Una Introducción

a la Programación en Emacs Lisp

Una Introducción a la Programación en Emacs Lisp

Revisada la tercera edición

Escrito por Robert J. Chassell. Traducido por David Arroyo Menéndez.

Esto es una *Introducción a la Programación en Emacs Lisp*, para personas que no son programadoras.

Edición 3.10, 28 October 2009

Publicado por:

Libremanuals, http://www.libremanuals.net/

Resumen del Contenido

Índice General

Prefacio

La mayoría del entorno integrado GNU Emacs está escrito en el lenguaje de programación llamado Emacs Lisp. El código escrito en este lenguaje de programación es el software—el conjunto de instrucciones—que cuenta al ordenador qué hacer cuando tu le das comandos. Emacs está diseñado de forma que se puede escribir nuevo código en Emacs Lisp y fácilmente instalarlo como una extensión al editor.

(GNU Emacs es algunas veces llamado un "editor extensible", pero hace mucho más que proporcionar capacidad de edición. Es mejor referirse a Emacs como un "entorno de computación extensible". Sin embargo, esta frase es un poco pretenciosa. Es más fácil referirse a Emacs simplemente como un editor. De hecho, cada cosa que se hace en Emacs—encontrar la fecha Maya y fases de la luna, simplificar polinomios, depurar código, administrar ficheros, leer cartas, escribir libros—todas estas actividades son maneras de editar tal y como se entiende en el mundo.)

Aunque Emacs Lisp normalmente se asocia solo con Emacs, es un lenguaje de programación completo. Se puede usar Emacs Lisp como harías con cualquier otro lenguaje de programación.

Quizás se quiere comprender la programación; quizás se quiere extender Emacs; o quizás se quiere llegar a ser un programador. Esta introducción a Emacs Lisp está diseñada para ayudar a empezar: para guiarse en el aprendizaje de los fundamentos de programación, y de manera más importante, para enseñar como uno mismo puede ir más allá.

Leyendo este texto

A través de este libro, se verá un pequeño ejemplo de programas que se pueden ejecutar dentro de Emacs. Si se lee este documento en Info dentro de GNU Emacs, se pueden ejecutar los programas como ellos aparecen. (Esto es fácil de hacer y se explica cuando los ejemplos se presentan). Alternativamente, se puede leer esta introducción como un libro impreso mientras se está sentando con un ordenador ejecutando Emacs. (Esto es lo que me gusta hacer; me gustan los libros impresos.) Si no se está ejecutando Emacs, todavía se puede leer este libro, pero en este caso, lo mejor es tratarlo como una novela, o como una guía para un país no visitado todavía: interesante, pero no es lo mismo allí.

Mucha de esta introducción se dedica a paseos guiados de código usado en GNU Emacs. Estos paseos están diseñados para dos propósitos: primero, familiarizarse con código real que funciona (código que se usa cada día); y, segundo, familiarizarse con la manera con la que Emacs funciona. Es interesante ver cómo un entorno en funcionamiento se implementa. También, espero que se escoja el hábito de navegar a través del código fuente. Se puede aprender comparando código de otros con el propio para ganar ideas. Tener GNU Emacs es como tener una cueva del dragón de los tesoros.

Además de aprender acerca de Emacs como un editor y Emacs Lisp como un lenguaje de programación, las guías de ejemplos guiados darán una oportunidad para familiarizarse con Emacs como un entorno de programación Lisp. GNU Emacs soporta programación y provee herramientas que llegarán a ser confortables usando

cosas como `M-.` (la clave que invoca el comando `find-tag`). También se aprenderś acerca de búffers y otros objetos que son parte del entorno. Aprendiendo acerca estas funcionalidades de Emacs es como aprender nuevas rutas alrededor de tu hogar.

Finalmente, son convenientes algunas habilidades usando Emacs para aprender aspectos de programación que no se conocen. Con frecuencia se puede usar Emacs para ayudar a comprender qué puzzles encontrar o como hacer alguna cosa nueva. Esta auto-confianza no es solo un placer, también es una ventaja.

Para quien está esto escrito

Este texto está escrito como una introducción elemental para personas que no son programadoras. Quien ya programa, puede no estar satisfecho con este libro. La razón es que puede tener que llegar a ser experto leyendo manuales de referencia y ese no es el camino para el que este texto está organizado.

Un programador experto que revisó este texto me dijo a mí:

> *Yo prefiero aprender desde manuales de referencia. Yo "escarbo" cada párrafo, y "vienen por aire" entre párrafos.*
>
> *Cuando llego al fin de un párrafo, asumo que este asunto está hecho, finalizado, que conozco cada cosa que necesito (con la posible excepción del caso en el que el siguiente párrafo empieza hablando acerca de eso en más detalle). Yo espero que un manual de referencia bien escrito no tendrá un montón de redundancia, y tendrá excelentes punteros a (un) lugar donde la información que quiero.*

¡Esta introducción no está escrita para esta persona!

