente al elif como la del else son opcionales. El siguiente ejemplo, similar al usado para las listas-Y, imprime el archivo sólo si se trata de un fichero ordinario. En caso contrario, muestra por la pantalla un mensaje de error (mandato echo).

```
if test -f archivo
then
    lpr archivo
else
    echo "Error: Se ha intentado imprimir un fichero no regular!"
fi
```

**El mandato condicional case** Este mandato condicional compara la palabra especificada con los distintos patrones y ejecuta el mandato asociado al primer patrón que se corresponda con dicha palabra. La comparación entre la palabra y los distintos patrones seguirá las mismas reglas que se usan en la expansión de nombres de archivos que se verá más adelante. Su sintaxis es la siguiente:

```
case palabra in
    patrón1) lista1;; # ejecutada si palabra encaja en patron1
    patrón2|patrón3) lista2;; # ejecutada si palabra encaja en
                           # patron2 o patron3
esac
```

En el apartado correspondiente a las funciones se mostrará un ejemplo del uso de esta construcción.

**El bucle until** Esta construcción ejecutará un mandato hasta que la ejecución de otro mandato devuelva un valor igual a verdadero. Su sintaxis es la siguiente:

```
until lista1
do
    lista # ejecutada hasta que lista1 devuelva verdadero
done
```

El siguiente ejemplo ejecuta en background un bucle until que muestrea cada 5 segundos (mandato `sleep`) la llegada de un mensaje (mandato `mail`) en cuya cabecera aparezca la palabra `URGENTE`. Cuando se detecta, se escribe (mandato `printf`) un pitido y se termina el bucle.

```
until mail -H | grep URGENTE && printf '\a'
do
    sleep 5
done &
```

**El bucle while** Esta construcción ejecutará un mandato mientras que la ejecución de otro mandato devuelva un valor igual a verdadero. Su sintaxis es la siguiente:

```
while lista1
do
    lista # ejecutada mientras verdadero lista1
done
```

**El bucle for** En cada iteración de este tipo de bucle se ejecutará el mandato especificado tomando la variable el valor de cada uno de los sucesivos elementos que aparecen en la lista de palabras. Su sintaxis es la siguiente:

```
for VAR in palabra1 ... palabran
do
    lista # en cada iteración VAR toma valor de sucesivas palabras
done
```

Si no aparece la palabra reservada `in` ni la secuencia de palabras, equivale a especificar la variable `$*` que, como se verá en la sección dedicada a las variables, se corresponde con la lista de parámetros de un archivo de mandatos o una función. Así, el siguiente fragmento:

```
for VAR
do
    lista
done
```

Equivaldría a:

```
for VAR in $*
do
    lista # en cada iteración VAR toma el valor de un argumento
done
```

En el siguiente ejemplo se usa un bucle `for` para realizar una copia de seguridad (con la extensión `.bak`) de un conjunto de archivos.

```
for ARCHIVO in m1.c m2.c cap1.txt cap2.txt resumen.txt
do
    test -f $ARCHIVO && cp $ARCHIVO $ARCHIVO.bak
done
```

**Funciones** La definición de una función permite asociar con un nombre especificado por el usuario un mandato (simple, lista de mandatos o mandato compuesto). La invocación de dicho nombre como si fuera un mandato simple producirá la ejecución del mandato asociado que recibirá los argumentos especificados. Más adelante, en el apartado dedicado a los parámetros, se tratará en detalle este tema. El valor devuelto por la función será el del último mandato ejecutado dentro de la misma (por legibilidad, se suele usar el mandato interno **return** para terminarla). La sintaxis para definir una función es la siguiente:

```
nombre_funcion() lista
```

Por motivos de legibilidad, se usan normalmente las llaves para delimitar el cuerpo de la función. Así, la estructura resulta similar a la del lenguaje C.

```
nombre_funcion()
{
    lista
}
```

En cuanto a la invocación de la función, como se comentó antes, se realiza utilizando el nombre de la misma como si fuera un mandato simple:

```
nombre_funcion arg1 arg2 ... argn
```

El siguiente ejemplo muestra una función que recibe como argumentos un conjunto de archivos y que, de forma similar al ejemplo anterior, realiza una copia de seguridad de cada uno de ellos. Nótese el uso del **case** para evitar sacar copias de seguridad de las propias copias (archivos con extensión **bak** o **BAK**). En dicha construcción se ha usado el patrón **\*** que, como se verá en la sección dedicada a la expansión de nombres de archivos, representa a cualquier cadena de cero o más caracteres.

```
copia_seguridad()
{
    for ARCHIVO
    do
        case $ARCHIVO in
            *.BAK|*.bak) ;;
            *) test -f $ARCHIVO && cp $ARCHIVO $ARCHIVO.bak;;
        esac
    done
}
```

La función se invocaría como si fuera un mandato simple, especificando los archivos de los que se quiere sacar una copia de seguridad:

```
copia_seguridad m1.c m2.c cap1.txt cap2.txt resumen.txt
```

## Redirecciones

El shell permite al usuario redirigir la entrada y salida que producirá un mandato durante su ejecución. Con este mecanismo, el usuario puede invocar un mandato especificando, por ejemplo, que los datos que lea el programa los tome de un determinado archivo en vez del terminal. Normalmente, este mecanismo se usa para redirigir alguno de los tres descriptores estándar.

La mayoría de los mandatos y, en general, de los programas de UNIX leen sus datos de entrada del terminal y escriben sus resultados y los posibles mensajes de error también en el terminal. Para simplificar el desarrollo de estos programas y permitir que puedan leer y escribir directamente en el terminal sin realizar ninguna operación previa (o sea, sin necesidad de realizar una llamada OPEN), los programas reciben por convención tres descriptores, normalmente asociados al terminal, ya preparados para su uso (esto es, para leer y escribir directamente en ellos):

- Entrada estándar (descriptor 0): De donde el programa puede leer sus datos.
- Salida estándar (descriptor 1): Donde el programa puede escribir sus resultados.
- Error estándar (descriptor 2): Donde el programa puede escribir los mensajes de error.

Anteriormente se presentó un primer tipo de redirección asociado a la lista con tuberías que permitía que la salida estándar de un mandato quedase conectada a la entrada estándar de otro. En este apartado nos centraremos en las redirecciones a archivos.

**Redirección de salida** El usuario puede redirigir la salida estándar de un programa a un archivo usando el carácter `>` delante del nombre del archivo. En general, la expresión `n>arch` permite redirigir el descriptor de salida `n` al archivo `arch`. Con este tipo de redirección, si el archivo existe, su contenido inicial se pierde, esto es, el shell trunca el archivo a una longitud de cero antes de que se produzca la ejecución del mandato. Si lo que el usuario desea es que, en el caso de que el archivo exista, la salida producida por el mandato se concatene detrás del contenido inicial del archivo, deberá usar `>>` (en general `n>> arch`) para especificar la redirección.

