Manual de supervivencia en Linux

Canek Peláez, editor
Facultad de Ciencias, UNAM

Autores:

Verónica Arriola Ríos
José Galaviz Casas
Emiliano Galeana Araujo
Karla García Alcántara
Luz de María Gasca Soto
Salvador Hernández López
Alejandro Hernández Mora
Canek Peláez Valdés
Luis Soto Martínez
Elisa Viso Gurovich
Víctor Zamora Gutiérrez

Basado en el manual original de:

Editores: Elisa Viso y Francisco Solsona
Autores: Mauricio Aldazosa, José Galaviz, Karla García, Iván Hernández, Canek Peláez, Karla Ramírez, Fernanda Sánchez Puig, Francisco Solsona, Manuel Sugawara, Arturo Vázquez y Elisa Viso

Índice general

Prefacio

Este libro presenta las aplicaciones y el enfoque que un estudiante de computación o un futuro programador debe dominar para sacar el mayor provecho de su enseñanza profesional en programación y, en general, en las ciencias de la computación. Hemos vertido aquí la experiencia de más de una década impartiendo cursos propedéuticos de *supervivencia* para estudiantes de la licenciatura en Ciencias de la Computación de la Facultad de Ciencias de la Universidad Nacional Autónoma de México.

A quién está dirigido

Este texto es un resumen (y en algunos casos, actualización) del libro *Manual de supervivencia en Linux*, el cual está dirigido a estudiantes y profesionales de la computación o futuros programadores que estén interesados en incursionar en el mundo del software libre, conocer el sistema operativo Linux y que requieran dominar rápidamente herramientas orientadas al desarrollo y productividad en este ambiente.

Es un libro práctico, por lo que es importante resaltar que todo lo que se ve en los distintos capítulos puede repetirse en una computadora personal con sistema operativo Linux.

Introducción a Unix 1

1.1. Historia

La primera versión de UNIX, llamada *Unics*, fue escrita en 1969 por *Ken Thompson*. Corría en una computadora PDP-7 de Digital. Más adelante, los laboratorios Bell, el MIT y General Electric se involucraron en la creación del sistema Multics, grande en tamaño, y el primer sistema de tiempo compartido que incluía muchas ideas innovadoras acerca de sistemas operativos. Thompson y algunos de sus colegas admiraban las capacidades de Multics; sin embargo, pensaban que era demasiado complicado. Su idea era demostrar que era posible construir un sistema operativo que proporcionara un ambiente de desarrollo cómodo de una forma más simple. Y afortunadamente, tuvieron un éxito admirable en el desarrollo de esta idea; a pesar de esto, UNIX es irónicamente mucho más complicado de lo que Multics nunca llegó a ser.

En 1970 Thompson, junto con *Dennis Ritchie*, portó UNIX a la PDP-11/20. Ritchie diseñó y escribió el primer compilador de *C* para proveer un lenguaje que pudiera ser usado para escribir una versión portátil del sistema. En 1973, Ritchie y Thompson reescribieron el kernel de UNIX, el corazón del sistema operativo, en C.

Inicialmente se otorgaron licencias gratuitas para utilizar UNIX a universidades con propósitos meramente educativos (en 1974). Esta versión fue la quinta edición (las ediciones hacen referencia a las ediciones del manual de referencia de UNIX). La sexta edición fue liberada en 1975; sin embargo, fue hasta la séptima edición, liberada por los laboratorios Bell en 1979, donde se logró la meta deseada: *portabilidad*; esta edición fue la que sirvió como punto de partida para la generación de este nuevo y maravilloso mundo: el mundo UNIX. Ésta es considerada la edición clásica de UNIX. Las dos vertientes más

fuertes creadas a partir de esta edición de UNIX son: System V (no confundirlo con la quinta edición) y el sistema BSD (Berkeley Software Distribution).

A pesar de que ambos sistemas surgieron de la misma influencia y comparten muchas características, los dos sistemas se desarrollaron con personalidades muy distintas. System V es más conservador y sólido; mientras que BSD es más innovador y experimental. System V era un sistema comercial, mientras que BSD no.

La historia es mucho más detallada y aún hay mucho que contar acerca del desarrollo de cada una de estas vertientes, de las cuales surgieron todavía muchas vertientes más de UNIX.

En el caso de la licenciatura en Ciencias de la Computación en la Facultad de Ciencias, como en muchas otras universidades de prestigio, el servicio al que acceden los estudiantes y la mayoría de los profesores es utilizando una implementación de UNIX, Linux, con computadoras conectadas en una red, por lo que todo lo que sucede entre los estudiantes y la computadora será en el contexto de esta red.

1.1.1. Sistemas UNIX libres

Los sistemas compatibles UNIX libres[1]para la arquitectura i386, notablemente Linux y FreeBSD, han hecho de UNIX un sistema al alcance de cualquiera. Estos sistemas proveen excelente rendimiento y vienen con un conjunto de características y herramientas estándar de UNIX. Son tan robustos que incluso muchas empresas han basado su desarrollo en alguno de estos sistemas.

Dado que estos sistemas no contienen software propietario pueden ser usados, copiados y distribuidos por cualquiera (incluso gratis). Las copias de estos sistemas se pueden conseguir en CD-ROM o bien por medio de Internet.

Linux. Linux es una versión de UNIX libre, originada por Linus Torvalds en la Universidad de Helsinki en Finlandia. Fue inspirado por Minix – escrito por Andrew Tanenbaum – y sigue más o menos las convenciones de System V. Su desarrollo ha sido llevado a cabo por programadores de todo el mundo, coordinados por Linus a través de Internet. Una gran cantidad de código incluida en el "sistema operativo Linux"[2]es GNU.

Hay varias distribuciones de Linux y algunas de las más importantes son:

- Fedora: `http://fedora.redhat.com/`, que es la utilizada en este trabajo.
- Ubuntu: `http://www.ubuntu.com/`
- Debian: `http://www.debian.org/`

[1]En este contexto *libre*, es una traducción literal de *free* que no significa *gratis*, (como en cervezas gratis), sino significa *libre* (como en libertad de expresión).

[2]Un *sistema operativo* es un conjunto de programas que controla y administra los recursos de la computadora, como son el disco, la memoria, los dispositivos de entrada y salida.

- openSUSE: `http://www.opensuse.org/`
- Arch: `http://www.archlinux.org/`, para usuarios avanzados.

Linux se distribuye bajo la licencia GPL (*G*eneral *P*ublic *L*icence), que es conocida como *CopyLeft* y que representa gran parte de la idiosincrasia del mundo UNIX libre.

FreeBSD. FreeBSD es derivado del sistema BSD (de la versión 4.4 para ser exactos); fue creado, igual que Linux, para la arquitectura i386. Su idea fundamental de desarrollo es tener un ambiente estable de desarrollo para esa arquitectura. Está soportado por un grupo internacional de voluntarios. La gran mayoría de los programas que conforman los sistemas FreeBSD, a diferencia de Linux, están gobernados por licencias estilo Berkeley, las cuales permiten redistribuciones siempre y cuando el código incluya una nota reconociendo el derecho de autor de *Regents of the University of California and the FreeBSD project*. Algunas otras partes de FreeBSD son GNU y están cubiertas por la GPL.

En este manual nos centraremos únicamente en Linux.

1.1.2. El proyecto GNU

El proyecto GNU buscaba desarrollar un sistema compatible con UNIX, totalmente libre[3]. Es operado por la FSF (*F*ree *S*oftware *F*oundation), organización fundada por *Richard Stallman*, quien ha escrito gran parte del código de GNU. GNU es una anagrama de *"GNU's not UNIX"*. Algunas herramientas invaluables para el trabajo diario de un estudiante de Ciencias de la Computación (al menos en esta Universidad) que han sido desarrolladas dentro del proyecto GNU son: un editor de texto (Emacs), el compilador de C (gcc), un depurador (gdb), y versiones de prácticamente todas las herramientas estándar de UNIX.

Todo el software distribuido por GNU se libera bajo la licencia GPL. De acuerdo con los términos de esta licencia, cualquiera puede modificar el software y redistribuir su propia versión. La restricción principal es que te obliga a redistribuirlo incluyendo el código fuente, aún en el caso de que tú hayas hecho las modificaciones.

UNIX es un sistema operativo multi-usuario y multi-tarea, es decir, permite a muchos usuarios conectarse al sistema y cada uno de estos usuarios puede ejecutar varios programas a la vez.

[3] A varios años de haber sido creada, lo han logrado. Su nombre es *Hurd* y promete muchas cosas interesantes en el mundo de los sistemas operativos. Sin embargo, Linux, (que sigue mucha de la filosofía de GNU, pero no es el proyecto GNU), le lleva mucha ventaja. Sin duda alguna ésta es una época interesante para estar metido en el software libre: nos esperan cosas buenas.

1.2. Sistemas de tiempo compartido

UNIX es un sistema de tiempo compartido. Esto significa que las computadoras que trabajan con UNIX pueden aceptar más de un usuario al mismo tiempo. La computadora principal (la parte del hardware que ejecuta realmente la mayoría del trabajo) recibe el nombre de computadora anfitrión. Una de las tareas del sistema operativo es asegurar que los recursos de la computadora anfitrión se compartan entre todos los usuarios.

Para trabajar con UNIX normalmente usarás una terminal. Una terminal UNIX básica consta de una pantalla, un teclado y algunas cosas más. No obstante, como se verá más adelante algunas terminales son más complejas que otras.

Cada vez que oprimes una tecla, la terminal envía una señal a la computadora anfitrión. Como respuesta, esta última envía su propia señal de regreso a la terminal, indicándole que despliegue el carácter correspondiente en la pantalla. UNIX fue configurado de tal forma que cuando la computadora anfitrión efectúa el eco, puedes ver que lo que teclea se recibe bien y que la conexión entre la computadora anfitrión y la terminal está intacta.

1.2.1. Los sistemas multiusuario

Muchas computadoras anfitrión están conectadas directamente a las terminales. No obstante, hay muchos centros de cómputo que ofrecen una variedad de computadoras anfitrión. En tales casos, la terminal podría estar conectada a una computadora especial, denominada servidor.

Casi todas las computadoras anfitrión cuentan con un teclado y una pantalla, que son parte de la computadora. Para UNIX, teclado y pantalla son sólo otra terminal. Sin embargo, dichas partes reciben el nombre especial de *consola*.

Un sistema UNIX clásico puede usar una computadora anfitrión que se encuentre en la oficina del administrador del sistema. Dicha computadora podría estar conectada a una sala llena de terminales o computadoras al otro extremo del corredor o en otro piso e incluso en otro edificio.

Básicamente, todos los sistemas UNIX aceptan varios usuarios a la vez. No obstante, algunas computadoras UNIX sólo pueden ser usadas por una persona simultáneamenente y son mejor conocidas como *estaciones de trabajo*[4].

[4]Esto no significa que la estación de trabajo no tenga capacidades multiusuario o multitarea; la estación de trabajo es una máquina con capacidades limitadas tanto en espacio en disco como en velocidad de procesamiento, razón por la cual es conveniente que sólo una persona la utilice.

1.3. Inicio de sesión

Dentro de los sistemas UNIX, todos los usuarios del sistema, sea éste uniusuario o multiusuario, se identifican con un nombre de usuario, también conocido como su *login* o *logname*, que será provisto por el administrador del sistema. La clave de usuario generalmente está relacionada con el nombre y apellido del usuario. Dos usos comunes es la de dar al usuario la primera letra de su nombre junto con su apellido, o las tres iniciales de su nombre y dos apellidos. Por ejemplo, supongamos que el nombre completo de un usuario es *Juan Pérez* y su nombre de usuario podría ser *juan* o *jp*. En este libro, utilizaremos *juan* como tu nombre de usuario.

Para usar un sistema UNIX, en primer lugar debes iniciar una sesión con un nombre de usuario; esto generalmente será a través de una interfaz gráfica de usuario; cubriremos esto en parte en el capítulo 2.

Ambientes gráficos en Linux | 2

2.1. Ambientes de escritorio

Un escritorio ofrece una interfaz gráfica de usuario[1] para interactuar con la computadora. La otra opción para interactuar con la computadora es mediante una interfaz de línea de comandos[2]. Para la mayoría de las personas el escritorio más común es Windows de Microsoft, para otras cuantas MacOS X. En Linux se tienen muchos escritorios que puedes utilizar en tu computadora. La base de varios estos escritorios es el *sistema de ventanas X*, que es un sistema originalmente de los ochentas que provee los elementos básicos para que se dibujen las ventanas y se interactúe con ellas. Una opción más moderna es *Wayland*, que es lo que utilizan los escritorios más importantes, incluyendo el que usaremos en el manual.

Existen varios escritorios para Linux; como suele ser el caso en UNIX, existe más de una aplicación para realizar la misma tarea. Algunos de los escritorios más comunes son:

- GNOME
- KDE
- Xfce
- Mate

[1] GUI, del inglés *Graphic User Interface*
[2] CLI, del inglés, *Command Line Interface*

2.2. GNOME

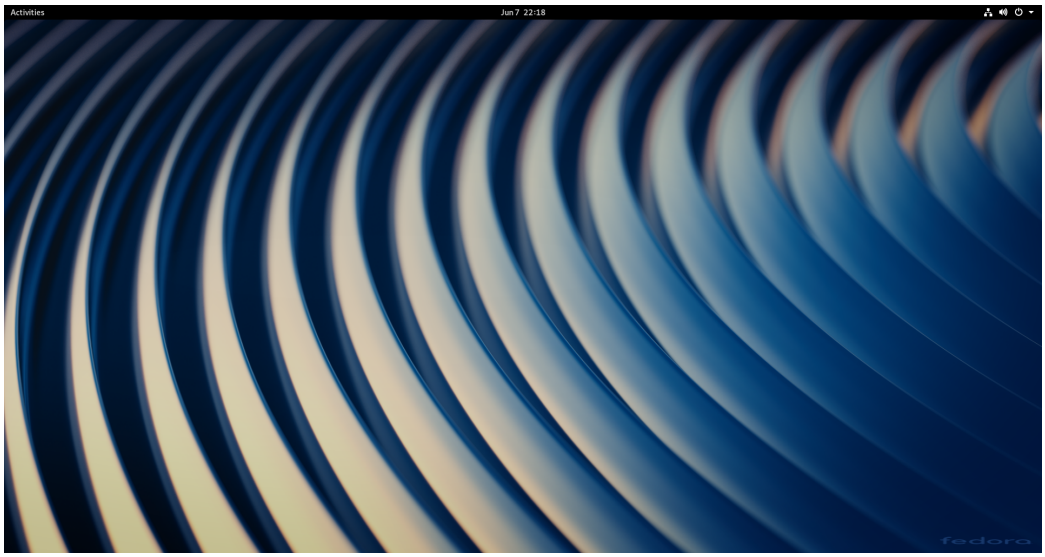
El sitio del proyecto GNOME[3], describe a la versión 3 del mismo como “una manera fácil y elegante de usar la computadora”.

La filosofía de GNOME es poder proporcionarle al usuario un escritorio fácil de usar para que pueda realizar su trabajo. En ese sentido, GNOME en general evita ofrecer muchas opciones de configuración, bajo la idea de que un ambiente de trabajo debe de funcionar sin que el usuario tenga que configurar nada: las cosas deben funcionar de manera automática.

2.2.1. El escritorio GNOME

El ambiente de trabajo de GNOME (conocido como el *shell*) consiste en únicamente un fondo de pantalla, y una barra superior donde están las actividades, el calendario y reloj, e indicadores distintos del sistema, como se muestra en la figura 2.1.

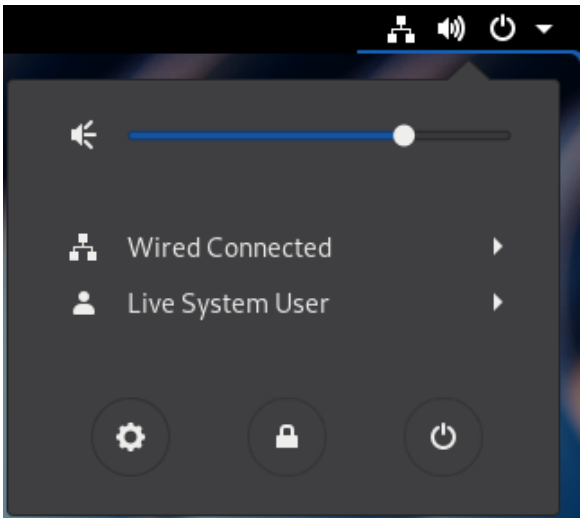
Figura 2.1 GNOME 3



Cada una de las zonas de la barra superior cumple una función distinta: los indicadores nos permiten verificar y configurar cosas como el idioma del teclado, el volumen del sonido, el tipo de conexión de red que estamos usando, así como opciones para cerrar la sesión o apagar la computadora, como se ve en la figura 2.2.

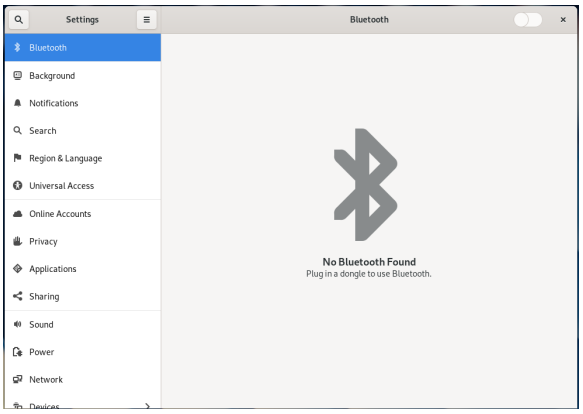
[3] `http://www.gnome.org/`

Figura 2.2 Indicadores



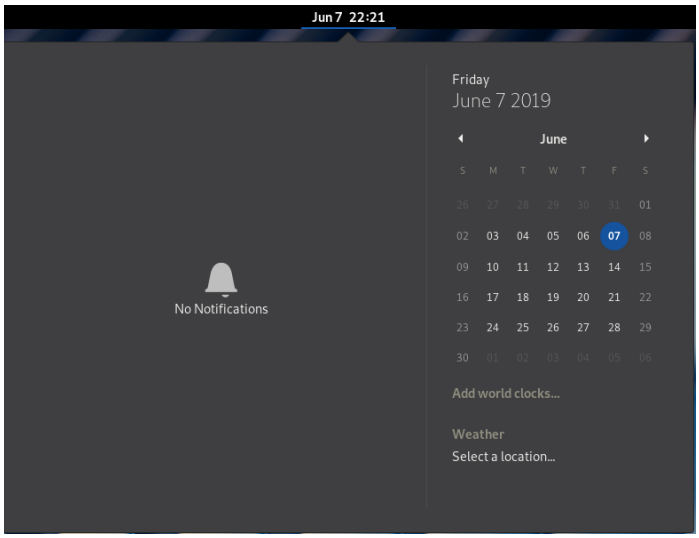
El icono de abajo a la izquierda nos permite acceder a las configuraciones del sistema (figura 2.3).

Figura 2.3 Configuraciones



El calendario y reloj nos permiten ver rápidamente la fecha y hora, así como citas o eventos que tengamos apuntados en nuestro calendario (figura 2.4).

Figura 2.4 Calendario



Y las actividades serán lo que más usaremos, así que se describiran en la siguiente sección:

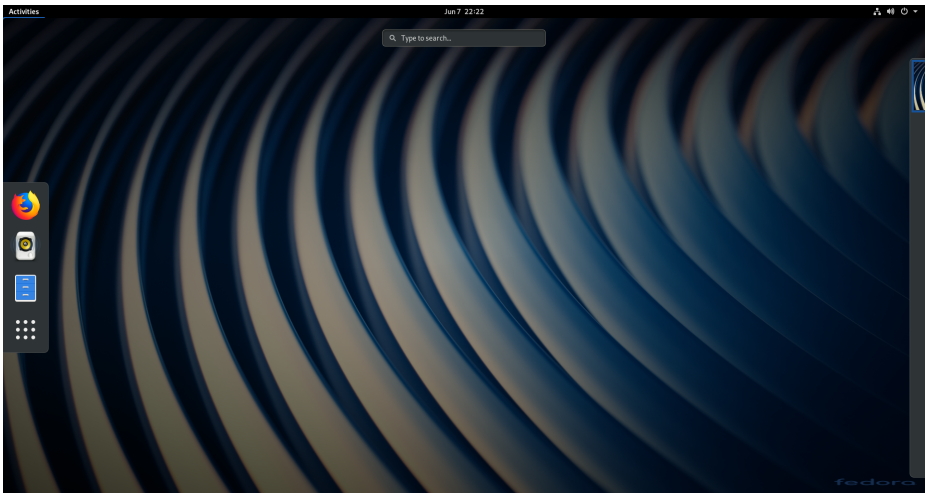
2.2.2. Actividades

Las actividades nos permiten organizar, lanzar, e interactuar con las aplicaciones con las estemos trabajando en nuestra computadora. Se puede acceder a las actividades de varias maneras: una es haciendo click con el ratón en la barra superior donde dice “Actividades”; otra es moviendo el ratón a la esquina superior izquierda; y una tercera más es presionando la tecla `Super`[4].

No importa como entremos a las actividades, se nos presentará una pantalla parecida a la figura 2.5.

[4]En los teclados de casi todas las PCs, es la tecla con el logo de Windows.

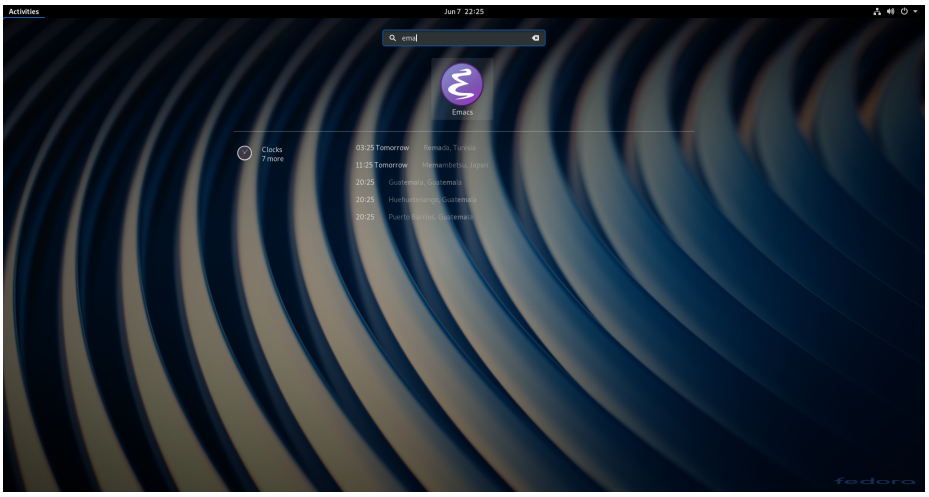
Figura 2.5 Actividades



La barra vertical a la izquierda es el tablero, y ahí podremos poner las aplicaciones que más usemos. Veremos esto en un momento.

Para lanzar una aplicación, si sabemos el nombre de la misma podemos sencillamente escribir su nombre; GNOME shell comenzará a buscar todas las aplicaciones, contactos y documentos que cacen con la cadena que escribamos (figura 2.6).

Figura 2.6 Buscando Emacs



Si no queremos escribir el nombre de la aplicación, o si no lo conocemos, podemos

buscarla en las aplicaciones, que es el último icono de arriba hacia abajo en el tablero (figura 2.5). Esto nos presentará todas las aplicaciones instaladas en el sistema (figura 2.7).

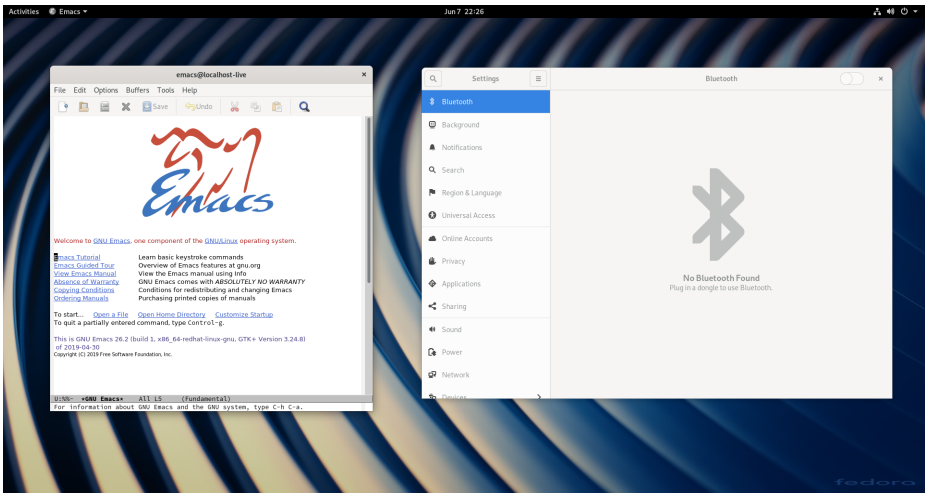
Figura 2.7 Aplicaciones

De cualquiera de estas maneras, podemos arrastrar el icono de Emacs al tablero, para poder lanzarlo de forma más rápida (figura 2.8).