Primeramente, intento decir cada cosa al menos tres veces: primero, introducirlo; segundo, mostrarlo en contexto; y tercero, mostrarlo en un contexto diferente, o revisarlo.

Segundo, yo no siempre pongo toda la información acerca de un asunto en un lugar, ni mucho menos en un párrafo. Desde mi punto de vista, se necesita una carga bastante fuerte en el lector. En vez de eso intento explicar solo lo que se necesita saber en el momento. (Algunas veces incluyo una pequeña información extra así no se sorprenderá más tarde cuando la información adicional se presente formalmente.)

Cuando uno lee este texto, no espera aprender todo la primera vez. Frecuentemente, solo necesita hacer un 'reconocimiento' con alguno de los elementos mencionados. Mi esperanza es que haber estructurado el texto y dar suficientes indicios que dejarán alerta de lo que es importante y concentrarse en ellos.

Es necesario "sumergirse" de algunos párrafos; no hay otro modo de leerlos. Pero yo he intentado guardar el número de tales párrafos. Este libro pretende ser como una colina que se acerca, en vez de una montaña.

Esta introducción de *Programación en Emacs Lisp* viene acompañado de un documento complementario. *El Manual de Referencia de GNU Emacs*. El manual de referencia tiene más detalles que esta introducción. En el manual de referencia, toda la información sobre un asunto está concentrado en un lugar. Se debe cambiar si se es como el programador citado arriba. Y, de acuerdo, después de que se ha leido

esta *Introducción*, se encontrará el *Manual de Referencia* útil cuando se escriben propios programas.

Historia de Lisp

Lisp fué desarrollado primero en los 50 en el Instituto Tecnológico de Massachusetts para investigar en inteligencia artificial. El gran poder del lenguaje Lisp lo hace superior para otros propósitos también, tal como escribir comandos de edición y entornos integrados.

GNU Emacs Lisp está fuertemente inspirado en Maclisp, que está escrito en el MIT en los sesenta. Está en cierto modo inspirado en Common Lisp, que llega a ser un estándar en los 80. Sin embargo, Emacs Lisp es mucho más simple que Common Lisp. (La distribución estándar de Emacs contiene un fichero de extensiones opctional, `cl.el`, que añade muchas funcionalidades a Emacs Lisp.)

Una nota para principiantes

Aunque no se conozca GNU Emacs, se puede leer este documento de manera útil. Sin embargo, es mejor aprender Emacs, al menos aprender a moverse alrededor de la pantalla del ordenador. Uno puede enseñarse a sí mismo cómo usar Emacs con el tutorial on-line. Para usarlo, se debe escribir *`C-h t`*. (Esto significa que se presione la tecla `CTRL` y la *`h`* al mismo tiempo, y después se presiona *`t`*

También, con frecuencia, me refiero a uno de los comandos de Emacs estándar listando las teclas que se presionan para invocar el comando y entonces dar el nombre del comando entre paréntesis, como este: *`M-C-\`* (`indent-region`). (Si lo deseas, se pueden cambiar las teclas que son escritas para invocar el comando; esto se llama *rebinding*. Véase Sección 16.8 “Mapas de Teclado”, página 199.) La abreviación *`M-C-\`* significa que se escribe la tecla `META`, `CTRL`, y `\` todo al mismo tiempo. (En muchos teclados modernos la tecla `META` es etiquetada con `ALT`.) Algunas veces una combinación como esta es llamada keychord, puesto que es similar al camino de tocar un acorde en un piano. Si tu teclado no tiene una tecla `META`, la tecla con prefijo `ESC` es usada en lugar de la otra. En este caso *`M-C-\`* significa que se presiona `ESC` y entonces escribe `CTRL` y la tecla `\` al mismo tiempo. Pero normalmente *`M-C-\`* significa presionar la tecla `CTRL` alrededor con la tecla que está etiquetada `ALT` y, al mismo tiempo, se presiona la tecla `\`.

Además de escribir una sola tecla, se puede prefijar lo que se escribe con *`C-u`*, que es llamado el ‘argumento universal’. El atajo *`C-u`* pasa a ser un argumento para el comando subsiguiente. De este modo, para indentar una región de texto plano a 6 espacios, se marca la región, y entonces se escribe *`C-u 6 M-C-\`*. (Si no se especifica un número, Emacs pasa el número 4 al comando o de otra manera ejecuta el comando de manera diferente). Véase Sección “Argumentos Numéricos” in *El Manual de GNU Emacs*.