Además de poder redirigir la salida a un archivo, el usuario puede especificar que la salida asociada a un determinado descriptor se redirija al mismo destino que tiene asociado otro descriptor. Así, la expresión `n>&m` indica que la salida asociada al descriptor `n` se redirige al mismo destino que corresponde al descriptor `m`. Si, por ejemplo, el usuario quiere que tanto la salida estándar como la salida de error de un mandato se redirijan a un archivo, puede usar la siguiente línea:

```
mandato > archivo 2>&1
```

**Redirección de entrada** De manera similar a lo visto para la salida, el usuario puede redirigir la entrada estándar de un programa a un archivo usando el carácter `<` delante del nombre del archivo. En general, la expresión `n< arch` permite redirigir el descriptor de entrada `n` al archivo `arch`. Asimismo, el usuario puede especificar que la entrada asociada a un determinado descriptor se redirija a la misma fuente que tiene asociada otro descriptor.

Así, por ejemplo, la expresión `n<&m` indica que la entrada asociada al descriptor `n` se redirige a la misma fuente que corresponde al descriptor `m`.

A continuación se muestra un ejemplo en el que se redirigen los tres descriptors del mandato `sort` para ordenar los datos contenidos en el archivo `entrada` dejando el resultado en el archivo `salida` y almacenando los posibles mensajes de error en el archivo `error`:

```
sort > salida < entrada 2> error
```

Otro tipo de redirección de entrada, usada principalmente en shell scripts, se corresponde con los denominados documentos *in-situ* (*here documents*). Con este mecanismo, el usuario especifica que la entrada estándar de un mandato se tomará de las líneas que siguen al propio mandato hasta que aparezca una línea que contenga únicamente una determinada palabra que especificó el usuario en la redirección. Dicho de manera informal, es como si el usuario pegará al mandato los datos que quiere que éste lea. Este tipo de redirección tiene la siguiente estructura:

```
mandato << marca_fin
linea de entrada 1
linea de entrada 2
.....
linea de entrada N
marca_fin
```

Las líneas que se tomarán como entrada del mandato sufrirán previamente las expansiones que realiza habitualmente el shell (ver la sección sobre la expansión de argumentos), a no ser que la palabra que aparece en la especificación de la redirección tenga algún carácter con *quoting* (este concepto se trata en el siguiente apartado).

## Quoting

Algunos caracteres o palabras tienen un significado especial para el shell. Cuando al procesar una línea, el shell encuentra una de estas secuencias, en vez de pasársela directamente al mandato correspondiente, la procesa transformando la línea leída de acuerdo con las acciones asociadas a la misma. El mecanismo de *quoting* permite que el usuario especifique que el shell no procese un determinado carácter (o secuencia de caracteres), a pesar de que éste tenga un significado especial para el shell. Para ello usará a su vez un metacarácter de protección:

- Colocará el símbolo `backslash` delante del carácter que pretende proteger.
- Encerrará entre comillas simples la secuencia de caracteres que pretende proteger.
- Encerrará entre comillas dobles la secuencia de caracteres que pretende proteger. En este caso la protección no es completa y permite que se realicen algunos tipos de procesamiento. Concretamente, no impide la expansión de variables y la sustitución de mandatos.

El siguiente ejemplo muestra como se pueden borrar los archivos `mio;tuyo` y `nombre`; `>raro`:

```
rm mio\;tuyo nombre';>'raro
```

Nótese que por comodidad el usuario normalmente aplicaría las comillas a todo el nombre del segundo archivo.

## Expansión de argumentos

Ciertos caracteres tienen un significado especial para el shell y, siempre que aparecen sin proteger en una determinada línea, el shell llevará a cabo el procesamiento correspondiente transformando la línea original de acuerdo con la funcionalidad asociada a cada uno de ellos. El shell ejecutará la línea resultante del tratamiento como si la hubiese introducido de esta forma el usuario. A esta acción de procesar los metacaracteres presentes en un argumento la denominaremos expansión de argumentos y en esta sección trataremos los siguientes tipos de expansión: expansión de tilde, expansión de variables, sustitución de mandatos, expansión aritmética y expansión de nombres de archivos.

**Expansión de tilde** El carácter tilde, no disponible en el Bourne shell pero sí en el resto de los comentados, se sustituye (se expande) por el nombre del directorio base (HOME) del usuario (o de otro usuario). Por ejemplo, el siguiente mandato muestra todos los archivos (mandato `ls`) contenidos en el subdirectorio `bin` del directorio base del usuario y en el subdirectorio `pub` del directorio base del usuario `jgarcia`.

```
ls ~/bin ~jgarcia/pub
```

**Expansión de variables** Como es habitual en cualquier lenguaje de programación convencional, cada variable está identificada por un nombre y almacena un valor. Para obtener el valor de una variable debe especificarse el carácter `$` delante de su nombre. Cuando ocurre esto en una línea, el shell lo sustituye por su valor asociado. Más adelante, en la sección que trata sobre los parámetros, se volverá a tratar este tema.

**Sustitución de mandatos** Este mecanismo permite que el usuario especifique que en una línea se reemplace un determinado mandato por la salida que produce su ejecución. Para ello el mandato debe estar encerrado entre `'` (o también dentro de la construcción `$(mandato)` en el caso del `ksh` y `bash`). Cuando el shell detecta esta construcción durante el tratamiento de una línea, ejecuta el mandato afectado por la misma y sustituye en la línea original el mandato por la salida que éste produce al ejecutarse. El siguiente ejemplo borra (mandato `rm`) los archivos que contengan la cadena de caracteres `CADENA` (el mandato `grep` con la opción `-l` imprime los nombres de los archivos que contengan la cadena especificada).

```
rm -f `grep -l CADENA arch1 arch2 arch3 arch4 arch5`
```

Suponiendo que `arch2` y `arch4` contienen dicha cadena de caracteres, la sustitución del mandato produciría la siguiente transformación:

```
rm -f arch2 arch4
```

**Expansión aritmética** La mayoría de los shells, a excepción del de Bourne, poseen la capacidad de evaluar expresiones aritméticas. En el caso del `ksh` y `bash` la expresión debe estar incluida en la construcción `$((expresión))`. Cuando el shell detecta esta construcción durante el tratamiento de una línea, evalúa la expresión y la sustituye en la línea original por el resultado de la misma. El siguiente ejemplo muestra el uso combinado de la sustitución de mandatos y de la expansión aritmética para mostrar por la pantalla la

primera mitad de las líneas del archivo `archivo` (el mandato `wc -l` cuenta el número de líneas de un archivo).

```
head -${('wc -l < archivo' / 2)} archivo
```