Figura 2.8 Agregando Emacs al tablero

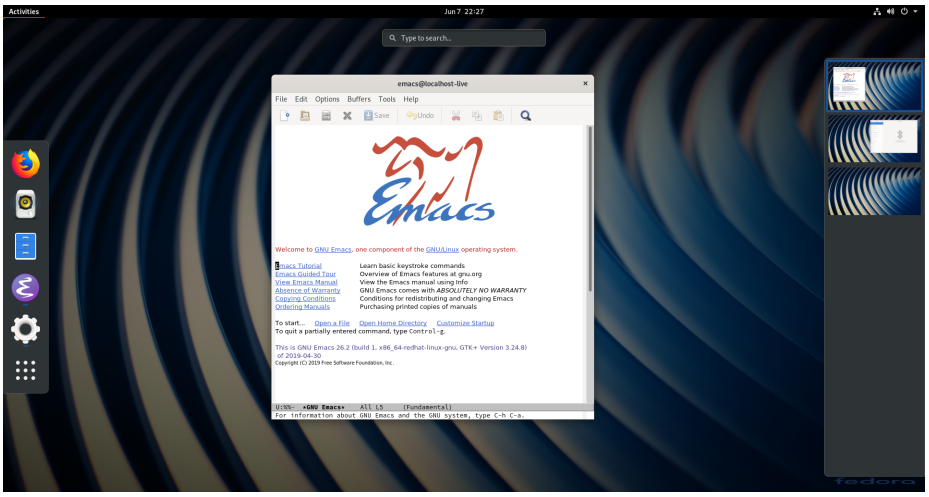
Ya que tengamos varias aplicaciones abiertas, en las actividades podemos verlas todas si están en el mismo escritorio virtual (figura 2.9).

Figura 2.9 Varias aplicaciones



El shell de GNOME soporta múltiples escritorios virtuales dinámicos, lo que nos permite organizar las aplicaciones que tengamos abiertas (figura 2.10).

Figura 2.10 Escritorios virtuales



Debe quedar claro que para usar GNOME 3 de forma eficiente, se prefiere el uso del teclado por encima del uso del ratón; se recomienda que consultes el sitio de atajos para el GNOME shell[5], pero aquí te damos los más comúnmente usados:

[5] `https://wiki.gnome.org/Projects/GnomeShell/CheatSheet`

Atajo	Descripción
System (Windows)	Cambia entre el escritorio y las actividades.
Alt + F1	Cambia entre el escritorio y las actividades.
Alt + F2	Lanza diálogo para ejecutar comando
Alt + Tab	Cambia aplicaciones
Alt + Shift + Tab	Cambia aplicaciones en el orden inverso
Alt + [tecla arriba de Tab]	Cambia entre ventanas de la misma apliación
Ctrl + Alt + Tab	Cambia entre elementos de shell
Ctrl + Shift + Alt + R	Inicia y detiene grabar el escritorio en un video
Ctrl + Alt + ↑ / ↓	Cambia escritorios virtuales
Ctrl + Alt + Shift + ↑ / ↓	Mueve la ventana actual a un escritorio virtual distinto

Sistema de Archivos 3

3.1. El sistema de archivos

Antes hablar sobre el sistema de archivos, debemos establecer lo que es un archivo. Un archivo puede entenderse como una secuencia de bytes con un significado concreto, que almacena cierta información, se guarda en un dispositivo de memoria(como un disco duro, un CD, un DVD o una unidad de almacenamiento USB) y tiene un nombre que lo identifica. Dentro de Unix podemos encontrar 3 tipos de archivos:

- Archivos regulares. Son todos aquellos que guardan información en sí mismos, y pueden tener una extensión en el nombre que indica de qué clase es la información que este almacena. Por ejemplo:

`txt`	Se refiere a archivos de texto
`jpg`	Se refiere a imágenes en formato JPEG
`java`	Programas de `Java`
`cc`	Programas de `C++`
`tex`	Documentos para LaTeX
`c`	Programas de C

- Archivos especiales de varias clases:
  - Archivos especiales de dispositivo, que proveen acceso a dispositivos tales como terminales o impresoras.

  - Archivos especiales FIFO (*First In First Out*, cola), algunas veces llamados *named pipes*, que proveen un canal para comunicación entre procesos independientes y que tienen un nombre asociado.

- Directorios[1], pueden contener archivos, incluso otros directorios y archivos especiales. El nombre de un directorio identifica a los archivos que éste agrupa.

Dentro de la computadora, toda la información que se guarda es vista como archivo. Debido a esto, es necesario definir una forma de organizarlos de manera que podamos trabajar con ellos. De no existir una organización, nuestros dispositivos de almacenamiento serían un caos donde se guarda cualquier cosa donde sea y como sea, además de ser necesario conocer el "idioma" de cada dispositivo de almacenamiento que se quiera usar.

El sistema de archivos es un componente muy importante en un sistema operativo; nos permite hacer uso de los dispositivos de almacenamiento de manera transparente y sin lidiar con elementos específicos del hardware. Establece cómo se administran los dispositivos de memoria, la forma en que los archivos se guardan en dichos dispositivos, entre otras cosas que van mas allá del alcance de este manual. Para fines prácticos, cuando hablemos del sistema de archivos nos referiremos a la estructura en que se organizan los archivos para ser manipulados por el usuario.

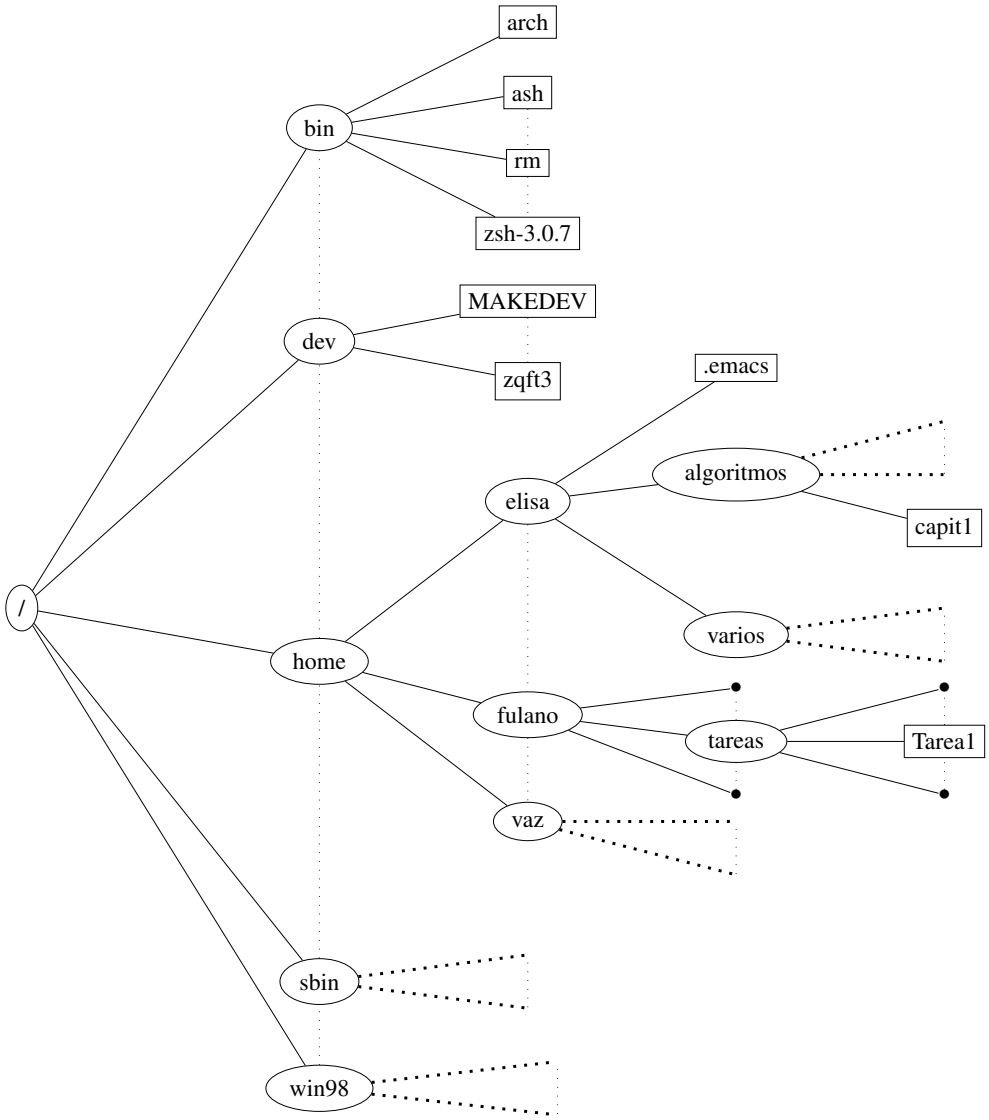
Unix organiza su sistema de archivos como una jerarquía de directorios. La jerarquía de directorios es un árbol (*tree*), pero dibujado de cabeza o de lado. Un directorio especial, `root`, es la raíz de la jerarquía y se denota por el carácter '`/`'. Dicho directorio almacena todos los archivos del sistema.

En los sistemas Unix, cada usuario tiene un directorio personal, en el cual almacena sus archivos. Dicho directorio se conoce como el *hogar (home)*. Una vez que entres en sesión, cada programa que se ejecuta se encuentra situado en un *directorio de trabajo*. El *shell* inicia, por omisión, en el directorio hogar del usuario. Un ejemplo simplificado de un directorio en un sistema Unix se muestra en la figura 3.1. En esta figura encerraremos en óvalos a los directorios y en rectángulos a los archivos. Mostramos puntos suspensivos cuando suponemos que en ese lugar aparecen un número considerable de archivos y/o directorios, pero por razones de espacio no los mostramos.

El nombre de un archivo puede contener cualquier carácter excepto '`/`' o el carácter nulo; sin embargo, algunos otros como '&' y ' ' (espacio en blanco) pueden causar problemas por sus significados especiales para su interpretación en la línea de comandos. Si tu sistema permite caracteres fuera del conjunto ASCII, tales como letras acentuadas (como nuestra letra *ñ*), también puedes usarlas para nombrar archivos. Los nombres de archivos son sensibles a las mayúsculas y minúsculas; esto es, `archivo` y `Archivo` identifican a dos archivos diferente. Por convención, los archivos cuyo nombre comienza con un punto ('.'), llamados *dot files*, por lo general se usan como archivos de configuración para programas.

[1] Los directorios también se conocen como carpetas.

Figura 3.1 Sistema de archivos en un sistema Unix



3.1.1. Rutas absolutas y relativas

Las *rutas* son expresiones que se usan para identificar un archivo o un conjunto de archivos. Se construyen bajando por el árbol del sistema de archivos, como el que se muestra en la figura 3.1.

Para referirse al directorio raíz se utiliza el carácter '/'; a partir del directorio raíz se puede descender en la jerarquía de directorios, siguiendo ramas del árbol, hasta llegar al

archivo o directorio deseado, separando cada directorio con el carácter '/'. Por ejemplo, para referirse al archivo llamado `ash` que se encuentra en el directorio `bin`, podemos usar la ruta `/bin/ash`.

Cuando se hace referencia a un archivo o directorio a partir del directorio raíz utilizando una ruta, se dice que es una *ruta absoluta*. No es necesario referirse a un archivo o directorio a partir de la raíz, sino que también se puede hacer a partir del directorio de trabajo, es decir, en el directorio en el que el usuario se encuentra parado. Cuando una referencia a un archivo o directorio se hace a partir del directorio de trabajo, se dice que es una *ruta relativa*.

La forma de presentar una ruta relativa es similar a la usada por las rutas absolutas, con la diferencia de que al principio no se pone el símbolo `/`. Así, el descenso en la jerarquía por las ramas del árbol no se lleva a cabo a partir de la raíz sino a partir del directorio de trabajo. Por ejemplo, supón que tu clave de usuario es `fulanito` y tu directorio hogar es `/home/fulanito`. Al iniciar una sesión te encontrarás en este directorio. Supongamos que te quieres referir al archivo `Tarea1` que está en el directorio `tareas` de tu propio directorio. Dicho archivo se representa usando la ruta absoluta

```
/home/fulanito/tareas/Tarea1
```

y usando una ruta relativa

```
tareas/Tarea1
```

El directorio actual siempre se denota por '`.`' y con '`..`' denotamos al directorio padre. Haciendo uso de dicha notación puedes usar trayectorias que vayan hacia arriba o hacia abajo en la jerarquía de directorios. Siguiendo con el ejemplo anterior, si el directorio de trabajo es `/home/fulanito/tareas`, entonces la ruta relativa del archivo `Ejemplo.java` en el directorio `algoritmos` del usuario `fulanito` es

```
../algoritmos/Ejemplo.java
```

El significado de esta ruta es: estando en el directorio `tareas`, utilizando '`..`' vamos al padre del mismo que es `fulano`; de ahí bajamos por la rama `algoritmos`; siguiendo esa ruta es que encontramos al archivo `Ejemplo.java`. A esta ruta relativa es equivalente la ruta absoluta

```
/home/fulano/algoritmos/Ejemplo.java
```

Nota que si el usuario se encontrara en otro lugar del árbol, la ruta relativa para alcanzar a `Ejemplo.java` sería otra. Es, precisamente, relativa al punto en el árbol en el que se encuentre el usuario.

Dado que el concepto de directorio base o *home* es tan importante, hay una notación especial para él, reconocida por casi cualquier intérprete de comandos de Unix: el carácter '~' al inicio de una trayectoria se refiere al directorio hogar del usuario actual, mientras

que '~vaz' hace referencia al directorio hogar del usuario vaz. También utilizaremos esa notación extensamente. Por ejemplo, para el usuario elisa, '~/alguno.txt' se refiere al archivo /home/elisa/alguno.txt.

Una notación menos conveniente para referirte al directorio *home* es $HOME, que nos regresa el valor de la variable de ambiente HOME que cualquier intérprete de comandos soporta. Por lo tanto, $HOME/alguno.c se refiere al archivo alguno.c en tu directorio *home*.

El *sistema de archivos* de Unix se puede dividir en distintos discos o particiones de un mismo disco o incluso en distintos medios de almacenamiento. El sistema de archivos se encarga de hacer esto transparente a los usuarios, para que parezca que todo vive en el mismo lugar. Existen distintas razones para hacer esto:

- Los discos y demás dispositivos que contienen los directorios y sus archivos pueden ser insuficientes para mantener la jerarquía completa, así que puede surgir la necesidad de distribuir la jerarquía en varios dispositivos.
- Para conveniencia en la administración, puede ser útil partir un disco muy grande en varias particiones.
- Los dispositivos de almacenamiento pueden no estar permanentemente unidos a la computadora. Un disco USB externo puede contener su propia estructura de directorios. El sistema de archivos de red permite que si las computadoras están conectadas entre sí tengan un sistema de archivos compartido.

Unix acomoda los sistemas de archivos externos asociándoles *puntos de montaje*. Un punto de montaje es un directorio en un sistema de archivos que corresponde al directorio raíz del sistema de archivos externo. El sistema de archivos primario es el que emana del directorio raíz real y se llama '/'. Un sistema de archivos externo se liga al sistema primario por medio del comando mount. Generalmente, todo esto es transparentes para los usuarios comunes, así que no debes preocuparte demasiado al respecto, aunque es bueno saberlo.

3.2. Moviéndose en el árbol del sistema de archivos

Para las siguientes secciones nos apoyaremos en una herramienta fundamental en Linux, la terminal. Se mencionarán conceptos que detallaremos en el respectivo capítulo.

Por omisión, la terminal no muestra en qué directorio estás trabajando. Es conveniente poder preguntar cuál es el directorio de trabajo (o directorio actual) y esto se hace con el comando:

```
~ $ pwd
```

(*Print Working Directory*), a lo que el sistema responde con la ruta absoluta del directorio en el que te encuentras.

Para cambiarte de directorio puedes utilizar rutas relativas, si es que a donde te quieres mover está hacia abajo en el árbol desde el directorio de trabajo; o bien, utilizar una ruta

absoluta para moverte arbitrariamente a cualquier otro directorio de trabajo. El comando que te permite cambiar de directorio de trabajo es

```
~ $ cd [ruta]
```

(*Change Directory*). Es importante mencionar que no te puedes mover a un archivo, sino únicamente a un directorio. Esto es porque al moverte, lo que se pretende es cambiar de directorio de trabajo.

Por ejemplo, si estando en el directorio de trabajo `home` quieres ubicarte en el directorio `varios` del usuario `elisa`, lo puedes hacer de las dos maneras siguientes:

```
home $ cd elisa/varios/
~ $ cd /home/elisa/varios/
```

El primer comando usa una ruta relativa, mientras que el segundo comando usa una ruta absoluta.
Para listar el contenido de un directorio usaa:

```
~ $ ls [parámetros] [ARCHIVOS]
```

Ambos modificadores se pueden omitir. Los *[parámetros]* se refieren al formato que puedes querer para la lista de archivos, mientras que *[archivos]* se refiere al lugar en el árbol del sistema de archivos del que quieres la lista. Si omites los *[parámetros]* la lista que se muestre será breve, sin mayor información más que el nombre del archivo o de los archivos del directorio de trabajo. Dos de los parámetros más comunes se refieren a pedir una lista extendida, que contenga, entre otra información, el tipo de archivo, su dueño, su tamaño, el día de su última modificación. Puedes querer que se listen también aquellos archivos que empiezan con algo que no es una letra, como un punto. Para ello, tecleamos el comando de la siguiente manera:

```
ls -al /home/fulanito/
```

y se mostrará en pantalla la lista de archivos y directorios que se encuentran en el directorio `/home/fulanito/`, pero únicamente los que cuelgan directamente de ese nodo. Si omites *[archivos]*, la lista que se dará será de los archivos y directorios inmediatamente colgando del directorio de trabajo.

Como puedes ver de la actividad anterior, muchas veces los archivos y directorios en un directorio son tantos, que si pides la lista extensa pudieses no alcanzar a verla completa. Para ello, haces lo que se llama *entubar* (en inglés, *pipe*), que consiste en redirigir la salida de un comando y usarla como entrada de un segundo comando; en este caso convendría que el segundo comando sea uno que nos enseñe la información por páginas. Tenemos dos de estos comandos, que se usan precedidos por el símbolo de entubamiento `'|'`: `less` y `more`.

Si utilizas el comando `less` para entubar:

```
fulanito@[estnum fulanito]% ls -al /bin | less
```

la respuesta se muestra "por páginas", esto es, muestra el número de renglones que le cabe en la pantalla, y se espera en la parte inferior de la misma a que teclees un espacio o que oprimas la tecla AvPág, para seguir con la siguiente página. Si tecleas q, el listado se suspende. El resultado que se te muestra con el comando anterior se puede ver en la figura 3.2.

Si el comando que se utiliza es `less`, entonces se podrá regresar sobre el listado, oprimiendo RePág; y avanzar tecleando espacio o AvPág.

También puedes "entubar" a que vaya a la impresora, si es que hay alguna conectada accesible, con el comando `lpr`.

Figura 3.2 Resultado de entubar `ls -al` con `less`

```
total 6364
drwxr-xr-x    2 root    root       4096 Apr 19  2000 .
drwxr-xr-x   22 root    root       4096 Jun  6 05:08 ..
-rwxr-xr-x    1 root    root       2612 Mar  7  2000 arch
-rwxr-xr-x    1 root    root      60592 Feb  3  2000 ash
-rwxr-xr-x    1 root    root     263064 Feb  3  2000 ash.static
-rwxr-xr-x    1 root    root       9968 Feb  3  2000 aumix-minimal
lrwxrwxrwx    1 root    root          4 Feb 17  2000 awk -> gawk
-rwxr-xr-x    1 root    root       5756 Mar  7  2000 basename
-rwxr-xr-x    1 root    root     316848 Feb 27  2000 bash
-rwxr-xr-x    1 root    root     446800 Feb  2  2000 bash2
lrwxrwxrwx    1 root    root          3 Feb 17  2000 bsh -> ash
-rwxr-xr-x    1 root    root       9528 Feb  7  2000 cat
```

Continúa en la siguiente página

Figura 3.2 Resultado de entubar `ls -al` con `more`

Continúa de la página anterior

```
-rwxr-xr-x    1 root    root      12044 Mar  7  2000 chgrp
-rwxr-xr-x    1 root    root      13436 Mar  7  2000 chmod
-rwxr-xr-x    1 root    root      11952 Mar  7  2000 chown
-rwxr-xr-x    1 root    root      49680 Mar  6  2000 consolechars
-rwxr-xr-x    1 root    root      33392 Mar  7  2000 cp
-rwxr-xr-x    1 root    root      45712 Feb  9  2000 cpio
lines 1-18
```

3.2.1. Permisos de archivos

La forma de manipular los archivos en Linux es a través de tres operaciones básicas: *lectura* (*read*), *escritura* (*write*) y *ejecución* (*execute*). Todo lo que necesites de los archivos lo puedes hacer con estas operaciones. Ahora bien, cómo dijimos antes, Linux es un

sistema multiusuario, por lo que sería potencialmente caótico que todos los usuarios tengan libertad para operar cualquier archivo de cualquier usuario. Para evitar esto contamos con los permisos.

Los *permisos de acceso* a un archivo, o simplemente *permisos*, especifican quién puede hacer qué a un archivo, ya sea para evitar que un archivo importante sea borrado o que un archivo confidencial sea visto. Dependiendo de si se trata de un archivo normal o directorio, los permisos de comportan un tanto diferente; los permisos de un archivo son asignados por su dueño y se pueden ver con el comando `ls -l` y modificarlos con el comando `chmod`.

Dado que los permisos de un archivo son un atributo del archivo mismo, todas las ligas a ese archivo tienen los mismos permisos.

Los permisos requeridos para realizar estas operaciones están denotados por '`r`', '`w`' y '`x`' respectivamente. El permiso `x` es necesario para programas compilados y cualquier guión (*script*) de shell que intentes usar como comando. Por ejemplo, si el archivo `qq` contiene un guión de shell pero no tiene el permiso `x`, no puedes ejecutar el comando `qq` por sí mismo. Si lo intentas, obtendrás un mensaje de error como: `command not found: qq`.

Sin embargo, puedes ejecutar `qq` con el comando

```
sh qq
```

aun cuando `qq` no tenga permiso de ejecución. En este ejemplo también necesitas que el archivo `qq` tenga permisos de lectura para poder ejecutar el script ya sea directa o indirectamente.

Los permisos `r`, `w` y `x` pueden especificarse independientemente para el dueño del archivo, para aquellos que se encuentran en el mismo grupo que el dueño y para todos los demás. Estas clases están representadas simbólicamente por `u` (*user*), `g` (*group*) y `o` (*others*; o `ugo`).

Los permisos que asignas a un archivo pueden protegerlo no sólo de acceso no deseado sino también de tus propios errores. Por ejemplo, puedes proteger un archivo contra modificaciones no intencionales o de borrarlo accidentalmente suprimiendo su permiso `w`.

Permisos para directorios

Los permisos de archivos también se aplican a los directorios.

- El permiso `r` para un directorio te permite ver qué hay dentro, pero es insuficiente para acceder a los archivos que se encuentran dentro.
- El permiso `w` para un directorio se requiere para poder añadir o borrar archivos de él. Sin embargo, `w` no es necesario para modificar un archivo listado en el directorio o borrar su contenido.
- El permiso `x` para un directorio te permite operar con los nombres en ese directorio si los conoces, pero no te permite ver cuáles son si no los conoces. Normalmente siempre que `r` se concede, también es así con el permiso `x`.

Otros permisos de archivos

Además de los permisos `rwx`, Unix tiene otras clases de permisos usados principalmente (pero no exclusivamente) por el administrador del sistema. Los describimos a continuación.

El set-uid bit. El bit `set-uid` le permite a un programa correr con los permisos de su dueño en lugar de con los permisos del usuario que lo llama. Puedes especificar el bit `set-uid` con `s` cuando usas el comando `chmod`. En un listado, se indica reemplazando la `x` con una `s`.

Normalmente, cuando llamas un programa y ese programa accede a un archivo, los privilegios de acceso del programa son los mismos que tú tienes. Como ejemplo, supón lo siguiente:

- El usuario `patito` es dueño del ejecutable `qq`.
- El programa `qq` utiliza el archivo privado de `patito` llamado `tarea`.
- El usuario `mamá-pato` ejecuta el programa `qq`.

Los permisos que se aplican a `tarea` durante la ejecución son aquéllos de "otros" (ya que `mamá-pato` no es dueña de `tarea`); no se utilizan los permisos del usuario `patito`. Por lo tanto, si `patito` no le dio permisos a su archivo `tarea` para que "otros" lo pudieran leer (todo el mundo), el programa `qq` no puede acceder a él. Una posible solución a esto es que `patito` le dé permisos de acceder al archivo `tarea` a todo el mundo, pero entonces `mamá-pato` podría hacer cualquier cosa con el archivo (incluso, cosas que `patito` no tenía intensión de permitir.)