Si se está leyendo esto en Info usando GNU Emacs, se puede leer a través de este documento completo solo presionando la barra de espacio, `SPC`. (Para aprender acerca de Info, escribe *`C-h i`* y luego selecciona Info.)

Una nota en terminología: cuando yo uso la palabra Lisp sola, con frecuencia me estoy refiriendo a los dialectos de Lisp en general, pero cuando yo hablo de Emacs Lisp, yo estoy refiriéndome a GNU Emacs Lisp en particular.

Se agradece

Gracias a todos quienes me ayudaron con este libro. Especialmente agradecido a Jim Blandy, Noah Friedman, Jim Kingdon, Roland McGrath, Frank Ritter, Randy Smith, Richard M. Stallman, and Melissa Weisshaus. Gracias también a Philip Johnson y David Stampe por su for ánimo paciente. Mis errores son míos.

Gracias al equipo inicial de traductores al español Antonio Barrones, Suso Pereira y David Vázquez Pérez por el primer impulso.

Robert J. Chassell
bob@gnu.org

1 Procesamiento de listas

Para las personas nuevas, Lisp es un lenguaje de programación extraño. En código Lisp hay paréntesis en cada lugar. Algunas personas incluso reclaman que el nombre signfica 'Lots of Isolated Silly Parentheses' ('Montones de Paréntesis Aislados Estúpidos'). Pero la advertencia es sin garantías. Lisp es para procesamiento de listas, y el lenguaje de programación maneja *listas* (y listas de listas) poniéndolas entre paréntesis. Los paréntesis marcan los límites de la lista. Algunas veces una lista es precedida por un apóstrofe simple o una marca de cita, '' [1] Las listas son las bases de Lisp.

1.1 Listas Lisp

En Lisp, una lista como esta: `'(rosa violeta margarita mantequilla)`. Esta lista es precedida por un apóstrofe simple. Podría estar bien escrita como sigue, que mira más como el tipo de lista con la que se está familiarizado:

```
'(rosa
  violeta
  margarita
  mantequilla)
```

Los elementos de esta lista son los nombres de las 4 flores diferentes, separadas por espacios en blanco y alrededor de paréntesis, como flores en un campo con un muro de piedras alrededor de ellas.

Las listas pueden también tener números dentro, como en esta lista: `(+ 2 2)`. Esta lista tiene un signo más, '`+`', seguido por dos '`2`', cada uno separado por espacios en blanco.

En Lisp, datos y programas están representados por el mismo camino; que son ambas listas de palabras, números, u otras listas, separadas por espacios en blanco y alrededor de paréntesis. (Desde que un programa mira como datos, un programa puede fácilmente servir como datos para otros; esta es una funcionalidad muy poderosa de Lisp.) (Incidentalmente, estas dos marcas de paréntesis *no* son listas Lisp, porque contienen '`;`' y '`.`' como marcas puntuación.)

Aquí está otra lista, esta vez con una lista dentro:

```
'(esta lista tiene (una lista dentro de ella))
```

Los componentes de esta lista son las palabras '`esta`', '`lista`', '`tiene`', y la lista '`(una lista dentro de ella)`'. La lista interior es hecha de las palabras '`una`', '`lista`', '`dentro`', '`de`', '`ella`'.

1.1.1 Átomos Lisp

En Lisp, hemos estado llamando a las palabras *átomos*. Este término viene desde el significado de la palabra átomo, que significa indivisible. Tan lejos como Lisp es concebido, las palabras que hemos estado usando en las listas no pueden ser

[1] Los apóstrofes simples o las marcas de citas son una abreviación para la función `quote`; tu no necesitas pensar acerca de funciones ahora; las funciones son definidaas en Sección 1.3 "Generar un mensaje de error", página 4.

divididas dentro de pequeñas partes y todavía significan la misma cosa como parte de un programa; como con números y caracteres de símbolos simples como '`+`'. Por otro lado, no como un antiguo átomo, una lista puede ser dividida en pequeñas partes. Ver Sección "`car`" in *Funciones Fundamentales*.

En una lista, los átomos son separados de otros por espacios en blanco. Ellos pueden estar a la derecha y cerca de un paréntesis.

Técnicamente hablando, una lista en Lisp consiste de paréntesis alrededor de átomos separados por espacios en blanco o alrededor de otras lista o alrededor de ambos átomos u otras listas. Una lista puede tener solo un átomo en ella o no tener nada en ella para todo. Una lista con nada dentro se ve como esto: `()`, y está llamado la *lista vacía*. No como cualquier otra cosa, un lista vacía es considerad ambos un átomo y una lista al mismo tiempo.