Suponiendo que `archivo` tiene 16 líneas la sustitución del mandato produciría la siguiente transformación:

```
head -$(16 / 2) archivo
```

Y la expansión aritmética tendría como resultado:

```
head -8 archivo
```

Como se comentó previamente el shell de Bourne no incluye esta facilidades por lo que hay que recurrir al mandato `expr` para realizar operaciones aritméticas. El ejemplo anterior habría que reescribirlo de la siguiente forma:

```
head -'expr `wc -l < archivo` / 2' archivo
```

**Expansión de nombres de archivos** El shell facilita la labor de especificar un nombre de archivo proporcionando al usuario un conjunto de caracteres que actúan como comodines (*wildcards*). Cuando el shell encuentra uno de estos caracteres en una palabra, considera que se trata de un patrón de búsqueda y sustituye la palabra original por todos los nombres de archivo que encajan en ese patrón. En este tipo de expansión están involucrados los siguientes metacaracteres:

- El carácter `*` representa a cualquier cadena de cero o más caracteres dentro del nombre de un archivo.
- El carácter `?` se corresponde con un único carácter sea éste cuál sea.
- Una secuencia de caracteres encerrada entre llaves cuadradas se corresponde con único carácter de los especificados, presentándose las siguientes excepciones:
  - Si el primer carácter de la secuencia es `!`, se corresponde con cualquier carácter excepto los especificados.
  - Si aparece un `-` entre dos caracteres de la secuencia, se corresponde con cualquier carácter en el rango entre ambos.

El siguiente ejemplo borraría los archivos cuyo nombre comience y termine por `A` (incluido el archivo `AA`), aquéllos cuyo nombre empieza por la letra `p` seguida de otros cuatro caracteres cualquiera, los que tengan un nombre de dos caracteres ambos comprendidos entre la `a` y la `z` minúsculas y, por último, los archivos cuyo nombre tenga una longitud de al menos dos caracteres tal que el primer carácter sea `1` o `2` y el segundo no sea numérico.

```
rm -f A*A p???? [a-z][a-z] [12][!1-9]*
```

La siguiente línea invoca la función `copia_seguridad`, antes presentada, pasándole como argumentos todos los archivos del directorio actual de trabajo.

```
copia_seguridad *
```

## Parámetros

Dado que el shell es una herramienta de programación, además de proporcionar los mecanismos para estructurar programas que se presentaron previamente, ofrece al usuario la posibilidad de utilizar variables, que en este entorno se las suele denominar de forma genérica como parámetros. Para obtener el valor almacenado en un parámetro se debe colocar un `$` delante del nombre del mismo (o delante del nombre encerrado entre llaves). Hay tres tipos de parámetros: parámetros posicionales, parámetros especiales y variables propiamente dichas.

**Parámetros posicionales** Se corresponden con los argumentos con los que se invoca un script o una función. Su identificador es un número que corresponde con su posición. Así, `$1` será el valor del primer argumento, `$2` el del segundo y, en general, `$i` se referirá al *i*-ésimo argumento. El usuario no puede modificar de forma individual un parámetro posicional, aunque puede reasignar todos con el mandato interno `set`. Después de ejecutar este mandato, los parámetros posicionales toman como valor los argumentos especificados en el propio mandato `set`.

**Parámetros especiales** Se trata de parámetros mantenidos por el propio shell por lo que el usuario no puede modificar su valor. A continuación se muestran algunos de los más frecuentemente usados:

- 0: Nombre del script
- #: Número de parámetros posicionales
- \*: Lista de parámetros posicionales
- ?: Valor devuelto por el último mandato ejecutado
- \$: Identificador de proceso del propio shell
- !: Identificador de proceso asociado al último mandato en background arrancado

**Variables** Este tipo de parámetros se corresponden con el concepto clásico de variable presente en los lenguajes de programación convencionales, pero dada la relativa simplicidad de este entorno, la funcionalidad asociada a las variables es reducida. En primer lugar, las variables no se declaran, creándose cuando se les asigna un valor usando la construcción `variable=valor` (o con el mandato interno `read`). No existen diferentes tipos de datos: todas las variables son consideradas del tipo cadena de caracteres. Cuando se intenta acceder a una variable que no exista, no se producirá ningún error sino que simplemente el shell la expandirá como un valor nulo.

Otro aspecto importante relacionado con las variables es su posible exportación a los procesos creados por el propio shell durante la ejecución de los distintos mandatos. Por defecto, las variables no se exportan y, por lo tanto, los procesos creados no obtienen una copia de las mismas. Si el usuario requiere que una variable sea exportada a los procesos hijos del shell, debe especificarlo explícitamente mediante el mandato interno `export`. Nótese que, aunque una variable se exporte, las modificaciones que haga sobre ella el proceso hijo no afectan al padre ya que el hijo obtiene una copia de la misma.



El shell durante su fase de arranque crea una variable por cada una de las definiciones presentes en el entorno del proceso que ejecuta el shell. Estas variables se consideran exportadas de forma automática. Como ejemplo típico de este tipo de variables, podemos considerar la variable `HOME`, que contiene el nombre del directorio base del usuario, y la variable `PATH`, que representa la lista de directorios donde el shell busca los mandatos.

## Shell scripts

El comportamiento del shell va a depender de los argumentos y opciones que se especifiquen en su invocación:

- Si recibe la opción `-i` o no se le especifican argumentos, el shell trabajará en modo interactivo, leyendo de su entrada estándar los mandatos que el usuario introduce.
- Si recibe la opción `-c cadena_de_caracteres`, el shell ejecutará únicamente los mandatos contenidos en `cadena_de_caracteres` y terminará.
- Si recibe como argumento el nombre de un archivo (shell script), el shell ejecutará los mandatos contenidos en el mismo.

En este apartado, nos centraremos en este último caso ya que, como se comentó al principio de la sección dedicada al shell de UNIX, esta exposición está centrada en los aspectos relacionados con la programación de scripts dejando a un lado las cuestiones asociadas al modo de trabajo interactivo.

La ejecución de un script, por lo tanto, se realiza de la siguiente manera:

```
sh script arg1 arg2 ... argn
```

Con lo que quedaría el parámetro `$0` con un valor igual a `script`, `$1` igual a `arg1` y así sucesivamente. El valor devuelto por el script será el del último mandato ejecutado (por legibilidad, se suele usar el mandato interno `exit` para terminarla).