El bit `set-uid` es una solución a esto. Cuando un proceso ejecuta un programa, invoca la función `exec` del sistema, especificándole a ésta la ruta del programa a ser llamado. Si los permisos del programa incluyen el bit `set-uid`, entonces el *ID* efectivo del proceso se convierte en aquél del dueño del programa, así que el programa se ejecuta con los privilegios del dueño del programa.

En efecto, el bit `set-uid` habilita a un usuario para acceder a un archivo como `tarea` indirectamente – con `mamá-pato` como agente – pero no directamente.

El ejemplo clásico del uso del bit `set-uid` es el comando `passwd`. El dueño del archivo `/etc/passwd` es `root` (el superusuario) y debe ser escrito solamente por el programa `passwd`. Los usuarios ordinarios deben ser capaces de poder ejecutar el comando `passwd`, pero no de modificar directamente el archivo `passwd`. Por lo tanto, el bit `set-uid` está encendido[2] para el programa `passwd`, lo que le permite escribir en el archivo `/etc/passwd`, mientras que esa habilidad permanece negada para usuarios ordinarios.

[2]En este contexto, utilizaremos indistintamente prendido, encendido, arriba o puesto, para indicar que ese bit ha sido activado por medio de alguna utilería válida como el comando `chmod`.

El bit set-gid. El bit `set-gid` es como el `set-uid` excepto que se aplica a los permisos de grupo en lugar de a los permisos de dueño. El bit `set-gid` se especifica con una `s` en la posición de la `x` en los permisos del grupo.

El locking bit. Si el `locking bit` de un archivo está prendido, un programa que esté leyendo o escribiendo el archivo puede bloquear otros intentos de leer o escribir en él al mismo tiempo. Prender este bit previene que accesos simultáneos a un archivo puedan corromper su significado y dejarlo en un estado inconsistente. El *locking bit* está representado por una `l` reemplazando a la `x` en los permisos del grupo.

El sticky bit. Cuando se aplica a un directorio, el `sticky bit` previene que los archivos dentro de él sean borrados o renombrados por otra persona que no sea su dueño. Cuando se aplica a un archivo ejecutable, es la herramienta adecuada para retener al programa en memoria cuando dicho programa puede ser compartido por varios usuarios. Sólo el superusuario o el dueño del archivo puede activar el *sticky bit*. Este bit está indicado por una `t` en el lugar de la `x` en los permisos de otros.

Construcción de permisos

Los comandos `chmod`, `umask` y `find` hacen uso de la habilidad para construir conjuntos de permisos simbólicamente. Esto lo puedes lograr especificando un modo *simbólico* que les indique cómo modificar un conjunto existente, posiblemente vacío, de permisos.

Un modo simbólico consiste de una o más cláusulas separadas por comas. Cada cláusula, en turno, consiste de cero o más letras "quién" seguidas por una secuencia de una o más acciones a aplicar a la categoría designada por las letras "quién". Cada acción consiste de un operador seguido ya sea de una o más letras de permisos, de una letra "quién" o de nada en absoluto. Estas son las letras "quién":

u Permiso para el usuario del archivo.
g Permiso para el grupo del archivo.
o Permiso para otros (i.e. el resto del mundo.)
a Permiso para cualquiera (equivalente a `ugo`.)

Si omites las letras "quién" que preceden al operador, se supone `a` y la máscara de creación (de la cual hablaremos más adelante). Éstos son los operadores:

+ Añade estos permisos.
- Quita estos permisos.
= Prende exactamente estos permisos, quitando cualquier otro para las letras "quién" indicadas.

Éstas son las letras de los permisos:

r *Lectura*
w *Escritura*
x *Ejecución*
X *Ejecución* sólo si el archivo es un directorio o algún permiso `x` ya está puesto

s *Establece el ID de usuario o grupo*
t *Bit pegajoso*
l *Bloqueo durante el acceso*

Si especificas una letra "quién" después de un operador, entonces se copian los permisos asociados con la letra indicada.

Representación octal de los permisos

Los permisos para un archivo pueden ser especificados como un número octal. Esta forma es obsoleta y no recomendada, pero mucha documentación antigua de Unix utiliza la forma octal muy seguido; también algunos sistemas (no nuestro caso, en la Licenciatura en Ciencias de la Computación) no soportan aún la forma simbólica en todos los contextos. El número octal se obtiene sumando lógicamente los números de la siguiente lista:

4000 Establece el ID del usuario en ejecución.
20d0 Establce el ID del grupo en ejecución si *d* es 7,5,3 o 1 (permiso de ejecución otrogado); habilitar el bloque de otra forma.
1000 Habilita el bit pegajoso.
0400 Establece el permiso de lectura para el dueño.
0200 Establece el permiso de escritura para el dueño.
0100 Establece el permiso de ejecución para el dueño.
0040,0020,0010 Establece permisos de lectura, escritura o ejecución para el grupo.
0004,0002,0001 Establece permisos de lectura, escritura o ejecución para el resto de los usuarios.

Permisos para archivos recién creados

Cuando un programa crea un archivo, especifica un conjunto de permisos para ese archivo. Conjuntos típicos son `u=rw g=rw o=rw` (`rw-rw-rw-`, o 666 en octal) para archivos de datos y `u=rwx g=rwx o=rwx` (`rwxrwxrwx`, o 777 en octal) para archivos ejecutables. Los permisos iniciales son pocos debido a la *máscara de creación de archivos*, también conocida como la *umask* ("user mask").Tú puedes cambiarla con el comando `umask`. En otras palabras, los bits en la máscara de creación de archivos representan permisos que serán negados a los archivos recién creados.

Los permisos de un archivo recién creado se calculan restando lógicamente los bits en la máscara de creación de archivos de los permisos especificados por el programa que lo crea. Por lo tanto, si tu máscara vale algo como `rwxrwxr-x` (octal 002), que excluye el permiso de escritura de otros, aquéllos fuera de tu grupo no podrán escribir o borrar archivos que tú crees a menos que les cambies los permisos más tarde (con `chmod`, por ejemplo).

Un valor típico para `umask` es `rwxr-xr-x` (octal 022), que niega los permisos de escritura a todos menos a ti. Con este valor de máscara, un archivo creado con permisos especificados como `u=rw g=rw o=rw` es en realidad creado con permisos `u=rw g=r o=r`.

La reducción de permisos por el valor de *umask* se aplica tanto a la creación de nuevos directorios como de nuevos archivos.

La Terminal | 4

En este capítulo exploraremos una de las herramientas más útiles e importantes en un sistema Unix y algunas de las razones por las cuales resulta extremadamente conveniente familiarizarce con ella. Comenzaremos por definir lo que es una terminal, luego explicaremos algunos detalles para entenderla mejor, también algunos trucos que facilitan su uso y ayudan a realizar las tareas en unos cuantos comandos. Finalmente, mostraremos una alternativa para sustituir todos los comandos por una sola instrucción que teclear.

La terminal, línea de comandos, o consola es un interprete de comandos que nos permite interactuar con la computadora en modo de texto, es decir sin la necesidad de un entorno gráfico. Por otro lado, los comandos son instrucciones que se le dan a la computadora y ejecutan uno o varios programas. Normalmente veremos la terminal como una ventana negra que solo puede desplegar texto y muestra un *prompt* que indica que está en espera de recibir comandos.

El término *comando* se utiliza también para referirse a instrucciones que esperan los editores de texto como `vi` o `emacs` y otro tipo de programas.

4.1. Sintaxis estándar de comandos

POSIX[1] define una sintaxis estándar para comandos, pero no todos los comandos la siguen. Muchos comandos definidos por POSIX tienen una forma moderna y una sintaxis obsoleta, por compatibilidad con sistemas viejos. La razón de esto es que muchos programas importantes – y muy viejos – que aún están en uso, siguen sintaxis vieja de algunos comandos. Siempre que hablemos de la sintaxis de un comando, trataremos de apegarnos a la sintaxis moderna de POSIX, pero mencionaremos de vez en cuando la sintaxis vieja

para algunos de ellos.

Aun en sistemas modernos hay una gran cantidad de comandos que no se apegan al estándar. Por ejemplo, el estándar especifica que las opciones de los comandos (no así, los operandos) pueden aparecer en cualquier orden, pero algunos comandos pueden requerir que las opciones aparezcan en un orden particular y muchas veces esto no está mencionado explícitamente en la página del manual del comando. En estos y otros casos hacer experimentos siempre es una buena idea.

Un comando consiste de una sucesión de palabras separadas por espacios en blanco (estos espacios en blanco pueden ser espacios sencillos, tabuladores e incluso – en algunos shell, solamente – nuevas líneas escapadas[2]). La primera palabra es el nombre del comando y las palabras subsecuentes son sus *argumentos* u *opciones*. El nombre del comando se refiere a un programa o guión (*script*) de shell a ser ejecutado. Los operandos consisten de las opciones del programa, si hay alguna, seguidas de sus operandos, si hay alguno. Las opciones controlan o modifican lo que el comando hace. Los operandos especifican nombres de trayectorias o con lo que va a trabajar el comando.

Especificación de opciones. Las opciones se especifican con una sucesión de palabras. Cada palabra es un *grupo de opciones* o una *opción argumento*. Las opciones generalmente se denotan por letras. Se pueden escribir en cualquier orden y se pueden combinar varias opciones en un solo grupo (siempre y cuando no reciban ningún argumento). Cada grupo de opciones está precedido por un guión (-). Por ejemplo, los comandos

```
ls -al
ls -l -a
```

son equivalentes e indican que el comando `ls` (*list archives*) sea llamado con las opciones `a` (*all files*) y `l` (*long listing*).

La última (o única opción) de un grupo de opciones puede ir seguida por una palabra que especifique uno o más argumentos opcionales para ella. Por ejemplo, el comando

```
sed -f qqscript qqarchivo
```

causa que el editor `sed` edite el archivo `qqarchivo` usando el guión de shell `qqscript`; en este caso, `qqscript` es un argumento para la opción `-f` y `qqarchivo` es un argumento para el comando `sed` mismo.

[1]Portable Operating System Interfaz es un estándar definido por la IEEE – Institute of Electrical and Electronics Engineers – y ANSI – American National Standards Institute –. POSIX tiene como finalidad definir un sistema operativo que se *comporte como* Unix, sea o no Unix. Han existido varias versiones de POSIX, siendo la más importante POSIX.2, parte 2.

[2]Una nueva línea "escapada" es una que está precedida por una diagonal invertida (\).

Cuando se reciben varios argumentos para una misma opción, generalmente se separan por comas, pero se pueden separar por blancos siempre y cuando se encierre la lista de argumentos entre comillas (dobles o sencillas), o si se pone una diagonal inversa frente a cada espacio en blanco (escapando los espacios en blanco). En cualquier caso, los argumentos deben formar una sola palabra. Los dos ejemplos que siguen muestran ambas convenciones:

```
prog -o zed1,zed2 -y "eres o no eres"
```

o alternativamente,

```
prog -o zed1,zed2 -y eres\ o\ no\ eres
```

Otras convenciones para argumentos. Hay otras dos convenciones muy comunes para especificar argumentos:

'- -' Indica el final de las opciones. Esto es muy útil cuando quieres pasar argumentos que empiezan con `-` a un comando. Por ejemplo,
`rm -- -qq`
es una forma de indicar que `-qq` es un archivo y no una opción, por lo que el comando borrará el archivo `-qq`. Si sólo se escribe `rm -qq`, obtendríamos un error sobre una opción `q` que no reconoce.

'-' Un solo guión representa a la entrada estándar en un contexto en que el programa espera una trayectoria. Por ejemplo, el comando
`diff - qq`
encuentra las diferencias entre la entrada estándar y el comando `qq`.

4.2. Agrupando nombres de archivos

Usualmente, es necesario hacer referencia a un conjunto de archivos o directorios. Para este propósito se utilizan construcciones con comodines. Un comodín es una construcción que se puede remplazar por un conjunto o secuencia de caracteres.

El comodín '?' se usa para representar un carácter cualquiera. Por ejemplo, todos los archivos en el directorio `/bin/` que empiezan con cualquier carácter al que le sigue nada más `sh` se pueden referir por medio de la expresión `/bin/?sh`.

El carácter '$*$' es otro comodín que se reemplaza por cualquier secuencia de caracteres, incluso la secuencia vacía, que es aquella que no tiene ningún carácter.

La tercera y última forma de expresiones con comodines son las que usan la expresión `[...]`; los caracteres dentro de los corchetes proporcionan una lista para que se trate de casar con alguno (y sólo uno) de ellos.

4.3. Archivos estándar y redireccionamiento

Muchos comandos de Unix leen su entrada de la *entrada estándar* a menos que especifiques archivos de entrada; y escriben su salida a la *salida estándar*. Por omisión, tanto la entrada como la salida están asociados a tu terminal (generalmente la salida es el monitor y la entrada es el teclado). Otro archivo estándar es el *error estándar*, que se usa para mensajes de error y otra información acerca del funcionamiento de un comando. La información enviada al error estándar también llega a la terminal.

Esta convención funciona bien porque puedes redireccionar tanto la salida como la entrada estándar. Cuando redireccionas la entrada estándar para ser tomada del archivo *archivo*, el programa que lee la entrada estándar leerá del archivo *archivo* en lugar de hacerlo desde la terminal. Algo similar sucede cuando redireccionas la salida estándar. Cuando se redirecciona la entrada se usa el operador < precediendo a una trayectoria, mientras que para el redireccionamiento de la salida se usa el operador >.

Por ejemplo, el comando `cat` copia la entrada estándar a la salida estándar, por lo tanto el comando

```
cat < felix > gato
```

copia el archivo `felix` al archivo `gato`[3]. Si utilizas el operador ≫ antes de una trayectoria, entonces la salida es agregada al final del archivo en lugar de escrita directamente. En el ejemplo anterior, el contenido de `gato` es sobrescrito con el contenido de `felix`; pero si haces en su lugar

```
cat < felix ≫ gato
```

el contenido de `felix` es agregado al final del contenido de `gato`. Discutiremos más formas de redireccionamiento en breve.

El error estándar también se puede redireccionar pero la sintaxis para hacerlo es dependiente del tipo de shell que se utilice.

- Para el shell Bourne (`sh`) y la mayoría de los que pertenecen a su familia, como `zsh`, `bash`, `korn`, etc., la construcción '`2> archivo`' especifica que cualquier cosa que sea enviada al error estándar será redireccionada al archivo `archivo`.

- Para el shell C, se utiliza la construcción `>& archivo`, que manda tanto la salida estándar como el error estándar al archivo `archivo`. Este shell (y de su familia sólo `tcsh`) no permite enviar la salida estándar y el error estándar a distintos archivos. De hecho, el uso del shell C está en decadencia, pero aún tiene algunos seguidores.

Una propiedad importante y valiosa del redireccionamiento es que un programa cuya entrada o salida es redireccionada no tiene por qué saberlo. En otras palabras, un programa

[3]Nota que como no das más que el nombre del archivo, la trayectoria se refiere al directorio de trabajo.

escrito bajo la suposición de que lee de la entrada estándar y escribe a la salida estándar también lee y escribe a archivos, siempre y cuando estos últimos sean pasados como redireccionamientos. La misma propiedad se aplica a guiones de shell.

Un uso común de `cat` es la creación de archivos nuevos, que generalmente contienen unas pocas líneas que se pueden teclear directamente de la pantalla. Por ejemplo,

```
~ $ cat > felix
Esta es una prueba de cat
para ver exactamente como funciona
^D
```

crea al archivo `felix` en el directorio de trabajo, y el contenido del archivo se tomó de la entrada estándar, que se dio por terminada al teclear `C-d` (representado por `^D` en el listado), que corresponde a un fin de archivo.
Otra manera de crear archivos es mediante el comando

```
touch mimosa
```

En realidad este comando lo que hace es actualizar la fecha de última modificación del archivo `mimosa`. Sin embargo, si el archivo no existe lo crea.

A lo largo de estas notas diremos que un programa produce un *resultado* cuando manda algo a la salida estándar. En la terminología tradicional de Unix se dice que *imprime* el resultado. Esta terminología viene de los días en los que la mayoría de las estaciones de trabajo estaban conectadas a un teletipo en lugar de un monitor, pero no debe confundirse con la acción de enviar un archivo a la impresora.

4.4. Entubamientos y filtros

Puedes usar la salida de un comando como la entrada de otro comando, conectando ambos comandos con un tubo (*pipe*). La construcción resultante es un *pipeline* o *entubamiento*. Al igual que con las formas de redireccionamiento discutidas anteriormente, los programas conectados no tienen por qué saber de dichas conexiones.

Sintácticamente, creas un entubamiento escribiendo dos comandos con un *pipe* (el símbolo |) entre ellos. Por ejemplo, la línea de comandos

```
~ $ grep pest phones | sort
```

llama al programa `grep`, que extrae del archivo `phones` todas las líneas que contengan la cadena `pest` y produce esas líneas como su salida estándar. Esta salida se entuba a la entrada del programa `sort`, que ve dicha salida como su entrada. La salida de la línea de comandos completa es una lista ordenada de todas las líneas en `phones` que contienen `pest`.

En una línea que contenga tanto redireccionamientos como entubamientos, los redireccionamientos tienen mayor precedencia: primero los redireccionamientos se asocian con comandos y después los comandos con sus redireccionamientos se pasan por los entubamientos.

La creación de un entubamiento implica la creación de un par de procesos, uno para el comando que produce la información entubada y otro para el proceso que consume esta información. Estos procesos los crea el shell que interpreta la línea de comandos.

Un *filtro* es un programa que lee datos de la entrada estándar, los transforma de alguna manera y luego escribe esta información transformada a la salida estándar. Uno puede construir un entubamiento como una sucesión de filtros; estas sucesiones proveen una forma muy poderosa y flexible de usar comandos simples para que al combinarlos alcanzar grandes metas, posiblemente muy complejas. Generalmente este tipo de entubamientos se citan como ejemplos de la *filosofía Unix*.

Hay otras herramientas en los Unix estándar para simular estos entubamientos: archivos especiales *FIFO*, *sockets* y *streams*. Pero son mucho más específicos y sofisticados y definitivamente fuera del alcance del presente trabajo.

4.5. Citas (*quotations*)

Todos los shells estándar asignan significados especiales a ciertos caracteres, llamados *meta-caracteres*. Para usar estos caracteres como caracteres ordinarios en la línea de comandos, debes citarlos (marcarlos de manera especial para que el shell **no** los interprete). Cada shell tiene su propio conjunto de meta-caracteres y sus propias convenciones de cita, pero algunas reglas generales siempre se cumplen:

- Una diagonal inversa (\) siempre sirve como carácter de escape para uno o más caracteres que le sucedan, lo cual le da a esos caracteres un significado especial. Si un carácter es un meta-carácter, el significado especial es – generalmente – el carácter mismo. Por ejemplo, \\ usualmente significa una sola \ y \$ el signo de pesos. En estos casos, la diagonal inversa le quita su significado a los caracteres, generalmente llamados caracteres escapados. La diagonal inversa misma es **el** *carácter de escape*.
- Cuando un texto se encierra entre comillas (" "), la mayoría de los meta-carácteres que estén en dicho texto son tratados como caracteres normales, excepto por $, que generalmente indica sustituciones a realizar.
- Otra forma de citar muy similar a la anterior, pero más fuerte es usar comillas sencillas ya que ni siquiera el carácter $ es interpretado.

Los espacios y tabuladores, si se encuentran dentro de algún texto citado, siempre son tratados como caracteres normales.

4.6. Utilidades de la terminal

A lo largo de la sección, podríamos notar que lo que hicimos con la terminal, se puede realizar igualmente con el entorno gráfico. La ventaja es que la terminal nos permite hacerlo mucho más rápido. En el tiempo que localizas el cursor, lo posicionas en el icono para abrir el explorador de archivos, y das clic; ya pudiste acceder a casi cualquier directorio del árbol. Más aún, puedes tener cosas muchos mas cómodas con la terminal, que resultan engorrosas en el entorno gráfico.

Por ejemplo, entrar al directorio Escritorio y ver los archivos que tiene y los permisos de los mismos puede hacerse con

```
~ $ ls ~/Documentos/
```

Durante tu paso por el mundo de Linux te encontrarás conviviendo de forma continua con la terminal de Linux, por lo cual es bueno conocer ciertas características de esta que pueden hacer que tal convivencia sea mucho mas amena. A continuación te presentamos algunos “trucos” que son desde muy útiles hasta fundamentales en la terminal.

- Manual de los programas

  Muchos programas vienen con una descripción de su funcionamiento y las opciones con las que cuentan. Para poder visualizar esta descripción puedes usar el comando "man"luego del nombre del programa. Para desplazarte en la descripción de un programa puedes usar las flechas del teclado o la tecla `Enter` y para salir debes presionar la tecla `q`.

  ```
  ~ $ man ls
  ```

- Historial de comandos

  Los comandos que ingresas en la terminal se guardan en un archivo en el directorio home. El archivo se llama .bash_history y puedes revisar su contenido con cualquier editor de texto.

  ```
  ~ $ emacs .bash\_history
  ```

- Historial de comandos

  Otra forma de visualizar el historial, es mediante el comando history, el cual te permite, entre otras cosas, ver una cierta cantidad de comandos.

  ```
  ~ $ history 20
  ```

- Comando anterior

  Si lo que necesitas es reutilizar el último comando que escribiste, puedes presionar ↑ para pasar al comando anterior. Puedes repetir esta acción para recorrer uno por uno los comandos que hayas ingresado a la terminal.

- Búsqueda por coincidencia

  En caso de que recuerdes parte del comando que quieras buscar, puedes presionar `C-r`. Esto despliega una entrada de texto que muestra comandos según encuentre coincidencias con el texto escrito. Esto es especialmente útil cuando buscas un comando largo, no recuerdas como deben usarse las opciones, o requieres una invocación con ciertos argumentos que ya usaste previamente.

  ```
  (reverse-i-search)`ls': ls | grep correos.txt
  ```

  Y para seleccionar el comando, presiona `C-a`

- Auto completar

  Siguiendo con la búsqueda de la comodidad, la terminal puede terminar de escribir un comando o nombre de archivo por ti. Luego de escribir algo en la terminal, si presionas `Tab`, se buscaran nombres de archivos o programas que empiecen con el texto previo. En caso de existir solo una coincidencia, la completará automáticamente, si existen mas, te mostrará las diversas opciones.

- Copiar y pegar

  Como ya mencionamos antes, la combinación de teclas `C-c` termina un proceso, por lo que no podemos usarla para copiar texto. Las teclas que se deben presionar para copiar texto de la terminal son `S-C-c`. De manera similar si se quiere pegar texto en la terminal se debe teclear `S-C-v`.

4.7. *Scripts* para la terminal

Es posible que te encuentres en situaciones en que requieras de la ejecución de la misma secuencia de comandos de forma repetida. Para evitar escribir varias veces lo mismo o en su defecto navegar por tu historial de comandos, es posible juntar toda la secuencia en un archivo de modo que este se ejecute como si fuera un guión de lo que la terminal debe hacer. A estos archivos que contienen comandos para la terminal se les conoce como *scripts* o guiones.

- Como crear un script

  Para hacer un script, basta con escribir en un archivo de texto plano los comandos que quieras ejecutar. Este archivo debe llevar la extensión .sh, además, dado que la idea es que el archivo sea ejecutado, es necesario agregar el permiso de ejecución. También es altamente recomendable que la primer línea sea `!#/bin/bash`. Esto indica que programa interpretará el archivo. No es necesario si el script se ejecuta desde la terminal, pero en caso contrario no será posible.

  La ejecución del script se realiza escribiendo la ruta del archivo que contiene las instrucciones. En caso de que el archivo esté guardado en el directorio de trabajo, es necesario agregar la referencia al directorio actual, es decir $./$, seguido del nombre del archivo.

  ```
  ~ $ ./guion.sh
  ```

- Argumentos

  Aún cuando ya sea posible usar un solo script para englobar todos los comandos que necesitamos, si estos necesitan una variación en los argumentos, el guión que tenemos no funcionará por si solo. Para evitar la creación de un nuevo script para cada posible entrada, podemos manejar argumentos. Para usar un argumento en el script, solo debes poner el símbolo `$` seguido del numero de argumento que quieres usar.

  ```
  #!/bin/bash
  mkdir $1
  ls $2 > $3.txt
  ```

  Básicamente, la instrucción recibe al menos tres argumentos, crea un directorio que lleva por nombre el primer argumento, obtiene los archivos que contenga el directorio cuya ruta es el segundo argumento y guarda el resultado en un archivo que llevará por nombre lo que se reciba como tercer argumento con extensión `.txt`.

- Variables

  Las variables son espacios de memoria que guardan información y tienen un identificador. Para definir una variable se debe escribir el identificador, el signo de igualdad, sin espacios, y la información que guardará la variable. Por ejemplo podemos hacer una variable que guarde una ruta

  ```
  ~ $ ruta=~/Documentos/propedeutico
  ```

  Y para poder acceder al valor de la variable, al igual que con los argumentos, usamos `$`.

```
~ $ ls $ruta
```

En caso de que el valor que se desea guardar contenga espacios, es necesario hacer un ajuste, ya que la terminal considera los espacios como limite de expresiones. Para poder guardar un valor con espacios es necesario ponerlo entre comillas. En el caso de la terminal, tanto las comillas simples como dobles funcionan pero hay una diferencia importante entre ambas que hay que resaltar: Cuando se usan comillas dobles, es posible agregar variables y que están sustituyan su valor por el que guardan. En caso de usar comillas simples, no se hará esta sustitución y dicho valor se mantendrá tal cual se escribió.

Por ejemplo:

```
~ $ msg1='Esta es la ruta $ruta'
~ $ msg2="Esta es la ruta $ruta"
~ $ echo $msg1
~ $ echo $msg2
```

Dará como resultado

```
Esta es la ruta ~/Documentos/propedeutico
Esta es la ruta $ruta
```

Algo interesante que podemos guardar en una variable, es el resultado de la ejecución de un comando. Esto se hace poniendo dicho comando entre paréntesis y todo precedido por el símbolo `$`

```
~ $ lista=$(ls -al $ruta)
```

4.8. Variables de entorno y alias

Para este punto, resulta ya muy cómodo haber pasado de pelearnos con los clics y selección de texto, a solo uno cuantos comandos y finalmente a solo ejecutar un programa. El único inconveniente que puede surgir es la necesidad de saber la ruta del archivo con el script y escribirla cada vez que usamos el programa. Sin embargo, es posible resumir todo en una sola línea a través de los alias.

Un alias es un nombre alternativo que le puedes dar a una cadena de texto. Esta cadena puede ser una ruta, un comando, un valor, etc. Para poder crear un alias usamos el comando `alias` seguido del nombre alternativo que queremos dar, luego el símbolo `=` sin dejar espacios, y finalmente, entre comillas simples `(')` la cadena a la que queremos poner el alias.

```
~ $ alias lal='ls -al'
```

El resultado de lo anterior es que al escribir `lal` en la terminal, se ejecutará el comando `ls` con las opciones que se indicaron.

Los alias funcionan únicamente en la terminal en que se crearon, lo que significa que es necesario crearlos cada vez que se inicia a menos que hagamos una pequeña modificación en el lugar adecuando.

En Linux existen una serie de archivos de configuración que se ejecutan cada vez que se inicia sesión. Uno de esto es el archivo es .bashrc que se encuentra en el directorio home y configura características de la terminal. Es altamente recomendable no borrar o modificar esta clase de archivos a menos que se tenga plena conciencia de lo que se hará. Sin embargo, la modificación necesaria es muy pequeña e inofensiva(siempre y cuando no se borre nada durante el proceso). Si agregas la definición de un alias al final de este archivo, podrás usar el alias en cualquier terminal sin necesidad de volver a definirlo.

Variables de entorno

Como mencionamos antes, para ejecutar un programa, es necesario escribir la ruta en que se encuentra el archivo que contiene dicho programa. Por otro lado, los comandos de la computadora también son programas y por tanto requieren saber donde está guardado el archivo donde se ejecutarán.

Resultaría muy complicado y molesto tener que buscar, aprender y escribir la ruta de cada comando por lo que esto no suena una buena opción. Tampoco es adecuando pensar en alias pues, además de que es necesario crear uno para cada caso, no es posible pasar argumentos.

Para solucionar este problema, en Linux contamos una variable que guarda las rutas donde están los posibles comandos y con esto ya no es necesario que el usuario lidie con eso. Esta variable es llama PATH y es lo que se conoce como una variable de entorno.

Las variables de entorno son aquellas que guardan información importante para el sistema operativo. Para poder verlas vasta con escribir su nombre precedido del símbolo `$`. Por ejemplo `$PATH`.

Otras variables importantes son

Tabla 4.1 Variables de entorno básicas

Nombre	Descripción
`$HOME`	Guarda la ruta del directorio hogar del usuario
`$SHELL`	Guarda el programa que se usa como terminal por omisión

Continúa en la siguiente página

Tabla 4.1 Variables de entorno básicas *Continúa de la página anterior*

Nombre	Descripción
$PATH	Guarda las ruta de los directorios que tienen los programas que son comandos. Si un directorio aparece en esta variable, sus programas pueden ser invocados desde la terminal sin necesidad de especificar una ruta
$PS1	Guarda la cadena que forma el promt principal que se muestra en la terminal
$USER	Guarda el nombre de usuario actual

Emacs | 5

5.1. Introduccion

La elección de un editor de texto para programar es algo muy personal y que cada programador debe tomar por su cuenta. Varios de los autores de este texto utilizamos Emacs, que es un editor que sirve para editar cualquier tipo de texto estructurado, ya sea en un lenguaje de programación o un lenguaje de marcado. En este manual utilizamos Emacs porque nos parece el editor más flexible y más versátil que existe para un programador; lamentablemente el precio que se paga por esto es que la curva de aprendizaje del mismo es más empinada.

En este capítulo daremos una breve introducción a Emacs, dejando claro que es un editor muy poderoso y que se requiere tiempo y práctica para aprenderlo a utilizar a profundidad. Te enseñaremos a utilizar una plétora de comandos para edición con Emacs; no tienes que aprenderlos todos para utilizar el editor, pero es importante los conozcas y practiques cada vez que tengas oportunidad.

5.2. Comandos

Los comandos en Emacs tienen un nombre auto-descriptivo; en muchas ocasiones estos comandos están formados por dos o más palabras separadas por un guión y, como puedes imaginar, ejecutar estos comandos llamándolos por su nombre puede resultar engorroso. Por esta razón, Emacs crea un vínculo entre un comando y una secuencia de teclas y puedes

ejecutarlo ya sea invocándolos por su nombre, por ejemplo `Meta-x` `find-file`, o bien tecleando la secuencia asociada, en este caso `C-x` `C-f`.

Las secuencias asociadas lucen complicadas al principio, pero después de utilizarlas un tiempo te darás cuenta que están ahí por muy buenas razones. De hecho, notarás que los comandos más usuales tienen asociadas secuencias de teclas cortas. Emacs fué diseñado por programadores, para programadores, por lo que la eficiencia lo es todo.

Cada vez que tienes que despegar una de tus manos de la parte central del teclado para buscar con el ratón un comando en los menús, pierdes tiempo valioso. Intenta: compara el tiempo que te toma teclear la secuencia `C-x` `C-f` contra el tiempo que te toma encontrar en el menú el mismo comando `find-file`. Ahora imagina una sesión de edición donde escribirás una decena de cuartillas y ejecutarás varios cientos de comandos.

Entre más utilices Emacs menos tendrás que pensar en las secuencias de los comandos, tus manos se harán cargo. Esto se conoce como memoria dactilar. Una vez que te sientas cómodo con los comandos de Emacs que dominas, regresa a este capítulo e investiga algunos de las secciones que a continuación presentaremos y aprende nuevas secuencias.

Las teclas `Control` y `Meta` (`Alt` y `AltGr`) son usualmente utilizadas en secuencias de teclas. De hecho, ya hemos utilizado algunas de éstas a lo largo de este material, pero en nomenclatura de Emacs no mostramos el nombre de estas teclas completo, sino que las sustituimos así: `Control` es `C` y `Meta` es `M`. Entonces una secuencia así: `C-x`, significa mantener presionada `Control` y sin soltar presionar `x`.

Sucede lo mismo con `M`, aunque aquí tenemos una pequeña variante y es que la tecla de escape, `esc`, puede utilizarse en sustitución de `Meta`. Cuando se ejecuta una secuencia `M-x` utilizando `esc`, sin embargo, no se mantienen presionadas de manera simultánea. Es decir, primero presionamos `Esc`, soltamos, y a continuación `x`.

El guión nos sirve para indicar que esas teclas se presionan juntas, los espacios en la secuencia nos indican que debemos soltar la(s) tecla(s) anterior(es) antes de iniciar con las teclas después del espacio. Por ejemplo, `C-x` `C-f` se teclea así: simultáneamente `Control` y `x`, liberamos ambas y de manera simultánea presionamos `Control` y `f`.

Escribir texto o ejecutar comandos

Puedes estarte preguntando si eventualmente Emacs sirve para editar texto o sólo para ejecutar comandos. Por raro que parezca, Emacs sirve exclusivamente para ejecutar comandos y el más ejecutado es `self-insert-command`, que es ejecutado por cada tecla y cuya única función es imprimir el carácter que representa la tecla en el buffer actual, o sea, la que acabas de oprimir.

Para redactar una carta, escribir tus tareas, programar o hacer una presentación, simplemente tienes que ejecutar Emacs, visitar un archivo (con nuestro ya viejo amigo: `C-x` `C-f`) y comenzar a teclear. Más adelante te decimos como salvar el contenido del buffer (sección 5.12); recuerda teclear `C-x` `C-c` para terminar la ejecución de Emacs.

5.3. Movimiento

Existe una posición privilegiada que es donde se efectúan las operaciones de Emacs, conocida como *el punto*. Generalmente, este punto está marcado con el principio de un cursor, que es un simpático cuadrado que nos indica dónde estás parado con respecto al buffer que estás editando. El punto en realidad se refiere a la posición entre el carácter cubierto por el cursor y el inmediato anterior.

Como es natural, las primeras operaciones que desearás hacer son las de movimiento del punto dentro del buffer, las cuáles se pueden efectuar a muchos niveles, desde carácter por carácter, línea por línea, palabra por palabra, por párrafo o por pantalla.

La mayoría de las operaciones de movimiento se pueden efectuar tanto hacia adelante como hacia atrás. A continuación presentaremos una tabla de movimientos tratando de explicar, cuando no sea claro, las funciones de estos comandos.

Tabla 5.1 Emacs: comandos de movimiento

Unidad	Adelante	Atrás
carácter	`C-f`	`C-b`
palabra	`M-f`	`M-b`
línea	`C-p`	`C-n`
enunciado	`M-a`	`M-e`
párrafo	`M-{`	`M-}`
pantalla	`C-v`	`M-v`

Es posible que necesites desplazarte de manera instantánea a ciertas posiciones privilegiadas del texto. Emacs ofrece los siguientes comandos para realizarlo:

Tabla 5.2 Emacs: movimiento a lugares especiales

Elemento	Inicio	Fin
línea	`C-a`	`C-e`
buffer	`M-<`	`M->`

5.4. Matando y borrando

Hasta el momento sabes cómo agregar texto al buffer y cómo moverte en él, pero ¿qué opciones ofrece Emacs para borrar texto? Ésas te las mostramos aquí, pero antes te presentamos un comando muy útil undo o `C-x` `u`, que deshace lo que hizo el último comando. Si ejecutas repetidamente este comando puedes retroceder tanto como quieras en el proceso de edición del buffer actual.

Los comandos de Emacs para borrar se dividen en dos grandes categorías, los que *matan* y los que *borran*. Como una manifestación del optimismo de Emacs, la muerte no es para siempre, por lo que todo aquello que matas puede *reencarnar*. Para lograr esto, Emacs envía a todos los muertos a una estructura conocida como el anillo de la muerte (`kill-ring`).

Por el contrario, cuando borras algo es para siempre, a menos, claro, que utilices el comando undo. Para evitar que los usuarios de Emacs pasemos malos ratos por eliminar cosas sin quererlo, los únicos comandos para borrar que veremos funcionan sobre caracteres.

En la siguiente tabla te mostramos los comandos para borrar y matar.

Tabla 5.3 Emacs: comandos para matar y borrar

Unidad	Adelante	Atrás
carácter (borrado)	`C-d`	`BackSpace`
palabra	`M-d`	`M-BackSpace`
línea	`C-k`	`C-u` `0` `C-k`
enunciado	`M-k`	`C-x` `Supr`

5.5. Reencarnación de texto

Cuando matas alguna sección, los caracteres que la conforman entran al `kill-ring`. Si eres perspicaz te preguntarás: *¿de qué sirve que se vayan a este lugar si no sé como traerlas de regreso?* Ésta es una pregunta válida: en efecto existe una manera de traer de regreso aquellos caracteres que han muerto en este buffer (o en cualquier otro buffer); a esta operación se le conoce como yank y el comando para realizarla es `C-y`. La ejecución de este comando inserta el último texto muerto en el punto actual.

Como es de esperarse, puedes reencarnar texto que hayas matado con anterioridad. Esto se logra con el comando `M-y`, el cual, si el comando inmediato anterior fue un yank,

reemplaza el texto que éste insertó con el texto inmediato anterior en el `kill-ring`. Esto último se puede repetir sacando sucesivamente lo que esté guardado en el `kill-ring`.

Los elementos del `kill-ring` son el resultado de acomodar adecuadamente bloques de operaciones *matar*; por ejemplo si matas dos líneas seguidas, estas dos líneas forman parte de un mismo bloque en el `kill-ring`.

5.6. Regiones

Las divisiones o unidades que hemos revisado no son todas; durante una sesión de edición puedes requerir tomar partes arbitrarias del texto y operar sobre ellas. A estas divisiones se les conoce como *regiones*.

Una región comprende los caracteres que se encuentren entre una *marca* y el punto. La marca es otra posición especial dentro del buffer y se fija por medio del comando `C-Space`. El punto es la posición que tenga el cursor. Es decir, se coloca la marca en la posición donde se encuentra el punto y mueves el punto ampliando o contrayendo la región, de nuevo entendida como los caracteres entre la posición donde está la marca y la posición actual del punto. Emacs indicará visualmente la extensión de la región, ya sea mostrándola sombreada o cambiando el color del texto.

Existen muchas operaciones que puedes hacer sobre una región. Por el momento sólo mencionaremos dos. La primera es *matar* una región, que se lleva a cabo con el comando `C-w` y la otra, copiar una región al `kill-ring` sin matarla, que se lleva a cabo con el comando `M-w`. Esta última operación sólo mete la región al `kill-ring` sin quitarla de su ubicación actual.

5.7. Rectángulos

En Emacs, además de poder hacer selecciones de caracteres contiguos, también puedes seleccionar áreas con forma de rectángulos, con la esquina superior izquierda en la marca y la esquina inferior derecha en el punto, o viceversa. Los comandos que se usan para lidiar con rectángulos son:

Tabla 5.4 Emacs: comandos para manejar rectángulos

Enlace	Operación
`C-x` `r` `k`	Cortar (kill) un rectángulo

Continúa en la siguiente página

Continúa de la página anterior

Enlace	Operación
C-x r y	Pegar (yank) un rectángulo
C-x r o	Abrir un rectángulo
M-x clear-rectangle	Borrar un rectángulo

5.8. Registros

En Emacs existen una especie de *cajones*, en donde puedes guardar cadenas de texto, rectángulos o posiciones. Estos cajones reciben el nombre de *registros*, y sus principales comandos son:

Tabla 5.5 Emacs: comandos para manejar registros

Enlace	Operación
C-x x	Copia la región en un registro. Te pregunta por el nombre del registro.
C-x r r R	Guarda el rectángulo seleccionado en el registro R, donde R puede ser un carácter o un número.
C-x r s R	Guarda la selección en el registro R.
C-x r Space R	Guarda la posición del cursor en el registro R.
C-x r i R	Inserta el contenido del registro R, si éste es un rectángulo o una cadena.
C-x r j R	Salta al punto guardado en el registro R.
C-x r t	Agrega el texto (que te será pedido en el minibuffer) a todas las líneas en la región rectangular, desplazando el texto hacia la derecha.

5.9. Archivos

Como mencionamos en el capítulo anterior, para abrir un archivo en Emacs utilizamos la secuencia: C-x C-f (find-file). Si te equivocas de archivo, inmediatamente después

de utilizar C-x C-f, puedes utilizar el comando C-x C-v (find-alternate-file).

5.10. Buscar

Emacs incluye varios tipos de búsqueda, el comando search-forward utiliza el *mini-buffer* para solicitarte la cadena que deseas buscar.

Probablemente el comando más útil para buscar en Emacs es isearch-forward (nota la "i" al inicio del comando), que sirve para realizar una búsqueda de manera incremental. Esto significa que Emacs comienza a buscar conforme tú escribes la cadena de búsqueda. Es más usual utilizar el enlace C-s que el comando anterior.

Para buscar hacia atrás (o hacia arriba) de manera incremental utilizamos el comando isearch-backward o C-r.

Una vez que inicias una búsqueda incremental, ya sea hacia adelante o hacia atrás, puedes utilizar nuevamente C-s o C-r para buscar la siguiente presencia hacia adelante o hacia atrás, respectivamente. Cuando llegas al final (o inicio) del buffer, Emacs utiliza el área de eco para avisarte que no existen más presencias de la cadena buscada; si vuelves a insistir en buscar hacia ese mismo lado, Emacs *da la vuelta* y continúa la búsqueda desde el otro extremo del buffer.

5.11. Reemplazar

Habrá ocasiones en las que quieras buscar y cambiar una, algunas o todas las presencias de una cadena de texto por otra. Realizar esta tarea manualmente es tedioso, aun cuando puedes utilizar los comandos de búsqueda para localizar rápidamente el texto a cambiar.

Por este motivo Emacs te ofrece dos comandos principales para reemplazar texto. El primero replace-string, cambia todas las ocurrencias de una cadena por otra (utiliza el *mini-buffer* para solicitar ambas cadenas).

Existe sin embargo, otro comando llamado query-replace-string que se comporta similar, pero antes de realizar cada cambio presenta una serie de opciones para decidir si esa presencia particular se cambia o no y también para decidir reemplazar todas las presencias que se encuentren más allá del punto actual. Este último comando es más popular y por ello está asociado a una secuencia corta de teclas: M-%.

5.12. Guardar

Ya te hemos comentado que mientras editas un documento los cambios existen únicamente dentro de Emacs, en el buffer. Para hacer estos cambios permanentes debes salvar el archivo y esto puedes lograrlo con el comando: `save-buffer` o C-x C-s.

Con el comando `save-some-buffers` (C-x s) Emacs preguntará si deseas guardar cada buffer modificado en tu sesión. Finalmente, el comando `write-file` o C-x C-w te permite salvar el contenido del buffer en un archivo distinto. Emacs te preguntará el nombre de dicho archivo utilizando el *mini-buffer*.

5.13. Ventanas

Una ventana en Emacs es una subdivisión del área de trabajo en donde podemos desplegar un buffer. Con C-x 2, se crea otra ventana en forma horizontal. Para crear una ventana vertical puedes utilizar C-x 3.

Para cambiar entre ventanas debes utilizar C-x o. En cualquiera de estas ventanas puedes realizar todo tipo operaciones, ejecutar comandos de Emacs, etc.

Para cerrar todas las ventanas excepto en la que estás parado teclea C-x 1; para cerrar la ventana en la que estás parado teclea C-x 0.

5.14. Marcos (*frames*)

Además de las ventanas, en en escritorio también podemos crear lo que Emacs llama *frames*, que en terminología estándar de X se llama ventanas. ¿Confundidos? Es muy fácil: estos frames de Emacs lucen como ejecuciones distintas de Emacs, mientras que las ventanas son sub-divisiones de un mismo frame.

5.15. Ayuda en línea

No podríamos decir que Emacs es un editor de texto auto–documentado si no contara con un sistema de ayuda en línea. Si tecleas la secuencia C-h ? ?, Emacs te mostrará una lista con las opciones de ayuda que tiene a tu disposición. Por ejemplo, con la secuencia C-h k, Emacs te pedirá teclear una secuencia de teclas y te dirá todo lo que sabe de esa secuencia, a qué comando está asociada y la documentación del comando.

5.16. Lista de buffers

En Emacs podemos tener una gran cantidad de buffers sobre los cuales estamos trabajando, tantos que es fácil perder de vista o recordar el nombre exacto del buffer que deseamos editar. Para estas situaciones existe el comando `list-buffers` o su enlace `C-x` `C-b`, el cual despliega una lista de los buffers que tenemos abiertos en ese momento. En este buffer se despliega información básica de los buffers, su nombre, su tamaño, y, en caso de estar asociados con un archivo, el nombre del archivo. Otra información muy útil que nos proporciona este comando es si el buffer ha sido modificado desde la última vez que se guardó en disco.

También desde este buffer podemos ir a los diferentes buffers, posicionándonos en el reglón correspondiente y presionando `Enter`.

Control de versiones 6

Cuando se edita un archivo de texto, especialmente si está relacionado con tareas computacionales (código fuente de un programa, *scripts*, archivos de configuración en Unix), esto suele ser una tarea *destructiva*. No nada más se agregan nuevas líneas de texto a un archivo; muchas veces se destruyen (eliminan) o modifican muchas de las líneas que ya existían.

Esto es completamente normal; pero a veces nos gustaría mantener una versión que ya funciona (o con varias partes avanzadas), antes de comprometernos a modificarla.

Es en este tipo de situaciones donde entra el concepto de control de versiones.

6.1. Control de versiones

La necesidad de mantener un control de versiones fue reconocida en el ámbito de la programación básicamente desde el inicio de la misma. Lo que hacen los sistemas de control de versiones es, como su nombre indica, controlar versiones: un grupo de programadores puede decidir que los archivos de un proyecto han llegado a un punto que vale la pena preservar (o asignar versiones), así se lo hacen saber al sistema de control de versiones y a partir de ese momento ese estado (o *versión*) del proyecto se preserva para toda la posteridad. Se pueden modificar los archivos del proyecto (o agregar e incluso elimiar algunos), pero si así se desea se puede recuperar (de preferencia *fácilmente*) la versión anterior.

Esto además es un proceso acumulativo; si después de haber preservado una versión se

decide preservar una nueva, ambas estarán disponibles siempre[1], así como cualquier otra versión futura que se quiera preservar. Un sistema de control de versiones entonces mantiene un historial de *todas* las versiones preservadas de un proyecto, con la única limitación siendo el espacio en disco duro disponible en la computadora: el kernel de Linux utiliza 7 gigabytes, a pesar de que su sistema de control de versiones utiliza varias optimizaciones para ahorrar espacio en disco. Sin embargo es un ejemplo extremo: el kernel de Linux consiste de aproximadamente 25 millones de líneas de código desarrolladas a lo largo de 26 años por miles de autores.

El control de versiones es útil en sí mismo; sin embargo se vuelve *particularmente* útil al trabajar en equipo: cada integrante del equipo puede tener una copia del proyecto y preservar los cambios que le parezcan convenientes. Si hay conflictos o se introducen errores, es (relativamente) sencillo regresar a una versión que se sabe que funciona.

6.1.1. Sistemas centralizados

En 1972 Marc Rochkind creó en Bell Labs el Source Code Control System (SCCS), pero no estuvo disponible al "público" hasta 1977 (entre comillas porque el uso de las computadoras era muy limitado en ese entonces para el público en general). En 1982 SCCS fue en general reemplazado por el Revision Control System (RCS) escrito por Walter Tichy.

En ese entonces tanto SCCS como RCS funcionaban a nivel de archivo, no de proyecto; eso quiere decir que uno mantenía versionados archivos individuales, no conjuntos de los mismos. No era extraño en esa época que muchos proyectos consistieran de un puñado de archivos, si no es que uno solo.

En 1986 Dick Grune comenzó a trabajar en una serie de *scripts* de shell que eventualmente se convertirían en el Concurrent Versions System (CVS), liberado oficialmente en 1990. CVS ya podía trabajar con directorios (rudimentariamente), lo que permitió mantener control de versiones de proyectos mucho más complejos. A un directorio o conjunto de directorios que estén bajo control de versiones se le suele denominar un *repositorio*.

El proyecto GNU que mencionamos en el capítulo 1 fue anunciado en 1983 por Richard Stallman; el control de versiones es *fundamental* para el desarrollo del Software Libre, por razones tanto técnicas como sociales (el control de versiones permite, entre otras cosas, determinar de manera clara quién es el autor de qué modificaciones). CVS se convirtió en miembro del proyecto GNU.

CVS es un sistema de control de versiones *centralizado*; esto quiere decir que existe una copia "central", que es realmente la que está bajo el contro de versiones, y múltiples usuarios pueden bajar una copia de la última versión del proyecto. Cuando un usuario a realizado cambios a su copia, puede enviar los mismos al repositorio central para que el resto a su vez los tengan disponibles; sin embargo, el historial de versiones anteriores únicamente se encuentra en el repositorio central.

[1] Siempre y cuando no se elimine el proyecto mismo, obviamente.

CVS fue el sistema de control de versiones utilizado por básicamente todos los proyectos de software libre desarrollados en y para Linux en la década de los noventas, a pesar de todas las limitaciones que tiene. Sin embargo, una excepción muy importante fue el mismo kernel de Linux: a pesar de que siempre estuvo disponible un repositorio CVS para el mismo, no es lo que utilizaba Linus Torvalds, el creador y dictador benevolente de Linux que es el que tiene la decisión final de qué se incluye o no en el kernel.

Inicialmente Linus utilizaba su correo electrónico para recibir *parches* que después aplicaba manualmente. Un parche o archivo diferencial (*diff*) es un archivo de texto con un formato especial que permite ver las diferencias entre dos archivos de texto. En lo general en un sistema de control de versiones, las distintas versiones de un archivo pueden describirse con una secuencia de parches. Se les denomina parches porque justamente "parchan" un proyecto, con la idea de que un parche repare o alivie un error en el código.