Por motivos de comodidad para el usuario y para proporcionar una manera uniforme de invocar todos los programas en el sistema, los scripts también pueden arrancarse de la siguiente forma:

```
script arg1 arg2 ... argn
```

Evidentemente, esta segunda forma es ventajosa pero presenta el problema de cómo determinar qué shell quiere el usuario que se arranque para ejecutar el script (nótese que normalmente en un sistema UNIX hay varios shells disponibles). Va a ser la primera línea del script la que especifique qué intérprete se arrancará usándose la siguiente sintaxis:

```
#!/interprete
```

Donde `interprete` debe especificar el nombre del archivo que contiene el ejecutable del intérprete. Así, si la primera línea del archivo `script` es la siguiente:

```
#!/bin/sh
```

Ejecutar la siguiente línea:

```
script arg1 arg2 ... argn
```

Equivaldría a lo siguiente:

```
/bin/sh arg1 arg2 ... argn
```

Nótese que se trata de un mecanismo genérico que permite especificar cualquier tipo de intérprete para procesar el archivo. Así, por ejemplo, el archivo podría contener mandatos de una determinada base de datos y el intérprete podría ser el programa de interfaz de la base de datos.

Existe una forma alternativa de ejecutar un script mediante el uso del mandato interno `..`. Cuando se invoca un script de esta forma, no se arranca un shell para ejecutar los mandatos contenidos en el mismo, sino que es el propio shell actual el que los lee y ejecuta. Esto hace que, a diferencia de lo que sucede cuando se ejecuta sin este mandato interno, tanto las funciones definidas en el script como las variables actualizadas afecten al shell actual una vez que finaliza la ejecución del script.

```
. script arg1 arg2 ... argn
```

A continuación se presentarán tres ejemplos de scripts cuyo estudio se deja al lector:

El primero, denominado `endir`, comprueba si en un determinado directorio, especificado como primer argumento, están presentes una serie de archivos que se le pasan como los siguientes argumentos.

```
#!/bin/sh
# Este script determina si un fichero esta presente en un
# directorio.
#   - Primer argumento: el nombre de un directorio
#   - Argumentos restantes: los ficheros que debe comprobarse
# Control de errores:
#   - Debe comprobarse que al menos se reciben dos argumentos.
#     Si no es asi se imprime un mensaje de error y termina
#     devolviendo un 1.
#   - Debe comprobarse que el primer argumento es un directorio.
#     Si no es asi se imprime un mensaje de error y termina
#     devolviendo un 2.
#   - Si alguno de los ficheros recibidos como argumentos no existen
#     en el directorio o existiendo no son ficheros regulares,
#     se imprime un mensaje de error y se continua
#     con el resto. Al final el script devolvera 3.
# NOTA: Este script solo tiene interes pedagogico. Esta no seria
#       la forma normal de hacer esta operacion

# Funcion que imprime un mensaje de error y termina. Recibe como
# parametro el valor que devolvera
Error()
{
    echo "Uso: 'basename $0' directorio fichero ...">&2
```

```

    exit $1
}

# Si el numero de argumentos es menor que 2, llama a Error (valor=1)
test $# -lt 2 && Error 1

# Si el primer argumento no es un directorio, llama a Error (valor=2)
test -d $1 || Error 2

# Variable que mantiene el valor que devolvera el script
ESTADO=0

# Obtiene el nombre del directorio
DIRECTORIO=$1

# Desplaza los argumentos mediante el mandato interno shift:
# $2->$1, $3->$2, ...
shift

# Bucle que hace una iteracion por cada fichero recibido como
# argumento. En cada una, FICHERO contiene dicho argumento
for FICHERO
do
    # Si el nombre del fichero contiene informacion del path
    # (p.ej. /etc/passwd) la elimina (dejaria passwd)
    FICHERO='basename $FICHERO'

    # Comprueba si existe un fichero regular con ese nombre
    if test -f $DIRECTORIO/$FICHERO
    then
        echo "$DIRECTORIO: $FICHERO"
    else
        echo "$DIRECTORIO: $FICHERO no existe o no es regular">&2
        ESTADO=3
    fi
done
exit $ESTADO

```

El segundo script, denominado listdir, muestra el contenido de un directorio.

```

#!/bin/sh
# Este script imprime el contenido de un directorio escribiendo un "/"
# al final de los nombres de los subdirectorios.
#     - Argumento: el nombre de un directorio
# Control de errores:
#     - Debe comprobarse que se recibe un solo argumento y que
#       este es un directorio.
#     Si no es asi se imprime un mensaje de error y termina

```

```

#         devolviendo un 1.

# NOTA: Este script solo tiene interes pedagogico. Esta no seria
#         la forma normal de hacer esta operacion

# Funcion que imprime un mensaje de error y termina.
Error()
{
    echo "Uso: 'basename $0' directorio">&2
    exit 1
}

# Si el numero de argumentos no es 1 o el argumento no es un
# directorio, llama a Error
test $# -eq 1 && test -d $1 || Error

# Obtiene el nombre del directorio
DIRECTORIO=$1

# Bucle que hace una iteracion por cada fichero existente (*) en
# DIRECTORIO.
for FICHERO in $DIRECTORIO/*
do
    if test -d $FICHERO
    then
        echo "$FICHERO/"
    else
        echo "$FICHERO"
    fi
done

```

El tercer ejemplo, denominado `elige1`, escribe los argumentos que recibe y permite que el usuario elija uno de ellos tecleando el número que le corresponde.

```

#!/bin/sh
# Este script escribe por la pantalla sus argumentos y permite que
# el usuario seleccione interactivamente uno de ellos.
# El dialogo con el usuario esta asociado directamente al terminal
# (/dev/tty) de esta forma se asegura que, aunque se redirija
# el script, el dialogo sera con el terminal.

# Comprueba que al menos recibe un argumento
test $# -ge 1 || exit 1
# Bucle que imprime los argumentos recibidos
NUMERO=0
for i
do
    NUMERO='expr $NUMERO + 1'

```

```

    echo "$i ($NUMERO)" > /dev/tty
done

ENTRADA=0
while test $ENTRADA -lt 1 || test $ENTRADA -gt $NUMERO
do
    echo -n "Elige introduciendo el numero correspondiente: ">/dev/tty
    read ENTRADA < /dev/tty
done

# imprime por la salida estandar el argumento seleccionado
shift `expr $ENTRADA - 1`
echo $1

```

## Ejecución de un mandato

Una vez que el shell ha analizado la línea leída, llevado a cabo todas las expansiones y sustituciones anteriormente explicadas, identificado el o los mandatos simples implicados y realizadas las redirecciones, se realizaría la siguiente labor por cada mandato simple involucrado:

- Si el nombre del mandato coincide con el de una función, se invoca la función que recibirá los argumentos especificados como sus parámetros posicionales.
- Si no es una función pero se trata de un mandato interno, se invoca el mandato interno con los argumentos especificados.
- Si no es ni función ni mandato interno y el nombre no incluye ningún carácter /, se busca si existe un fichero ejecutable con ese nombre en alguno de los directorios almacenados en la variable PATH. Si es así, se arranca un proceso (una llamada FORK seguida por una llamada EXEC) para ejecutar el programa. El programa arrancado recibirá los argumentos especificados y como entorno todas las variables exportadas. En el caso de que el ejecutable sea un script, los argumentos los obtendrá como sus parámetros posicionales y el entorno como variables.
- En el caso de que el nombre contenga algún /, el shell comprobará si existe un archivo con ese nombre y es ejecutable. Si es así, se procede de la misma forma que en el caso anterior, arrancando un proceso (una llamada FORK seguida por una llamada EXEC) para ejecutarlo.

### 4.6.2. Mandatos de UNIX

La interfaz de usuario no consiste únicamente en el shell, sino también en un gran número de programas del sistema disponibles para el usuario. Nótese que al tratarse de un intérprete con mandatos externos, el shell es meramente un lanzador de programas y, por lo tanto, el sistema debe incluir una serie de mandatos para que el usuario pueda realizar las operaciones que desea. El estándar POSIX 1003.2, por consiguiente, no sólo trata de los aspectos relacionados con el shell, sino que también especifica qué mandatos deben existir y cuál debe ser su comportamiento. Hay que recordar en este punto que, a pesar

del modelo de mandatos externos del shell, algunos mandatos se tienen que implementar como internos debido a que su efecto sólo puede lograrse si es el propio intérprete el que ejecuta el mandato. A grandes rasgos, los mandatos disponibles se pueden dividir en las siguientes categorías:

- Mandatos para manipular archivos y directorios
- Herramientas para el desarrollo de aplicaciones
- Filtros
- Administración del sistema
- Utilidades misceláneas

La mayoría de los mandatos son simples y realizan labores relativamente sencillas, pero están diseñados para poderse usar conjuntamente creando listas de mandatos que pueden llevar a cabo operaciones bastante complejas. Mención aparte merecen un tipo de mandatos a los que típicamente se les denomina filtros. Estos mandatos (como por ejemplo, **grep**, **sed**, **awk** o **sort**) leen unos datos de entrada, realizan un determinado procesamiento de los mismos (filtrado) y producen una salida. Este tipo de mandatos se usan muy frecuentemente para el desarrollo de shell scripts y, por lo tanto, los programadores de este entorno deben conocerlos a fondo.

En esta sección haremos una descripción muy breve (casi una mera enumeración) de algunos de los mandatos de uso más frecuente, tanto de los de carácter interno como de los externos. El lector se deberá remitir a las páginas correspondientes del manual o algunos de los libros incluidos en la bibliografía de la asignatura para obtener más información de los mismos.

## Mandatos internos

El shell ejecuta directamente este tipo de mandatos, o sea, el propio shell incluye código para realizar la funcionalidad asociada a un mandato de este tipo sin tener que activar ningún programa externo. Como se ha comentado previamente, este tratamiento especial se debe al carácter intrínsecamente interno de estos mandatos. Sin embargo, algunos intérpretes, por motivos de eficiencia y frecuencia de uso, implementan como internos algunos mandatos que no necesitarían serlo. Así, por ejemplo, el mandato **echo**, que imprime por la salida estándar un mensaje, está implementado en muchos shells como interno a pesar de no requerirlo. A continuación, se presenta una lista de algunos de los mandatos internos más frecuentemente usados.

- **break**: Termina la ejecución de un bucle
- **cd**: Cambia el directorio de trabajo actual
- **continue**: Prosigue con la siguiente iteración de un bucle
- **echo**: Escribe sus argumentos por la salida estándar
- **eval**: Construye un mandato simple concatenando sus argumentos y hace que el shell lo ejecute

- `exec`: Reemplaza el shell por otro programa. Si no se especifica ningún programa, permite redirigir los descriptores del propio shell
- `exit`: Termina la ejecución del shell
- `export`: Establece que las variables especificadas se exportarán
- `pwd`: Imprime el directorio de trabajo actual
- `read`: Lee una línea de entrada estándar asignando a variables lo leído
- `readonly`: Establece que las variables especificadas no pueden modificarse
- `return`: Termina la ejecución de una función
- `set`: Establece valor de opciones y de parámetros posicionales
- `shift`: Desplaza a la izquierda parámetros posicionales (`2- >1, ...`)
- `test`: Evalúa expresiones condicionales
- `trap`: Permite el manejo de señales
- `umask`: Establece la máscara de creación de archivos
- `unset`: Elimina una variable o función

## Mandatos externos

Antes de pasar a enumerar algunos de los mandatos externos más frecuentemente usados, es preciso resaltar que algunos de estos mandatos son complejos, incluso más que el propio shell, y requieren un esfuerzo considerable para lograr conocerlos bien. Así, por ejemplo, el programa `make`, que facilita el proceso de actualización de programas, puede tener un tamaño mayor que algunos shells.

- `ar`: Construye y mantiene bibliotecas de archivos
- `awk`: Lenguaje de procesamiento y búsqueda de patrones
- `bc`: Calculadora programable
- `basename`: Imprime nombre de archivo eliminando información de directorio
- `cat`: Concatena e imprime archivos
- `cc`: Compilador del lenguaje C
- `chmod`: Cambia el modo de protección de un archivo
- `chown`: Cambia el propietario de un archivo
- `cmp`: compara dos archivos
- `comm`: Imprime las líneas comunes de dos archivos

- cp: Copia archivos
- cut: Extrae los campos seleccionados de cada línea de un archivo
- date: Imprime la fecha y la hora
- dd: Copia un archivo realizando conversiones
- diff: Imprime las diferencias entre dos archivos
- dirname: Complementario de basename ya que devuelve la información de directorio de un nombre de archivo
- expr: Evalúa sus argumentos como una expresión
- false: Devuelve un valor de falso
- find: Encuentra los archivos que cumplan una condición
- grep: Busca las líneas de un archivo que sigan un patrón
- head: Imprime las primeras líneas de un archivo
- id: Imprime la identidad de un usuario
- kill: Manda una señal a un proceso
- ln: Crea un enlace a un archivo
- lpr: Manda un archivo a la impresora
- ls: Lista el contenido de un directorio
- make: Mantiene, actualiza y regenera programas
- man: Manual del sistema
- mkdir: Crea un directorio
- mkfifo: Crea un FIFO
- mv: Mueve o renombra un archivo
- od: Muestra el contenido de un archivo en varios formatos
- paste: Combina varios archivos como columnas de un único archivo
- pr: Formatea un archivo para imprimirlo
- printf: Escribe con formato
- ps: Muestra el estado de los procesos del sistema
- rm: Borra un archivo
- rmdir: Borra un directorio