El aplicar parches manualmente es, por supuesto, increíblemente ineficiente, pero Linus se negaba a utilizar sistemas de control de versiones porque (él aducía) ninguna hacía lo que él necesitaba. Esto continuó durante todo el final del siglo XX.

6.1.2. Sistemas distribuidos

En 1998 la compañía BitMover comenzó a trabajar en un sistema de control de versiones particularmente orientado a las necesidades de Linus y, por extensión, Linux. En el año 2000 BitKeeper fue liberado y dos años después comenzó a ser utilizado por Linus y, de manera gradual y a regañadientes, por el resto de la comunidad de desarrolladores del kernel de Linux.

La razón de que muchos desarrolladores se resistieran a usar BitKeeper es que era un sistema de control versiones que no es Software Libre; la compañía BitMover lo vendía, aunque a los desarrolladores del kernel de Linux se les proporcionaba automáticamente una licencia gratuita.

BitKeeper no es un sistema de control de versiones centralizado, como CVS o Subversion (que es una mejora incremental sobre CVS); es un sistema de control de versiones *distribuido*. Esto quiere decir que no existe un repositorio central; cada copia del mismo es un repositorio en sí misma, con toda la historia del proyecto incluida en ella.

La resistencia a BitKeeper y el hecho de que no fuera Software Libre resultó (junto con otras razones) en la creación de varios sistemas de control de versiones distribuidos, probablemente los más famosos siendo Mercurial y Bazaar.

En 2005 BitMover decidió retirar la versión gratuita de BitKeeper que proporcionaba a los desarrolladores del kernel de Linux, supuestamente porque algunos de ellos habían tratado de aplicar ingeniería reversa al protocolo de comunicación del mismo, así que Linus decidió tomarse (literalmente) unas semanas para escribir su propio sistema de control de versiones distribuido, porque ninguno en existencia hacía lo que BitKeeper.

Después de unas cuantas semanas de desarrollo, Linus liberó Git, que poco más de una

década después es básicamente lo que utilizan todos los proyectos de Software Libre en existencia (con algunas excepciones).

6.2. Git

Git es un sistema de control de versiones distribuido con dos características muy particulares: está pensado para hacer bifurcaciones (que explicaremos más adelante) de manera muy rápida y además utiliza criptografía para toda la historia de un repositorio.

Lamentablemente la facilidad de uso nunca fue una prioridad para Linus ni para ninguno de los siguientes desarrolladores de Git, así que el programa tiene la en general muy merecida fama de ser algo difícil de aprender a usar, especialmente sus características más oscuras. Sin embargo para cosas sencillas sólo es necesario aprender un puñado de comandos que serán los que cubriremos en esta sección.

Git es de hecho una colección de programas, los cuales en general serán invocados a través del programa principal `git`. Los comandos de Git generalmente son de la siguiente forma:

```
~ $ git comando [opciones]
```

Lo que hace el programa `git` es tomar el comando que recibe como parámetro, `comando` en el ejemplo de arriba, y ejecutar `git-comando`. En una instalación normal de Git, esto resulta en cientos de programas para los comandos de Git.

Para crear un repositorio podemos comenzar con un directorio vacío o que ya contenga archivos, es lo mismo. Nosotros crearemos un nuevo repositorio desde cero en nuestros ejemplos:

```
~ $ mkdir ejemplo
~ $ cd ejemplo
~/ejemplo $ git init .
```

Recordemos que usar "." sirve para referirnos al directorio actual, así que el comando que acabamos de usar se traduce en pedir a git que cree un repositorio asociado al directorio que pasamos como parámetro (en este caso .); si el directorio no existe, se crea uno nuevo vacío con el nombre que hemos pasado, por lo tanto una forma de inicializar un repositorio desde cero es:

```
~ $ git init ejemplo
~ $ cd ejemplo
```

El comando `init` de Git únicamente crea el directorio `.git` con una configuración por omisión. El repositorio está vacío; si listamos los archivos en el directorio no aparecerá nada excepto el directorio `.git`:

```
~/ejemplo $ ls -la
total 56
drwxr-xr-x   3 user user  4096 Jul 17 19:57 .
drwx------ 119 user user 45056 Jul 17 19:57 ..
drwxr-xr-x   6 user user  4096 Jul 17 19:57 .git
```

El estado del repositorio también es vacío, dado que no hemos hecho nada todavía; podemos revisar el estado de un repositorio con el comando `status`:

```
~/ejemplo $ git status .
On branch master
Initial commit
nothing to commit (create/copy files and use "git add" to track)
```

Más adelante veremos qué quiere decir exactamente "branch master". Vamos a agregar un archivo a nuestro repositorio; en Emacs crea el archivo `capitan.txt` dentro del directorio `ejemplo`, escribe lo siguiente en el mismo y guárdalo:

```
Va de cuento: nos regía
Un capitán que venía
Mal herido, en el afán
De su primera agonía.
```

Ahora revisa el estado de tu repositorio; algo del estilo debería aparecerte:

```
~/ejemplo $ git status .
On branch master
Initial commit
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        capitan.txt
nothing added to commit but untracked files present
(use "git add" to track)
```

Git nos dice que hay archivos fuera del control de versiones ("untracked files") y que podemos agregarlos usando el comando `add`, así que hagamos eso:

```
~/ejemplo $ git add capitan.txt
~/ejemplo $ git status .
On branch master
Initial commit
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   capitan.txt
```

Si en este paso hubiéramos creado más de un archivo en nuestro repositorio y quisiéramos agregar **todos** al control de versiones, la forma de agregar todos los cambios que hemos hecho es pasar el parámetro `-A` (contracción de `-a`) en vez del nombre de un archivo, el comando quedaría de la siguiente manera:

```
~/ejemplo $ git add -A
```

Existen comandos similares para agregar al control de versiones los cambios en más de un archivo a la vez:

`git add .` Agrega las modificaciones de archivos existentes y los creados después de la última vez que se cargaron cambios. **No carga las remociones de archivos**.

`git add -u` Agrega las modificaciones de archivos existentes y los eliminados después de la última vez que se cargaron cambios. **No carga los archivo que se crearon recientemente**.

Ahora Git nos dice que hay un archivo nuevo, pero que no hemos realizado el cambio. Esto es importante; Git no realizará ningún cambio (no quedará preservado en el historial de versiones) hasta que realicemos (o cometamos, "commit") el mismo. También nos dice el comando para eliminar `capitan.txt` del control de versiones con el comando `rm`.

Vamos a realizar el cambio para que quede en el historial de versiones:

```
~/ejemplo $ git commit capitan.txt -m "Inicio del verso"
[master (root-commit) ae1cf70] Inicio del verso
 1 file changed, 4 insertions(+)
 create mode 100644 capitan.txt
```

El parámetro `-m` nos permite agregar el mensaje que queremos asociar al realizar nuestros cambios; si no la especificamos, Git abrirá un editor para que escribamos el mensaje. No podemos realizar un cambio si no escribimos un mensaje asociado al mismo.

Si revisamos el estado del repositorio, Git nos informará que no existe ninguna modificación:

```
~/ejemplo $ git status .
On branch master
nothing to commit, working tree clean
```

A partir del momento en que realizamos el cambio, esta versión del archivo `capitan.txt` quedará preservada siempre que el repositorio exista; no importa cómo modifiquemos el archivo o incluso que lo eliminemos, siempre podremos recuperar esta versión.

Para ejemplificar esto, vamos a terminar el verso; en Emacs agrega las dos últimas líneas del verso para completarlo y guárdalo:

```
Va de cuento: nos regía
Un capitán que venía
Mal herido, en el afán
De su primera agonía.
¡Señores, qué capitán
el capitán de aquel día!
```

Si ahora verificamos el estado del repositorio, nos dirá que nuestro archivo ha sido modificado.

```
~/ejemplo $ git status .
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in
   working directory)
        modified:   capitan.txt
no changes added to commit (use "git add" and/or "git commit -a")
```

Este ejemplo con un único archivo es muy sencillo, pero si tuviéramos muchos archivos que hemos modificado a lo largo de varios días (lo cual es muy común al programar), es útil ver qué hemos modificado en el repositorio. Para esto podemos usar el comando `diff`, que nos permite ver la diferencia entre la última versión realizada y nuestras nuevas modificaciones:

```
~/ejemplo $ git diff .
commit 8b0d526a13abc49c5b3fcf4a9a1571dad3e3e69c (HEAD -> master)
Author: User <user@unam.mx>
Date:   Thu Jul 17 20:25:58 2017 -0500

    Verso completado

diff --git a/capitan.txt b/capitan.txt
index 8d51b69..3e3e69c 100644
--- a/capitan.txt
+++ b/capitan.txt
@@ -2,3 +2,5 @@ Va de cuento: nos regía
 Un capitán que venía
 Mal herido, en el afán
 De su primera agonía.
+¡Señores, qué capitán
+el capitán de aquel día!
```

En la terminal Git usará un programa paginador (generalmente `less`) para mostrar la diferencia. Podemos ver en la misma que el cambio que hicimos fue agregar las dos últimas líneas.

Para agregar este cambio al repositorio tenemos que usar de nuevo los comandos `add` y `commit`, aunque en este caso podríamos ahorrarnos el primero:

```
~/ejemplo $ git add capitan.txt
~/ejemplo $ git commit capitan.txt -m "Verso completado"
[master 8b0d526] Verso completado
 1 file changed, 2 insertions(+)
```

Git nos informa que un archivo fue modificado y que tuvo dos inserciones. Podemos ver el historial de modificaciones utilizando el comando `log`:

```
~/ejemplo $ git log .
commit 8b0d526a13abc49c5b3fcf4a9a1571dad3e3e69c (HEAD -> master)
Author: User <user@unam.mx>
Date:   Thu Jul 17 20:25:58 2017 -0500

    Verso completado

commit ae1cf70ddbd1bdd7b400267c046c2fe1571d3afb
Author: User <user@unam.mx>
Date:   Thu Jul 17 20:18:00 2017 -0500

    Inicio del verso
```

El número hexadecimal que aparece a la derecha de `commit` es el identificador de cada versión; podemos recuperar cualquier versión pidiéndole a git que revise su identificador con el comando `checkout`:

```
~/ejemplo $ git checkout ae1cf70ddbd1bdd7b400267c046c2fe1571d3afb
Note: checking out ae1cf70ddbd1bdd7b400267c046c2fe1571d3afb

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

  git checkout -b <new-branch-name>

HEAD is now at ae1cf70... Inicio del verso
~/ejemplo $ cat capitan.txt
Va de cuento: nos regía
```

```
Un capitán que venía
Mal herido, en el afán
De su primera agonía.
```

Cuando revisamos una versión Git nos da bastante información de que ya no estamos a la cabeza de nuestra rama; explicaremos esto más adelante. Para regresar a donde estábamos, con el archivo `capitan.txt` completo, sólo debemos revisar "master":

```
~/ejemplo $ git checkout master
Previous HEAD position was ae1cf70... Inicio del verso
Switched to branch 'master'
```

Por último y antes de explicar las bifurcaciones, vamos a ver cómo eliminar un archivo del repositorio; para esto sólo hay que usar el comando `rm`. Podemos hacer esto sin ningún temor; borramos el archivo del estado *actual* del repositorio, pero el mismo siempre va a estar en el historial de versiones y siempre podremos recuperarlo.

```
~/ejemplo $ git rm capitan.txt
rm 'capitan.txt'
~/ejemplo $ git commit capitan.txt -m 'Eliminado'
[master 2a10a91] Eliminado
 1 file changed, 6 deletions(-)
 delete mode 100644 capitan.txt
```

6.2.1. Bifurcaciones

Como hemos discutido el funcionamiento de Git, podría parecer que las versiones que se van guardando forman una progresión lineal: está el estado inicial de un repositorio, se le hacen cambios y se realizan, luego se le hacen más cambios que también se realizan, etc.

En los hechos es más complejo que esto porque Git está diseñado para trabajar con múltiples personas modificando un mismo repositorio en varias computadoras; muchos usuarios de Git lo que hacen es *bifurcar* un repositorio. Al bifurcar un repositorios, todos los cambios ocurren en una rama distinta a la principal (la que Git llama por omisión "master"), y uno puede realizar modificaciones ahí de forma independiente a la misma.

Para crear una rama uno utiliza el comando `branch`:

```
~/ejemplo $ git branch pruebas
```

Esto únicamente crea la rama, en este momento seguimos trabajando en la rama "master". Para cambiarnos a la rama que creamos, se utiliza el comando de revisión, `checkout`:

```
~/ejemplo $ git checkout pruebas
Switched to branch 'pruebas'
```

La creación de una rama y el cambio a ésta puede resumirse alternativamente a un sólo paso con la opción `-b` agregada al comando `checkout`:

```
~/ejemplo $ git checkout -b pruebas2
Switched to branch 'pruebas2'
```

Después de realizar modificaciones a nuestra rama `pruebas`, podemos ver las diferencias con la rama principal:

```
~/ejemplo $ git diff master
```

Y si estamos contentos con los cambios, podemos regresar a la rama principal con `checkout` y combinarle los cambios de la rama con el comando `merge`:

```
~/ejemplo $ git checkout master
Switched to branch 'master'
~/ejemplo $ git merge pruebas
Updating 2a10a91..b6a5eb4
Fast-forward
 capitan.txt | 2 ++
 1 file changed, 2 insertions(+)
 create mode 100644 capitan.txt
```

Hemos mezclado el trabajo de la rama `pruebas` con el de `master`, sin embargo la rama `pruebas2` ya no esta al día con el trabajo que hasta el momento nos convence como definitivo en `master` porque fue creada antes del comando `commit` más reciente. Para mezclar los cambios de éstas dos ramas en `pruebas2` y continuar trabajando usamos el comando `rebase`. En el caso de esta comando y `merge`, si hemos editado las mismas líneas con información diferente, tendremos un "conflicto" como mostramos a continuación:

```
~/ejemplo $ git checkout pruebas2
Switched to branch 'pruebas2'

~/ejemplo $ git rebase master
First, rewinding head to replay your work on top of it...
Applying: Cambio al final
Using index info to reconstruct a base tree...
M       capitan.txt
Falling back to patching base and 3-way merge...
Auto-merging capitan.txt
CONFLICT (content): Merge conflict in capitan.txt
Failed to merge in the changes.
Patch failed at 0001 Cambio al final
The copy of the patch that failed is found in:
```

```
   ~/ejemplo/.git/rebase-apply/patch

When you have resolved this problem, run "git rebase --continue".
If you prefer to skip this patch, run "git rebase --skip" instead.
To check out the original branch and stop rebasing, run
"git rebase --abort".
```

En este momento, como hemos editado las mismas líneas en el último `commit` de ambas ramas, tenemos un conflicto y nuestro archivo "capitan.txt" se ve así:

```
Un capitán que venía
Mal herido, en el afán
De su primera agonía.
<<<<<<< HEAD
¡Señores, qué capitán
el capitán de aquel día!
=======
No era el hombre más honesto
ni el más piadoso,
pero era un hombre valiente.
>>>>>>> Cambio al final
```

El conflicto se ve señalado con "`<<<<<<< HEAD`" y "`>>>>>>> Cambio al final`". Las líneas hasta antes de la separación "`=======`" corresponde al contenido del archivo "capitan.txt" en la rama master, después de esta vemos el contenido del último `commit` en la rama pruebas2. Para solucionarlo basta con elegir sobre lo que queremos seguir editando y modificar el archivo de acuerdo a esto. En este ejemplo queremos ambos contenidos, así que simplemente eliminaremos las señalizaciones de conflicto, agregaremos los cambios y continuaremos el rebase:

```
~/ejemplo$ git add capitan.txt
~/ejemplo$ git rebase --continue
Applying: Cambio al final
Applying: Cambio al final
```

Y éste será el resultado obtenido en el archivo "capitan.txt" en la rama `pruebas2`:

```
Un capitán que venía
Mal herido, en el afán
De su primera agonía.
¡Señores, qué capitán
el capitán de aquel día!

No era el hombre más honesto
ni el más piadoso,
pero era un hombre valiente.
```

El bifurcar un repositorio nos permite trabajar en un espacio "limpio", sin preocuparnos de lo que otras personas pudieran o no hacer, y sólo al final que ya estemos seguros de los cambios que queremos realizar, combinarlos con la rama principal. Es la manera recomendada para trabajar con Git, ya que está optimizado para poder bifurcar repositorios grandes muy rápidamente.

6.2.2. Deshacer cambios

Hasta el momento hemos hecho énfasis en la importancia de realizar un commit por cada cambio significativo realizado, para poder volver a versiones anteriores de nuestro trabajo en caso de ser necesario. Cuando queremos volver al estado que hay en algún commit previo usamos el comando `reset` con alguna de las opciones que tiene, aquí explicaremos las más usuales que son "`- -soft`", "`- -hard`" y "`- -mixed`".

Primero es importante aclarar algunos términos de nuestro espacio de trabajo con git:

- **HEAD**: Es el último commit cargado en el historial de nuestra rama.
- **Index**: Es el área de ensayo (normalmente llamada *staging area*), el conjunto de archivos que se convertirán en el nuevo commit, es decir, los que vamos agregando con `git add` antes de hacer commit.
- **Copia de trabajo**: Son los archivos sobre los que estás trabajando antes de agregarlos a *Index*.

Para regresar a un commit en específico necesitamos visualizar el regitro de commits que hemos hecho con su información corespondiente usamos el comando `log` como ya lo hemos visto en la sección 6.2.

Aquí una breve explicación de las opciones principales al hacer *reset*:

- **- -hard** Esta opción deshará **todos** los cambios que no hayan sido agregados a un commit, comunmente se utiliza de las siguiente formas:

  ```
  $ git reset -hard HEAD
  ```

  Borrará todos los cambios que hayas hecho desde el último commit y te posicionará en la "cabeza" del historial del repositorio, es decir, el último commit.

  Otra forma de utilizar esta opción es indicando a qué commit quieres regresar:

  ```
  $ git reset -hard f876b67
  ```

  Este comando descartará **todos** los commits y cambios que hayas realizado después del indicado, lo cual dejará a HEAD, *Index* y la *copia de trabajo* en exactamente el mismo estado.

- **- -soft** Pone a HEAD en el commit que indiques sin eliminar cambios posteriores:

  ```
  $ git reset -soft f876b67
  ```

  Este comando colocará a HEAD en el commit indicado y todos los cambios realizados después de esto (incluyendo los siguientes commits) quedarán en Index, así que si después de ejecutar este comando realizas un commit, éste tendrá todo lo que hayas realizado después de `f878b67`. En otras palabras, *Index* y la *copia de trabajo* quedan en el mismo estado.

- **- -mixed** Esta es la opción por omisión de `reset` cuando no hay alguna especificada. Coloca a HEAD en el commit indicado mientras que reinicia a *Index* para quedar en el mismo estado y todos los cambios posteriores pasan a la copia de trabajo. Un ejemplo de uso quedaría análogo a los de las otras opciones:

  ```
  $ git reset -mixed f876b67
  ```

Cuando nos queremos referir a commits específicos que conocemos "hace cuántos commits hicimos", podemos referirnos a ellos con "HEAD˜n", donde n representa cuántos commits atrás quieremos ir, si queremos referirnos al commit anterior a HEAD lo llamamos "HEAD˜1", al que hicimos dos antes como"HEAD˜2", y así sucesivamente. Por ejemplo, si quisiéramos hacer un reset por omisión (lo mismo que si usara `- -mixed`) al 4 commits antes de HEAD, usaríamos: `$ git reset HEAD˜4`
Con lo que HEAD e *Index* quedarían en el commit que antes estaba a 4 lugares de HEAD y *la copia de trabajo* contendrá los cambios de los 4 commits que estaban delante.

6.2.3. Usos avanzados

Git es una herramienta muy versátil y poderosa y, consecuentemente, a veces muy compleja. Podría escribirse un libro entero al respecto; y de hecho ya lo hicieron y está disponible en línea[2]. Les recomendamos que lo lean para ver una explicación más detallada y profunda acerca de Git.

6.3. GitHub

Todos los ejemplos cubiertos en este capítulo utilizan Git localmente, en un repositorio en la misma computadora en la que se trabaja. Este es un uso válido de Git, pero la herramienta es mucho más poderosa y útil cuando se utiliza con múltiples repositorios.

Por suerte existen muchos servicios en línea que proveen alojamiento para repositorios de Git (y en algunos casos otros sistemas de control de versiones). Les recomendamos que

[2] `http://www.tex.ac.uk/tex-archive/help/Catalogue/catalogue.html`

revisen GitHub[3] y GitLab[4], que les permiten tener repositorios privados, al menos para estudiantes en el caso de GitHub.

Esto es importante por dos razones: en primer lugar, si quieren tener documentos personales, deben entender que si están en un repositorio público cualquier persona podría verlos. En segundo lugar, porque si quieren mantener prácticas y proyectos escolares en repositorios en línea (que se los recomendamos), más vale que dichos repositorios sean privados: tenerlos en un repositorio público es equivalente a "pasarle la tarea" a sus compañeros.

Del otro lado de la moneda, mencionamos que les conviene tener proyectos de programación de Software Libre en un repositorio *público*; varias compañías de software suelen reclutar programadores únicamente de ver sus perfiles en portales como GitHub y GitLab.

Para mantener respaldos de nuestros repositorios locales de manera remota es necesario contar con una cuenta en GitHub o GitLab. En el caso de Github puedes hacerlo en el sitio github.com y una vez confirmada tu cuenta te recomendamos asociar tu cuenta a GitHub Education para tener acceso a repositorios privados (recuerda que no quieres "pasarle la tarea" a tus compañeros), esto lo puedes hacer en el sitio education.github.com/students

Una vez que tengas una cuenta, te recomendamos configurar a Git desde la consola con el usuario y correo que elegiste para tu cuenta en GitHub, de esta forma todos los cambios que hagas en los repositorios locales te tendrán como autor automáticamente. Para hacerlo existen los siguientes comandos sustituyendo el nombre de usuario y el correo con los tuyos:

```
~/ $ git config --global user.name "usuario"
~/ $ git config --global user.email "usuario@ciencias.unam.mx"
```

Opcionalmente puedes asociar una clave de autenticación remota entre tu computadora personal y tu cuenta de GitHub, esto te servirá para que al subir cambios a un repositorio remoto desde tu computadora no sea necesario introducir tus credenciales (usuario y contraseña).

Primero generamos una llave SSH, pulsamos Enter para generarla con valores predeterminados: (recuerda sustituir tu correo y usuario cuando sea necesario)

```
~/ $ ssh-keygen -t rsa -b 4096 -C "usuario@ciencias.unam.mx"
Generating public/private rsa key pair.
Enter file in which to save the key (/home/user/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/user/.ssh/id_rsa.
Your public key has been saved in /home/user/.ssh/id_rsa.pub.
The key fingerprint is:
```

[3] https://github.com
[4] https://gitlab.com

```
b0:8d:d6:4d:87:5e:ad:75:d9:18:27:d6:83:d8:c9:8d
usuario@ciencias.unam.mx
The key's randomart image is:
+--[ RSA 4096]----+
|                 |
|         + B . . |
|      . . E * o .|
|       * + + o *o|
|      + S . . +.o|
|     .       . . |
|                 |
|                 |
|                 |
+-----------------+
```

Inicializamos el agente SSH de nuestra computadora y agregamos la clave que acabamos de generar:

```
~ $ eval $(ssh-agent -s)
Agent pid 10875

~ $ ssh-add ~/.ssh/id_rsa
Identity added: /home/user/.ssh/id_rsa (/home/user/.ssh/id_rsa)
```

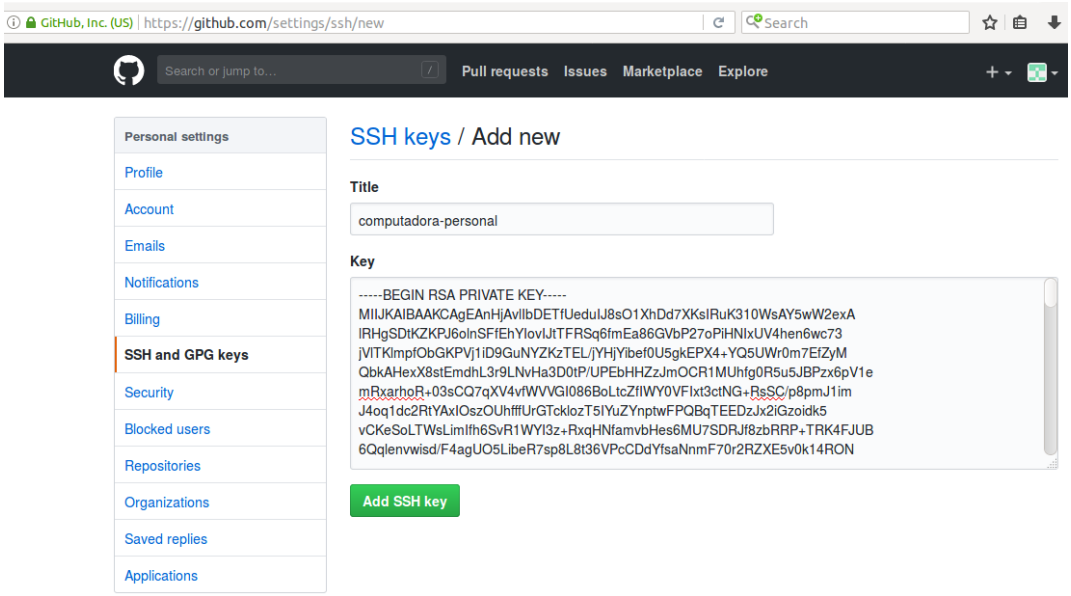
Verificamos que haya contenido y copiamos la llave en el portapapeles:

```
~ $ cat < ~/.ssh/id_rsa
-----BEGIN RSA PRIVATE KEY-----
MIIJKAIBAAKCAgEAnHjAvlIbDETfUeduIJ8sO1XhDd7XKsIRuK310WsAY5wW2exA
lRHgSDtKZKPJ6olnSFfEhYIovIJtTFRSq6fmEa86GVbP27oPiHNIxUV4hen6wc73
.
.
.
JdNaTHcuY6urSRhoqwFKMdby33eqte35gc5bP7iu/BlJe8UOMGEbPRnjrQUDwiF3
oRdQbm+mlUVlj8y7n399ETvQDEGuVo3pj0+chXheHl9s/X9UfE9nMdpBDqM01mm4
f5NgdBBzUe3B4e4oAk2dJ+pGxhD+yMtMpn+GdXDMQQf+tXD4sK+7UHkvTPY=
-----END RSA PRIVATE KEY-----
```

Vamos a la configuración de nuestra cuenta en GitHub y elegimos la opción `SSH and GPG keys` (github.com/settings/keys), seleccionamos `New SSH Key` y agregamos la llave que generamos en nuestra computadora:

Comprobamos la conexión con el comando `ssh -T git@github.com` y hemos terminado.

Figura 6.1 Asignamos un título mnemónico a la llave y la agregamos.



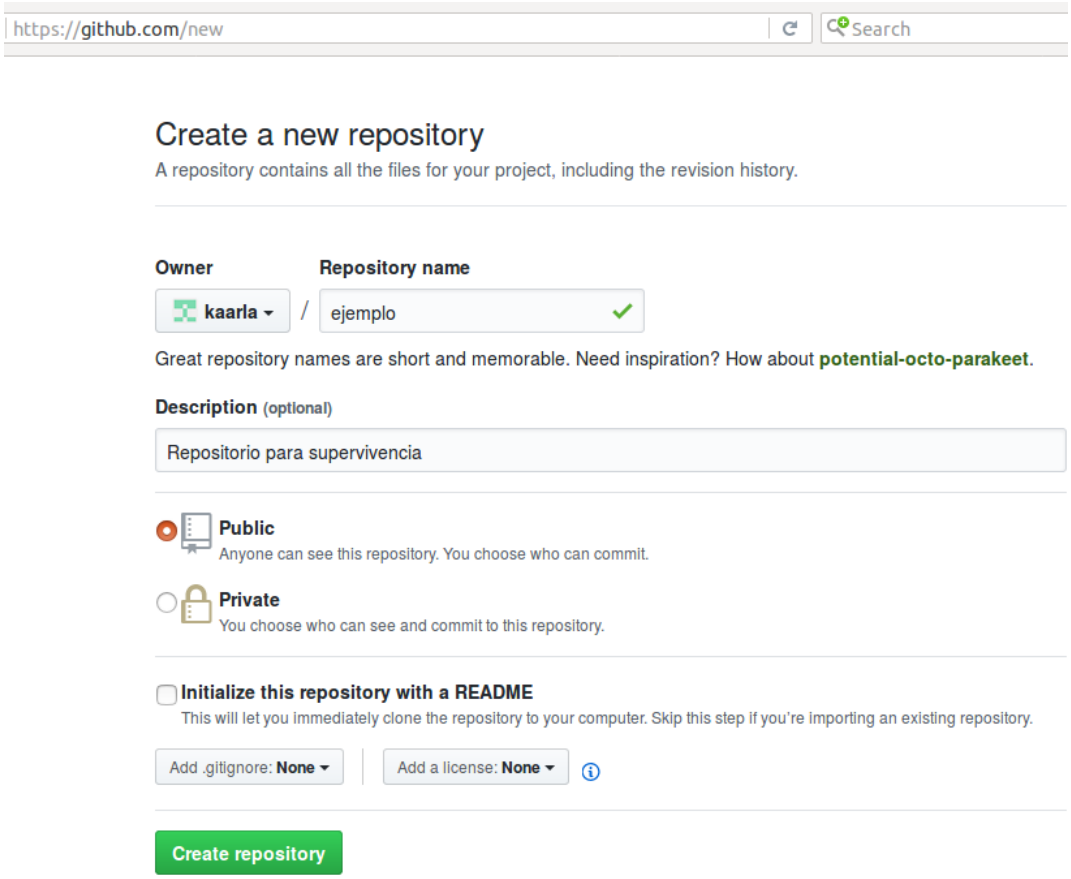
A partir de ahora la autenticación con github de nuestro usuario desde nuestra computadora será automática. Lo siguiente es ligar nuestro repositorio local `ejemplo` con uno remoto en GitHub. Tendremos que crear un repositorio desde nuestra cuenta de GitHub; lo primero es ir a GitHub, iniciar sesion y dar clic en el botón “New”, posteriormente asignar el mismo nombre de nuestro repositorio local al remoto, agregar una descripcion opcional y dar clic en crear:

Nuestro repositorio remoto por ahora esta vacío y nos da diferentes opciones para empezar a trabajar, nosotros eligiremos la segunda enlistada en la pantalla que corresponde a ligarlo a un repositorio local existente por medio de SSH, en caso de no haber configurado tu llave selecciona HTTP y la única variación será que tendrás que introducir tus credenciales (Figura 6.3).

Volvemos al repositorio sobre el que trabajamos en la sección anterior y usamos el comando `remote add` con la convención “origin” que suele usarse para asignar el repositorio remoto. En seguida usaremos el comando `push` que sirve para “empujar” los cambios de la rama actual a su compia remota:

```
~/ejemplo $ git remote add origin git@github.com:usuario/ejemplo.git
~/ejemplo $ git push -u origin master
Counting objects: 7, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (6/6), done.
Writing objects: 100% (7/7), 611 bytes | 0 bytes/s, done.
Total 7 (delta 1), reused 0 (delta 0)
remote: Resolving deltas: 100% (1/1), done.
```

Figura 6.2 Personalizamos la creación del nuevo repositorio.




```
To https://github.com/usuario/ejemplo.git
 * [new branch]      master -> master
Branch master set up to track remote branch master from origin.
```

Actualizamos la página de configuración de nuestro repositorio y tendremos una copia del local en su estado actual, así se ve en master el documento que hemos trabajado localmente (Figura 6.4).

Si estuviéramos trabajando en equipo y alguien mas actualizara la rama en la que queremos trabajar, master por ejemplo, usaríamos el comando `git pull origin master` desde nuestro repositorio local y automaticamente bajariamos el estado de la rama remota; sólo en caso de haber editado las mismas líneas sería necesario solucionar los confictos como vimos previamente.

Figura 6.3 Opciones para inicializar un repositorio.

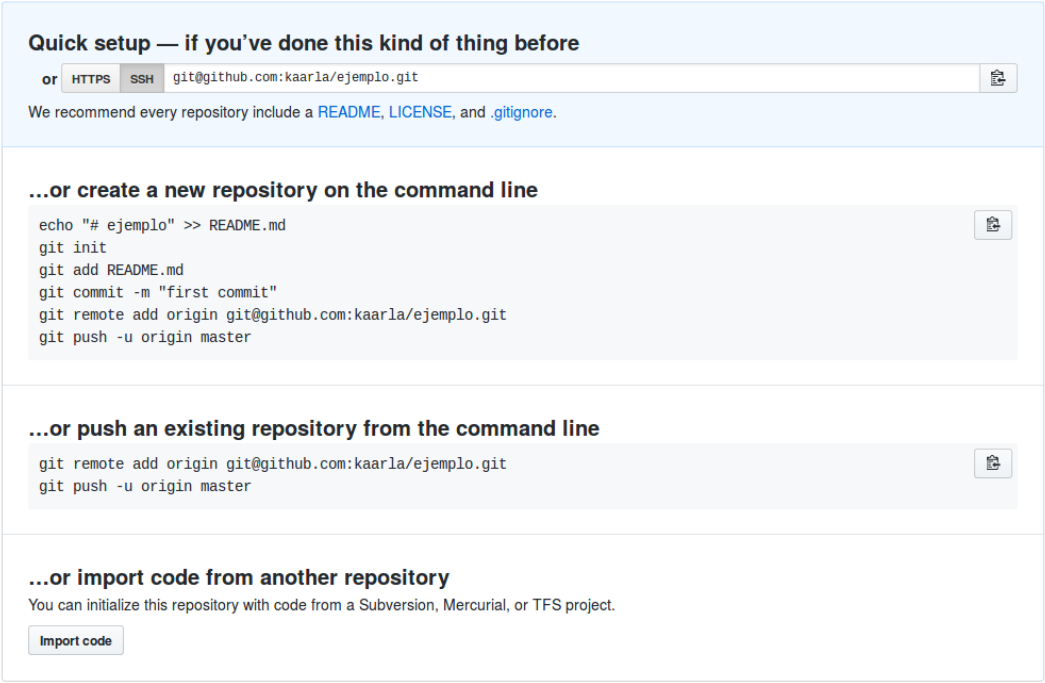
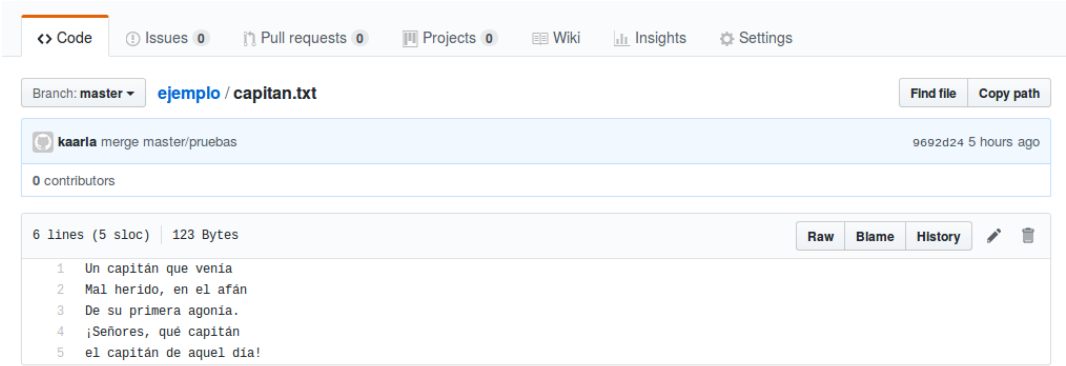


Figura 6.4 Vista final del contenido del archivo en línea.



LaTeX | 7

7.1. Introducción

Una de las aplicaciones más importantes de la computadora hoy en día es la preparación de texto, sea éste para notas, una carta, un libro, reporte, artículo, tarea, etc.

Un procesador de texto es un paquete al que le tecleas texto, con algunas "observaciones" y el paquete procede a formatear el texto. Entre las operaciones que deseamos haga con el texto están las siguientes:

1. Armar los renglones de tal manera que quede el texto bien repartido.
2. Armar páginas de tal manera que se vayan numerando y que tengan todas el mismo tamaño.
3. Facilitar el cambio de tipo de letra y proveer distintos tipos de letras como itálicas, negritas, etc..
4. Facilitar la edición de tablas de distintos tipos.
5. Facilitar la construcción de ecuaciones matemáticas.
6. Facilitar la inserción de figuras en el texto.

Muchos de los procesadores de texto pretenden hacer mucho trabajo por ti. Por ejemplo, al escribir una carta proveen maneras directas y fáciles de formar la carta, pidiendo únicamente el destinatario, el remitente, la firma, etc..

Hay fundamentalmente dos tipos de procesadores de texto, aquellos que van mostrando el resultado de la edición en la misma pantalla, conforme se va escribiendo – como es

el caso de Word – y aquellos que funcionan con comandos de control y que ejecutan un programa para poder ver cómo queda el texto ya formateado. LaTeX es de este último tipo. Llamemos a los primeros de tipo *intérprete*, porque van *interpretando* conforme van recibiendo el texto, y a los segundos de tipo *compilador*, porque primero reciben todo el texto y después lo transforman a las páginas correspondientes.

Cada uno de estos tipos de procesadores tiene sus ventajas y sus desventajas. La principal ventaja del intérprete es que es más fácil de usar y detectas inmediatamente cuando algo no es como lo quieres. La principal desventaja es que supone demasiados parámetros respecto a cómo quieres el texto, y toma muchas decisiones que son difíciles de deshacer. La principal ventaja de los compiladores es que te da un manejo más preciso del formato que quieres, permitiendo de esta manera una edición más rápida. La principal desventaja es que requiere un nivel de abstracción mayor, ya que como no estás *viendo*[1] el resultado de lo que haces, y tienes que compilarlo para verlo, trabajas más "a ciegas".
Se eligió LaTeX para trabajar acá por varias razones, entre ellas:

(a) Es un paquete de distribución gratuita, disponible en prácticamente todos los sistemas Unix y que está ampliamente documentado.
(b) La documentación viene con LaTeX, por lo que está accesible.
(c) Cuenta con un gran número de gente trabajando continuamente en el proyecto, por lo que hay muchos paquetes adicionales para casi cualquier cosa que se te pueda ocurrir, como graficación, música, animación, entre otros.
(d) Es el más usado en el medio de Ciencias de la Computación y Matemáticas para la elaboración de trabajos.

En suma, aun cuando no logres ver más que una pequeña parte, la estructura de LaTeX es tal que el resto de los paquetes los podrás aprender a manejar por su cuenta, conforme los vayas requiriendo.

7.2. Componentes de un texto

Muchos se refieren a los procesadores de texto como procesadores de palabra, reconociendo con esto que el componente básico de un texto son las palabras. Sin embargo, es en la forma de agrupar palabras donde realmente se formatea texto. Veamos algunos componentes importantes de un texto:

palabras	oraciones	párrafos
subsubsecciones	subsecciones	secciones
capítulos	partes	documento

[1] Esto, en realidad, ya no es del todo cierto, pues LaTeX cuenta ya con intérpretes que permiten ir viendo cómo queda el texto conforme lo vas escribiendo.

A partir de las subsubsecciones puedes poner títulos y subtítulos.

7.3. Ambientes para formato

Por el tipo de procesador que es LaTeX, debes indicarle en qué tipo de ambiente quieres elaborar el documento. Los ambientes definen los márgenes que quieres (arriba, abajo, derecho, izquierdo), qué componentes se vale que tenga el documento (capítulos, secciones, partes), el tamaño del renglón, la separación entre párrafos, la sangría. LaTeX cuenta, entre otros, con las siguientes clases de documentos:

Tabla 7.1 LaTeX: ambientes

Ambiente	Nombre	Descripción
Reporte	report	Se utiliza para textos menos formales que un libro o un artículo. No tiene capítulos, sino que agrupa a partir de secciones.
Artículo	article	Es, tal vez, el ambiente que más seguido utilizarán para la entrega de tareas, elaboración de pequeños manuales, etc. Permite escribir en dos columnas fácilmente.
Libro	book	Da toda la infraestructura para la publicación de un libro, con partes, capítulos, secciones, etc. Permite preparar las páginas para imprimir algo por los dos lados.
Acetatos	seminar	Supone un tamaño de letra de cuatro veces el normal, con lo que permite armar acetatos.
Cartas	letter	Permite elaborar cartas que acomodan de manera fácil el remitente, la firma, etc.

Cada una de las clases de documento supone un distinto ambiente. El ambiente es lo primero que le debes dar al compilador de LaTeX .
Un archivo fuente de texto para LaTeX consiste de:

- Una especificación de ambiente

```
\documentclass{report}
```

- El preámbulo, donde especificas algunos modificadores para este ambiente, como pueden ser paquetes adicionales que quieras usar.

```
\usepackage[spanish]{babel}
\usepackage{ucs}
\usepackage[utf8x]{inputenc}
\usepackage{amsmath}
```

También en el preámbulo puedes cambiar de manera explícita algunos de los parámetros de la hoja.

```
\setcounter{chapter}{5}
\addtolength{voffset}{2cm}
```

O, en general, establecer propiedades de los objetos con que vas a trabajar o incluir definiciones:

```
\input{definiciones}
\includeonly{segundo}
```

- El documento propiamente dicho que empieza con

```
\begin{document}
```

y termina con

```
\end{document}
```

y entre estas dos marcas puede haber texto;

```
Éste es un documento que trae conceptos básicos
```

comandos de LaTeX;

```
\begin{center}
    ...
\end{center}
```

o, en general, comandos que insertan en ese lugar texto de otros archivos:

```
\include{primero}
\include{segundo}
```

7.3.1. Posibles errores

En este primer ejercicio realmente tendrás pocos errores que reportar. Generalmente serán aquéllos ocasionados por errores de dedo. Nóta que LaTeX tiene varios tipos de comandos:

1. Aquellos que están entre `\begin{...}` y `\begin{...}`, donde lo que va entre llaves se repite. Veamos algunos ejemplos:

```
\begin{document}
  ....
\end{document}
```

```
\begin{compactenum}
\item ....
\item ....
\end{compactenum}

\begin{center}
    ...
\end{center}
```

A estos comandos les llamamos “de ambiente” porque delimitan a un texto para que se les apliquen determinadas reglas.

II. También tenemos comandos que marcan el inicio y final de un cierto tipo de datos; por ejemplo, caracteres matemáticos. Para esto, cuando quieres que la fórmula vaya insertada entre el texto, puedes usar cualquiera de los dos ambientes que siguen.

```
 $   .... $
\(   .... \)
```

y cuando quieres que aparezca en un párrafo separado, tienes para ello cualquiera de los dos ambientes que siguen.

```
$$   ....   $$
\[   ....   \]
```

III. Aquellos que marcan lo que sigue, y que aparecen únicamente como comandos de control. A estos les llamamos comandos de estado, porque cambian el estado del ambiente:

Ejemplos:

```
\bf
\large
```

Estos comandos funcionan “hasta nuevo aviso”. Puedes delimitar también su rango de acción si colocas entre llaves al comando y a todo a lo que quieres que se aplique:

```
{\large Esta es una prueba}
```

y entonces el comando se referirá únicamente a lo que está en las llaves más internas.

IV. Aquellos comandos que actúan sobre sus parámetros:

```
\section{Primera}
\setcounter{section}{4}
```

V. Y por último, aquellos comandos que simplemente se usan como abreviaturas convenientes o para denotar símbolos especiales. *Siempre* empiezan con \

```
\cdot
\uparrow
```

En el primero, segundo y tercer tipo de comandos es usual que se te olvide cerrar un ambiente, y entonces el compilador reportará que se encontró el final del documento antes de lo esperado, o algún mensaje donde indique que lo que sigue no tiene sentido donde está.

Es frecuente que tengas errores al teclear y LaTeX no reconozca el comando. En esos casos señalará exactamente el lugar donde no está entendiendo.

En el último tipo de comandos, además del primer tipo de error, como pudiera ser el de teclear mal el nombre del comando, puedes también darle menos parámetros de los que espera, o de distintos tipos. Por ejemplo, el comando

```
\setcounter{section}{4}
```

espera una número entero en su segundo parámetro, por lo que si le dieras otra cadena protestaría.

Muchas veces es necesario tener en el documento texto que no quieres que aparezca en la versión final. También pudiera suceder que tengas problemas con un comando particular y quieras ver qué pasa si lo quitas, pero sin quitarlo realmente del archivo. Lo que conviene en esos casos es usar comentarios.

LaTeX tiene dos tipos de comentarios. El primero, y el más sencillo de usar, es el que simplemente comenta el resto de una línea. Para ello se usa el carácter `%` (porciento), y desde el punto en el que aparece hasta el fin de línea (en la pantalla, aunque la línea se haya "doblado") el compilador de LaTeX no va a considerar eso para compilar y no aparecerá en el trabajo final.
La segunda manera es introduciendo el ambiente de comentario mediante

```
\begin{comment}
    ...
\end{comment}
```

y todo lo que quede en este ambiente quedará excluido de la compilación de LaTeX. Para que puedas usar esta forma de comentario, debes incluir el paquete `comment` en el preámbulo de tu documento:

```
\usepackage{comment}
```

Esta manera de incluir comentarios también puede ser muy útil si estás trabajando con un archivo de texto muy grande y deseas compilar por pedazos.

7.4. Formato general de un documento

LaTeX procesa los espacios en blanco de manera especial. Por ejemplo, entre dos palabras un espacio en blanco es lo mismo que 25 o que un cambio de renglón; similarmente, un renglón en blanco es equivalente a diez. En general, no respeta de manera estricta el formato que trae el documento fuente, pues su trabajo consiste, precisamente, en acomodar

adecuadamente el texto de acuerdo a los comandos que recibe. Distingue una palabra de otra por al menos un espacio en blanco, y un párrafo de otro por al menos un renglón en blanco. La forma como aparece el texto fuente, en ausencia de separaciones de palabras o párrafos, no tiene nada que ver con el formato final, ni en el tamaño del renglón, ni en la manera en que silabea.

LaTeX se encarga de generar espacios entre los párrafos cuando encuentra dos cambios de renglón seguidos (al menos un renglón en blanco). También, cada vez que empieza un capítulo, sección, subsección, etc. genera espacios verticales en blanco.

LaTeX ofrece la posibilidad de manipular directamente los espacios que se dejan entre párrafos o el cambio de renglón. En los comandos que siguen, <num> representa a un entero positivo o negativo y <mdda> representa una unidad de medida, que puede ser `pt`, `in`, `cm`, `mm`, `ex`, `em` o cualquier variable de longitud – `ex` representa la altura de una `x` mientras que `em` representa el ancho de la letra `m` – . Los comandos con que cuentas para manipular los espacios se muestran en la tabla 7.2.

Tabla 7.2 LaTeX: comandos para modificar espacios

Comando	Descripción
\␣	Indica que se deje incondicionalmente un espacio en blanco; puede ser necesario cuando LaTeX genera algún código y suprime los blancos que lo separan.
~	Produce también un espacio en blanco, pero que el compilador no puede usar para cambiar de línea.
\\	Cambia de renglón y el siguiente renglón empieza en el margen izquierdo.
\\[< num><mdda>] *[< num><mdda>]	Da un cambio de renglón, pero salta, además de al renglón siguiente, tanto como se le indique. Si el número es negativo, retrocede en la hoja. El * a continuación de \\\\ es para forzar a que se dé el espacio; de otra manera, LaTeX decide si aplica el espacio o no.
\vspace{<num><mdda>} \vspace*{<num><mdda>}	Da un espacio vertical del tamaño especificado. Si el número es negativo, retrocede en la página. Nuevamente, el * es para forzar a LaTeX a que recorra el espacio especificado.
\smallskip \medskip \bigskip	Permiten el desplazamiento vertical del texto en un cuarto del tamaño de la línea base (\\baselineskip), media línea base y una completa. Es muy útil porque son saltos relativos al tamaño de letra que estés usando.

Continúa en la siguiente página

Tabla 7.2 LaTeX: comandos para modificar espacios *Continúa de la página anterior*

Comando	Descripción
\noindent	Hace que un párrafo o renglón empiece en el margen izquierdo de la hoja, en lugar de hacerlo con sangría.
\indent	Obliga a dejar un espacio horizontal igual al de la sangría de párrafos.
\hspace{<num><mdda>} \hspace*{<num><mdda>}	Deja un espacio horizontal dado por el número. Si el número es negativo, retrocede en la línea. LaTeX puede decidir que el espacio no es adecuado en ese punto y no obedecer. Nuevamente, el * fuerza a LaTeX a obedecer la instrucción.

Tanto \vspace como \hspace pudieran ser inhibidos por LaTeX si es que el compilador considera que no se vería bien. Por ejemplo, si el desplazamiento vertical toca a principio de una página, decide no hacerlo.

Todo documento, menos las cartas, deben tener un título, autor(es) y la fecha en que fue impreso. Algunas veces se agrega una nota de agradecimiento. En general, a esto se le llama el título. En el preámbulo (antes de que empiece el documento propiamente dicho) debes colocar los datos para que LaTeX pueda armar el título que se desea. Los comandos a LaTeX son:

```
\title{Ejercicios de Latex}
```

Este es un comando con un parámetro que le indica a LaTeX que el título del documento es "Ejercicios de Latex". Nota que el argumento debe ir entre llaves.

```
\author{El Chapulín Colorado \and Pepe Grillo}
```

Si se trata de más de un autor, los nombres de los autores irán separados por la palabra \and.

```
\date{El día más bello del planeta\\
      \today}
```

En la fecha puedes poner lo que quieras, aunque se espera una fecha. Las dos diagonales inversas \\ sirven para cambiar de renglón, y las puedes utilizar en casi cualquier momento. El comando \today sirve para que LaTeX inserte la fecha de hoy.

```
\thanks{Gracias a los voluntarios de este taller}
```

Sirve para agregar, al pie de la página, algún comentario que se desee. Este comando deberá aparecer una vez iniciado el documento propiamente dicho.