- sed: Editor no interactivo
- sh: Invoca el shell de Bourne
- sleep: Suspende la ejecución durante un intervalo de tiempo
- sort: Ordena un archivo
- stty: Fija las opciones de un terminal
- tail: Imprime las últimas líneas de un archivo
- tee: Copia la entrada estándar a la salida estándar y a un archivo
- touch: Cambia las fechas de acceso y modificación de un archivo
- tr: Traduce (convierte) caracteres
- true: Devuelve un valor de verdadero
- tty: Devuelve el nombre del terminal del usuario
- uname: Devuelve el nombre del sistema
- uniq: Elimina líneas adyacentes idénticas
- wait: Espera por la finalización de procesos
- wc: Cuenta el número de caracteres, palabras y líneas de un archivo



## Capítulo 5

# Herramientas de desarrollo

Nos acercaremos al uso de diversas herramientas útiles para el programador.

Existen entornos integrados para el desarrollo de software pero no son de uso común. Clásicamente en UNIX se hace uso de diversas herramientas que cumplen con las diversas etapas del desarrollo.

Esta parte del curso es eminentemente práctica. El alumno deberá consultar la ayuda disponible. Como método rápido de consulta, se dispondrá de trípticos con resúmenes útiles de varias herramientas.

### 5.1. Editor

Vamos a ver tres editores tipo. El alumno deberá escoger aquella opción que más se adecue a sus necesidades.

En primera instancia, se recomienda que use el **pico**.

**pico** \_\_\_\_\_ *Simple Text Editor*

Es el editor que internamente usa la herramienta para correo electrónico **pine**. Es muy sencillo de usar pero poco potente.

A modo de introducción a su uso realice las siguientes tareas:

|                                                                                       |       |
|---------------------------------------------------------------------------------------|-------|
| <div style="border: 1px solid black; padding: 2px; display: inline-block;">pico</div> | T 5.1 |
| <i>Arranca el editor.</i>                                                             |       |

|                                                                                         |       |
|-----------------------------------------------------------------------------------------|-------|
| <div style="border: 1px solid black; padding: 2px; display: inline-block;">Ctrl G</div> | T 5.2 |
| <i>Accede a la ayuda para aprender los pocos controles que tiene.</i>                   |       |

|                                                                                         |       |
|-----------------------------------------------------------------------------------------|-------|
| <div style="border: 1px solid black; padding: 2px; display: inline-block;">Ctrl X</div> | T 5.3 |
| <i>Abandona el editor.</i>                                                              |       |

**vi** \_\_\_\_\_ *Visual Editor*

El **vi** es el editor más clásico y lo encontraremos en cualquier máquina UNIX. Es por eso que, a pesar de ser poco amigable, hay mucha gente que lo prefiere a otros más sofisticados. Existe una versión más moderna y potente llamada **vim**.

**vim***Arranca el editor.*T 5.5 **Esc***Cambia a “modo mandato”.*T 5.6 **:help***Accede a la ayuda navegable.*T 5.7 **:q***Abandona la edición actual.*T 5.8 **vimtutor***Arranca el editor en la realización de un tutorial del mismo.***emacs** \_\_\_\_\_ *GNU project Editor*





El **emacs**, desarrollado por FSF bajo licencia GNU, es mucho más que un simple editor. Es un entorno sobre el que se integran múltiples aplicaciones, correo electrónico, navegador web, news, etc., etc., etc. Está escrito sobre un intérprete de *lisp*.

Existe una versión llamada **Xemacs** más adaptada al entorno gráfico X, y totalmente compatible con el anterior.

T 5.9 **emacs***Arranca el editor.*T 5.10 **Ctrl G***Aborta la acción o mandato a medio especificar.*T 5.11 **Ctrl H T***Accede a la realización de un tutorial.*T 5.12 **Ctrl X Ctrl C***Abandona la edición actual.*

## Ejercicios

Para el editor de su elección, o para cada uno de ellos, realice los siguientes ejercicios prácticos:

T 5.13 *Arranque el editor para crear un nuevo fichero.*T 5.14 *Lea el fichero de ejemplo “estilos.c” desde el editor.*T 5.15 *Sin modificar su funcionalidad, formatee el texto con el estilo que considere más legible.*T 5.16 *Guarde el texto editado con el nombre “mi\_estilo.c”.*T 5.17 *Abandone el editor.*

## 5.2. Compilador

Compilar un lenguaje de alto nivel, léase C, es traducir el fichero o ficheros fuente en ficheros objeto, que contienen una aproximación al lenguaje máquina, más una lista de los símbolos externos referenciados pero no resueltos.

La etapa de montaje es capaz de enlazar unos ficheros objeto con otros, y a su vez con las bibliotecas necesarias, resolviendo todas las referencias a símbolos externos. Esta etapa produce finalmente un fichero ejecutable.

Un fichero ejecutable tiene un formato interno que el sistema operativo es capaz de entender, que describe como situar el proceso en memoria para su ejecución.

TODO: recuperar la figura

Figura 5.1: Fases de compilación y montaje.

En la figura 5.1 se observa la compilación separada de dos módulos `main.c` y `prim.c` y el montaje de los dos ficheros objeto resultantes para la obtención del fichero ejecutable `primes`.

### Extensiones

Ciertos sufijos o extensiones de los nombres de fichero, le indican al compilador los contenidos del mismo, y por lo tanto el tipo de tratamiento que debe realizarse:

- .c Fuente de C. Debe ser preprocesado, compilado y ensamblado.
- .h Fichero de cabecera de C. Contiene declaraciones de tipos de datos y prototipos de funciones. Debe ser preprocesado.
- .o Fichero objeto. Es el resultado de la compilación de un fichero fuente. Debe ser montado.
- .a Biblioteca de ficheros objeto. Se utiliza para resolver los símbolos de las funciones estándar que hemos usado en nuestro programa y que por lo tanto aún quedan sin resolver en los ficheros objeto.

### Errores y *Warnings*

Como ya habrá constatado, los lenguajes de programación, y entre ellos el C, presenta una sintaxis bastante estricta. Una de las misiones básicas del compilador es indicarnos los errores que hayamos cometido al codificar, dónde están y en qué consisten.