Como ya mencionamos, dependiendo de la clase de documento LaTeX se comportará de manera diferente al compilar el texto fuente (la especificación de \documentclass).

7.4.1. Cartas

Para escribir cartas, todo indica que el formato que le tiene que dar LaTeX es muy distinto al de los documentos que hemos visto hasta ahora. Puedes meter en un mismo documento varias cartas, ya que se espera que el remitente y la firma sean los mismos para todas las cartas incluidas en el documento. Por ello, una vez iniciado el documento, debes especificar estos dos parámetros. Veamos primero el remitente:

Remitente: Se refiere a lo que normalmente aparece en el tope de la carta y que es la dirección de aquél quien firma la carta:

```
\address{Dra. Genoveva Bienvista\\
         Cubículo 003. Edif. de Matemáticas\\
         Facultad de Ciencias, UNAM}
```

Puede consistir de uno o más renglones. Si deseas más de un renglón en el remitente, separas los renglones con \\.

Firma: Para la firma se especifica de manera similar al remitente:

```
\signature{ Dra. Genoveva Bienvista\\
            Profesor Titular B}
```

Una vez que tienes definidos estos parámetros, se procede a escribir cada una de las cartas. Se delimita cada una de éstas por

```
\begin{letter}
  . . .
\end{letter}
```

Dentro de cada carta colocarás los parámetros que se refieren a cada una de ellas. Veamos cómo se hace:

Destinatario: A continuación de \begin{letter} colocarás entre llaves los datos del destinatario:

```
\begin{letter}{Distinguido estudiante de Ciencias
               de la Computación\\
               Primer Ingreso\\
               Generación 2007}
```

Con esto, aparecerá el texto inmediatamente antes de la carta.

Introducción: Llamamos introducción a la frase con la que se saluda o inicia la carta. También se relaciona únicamente con la carta en cuestión:

```
\opening{Muy querido estudiante:}
```

y este renglón aparecerá al inicio de la carta.

Despedida: Esta es la frase que aparecerá antes de que aparezca la firma y "jalará" a la firma a continuación. Aparecerá dondequiera que coloques el comando, por lo que lo deberá colocar inmediatamente antes del fin de la carta (\end{letter}).

```
\closing{Quedo de Uds. atentamente,}
```

Copias: Muchas veces quieres poner al pie de la carta la lista de las personas a las que pretendes entregarle una copia de la carta. Esto se logra con el comando:

```
\cc{Mis mejores amigos\\
    Mis estimados colegas\\
    Primer ingreso en general}
```

También este comando, como el de despedida, aparecerá en donde lo coloques, por lo que lo deberás colocar también al final de la carta y después de la despedida.

7.5. Listas de enunciados o párrafos

Un párrafo es un conjunto de oraciones que empiezan y terminan con doble Enter. En el documento fuente puedes observar un renglón en blanco para finalizar cada párrafo. El aspecto en el documento formateado dependerá de los parámetros que tenga LaTeX para compilar ese texto – aunque se pueden modificar, no lo haremos por el momento–.

LaTeX provee mecanismos para presentar listas de párrafos, ya sean numerados, etiquetados o con una marca separándolos entre sí. En general, tienen el siguiente formato:

```
\begin{<tipo-de-lista>}
\item <párrafo>
\item <párrafo>
\item . . .
\end{<tipo-de-lista>}
```

Cada párrafo que desees marcar (numerar) deberá ir precedido del comando `\item`. Entre un `\item` y el siguiente, o entre `\item` y el final de la lista puede haber más de un párrafo.

Los <*tipo de lista*> que vas a manejar por lo pronto son los siguientes:

```
\begin{enumerate}
    <enumeración de párrafos>
 \end{enumerate}

\begin{itemize}
   <listado de párrafos>
\end{itemize}

\begin{description}
    <descripción de párrafos>
\end{description}
```

Cada uno de éstos puede aparecer anidado en el otro, o anidado en sí mismo. Veamos primero las numeraciones.

7.5.1. Numeraciones

Empezaremos por el tema de incisos numerados. Veamos un anidamiento de numeración:

```
\begin{enumerate}
\item Este es el primer párrafo en el primer nivel
  \begin{enumerate}
  \item Queremos ver qué pasa al anidar en el segundo nivel
    \begin{enumerate}
    \item Anidamos una vez más para el tercer nivel
    \item Otro inciso para que se vea qué pasa en el tercer nivel
      \begin{enumerate}
      \item Numeración en el cuarto nivel
      \end{enumerate}
    \end{enumerate}
  \item Otro inciso en el segundo nivel de numeración
  \end{enumerate}
\item Otro inciso en el primer nivel de numeración
\end{enumerate}
```

que produce el texto:

1. Este es el primer párrafo en el primer nivel

   a) Queremos ver qué pasa al anidar en el segundo nivel

      1) Anidamos una vez más para el tercer nivel
      2) Otro inciso para que se vea qué pasa en el tercer nivel

         a′ Numeración en el cuarto nivel

   b) Otro inciso en el segundo nivel de numeración

2. Otro inciso en el primer nivel de numeración

Como puedes observar, cada nivel lleva su propio contador y usa sus propios métodos de numeración. Los valores por omisión, como acabas de ver, son los números arábigos seguidos de un punto para el primer nivel; las letras minúsculas seguidas de un paréntesis para el segundo nivel; nuevamente números arábigos, pero seguidos de un paréntesis para el tercer nivel; y nuevamente letras mayúsculas, pero seguidas de un apóstrofo para el cuarto nivel. LaTeX admite únicamente cuatro niveles de anidamiento para las numeraciones. Las numeraciones también tienen definidas espacio entre los distintos niveles y sangrías para los mismos. Puedes definir qué tipo de numeración utilice. La manera más fácil es incluyendo el paquete para numeraciones en el preámbulo de tu documento fuente, \usepackage{enumerate} y entonces podrás marcar qué tipo de numeración deseas, colocando el que correspondería al primero de la lista:

```
\begin{enumerate}[(a)]
\item Es un paquete de distribución gratuita, disponible
        en prácticamente todos los sistemas Unix, y que está
        ampliamente documentado.
\item La documentación viene con \textnormal{\LaTeX{}},
      por lo que está accesible.
\end{enumerate}
```

Con esta opción, produces la numeración con las etiquetas que deseas:

(a) Es un paquete de distribución gratuita, disponible en prácticamente todos los sistemas Unix, y que está ampliamente documentado.

(b) La documentación viene con LaTeX, por lo que está accesible.

Mientras que

```
\begin{enumerate}[I.]
\item Es un paquete de distribución gratuita, disponible en
      prácticamente todos los sistemas Unix, y que está
      ampliamente documentado.
\item La documentación viene con \textnormal{\LaTeX{}},
      por lo que está accesible.
\end{enumerate}
```

produce

I. Es un paquete de distribución gratuita, disponible en prácticamente todos los sistemas Unix, y que está ampliamente documentado.

II. La documentación viene con LaTeX, por lo que está accesible.

Además del tipo de número a usar, indicas también una cierta manera de editarlo. Por ejemplo, en el caso anterior, cuando usas minúsculas las colocas entre paréntesis, mientras que cuando usas números romanos en mayúscula, al número lo haces seguir por un punto.

Las maneras que tienes de numerar presentan al tipo de letra que va a ir, precedido y/o seguido de algún separador, como `):,-`:

- Arábigos. El valor por omisión para el primer nivel. `[1.]`
- Letras minúsculas. `[a-]`
- Letras mayúsculas. `[A:]`
- Números romanos en minúscula. `[i.]`
- Números romanos en mayúscula. `[I.]`

Si aparece entre los corchetes algo que no sea identificado como el primer número de cierto tipo, LaTeX simplemente repetirá esa cadena frente a cada uno de los párrafos marcados con `\item`. Si deseas, por ejemplo, agregar un texto antes del número, como `Nota 1`, lo que debes hacer es encerrar entre llaves lo que no corresponde al número.

```
\begi
\item
\item
\end{
```

Nota 1. Primera nota.

Nota 2. Segunda nota.

Cabe aclarar que la sustitución que hagas para el tipo de numeración afecta únicamente al nivel en el que aparece. Los niveles anidados seguirán la convención por omisión descrita arriba.

7.5.2. Listas marcadas

Cuando simplemente quieres marcar a cada párrafo con algún carácter, como un guión, un punto o algo similar, puedes utilizar las listas de párrafos:

```
\begin{itemize}
\item . . .
\item . . .
 . . .
\end{itemize}
```

También estas listas las puedes anidar entre sí hasta cuatro niveles de profundidad, cambiando el carácter que marca a los distintos párrafos. Observa el siguiente ejemplo, con el código de LaTeX a la izquierda y el resultado a la derecha.

```
\begin{itemize}
\item Primero del primer nivel
  \begin{itemize}
  \item Primero del segundo nivel
    \begin{itemize}
    \item Primero del tercer nivel
      \begin{itemize}
      \item Primero del cuarto nivel
      \item Segundo del cuarto nivel
      \end{itemize}
    \item Segundo del tercer nivel
    \end{itemize}
  \item Segundo del segundo nivel
  \end{itemize}
\item Segundo del primer nivel
\end{itemize}
```

- Primero del primer nivel
  - Primero del segundo nivel
    - Primero del tercer nivel
      - Primero del cuarto nivel
      - Segundo del cuarto nivel
    - Segundo del tercer nivel
  - Segundo del segundo nivel
- Segundo del primer nivel

Al igual que en las numeraciones, no puedes tener más de cuatro anidamientos del mismo tipo de listas.

Puedes modificar el carácter que quieras que use como marca en la lista, colocando el carácter deseado entre corchetes a continuación de `\item` en el párrafo seleccionado. El código se encuentra a la izquierda y el resultado a la derecha.

```
\begin{itemize}
\\item[-] Primero del primer nivel
  \begin{itemize}
  \item[+] Primero del segundo nivel
    \begin{itemize}
    \item[:] Primero del tercer nivel
      \begin{itemize}
      \item[>] Primero del cuarto nivel
      \item Segundo del cuarto nivel
      \end{itemize}
    \item Segundo del tercer nivel
    \end{itemize}
  \item Segundo del segundo nivel
  \end{itemize}
\item Segundo del primer nivel
\end{itemize}
```

- Primero del primer nivel
  - + Primero del segundo nivel
    - : Primero del tercer nivel
      - > Primero del cuarto nivel
      - ◇ Segundo del cuarto nivel
    - ○ Segundo del tercer nivel
  - • Segundo del segundo nivel
- ■ Segundo del primer nivel

Como puedes observar, a diferencia de las numeraciones, únicamente cambia el párrafo de la lista que hayas marcado. El resto de las marcas se mantiene igual. Hay manera de cambiar la marca para todos los elementos del nivel de anidamiento, de la misma manera que se hizo con las numeraciones.

7.6. Tablas

En general, LaTeX provee comandos para definir tablas de los tipos más variados. Veremos acá únicamente las más sencillas, aunque no por eso poco poderosas.

Una tabla es un conjunto de renglones alineados por columnas. Por ello, para que LaTeX vaya acomodando las columnas le debes dar el número de columnas y el formato de de cada una de ellas. El formato general de una tabla de texto es el siguiente:

```
\begin{tabular}[<pos>]{<cols>}
    <renglones>
\end{tabular}
```

Es necesario aclarar que una tabla puede aparecer en cualquier posición donde aparezca una palabra. Por ello, es conveniente en la inmensa mayoría de los casos preceder y suceder a la tabla con un renglón en blanco.

El argumento que va entre corchetes, *<pos>* se refiere al caso en que desees que la tabla aparezca a continuación de una palabra, si la queremos alineada “desde arriba” ([t]), desde

abajo ([b]) o centrada [c], que es el valor por omisión. Este parámetro no tendrá sentido si se deja, como acabamos de mencionar, un renglón en blanco antes y después de la tabla.

El argumento <*cols*> se refiere al formato que deberán tener las columnas de la tabla, y que será aplicado a cada uno de los renglones. Debes especificar el formato para cada una de las columnas que deseas tener en la tabla. También especificas si se debe colocar algo entre las columnas. De esto, <*cols*> corresponde a una lista de especificadores, que se refieren a la sucesión de columnas e intercolumnas:

Tabla 7.3 Modificadores para la definición de columnas

Especificación	Descripción
l	Especifica que el contenido de esta columna se colocará pegado a la izquierda, y ocupará tanto espacio como requiera para quedar contenido en un renglón.
r	Similar al anterior, pero pegado a la derecha.
c	Similar al anterior, pero centrado.
p{<*ancho*>}	Da un ancho fijo para la columna. LaTeX acomodará el texto como si fuera un párrafo, ocupando tantos renglones como sea necesario.
m{<*ancho*>}	Igual que p, excepto que si hay columnas que dan un número distinto de líneas para el mismo renglón, centra verticalmente el contenido de las columna.

Para estas dos opciones, el *{<ancho>}* deberá estar dado en:

in	pulgadas
cm	centímetros
mm	milímetros
pt	puntos
em	ancho de carácter
Algún nombre de longitud	

|	Una línea vertical que separa columnas.
||	Dos líneas verticales separando las columnas.

En los primeros tres casos, el tamaño final de la columna será el máximo de todos los renglones de la tabla. Si es que va a haber mucho texto en cada columna, te recomendamos usar la cuarta o quinta descripción, para que el renglón se “doble” y se conforme un párrafo.

Los renglones se componen de material que quieres en cada columna. Cada vez que

deseas cambiar de columna, debes insertar un carácter & y cada vez que quieras cambiar de renglón deberás agregar al final del renglón \\.

Puedes colocar líneas entre cada dos renglones con el comando \hline. Cuando no es el primero de los renglones, debe venir a continuación de un cambio de renglón. Veamos un ejemplo sencillo a continuación.

```
\begin{tabular}[t]{cccc}
   letras y símbolos\\
   palabras & oraciones
   & párrafos  & subsecciones\\
   secciones  & capítulos
   & partes  & documento
\end{tabular}
```

Se tiene una tabla con cuatro columnas y dos renglones y se usa & para *separar* las columnas entre sí. Se utilizan \\ para *separar* los renglones. La tabla queda de la siguiente manera:

letras
pa
se

Es importante que notes que lo que corresponde a un renglón formado no forzosamente tiene que aparecer en el mismo renglón en el código. Sólo en el caso en que la especificación de la columna sea p o m, el introducir un renglón en blanco será interpretado como cambio de párrafo, que lo realiza dentro del mismo renglón de la tabla y en la misma columna.

7.7. Fórmulas matemáticas y símbolos especiales

Hasta ahora has escrito texto que si bien te permite escribir, por ejemplo, una novela, no te sirve de mucho para cuando quieres escribir texto un poco más especializado. En estas notas han aparecido muchos caracteres especiales. En general, deseas poder escribir caracteres especiales y, en particular, fórmulas matemáticas.

Para algunos caracteres especiales, como pudieran ser las vocales acentuadas del español, las diéresis en la u o la tilde en la n, hemos logrado que Emacs se comporte como máquina de escribir y haga que el apóstrofo ('), la tilde (~) y las comillas (") se comporten lo que se conoce como símbolos "sordos", esto es, que "no escuchen" hasta que se teclee el siguiente carácter – o bien está configurado el teclado para que la combinación de tecla con Alt produzca los caracteres acentuados –. Sin embargo, pudiera ser que no estuviera Emacs habilitado de esta manera. En general, LaTeX provee mecanismos para que se impriman una gran cantidad de símbolos que no aparecen en el inglés, pero que se usan en

idiomas como el francés, el español y el alemán. Casi todos empiezan con una diagonal inversa \ y siguen con el código especial dado a ese carácter. LaTeX los llama *símbolos*. Veamos a continuación una lista de los más usados:

Tabla 7.4 Acentos

`\`{o}`	ò	`\~{o}`	õ	`\v{o}`	ǒ	`\c{o}`	ǫ	`\'{o}`	ó
`\={o}`	ō	`\H{o}`	ő	`\d{o}`	ọ	`\^{o}`	ô	`\.{o}`	ȯ
`\t{oo}`	o͡o	`\b{o}`	o̱	`\"{o}`	ö	`\u{o}`	ŏ	`\'{\i}`	í

En todas estas combinaciones puede aparecer cualquier carácter alfabético. En el caso de que lo que siga a la \ sea un símbolo especial, no es necesario poner al argumento entre llaves: `\~a` ã y `\^d` d̂.

Los símbolos que siguen no tienen argumentos, sino que se sustituyen tal cual se especifican.

Tabla 7.5 Símbolos no ingleses

`\ae`	æ	`\oe`	œ	`\aa`	å	`\AA`	Å	`\AE`	Æ
`\OE`	Œ	`\O`	Ø	`\l`	ł	`\L`	Ł	`?``	¿
`!``	¡	`\o`	ø	`\ss`	ß				

Se tienen algunos símbolos de puntuación que resultan útiles. Se incluyen acá también los símbolos que tienen algún significado especial en LaTeX.

Tabla 7.6 Símbolos de puntuación y especiales

`\dag`	†	`\S`	§	`\copyright`	©			`\P`	¶
`\ddag`	‡	`\pounds`	£	`\#`	#			`\$`	$
`\%`	%	`\&`	&	`\_`	_	`\&`	&&verb+{ }+		{ }

Finalmente, cuando se trate de que un símbolo aparezca “tal cual”, sin que sea interpretado por LaTeX (por ejemplo, la diagonal inversa \) simplemente usas el comando `\verb+`*texto*`+` y todo el texto que aparece entre los símbolos “+” no será interpretado y aparecerá en la página tal cual lo tecleaste[2].

[2] Usando este comando es como se han escrito algunos de los comandos en todo este texto.

7.7.1. Fórmulas matemáticas

El tipo de texto que hemos escrito hasta ahora es lo que LaTeX denomina texto LR (*left to right*) y que consiste fundamentalmente de párrafos que se acomodan en la hoja como vienen. Cuando quieres escribir fórmulas matemáticas, éstas contienen símbolos que obligan al compilador a realizar un formato también vertical, no nada más en el renglón correspondiente. Para lograrlo deberás introducir un "ambiente" matemático. Tienes dos tipos de ambientes matemáticos:

I. En la línea, donde lo que buscas es poder intercalar una fórmula o un carácter ($\alpha_i = 2 \cdot \beta^3$). Esto se logra, de cualquiera de las siguientes maneras:

Se escribe:	**Se forma:**
$\alpha = 2 \cdot \beta^3$	$\alpha_i = 2 \cdot \beta^3$
\(\ alpha_i=2\cdot\beta^3\)	$\alpha_i = 2 \cdot \beta^3$
\begin{math}\alpha_i=2\cdot\beta^3\end{math}	$\alpha_i = 2 \cdot \beta^3$

II. En un "recuadro" que contiene a la fórmula matemática. Introduces también un ambiente que dejará espacio en blanco entre el párrafo anterior y el siguiente. Dentro de ese párrafo puedes escribir las fórmulas. Para mostrar esta modalidad, se mostrará del lado izquierdo el texto fuente y del derecho el texto formado:

\[\ alpha=2\cdot\beta^3\]	$\alpha_i = 2 \cdot \beta^3$
\begin{displaymath} \alpha_i=2\cdot\beta^3 \end{displaymath}	$\alpha_i = 2 \cdot \beta^3$

La diferencia fundamental entre el modo de despliegue (párrafo) y el de línea es el tamaño de los términos. Mientras que en el modo de línea se ajusta el tamaño a la altura normal de una línea, en el modo de párrafo ocupa tanto espacio vertical como requiera. Compara estos dos modos en el siguiente esquema.

\(2\ frac{n}{n+1\}\)	$2\frac{n}{n+1}$	\[2\ frac{n}{n+1}\]	$2\dfrac{n}{n+1}$

Cuando estás en modo matemático, ya sea de línea o de recuadro, los espacios en blanco únicamente tendrán sentido para separar comandos. No aparecerán en las fórmulas que escribas como tales, a menos que utilices un blanco precedido por la diagonal inversa. El texto que escribas se formará con un tipo parecido al de las itálicas, y si escribes varias palabras, aparecerán sin los espacios. Compara el modo matemático con el tipo itálico en los siguientes renglones.

\(Esta\ es\ una\ prueba\)	$Esta\ es\ una\ prueba$
{\ it Esta es una prueba}	*Esta es una prueba*

Nota que el lugar que ocupa cada carácter es ligeramente mayor en el modo matemático.

Revisemos ahora cuáles son los componentes principales de las fórmulas matemáticas.

Exponentes

Una de las acciones más comunes que quieres realizar es la de poner exponentes. Esto se logra fácilmente si donde quieres deseas un exponente colocas el símbolo ^ seguido de un sólo símbolo, para que ese símbolo aparezca como exponente; o bien seguido de varios símbolos, todos ellos entre llaves. No se permiten secuencias de más de una combinación de ^ con el exponente. Siguen algunos ejemplos:

$a^m b^n c^{n+m}$	$a^m b^n c^{n+m}$
x^{y^2}	x^{y^2}
x^{2y}	x^{2y}
x^2y	x^2y

En el segundo ejemplo nos vimos forzados a encerrar el segundo exponente entre llaves para que no se infringiera la regla de que hubiera más de uno seguido. De esta manera, el exponente de x es, simplemente, otra expresión con exponente.

Como puedes observar en todos estos ejemplos, las llaves sirven para indicar dónde empieza y termina el exponente. Es claro, por ejemplo en el último caso, que en ausencia de llaves LaTeX considerará únicamente al primer carácter que sigue a ^.

Subíndices

Funcionan de la misma manera que los exponentes, excepto que se usa el símbolo _ de subrayado en lugar del ^. Las mismas reglas aplican para subíndices que para exponentes. Siguen algunos ejemplos:

$a_m b_n c_{n+m}$	$a_m b_n c_{n+m}$
x_{y_2}	x_{y_2}
x_{2y}	x_{2y}
x_2y	x_2y

7.8. Imágenes y figuras

7.8.1. Tablas, figuras, etc.

Habrás notado a lo largo de este libro que aparecen figuras o tablas que se van numerando. Más aún, la numeración de las tablas, por ejemplo, es independiente de la de las gráficas o, en general, figuras que queramos incluir. Esto se hace mediante lo que LaTeX llama *flotantes*. Les llama así porque le puedes indicar al compilador que las acomode no forzosamente donde aparecen, sino en la primera página en la que no ocasionen que se deje espacio en blanco innecesario. Por supuesto que también le puedes indicar que las coloque exactamente donde aparecen, sin importar el armado de las páginas que las preceden, o haciendo tú el armado a pie.
Las flotantes que vamos a revisar acá son únicamente dos:

- Figuras (figure)
- Tablas (table)

Ambas flotantes tienen los mismos argumentos, que tienen que ver fundamentalmente con dónde se desea que se les acomode; ambas proveen un espacio con la posibilidad de ponerles un título (\caption)[3]. Definen un ambiente donde los cambios de tipo de letra o tamaño serán efectivos únicamente dentro de ese ambiente. En general, el ambiente table se usa para material que se presenta en forma de tabla o lista, mientras que el de figure se usa para material gráfico. La sintaxis de estos ambientes es la siguiente:

Para figuras:

```
\begin{figure}[posicionador]
    Contenido
    \caption{Título de la figura}
\end{figure}
```

Para tablas (o cuadros):

```
\begin{table}[posicionador]
    Contenido
    \caption{Título de la tabla}
\end{table}
```

Si el *contenido de la figura o tabla* va antes de \caption, entonces el título aparecerá abajo de la figura. Si va después, esto es que \caption sea el primer renglón después de que empieza la figura o tabla, el título irá arriba. La posición de la figura o tabla puede ser:

[3] También hay la posibilidad de marcarlas con una etiqueta para poder hacer referencias a ellas, pero dado lo poco del tiempo no entraremos en eso en esta ocasión.

Tabla 7.7 Posiciones posibles para los flotantes

letra	significado
t	*top:* Lo coloca en la parte superior de la primera página disponible.
b	*bottom:* Lo coloca en la parte inferior de la primera página donde quepa.
p	*page:* Lo coloca en una página de flotantes, la primera que pueda.
h	*here:* Lo coloca, si es que puede, en el punto donde aparece.
H	*Here:* Incondicionalmente donde aparece. Si es necesario, porque la figura no quepa en esa página, deja página en blanco y lo coloca en la siguiente página.
!	*Try harder:* Intenta de manera más firme acomodar la figura o tabla donde se especifica con los posicionadores que siguen a !.

Para acomodar una figura o tabla procedes a dar tus preferencias, que son combinaciones de las primeras tres letras. LaTeX tratará de acomodar la figura, colocándola en la misma página o posponiéndola, procurando dejar poco espacio en blanco. Si no se especifica, el valor por omisión es tbp. LaTeX podría no acomodar las figuras como le indicas porque tiene parámetros que le indican, por ejemplo, cuántas figuras pueden ir en una misma página o qué porcentaje del total de la página pueden ocupar. No entraremos en estos detalles por falta de tiempo. En la mayoría de los casos, y sobre todo en un principio, puedes trabajar con los valores por omisión, excepto posiblemente para el posicionamiento. Te recomendamos, en general, poner H y si no se forma "bonito", proceder a acomodar la figura o tabla a pie. Veamos algunos ejemplos sencillos, con el código de LaTeX que los produce.

Tabla 7.8 Conversión de grados Farenheit a Centígrados

Fahr	Cent
32	0
40	4.5
80	27

```
\begin{table}[!h]
  \caption{Conversión de Farenheit
       a Centígrados}
  \begin{tabular}[t]{|r|r|}
     \hline
     \bf Fahr & \bf Cent\\\hline
     32 & 0\\
     40 & 4.5\\
     80 & 27\\\hline
  \end{tabular}
\end{table}
```

Puedes cambiar varios aspectos de esta tabla. Si, como ya dijimos, quieres que la acomode en la parte superior de una página, conseguirías que coloque primero el texto que

dimos como código y después la tabla misma. Sin embargo, es poco recomendable dar una única posición como opción, porque entonces si LaTeX no puede acomodar a la tabla o figura en la siguiente página, lo más seguro es que ya no la coloque en ningún lugar y la perdamos. Observa cómo la tabla 7.9 que estás colocando inmediatamente después de estas líneas, queda acomodada de manera aparentemente "arbitraria" (está acomodada utilizando sólo la mitad del ancho de la página).

Tabla 7.9 Conversión de Farenheit a Centígrados

Fahr	Cent
32	0
40	4.5
80	27

La tabla 7.9 fue generada con el siguiente encabezado: \begin{table}[H]. Las líneas

```
\begin{center}
  ...
\end{center}
```

se pueden usar para centrar cualquier párrafo, línea, figura o tabla.

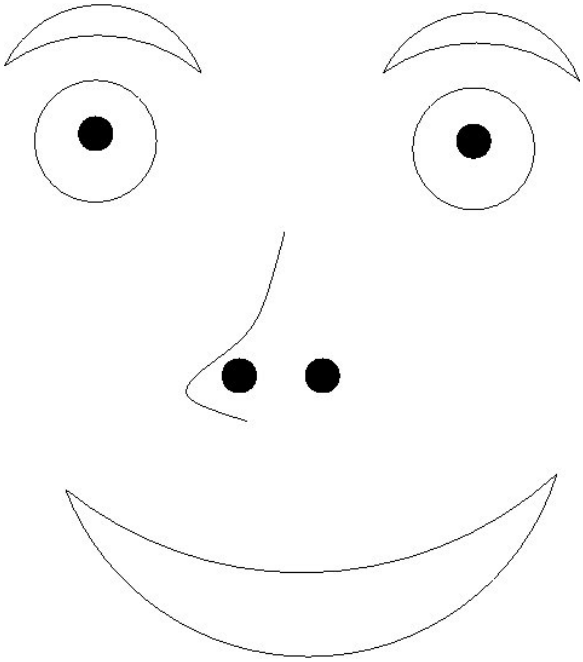
Cuando dejas que LaTeX acomode a la tabla o figura donde mejor pueda – en ausencia de H – deberá haber un renglón en blanco antes y después de la tabla, para evitar que se encimen o mezclen renglones que pusimos en párrafos separados.

En una flotante de tabla puede colocar uno o más array, cuidando nada más de declarar un ambiente matemático adentro.

Todo lo que dijimos de las tablas se aplica a las figuras. El contador de tablas es independiente del de figuras. LaTeX provee un mecanismo sencillo para incluir imágenes o figuras generadas desde fuera y que estén codificadas en postscript o imágenes .png. Esto se hace con el comando \includegraphics {...} que toma como argumento un archivo en el que se encuentre la imagen o figura que se desea incluir. En el archivo dummy.ps se encuentra una figura (hecha en xfig) que deseamos incorporar en nuestro documento. Para hacerlo, tenemos que incluir el uso del paquete graphics en el preámbulo del documento: \usepackage{graphics}. Este paquete cuenta con los mecanismos para poder manipular la fi-

gura:

Figura 7.1 Gráfico introducido con `includegraphics`



que fue incluida con el siguiente código:

```
\begin{figure}[H]
  \caption{Gráfico introducido con \texttt{includegraphics}}
  \includegraphics[width=.5\textwidth]{latex/dummy}
\end{figure}
```

Lenguajes de marcado 8

Los lenguajes de marcado (*markup languages*) utilizan texto común y corriente combinado con información adicional que sirve para definir su semántica o su presentación (a veces ambas). En palabras sencillas: esta información adicional nos dice *qué* significa el texto o *cómo* se presenta al usuario. La información adicional suele estar intercalada en el mismo texto utilizando *marcas* especiales; de ahí el nombre de lenguajes de marcado.

Los lenguajes de marcado tienen sus orígenes en la industria editorial, mucho antes de la creación de computadoras digitales: cuando un manuscrito se preparaba para impresión, un especialista (coloquialmente conocido como "marcador") escribía en los márgenes anotaciones especiales (marcas) que servían de guía a los que transcribían el texto en su forma final ya lista para imprimirse. Estas marcas eran un lenguaje de marcado primitivo (no había necesariamente un estándar, ni tenían una sintaxis formalmente definida), que servía para explicar la presentación del texto: el tamaño de la fuente, el tipo, el estilo, etc.

Con la llegada de las computadoras, los lenguajes de marcado ganaron además la capacidad de obtener automáticamente información semántica: por ejemplo, si en nuestro lenguaje de marcado definimos la marca AUTOR: para que con ella identifiquemos al autor del documento, es relativamente sencillo escribir un programa que obtenga el autor o autores de todos los documentos disponibles y haga una relación entre ellos.

Aunque hay muchísimos lenguajes de marcado actualmente (TEX y PostScript son lenguajes de marcado procedurales), de especial importancia es SGML, el Lenguaje de Marcado Estándar Generalizado (*Standard Generalized Markup Language*). SGML es un meta lenguaje para definir lenguajes de marcado, y de ahí se derivan los dos lenguajes de marcado probablemente más famosos y usados en la actualidad: XML y HTML. Nos centraremos en estos últimos durante el resto del capítulo.

8.1. XML

SGML es un metalenguaje grande y complejo. Implementar un programa (o sistema de programas) que pueda lidiar con *cualquier* lenguaje definido con SGML tendrá que ser igualmente grande y complejo. La mayor parte de los programas que se utilizan para manejar lenguajes definidos con SGML, en la práctica sólo soportan un subconjunto de todas las opciones que ofrece.

En gran medida por este motivo, y las necesidades especiales que presentan las aplicaciones que necesitan comunicarse por Internet (SGML precede por varios años a Internet), fue que se creó XML, el Lenguaje de Marcado Extensible (eXtensible Markup Language). XML es un subconjunto simplificado de SGML, lo que hace el escribir programas que lo manejen mucho más fácil, y además está pensado para compartir información, especialmente a través de la red.

XML es inmensamente popular y existen muchísimos lenguajes definidos con XML que se usan hoy en día; por nombrar sólo algunos: RSS (para noticias), MathML (para representar fórmulas matemáticas complejas), XHTML (el sucesor *de facto* de HTML), Scalable Vector Graphics (para gráficos escalares), MusicXML (para notación musical) y miles más. XML también es utilizado por muchos programas para guardar sus archivos, especialmente en el mundo del software libre: OpenOffice, AbiWord, Gnumeric y KOfficen usan XML para guardar sus documentos, por nombrar unos cuantos.

Un documento XML presenta su información en una estructura jerárquica, que podemos ver como un árbol. Todo documento XML tiene un *elemento raíz*, que a su vez puede tener más elementos y/o texto común y corriente. Cada elemento a su vez puede tener más elementos y/o texto y así sucesivamente. Además, cada elemento puede tener *atributos*. Veamos un ejemplo en el listado 8.1

Código 8.1 Documento en XML (1/2)

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Ejemplo de un libro -->
<libro>
 <titulo>
  El ingenioso hidalgo don
  Quijote de la Mancha
 </titulo>
```

Código 8.1 Documento en XML (2/2)

```
<autor>Miguel de Cervantes Saavedra</autor>
 <parte numero="1">
 <capitulo numero="1">
  <resumen>
   Que trata de la condici&oacute;n y ejercicio
   del famoso hidalgo don Quijote de la Mancha
  </resumen>
  <parrafo>
   En un lugar de la Mancha, de cuyo nombre no
   quiero acordarme, no ha mucho tiempo que
   viv&iacute;a un hidalgo de los de lanza en
   astillero, adarga antigua, roc&iacute;n flaco
   y galgo corredor.
  <parrafo>
   ...
 </capitulo>
 ...
</parte>
...
</libro>
```

La primera línea es la *declaración* XML, que sirve para definir qué versión del estándar estamos usando (generalmente la "1.0") y el conjunto de caracteres en que está el documento (generalmente "UTF-8"). La declaración XML es opcional; sin embargo, es recomendable utilizarla.

La segunda línea es un comentario. Los comentarios sirven para explicar detalles del archivo a un lector humano. Los comentarios siempre se declaran de la misma manera; comienzan con "$<! - -$" y terminan con "$- - >$". No es válido tener doble guion dentro de un comentario.

El ejemplo es de un libro: libro es el elemento raíz, titulo, autor, parte, capitulo, extracto y parrafo son elementos, mientras que numero es un atributo que poseen parte y capitulo. Los atributos siempre deben ir entre comillas, mientras que el texto plano no.

Como estamos definiendo un documento XML que utiliza el conjunto de caracteres UTF-8, sería perfectamente legal definir a los elementos como título, capítulo, párrafo, pero no es lo que suele acostumbrarse. De igual forma, podrían utilizarse acentos (o eñes o diéresis), pero utilizamos "ó" en lugar de "ó" para explicar lo que son las *entidades*.

Las entidades son usadas para escribir caracteres en XML que de otra forma sería complicado hacerlo. Por ejemplo, el símbolo de "menor que" ($<$) no es válido dentro de un documento XML porque es el utilizado para definir cuándo empieza una etiqueta de inicio o final; entonces se utiliza la entidad < (por *"less than"*, en inglés). De igual forma, para usar el símbolo de ampersand (&) se utiliza la entidad &. En todo documento XML están definidas las siguientes cinco entidades:

Entidad	Símbolo
&	&
>	>
<	<
'	"
"	’

pero se pueden definir más entidades y veremos cómo más adelante. Además, todo símbolo en UTF-8 puede utilizarse mediante su número identificador; por ejemplo, en lugar de ó para la “ó”, podríamos usar Č, porque 268 es el número que le corresponde a “ó” en UTF-8.

El documento de este ejemplo está *bien formado*; esto quiere decir que todos sus elementos tienen etiquetas de inicio y finales (por ejemplo <parte> y < /parte>) están bien anidados (todo elemento, excepto el raíz, está completamente contenido dentro de otro) y los valores de los atributos están entre comillas (también podrían estar entre comillas simples, como en ‘3’).

Para que un documento XML sea correcto, debe estar bien formado; pero eso no es suficiente; además, el documento debe ser *válido*. Un documento XML válido es aquel que cumple con un *esquema* particular. Un esquema nos dice la estructura que debe seguir un documento XML; hay varias formas de definir esquemas.

La forma más vieja de definir esquemas viene de tiempos de SGML y se llama DTD, Definición de Tipo de Documento (*Document Type Definition*). El DTD para los XML de libros sería como se ve en el ejemplo del listado 8.2.

Código 8.2 DTD para archivo de libro en XML

```
<?xml version="1.0" encoding="UTF-8"?>
<!ENTITY  iacute "&#262;">
<!ENTITY  oacute "&#268;">
...
<!ELEMENT libro ( titulo, autor+, parte+ ) >
<!ELEMENT titulo (#PCDATA) >
<!ELEMENT autor (#PCDATA) >
<!ELEMENT parte ( capitulo+  >)
<!ATTLIST parte numero CDATA #REQUIRED >
<!ELEMENT capitulo ( resumen, parrafo+ ) >
<!ATTLIST capitulo numero CDATA #REQUIRED >
<!ELEMENT resumen ( #PCDATA ) >
<!ELEMENT parrafo ( #PCDATA ) >
```

El DTD sencillamente nos dice qué puede contener cada elemento; también puede definir las entidades válidas en el documento. El elemento libro tiene un titulo, uno o más

autores (autor+) y una o más partes (parte+). El DTD también nos dice si un atributo es obligatorio (#REQUIRED) u opcional.

El problema con los DTDs es que son poco flexibles y que no soportan varias características específicas de XML que SGML no tiene (como espacios de nombres). Sin embargo, para documentos con estructuras sencillas, los DTDs cumplen razonablemente el trabajo.

Otra forma de definir esquemas es con XML Schema. Sin embargo, los esquemas de este estilo son algo más complejos que los DTDs; por razones de espacio no los veremos aquí.

Una gran ventaja que ofrece XML para manipular información, es que ya existen las herramientas necesarias para manejar los documentos, en casi cualquier lenguaje de programación existente. Comprobar que un documento XML esté bien formado y sea válido es muy sencillo, por lo que el programador ya sólo necesita encargarse de extraer y manipular la información contenida en el documento. Para esto último también ya existen las herramientas necesarias.

Una característica de XML es que, como su nombre lo dice, es *extensible*. Supongamos que tenemos un programa que lee el siguiene archivo:

Código 8.3 Descripción de un producto en XML.

```
<?xml version="1.0" encoding="UTF-8"?>
<producto>
 <nombre>
  Aletas de caucho
 </nombre>
 <precio>
   197.50
 </precio>
 <marca>
   Speed
 </marca>
<producto>
```

Y queremos agregar al listado del producto alguna otra característica. Por ejemplo, una categoría. Entonces debemos modificar el archivo para que se vea siguiente forma:

Código 8.4 Descripción de un producto en XML (extendida).

```
<?xml version="1.0" encoding="UTF-8"?>
<producto>
 <nombre>
  Aletas de caucho
 </nombre>
 <precio>
   197.50
 </precio>
 <marca>
   Speed
 </marca>
 <categoria>
   Deportes
 </categoria>
<producto>
```

Aunque el archivo es diferente, el programa seguirá reconociéndolo como lo hacía antes. Pero por supuesto, no será capaz de leer la categoria.

Es importante notar que un documento XML es inútil por sí mismo; es necesario tener un programa que sepa leerlo. En otras palabras, XML se utiliza únicamente para transportar información, y no tiene comportamiento definido.

8.2. HTML

HTML (*Hypertext Markup Language* o *Lenguaje de Marcas de Hipertexto*) nació casi al mismo tiempo que la World Wide Web, de la necesidad de poder presentar información en la red de forma rápida y sencilla. SGML era muy complejo para lo que se requería y XML todavía no se inventaba; así que HTML fue el resultado que se obtuvo para resolver este problema. Se utilizó una estructura similar a la de un lenguaje SGML, pero mucho menos estricta y con bastante tolerancia a fallos.

Eso, aunado a que al inicio no había un estándar propiamente sino sólo una serie de reglas no muy claramente especificadas, hizo que HTML terminara siendo un lenguaje poco consistente y demasiado permisivo en su manejo de errores (en general los navegadores desplegaban una página sin importar cuántos errores tuviera el documento HTML).

Con la fundación del Consorcio de la World Wide Web (*World Wide Web Consortium*), o W3C, se empezaron a corregir muchos de los problemas que inicialmente tenía HTML. Se creó el estándar XHTML, una versión más estricta de HTML que extiende a XML. Al ser una extensión a XML, los documentos en XHTML deben estar bien formados, y por tanto, es más fácil encontrar y corregir errores en ellos.

Por un tiempo, la W3C se enfocó en el desarrollo de XHTML, mientras que otro equipo se enfocaba en extender HTML. Eventualmente, la W3C se dio cuenta de que esta no era la mejor idea, y a pesar de todo el esfuerzo que se invirtió en XHTML, se decidió que el camino a seguir era seguir con el desarrollo de HTML.Esto fue principalmente para juntar esfuerzos con los desarrolladores de HTML; mucha gente prefería este proyecto por varias razones como simplicidad, facilidad de extensión, y retrocompatibilidad. La recomendación actual de la W3C es utilizar HTML 5.2.

HTML es un lenguaje de marcado que se utiliza para escribir páginas web. Básicamene consiste en una serie de etiquetas donde, de manera similar a XML, cada etiqueta representa un elemento en nuestra página. Por ejemplo, si queremos agregar una tabla a nuestra página web, podemos utilizar la etiqueta table.

El siguiente es un ejemplo de un documento en HTML.

Código 8.5 Documento en HTML

```
<!DOCTYPE html>
<html>
<head>
<title>Hola Mundo</title>
</head>
<body>

<h1>Encabezado del sitio</h1>
<p>Este es un ejemplo.</p>

</body>
</html>
```

En el documento anterior:

- La etiqueta <!DOCTYPE html> nos dice que el documento es de tipo HTML.
- La etiqueta <html> es llamada la etiqueta raíz y se recomienda (aunque no es necesario) ponerla siempre.
- La etiqueta <head> contiene información sobre el documento (en este caso, un título).
- La etiqueta <title> contiene un título para el documento.
- La etiqueta <body> contiene la parte visible de la página.
- La etiqueta <h1> define un encabezado.
- La etiqueta <p> define un párrafo.

En general, HTML es un lenguaje con un formato muy libre y poco estricto. Sin embargo, se recomienda siempre cerrar las etiquetas que se abren y siempre tener las etiquetas html, head y body.

8.3. CSS

Las Hojas de Estilo en Cascada (*Cascading Style Sheets*) es el medio a partir del cual se modifica cómo se ve un documento HTML; pero también se pueden utilizar para documentos XML arbitrarios o para documentos XHTML.
Veamos cómo funcionan siguiendo el ejemplo en el listado 8.6.

Código 8.6 Hoja de estilo en cascada (1/2)

```
<!DOCTYPE html>
<html>
 <head>
  <title>Pruebas de CSS</title>
 </head>
 <body>
  <h1>Pruebas de CSS</h1>
  <p>
   Texto normal.
   <em>Texto enfatizado.</em>
   <strong>Texto reforzado.</strong>
   Liga a <a href="http://www.google.com">Google</a>.
  </p>
  <h2>Lista no ordenada</h2>
  <ul>
   <li>Elemento 1</li>
   <li>Elemento 2</li>
   <li>Elemento 3</li>
   <li>Elemento 4</li>
  </ul>

  <h2>Lista ordenada</h2>
  <ol>
   <li>Elemento 1</li>
   <li>Elemento 2</li>
   <li>Elemento 3</li>
   <li>Elemento 4</li>
  </ol>
 </body>
</html>
```

Sin utilizar ninguna hoja de estilo, la página se vería como en la figura 8.1.

Figura 8.1 Página sin hoja de estilo

Pruebas de CSS

Texto normal. *Texto enfatizado*. **Texto reforzado.** Liga a Google.

Lista no ordenada

- Elemento 1
- Elemento 2
- Elemento 3
- Elemento 4

Lista ordenada

1. Elemento 1
2. Elemento 2
3. Elemento 3
4. Elemento 4

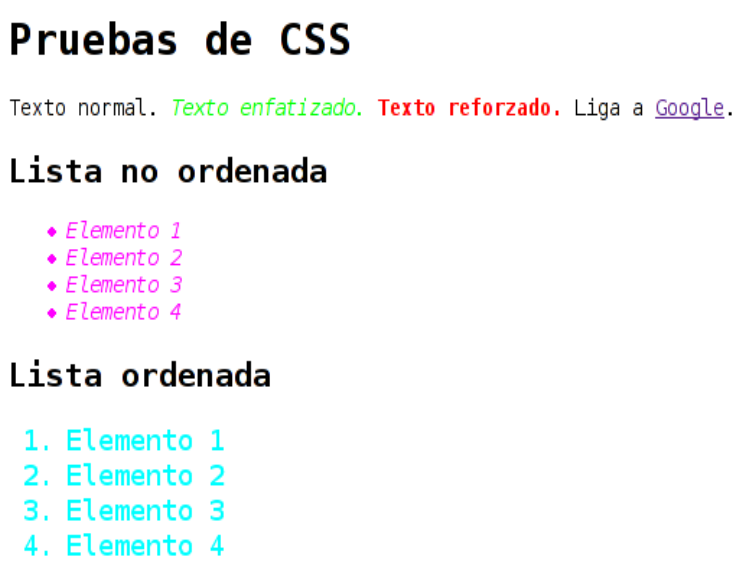
Sin modificar en *nada* el contenido del documento, podemos cambiar por completo cómo se ve, usando la hoja de estilo del listado 8.7.

Código 8.7 Documento en CSS

```
body {
    font-family: mono;
    font-size: 14px;
}
em {
    color: #0f0;
}
strong {
    color: #f00;
}
ul {
    color: #f0f;
    font-style: italic;
}
ol {
    color: #0ff;
    font-size: large;
}
```

Para que el documento HTML use la hoja de estilo, se utiliza el elemento link en la cabeza del mismo. El resultado puede apreciarse en la figura 8.2.

Figura 8.2 Página con hoja de estilo



Código 8.8 Documento en HTML con *link*

```
<!DOCTYPE html>
<html>
 <head>
  <title>Pruebas de CSS</title>
  <link rel="stylesheet" href="example.css" type="text/css" />
 </head>
```

Sin importar que tan bien o mal se vea la página, lo relevante es que su apariencia es completamente independiente de su contenido. El que lo primero sea ortogonal a lo segundo, permite a los creadores del contenido concentrarse sólo en la información y dejarle la presentación a diseñadores gráficos o a alguien que se encargue exclusivamente de eso (y así evitar que las páginas se vean como en el ejemplo).

Las hojas de estilo también sirven para poder determinar cómo se representará una página en distintos medios. Se puede utilizar una hoja de estilo para presentarla en un navegador normal; otra especialmente hecha pensando en impresoras; una más para lectores de pantalla (para usuarios ciegos o con visión limitada).

8.3.1. Javascript

JavaScript (que no tiene casi nada que ver con el lenguaje de programación Java) es un lenguaje de programación que se usa principalmente dentro de un navegador. Dado que es posible acceder al contenido de una página a través de JavaScript, es una opción muy sencilla para crear contenido dinámico y darle la posibilidad al usuario de interactuar con la página sin necesidad de comunicarse todo el tiempo con el servidor (porque el código JavaScript se ejecuta en el cliente, en el navegador).

Todo el contenido de una página está disponible a través del Modelo de Objeto del Documento (*Document Object Model* o DOM), que es básicamente una representación en memoria de la jerarquía tipo árbol que tiene un documento HTML o XML.

Por ejemplo, si tenemos la página en el listado 8.9,

Código 8.9 Documento en HTML

```
<!DOCTYPE html>
<html>
 <head>
  <title>JavaScript</title>
  <script type="text/javascript" src="javascript.js" />
 </head>
 <body>
  <h1>JavaScript</h1>
   <table id="myTable">
    <tr>
     <td><strong>Elementos</strong></td>
     <td><strong>N&uacute;mero</strong></td>
    </tr>
   </table>
   <input type="button" value=" Agregar rengl&oacute;n "
                    onclick="addRow(document);" />
   <input type="button" value=" Borrar rengl&oacute;n "
                    onclick="delRow(document);" />
 </body>
</html>
```

y el archivo javascript.js tiene definidas las funciones del listado 8.10.

Código 8.10 Documento en Javascript

```
function addRow(document)
{
        var table    = document.getElementById('myTable');
        var numRows = table.rows.length;
        var newRow   = table.insertRow(numRows);
        var cell1    = newRow.insertCell(0);
        var cell2    = newRow.insertCell(1);

        cell1.innerHTML = 'Nuevo Elemento';
        cell2.innerHTML = numRows;
}
function delRow(document)
{
        var table    = document.getElementById('myTable');
        var numRows = table.rows.length;
        if (numRows > 1)
                table.deleteRow(numRows-1);
}
```

Entonces cada vez que se haga click en el botón “Agregar renglón”, un nuevo renglón aparecerá en la tabla; y cada vez que se haga click en “Borrar renglón”, se le quitará un renglón a la tabla.

El primer botón de la página tiene definido (usando el atributo onclick) que cada vez que se le haga click, se mande llamar a la función addRow con el parámetro document. En JavaScript, document es una variable que representa a todo el documento HTML o XML. El segundo botón tiene definido de forma análoga que se mande llamar la función delRow.

La función addRow hace lo siguiente: primero obtiene el objeto de la tabla usando la función getElementById. Ésta es una función del DOM y nos permite obtener cualquier elemento, siempre y cuando tengamos su identificador (en la página, al definir la tabla usamos el atributo id).

Después, obtiene cuántos renglones tiene la página e inserta un nuevo renglón al final; los renglones se numeran a partir del 0, por lo que si hay n renglones están numerados de 0 a $n - 1$. Después añade dos celdas al renglón y define el código HTML de cada una de ellas.

La función delRow también obtiene el objeto de la tabla; si hay al menos dos renglones borra el último (esto es para no quedarnos con una tabla sin renglones).

Este ejemplo tan sencillo permite ver el poder que tiene JavaScript: nos permite modificar dinámicamente el documento HTML y XML, y de esta forma interactuar con el usuario sin necesidad de comunicarnos con el servidor, porque todo ocurre del lado del cliente (el navegador).