Los avisos que un compilador es capaz de notificar son de dos tipos:

**Errores.** Son problemas graves, tanto que evitan que el proceso de compilación pueda concluir. No obstante el compilador seguirá procesando el fichero o ficheros, en un intento de detectar todos los errores que haya.

Es imprescindible corregir estos errores para que el compilador puede terminar de hacer su trabajo.

**Warnings.** Son problemas leves o alertas sobre posibles problemas en nuestro código.

Aunque el compilador podrá terminar de hacer su trabajo, **es imprescindible corregir los problemas que originan estos warnings**, porque seguramente serán fuente de futuros problemas más graves.

La compilación de un programa correctamente realizado no debería dar ningún tipo de mensaje, y terminar limpiamente.

`gcc` \_\_\_\_\_ *GNU C and C++ Compiler*

El nombre que suele recibir el compilador de lenguaje C en UNIX es `cc`. Nosotros usaremos el `gcc` que es un compilador de C y C++, desarrollado por FSF bajo licencia GNU.

Es un compilador muy bueno, rápido y eficiente. Cubre las etapas de compilación y montaje. Parte de uno o más fuentes en C, objetos y/o bibliotecas y finalmente genera un ejecutable autónomo.

Admite multitud de opciones, entre las cuales cabe destacar:

- c *file.c* Realiza tan solo el paso de compilación del fichero indicado, pero no el de montaje.
- o *name* Solicita que el resultado de la compilación sea un fichero con este nombre.
- Wall Activa la detección de todo tipo de posibles errores. El compilador se convierte en más estricto.
- g Añade al fichero objeto o ejecutable información que (como veremos más adelante) permitirá la depuración simbólica del mismo.
- O Activa mecanismos de optimización que conseguirán un ejecutable más rápido, a costa de una compilación más larga.
- l*library* Solicita el montaje contra la biblioteca especificada.
- L*directory* Indica un directorio donde buscar las bibliotecas indicadas con la opción -l.

Tiene muchísimas más opciones. Consulte el manual si lo precisa.

T 5.18 `minimo.c`

*Escriba el fichero con el editor de su preferencia y los contenidos indicados en el capítulo anterior ??.*

T 5.19 `gcc minimo.c`

*Usamos el compilador para obtener el ejecutable resultante de este fuente.*

T 5.20 `ls`

*Observe que si no le indicamos al compilador otra cosa, el ejecutable resultante tomará el nombre por defecto `a.out`.*

T 5.21 `./a.out`

*Ejecutamos el fichero, indicando que se encuentra en el directorio actual. Esto será necesario si no tenemos el directorio “.” en la variable de entorno PATH.*

Ahora vamos a trabajar un programa más complejo.

`factorial.c`

T 5.22

*Escriba el fichero con el editor de su preferencia y los contenidos indicados en el capítulo anterior. El objetivo será mostrar el factorial de los números de 0 al 20 usando un bucle.*

`gcc -g -c factorial.c`

T 5.23

*Con este paso compilaremos, para obtener el fichero objeto `factorial.o`. Este fichero contendrá información simbólica.*

`gcc factorial.o -o factorial`

T 5.24

*Cuando compile correctamente, montaremos para obtener el ejecutable.*

`./factorial`

T 5.25

*Ejecute el mandato y compruebe su correcto funcionamiento.*

### 5.3. Depurador

La compilación sin mácula de un programa no implica que sea correcto. Es muy posible que contenga errores que sólo se harán visibles en el momento de su ejecución. Para corregir estos errores necesitamos una herramienta que nos permita supervisar la ejecución de nuestro programa y acorralar el error hasta localizarlo en el código.

Existen depuradores de bajo y de alto nivel. Los de bajo nivel nos permiten inspeccionar el código máquina o ensamblador, y su ejecución paso a paso. Los alto nivel, también llamados depuradores simbólicos, permiten visualizar y controlar la ejecución línea a línea del código fuente que hemos escrito, así como inspeccionar el valor de las variables de nuestro programa..

Para que un programa pueda ser depurado correctamente, su ejecutable deberá contener información que lo asocie al código fuente del que derivó. Para ello deberá haber sido compilado con la opción `-g`.

Algunas de las funciones del depurador son:

- Establecer puntos de parada en la ejecución del programa (*breakpoints*).
- Examinar el valor de variables
- Ejecutar el programa línea a línea

Nosotros usaremos el `gdb` desarrollado por FSF bajo licencia GNU.

También veremos en `ddd` que es un *frontend* gráfico para el `gdb`.

#### Fichero core

Trabajando en UNIX nos sucederá a menudo, que al ejecutar un programa este termine bruscamente y apareciendo algún mensaje de error como por ejemplo:

`segmentation fault: core dumped`.

Este mensaje nos informa de que nuestro programa en ejecución, esto es, el proceso, ha realizado un acceso ilegal a memoria y el sistema operativo lo ha matado. Así mismo nos indica que se ha generado un fichero de nombre `core`. El fichero `core` es un volcado de la

memoria del proceso en el preciso instante en que cometió el error, esto es una instantánea. Este fichero nos servirá para poder estudiar con todo detalle la situación que dio lugar al error.

`gdb prog [core]` \_\_\_\_\_ *GNU debugger*

Invocaremos al `gdb` indicándole el programa ejecutable que queremos depurar y opcionalmente indicaremos en fichero `core` que generó su anómala ejecución.

Una vez arrancado el depurador, proporciona una serie de mandatos propios para controlar la depuración:

`help` Da acceso a un menú de ayuda.

`run` Arranca la ejecución del programa.

`break` Establece un *breakpoint* (un número de línea o el nombre de una función).

`list` Imprime las líneas de código especificadas.

`print` Imprime el valor de una variable.

`continue` Continúa la ejecución del programa después de un *breakpoint*.

`next` Ejecuta la siguiente línea. Si se trata de una llamada a función, la ejecuta completa.

`step` Ejecuta la siguiente línea. Si se trata de una llamada a función, ejecuta sólo la llamada y se para al principio de la misma.

`quit` Termina la ejecución del depurador

Para más información sobre el depurador consulte su ayuda interactiva, el manual o el tríptico.

Más adelante tendremos oportunidad de hacer un uso extenso del depurador.

`ddd prog [core]` \_\_\_\_\_ *Data Display Debugger*

Es un *frontend* gráfico para el `gdb`, esto es, nos ofrece un interfaz gráfico bastante intuitivo para realizar la depuración, pero en definitiva estaremos usando el `gdb`. Como característica añadida, nos ofrece facilidades para la visualización gráfica de las estructuras de datos de nuestros programas, lo cual será muy útil. Para usar el `ddd` es preciso tener arrancadas las X-Windows.

## 5.4. Bibliotecas

Una regla básica para realizar software correcto es utilizar código que ya esté probado, y no estar reinventando sistemáticamente la rueda. Si existe una función de biblioteca que resuelve un problema deberemos usarla y no reescribir tal funcionalidad.

Existen muchas muchas bibliotecas con facilidades para realizar todo tipo de cosas: aritmética de múltiple precisión, gráficos, sonidos, etc., etc.

Para poder hacer uso correcto de esas funciones de biblioteca debemos consultar el manual que nos indicará la necesidad de incluir en nuestro fichero fuente cierto fichero de



cabecera (Ej. `#include <string.h>`), que contiene las declaraciones de tipos de datos y los prototipos de las funciones de biblioteca.

Se recomienda que consulte el tríptico de ANSI C.

Destacaremos aquí dos bibliotecas fundamentales:

**libc.a** Es la biblioteca estándar de C. Contiene funciones para: manejo de tiras de caracteres, entrada y salida estándar, etc.

Se corresponde con las funciones definidas en los ficheros de cabecera:

```
<assert.h>  <ctype.h>  <errno.h>  <float.h>  <limits.h>
<locale.h>  <math.h>  <setjmp.h>  <signal.h>  <stdarg.h>
<stddef.h>  <stdio.h>  <stdlib.h>  <string.h>  <time.h>
```

El montaje contra esta biblioteca se realiza de forma automática. No es preciso indicarlo, pero lo haríamos invocando al compilador con la opción `-lc`.

**libm.a** Es la biblioteca que contiene funciones de cálculo matemático y trigonométrico: `sqrt`, `pow`, `hypot`, `cos`, `atan`, etc. . Para hacer uso de las funciones de esta biblioteca habría que incluir en nuestro fichero `<math.h>`.

El montaje contra esta biblioteca se realiza invocando al compilador con la opción `-lm`.

`ar -opts [member] arch files... _____` *Manage Archive Files*

Cuando desee realizar medianos o grandes proyectos, podrá realizar sus propias bibliotecas utilizando esta herramienta. Es una utilidad para la creación y mantenimiento de bibliotecas de ficheros.

Para usar una biblioteca, se especifica en la línea de compilación la biblioteca (por convención `libnombre.a`) en vez de los objetos.

Algunas opciones frecuentemente usadas son:

`-d` Elimina de la biblioteca los ficheros especificados

`-r` Añade (o reemplaza si existe) a la biblioteca los ficheros especificados. Si no existe la biblioteca se crea

`-ru` Igual que `-r` pero sólo reemplaza si el fichero es más nuevo

`-t` Muestra una lista de los ficheros contenidos en la biblioteca

`-v` *Verbose*

`-x` Extrae de la biblioteca los ficheros especificados

A continuación se muestran algunos ejemplos del uso de este mandato.

Para obtener la lista de objetos contenidos en la biblioteca estándar de C, se ejecutaría:

```
ar -tv /usr/lib/libc.a
```

El siguiente mandato crea una biblioteca con objetos que manejan distintas estructuras de datos:

```
ar -rv $HOME/lib/libest.a pila.o lista.o
```

```
ar -rv $HOME/lib/libest.a arbol.o hash.o
```

Una vez creada la biblioteca, habría dos formas de compilar un programa que use esta biblioteca y además la matemática:

```
cc -o pr pr.c -lm $HOME/lib/libest.a
```

```
cc -o pr pr.c -lm -L$HOME/lib -lest
```

## 5.5. Constructor

Si el programa o aplicación que estamos desarrollando está convenientemente descompuesto en múltiples módulos, el proceso global de compilación puede ser automatizado haciendo uso de la herramienta **make**.

**make** \_\_\_\_\_ *Application Maintainer*

Esta utilidad facilita el proceso de generación y actualización de un programa. Determina automáticamente qué partes de un programa deben recompilarse ante una actualización de algunos módulos y las recompila. Para realizar este proceso, **make** debe conocer las dependencias entre los ficheros: un fichero debe actualizarse si alguno de los que depende es más nuevo.

La herramienta **make** consulta un fichero (**Makefile**) que contiene las reglas que especifican las dependencias de cada fichero objetivo y los mandatos para actualizarlo. A continuación, se muestra un ejemplo:

Makefile

```
# Esto es un comentario
CC=gcc                # Esto son macros
CFLAGS=-g
OBJS2=test.o prim.o

all: primes test      # Esta es la primera regla

primes: main.o prim.o # Esta es otra regla
        gcc -g -o primes main.o prim.o -lm
        # Este es el mandato asociado

test: ${OBJS2}         # Aquí usamos las macros
        ${CC} ${CFLAGS} -o $@ ${OBJS2}

main.o prim.o test.o : prim.h # Esta es una dependencia.

clean: # Esta no depende de nada, es obligatoria.
        rm -f main.o ${OBJS2}
```

Observe que las líneas que contienen los mandatos deben estar convenientemente tabuladas haciendo uso del tabulador, no de espacios. De no hacerlo así, la herramienta nos indicará que el formato del fichero es erróneo.

Ejecutaremos **make** para que se dispare la primera regla, o **make clean** explicitando la regla que queremos disparar. Entonces lo que sucederá es:

1. Se localiza la regla correspondiente al objetivo indicado.

2. Se tratan sus dependencias como objetivos y se disparan recursivamente.
3. Si el fichero objetivo es menos actual que alguno de los ficheros de los que depende, se realizan los mandatos asociados para actualizar el fichero objetivo.

Existen macros especiales. Por ejemplo, `$@` corresponde con el nombre del objetivo actual. Asimismo, se pueden especificar reglas basadas en la extensión de un fichero. `nas` de ellas están predefinidas (Ej. la que genera el `.o` a partir del `.c`).

### Ejercicio 5.1.

## 5.6. Otras herramientas

Existen variedad de otras herramientas que podrán serle de utilidad cuando desarrolle código en un sistema UNIX Citaremos tres:

#### `gprof`

Es una herramienta que nos permite realizar un perfil de la ejecución de nuestros programas, indicando dónde se pierde el tiempo, de manera que tendremos criterio para optimizar si es conveniente estas zonas de nuestro código.

#### `gcov`

Es semejante a la anterior, pero su objetivo es distinto. En la fase de verificación del programa, permite cubrir el código, es decir, asegurar que todas las líneas de código de nuestro programa han sido probadas con suficiencia.

#### `indent`

Permite endentar el ficheros fuente en C. Es muy parametrizable, para que el resultado final se corresponda con el estilo del propio autor.