La representación impresa de ambos átomos y listas se llaman *expresiones simbólicas* o, más concisamente, *s-expresiones*. La palabra *expresión* por sí mismo puede referir o bien la representación impresa, o para el átomo o la lista como ella es manejada internamente en el ordenador. Con frecuencia, la personas usan el término *expresión* indiscriminadamente. (También, en muchos textos, la palabra *forma* se usa como un sinónimo para la expresión.)

Incidentalmente, los átomos que hacen que nuestro universo fuese nombrado así cuando ellos fueron pensados para ser indivisibles; pero han sido encontrados átomos físicos que no son indivisibles. Las partes pueden dividir un átomo o puede fisionarse en 2 partes de igual tamaño. Los átomos físicos fueron nombrados prematuramente, antes de que su verdadera naturaleza fuese encontrada. En Lisp, ciertos tipos de átomos, como un array, pueden ser separados en partes; pero el mecanismo de hacer esto es diferente de el mecanismo para dividir una lista. Tan lejos como las operaciones de las listas son concebidas, los átomos de una lista son indivisibles.

Como en Inglés, los significado de las letras de componentes de un átomo Lisp son diferentes desde el significado de las letras que crean una palabra. Por ejemplo, la palabra Suramericana '`ai`', es completamente diferente de las dos palabras '`a`', e '`i`'.

Hay muchos tipos de átomos naturales, pero solo unos pocos en Lisp: por ejemplo, *números*, tales como 37, 511, o 1729, y *símbolos*, tales como '`+`', '`foo`', o '`forward-line`'. Las palabras que hemos listado en los ejemplos de debajo son todos símbolos. Cada día de conversación Lisp, la palabra "átomo" no se usa con frecuencia, porque los programadores normalmente intentan ser más específicos acerca de que tipo de átomo están tratando. La programación Lisp es sobre todo de símbolos (y algunas veces números) con listas. (De ese modo, tres palabras rodeadas de paréntesis son una apropiada lista en Lisp, desde que ello consiste en átomo, que en este caso son símbolos, separados por espacios en blanco y cerrados por paréntesis, sin cualquier puntuación no Lisp.)

Texto entre comillas — incluso frases de párrafos — son también un átomo. Aquí hay un ejemplo:

```
'(esta lista incluye "texto entre comillas.")
```

En Lisp, todo el texto citado incluyendo la marca de puntuación y los espacios en blanco son un átomo simple. Este tipo de átomo es llamado *string* (por 'cadena de caracteres') y es el tipo de cosa que es usada para mensajes que un ordenador puede imprimir para que un humano lea. Las cadenas son un tipo diferente de átomo en vez de números, o símbolo y son usados de manera diferente.

1.1.2 Espacios en blanco en listas

La cantidad de espacios en blanco en una lista no importa. Desde el punto de vista del lenguaje Lisp,

```
'(esta lista
   parece esto)
```

es exactamente lo mismo que esto:

```
'(esta lista parece esto)
```

Ambos ejemplos muestran que Lisp es la misma lista, la lista hecha de los símbolos '`esta`', '`lista`', '`parece`', y '`esto`' en este orden.

Espacios en blanco extra y nuevas líneas son diseñadas para crear una lista más legible por humanos. Cuando Lisp lee la expresión, asimila los espacios en blanco extra (pero necesita tener al menos un espacio entre átomos en orden para contarlos aparte.)

Aunque parezca raro, los ejemplos que hemos visto cubren casi todo lo que Lisp tiene. Cualquier otra lista en Lisp ve más o menos como uno de estos ejemplos, excepto que la lista puede ser más larga y más compleja. En resumen, una lista está entre paréntesis, una cadena está entre comillas, un símbolo parece como una palabra, y un número parece un número. (Para ciertas situaciones, corchetes, puntos y otros caracteres especiales pueden ser usados; sin embargo; iremos bastante lejos sin ellos.)

1.1.3 GNU Emacs te ayuda a escribir listas

Cuando se escribe una expresión Lisp en GNU Emacs usando bien el modo de Interacción Lisp o el modo Emacs Lisp, están disponibles varios comandos para formatear la expresión Lisp, de modo que sea fácil de leer. Por ejemplo, presionando la tecla `TAB` automáticamente se indenta la línea del cursor que está en la cantidad correcta. Un comando para indentar apropiadamente el código en una región está asociado a `M-C-\`. La indentación está diseñada de manera que se pueden qué elementos de una lista pertenencen a qué lista — los elementos de una sublista están más indentados que los elementos de una lista cerrada.

Además, cuando se escribe un paréntesis de cierre, Emacs momentáneamente salta el cursor atrás para hacer el matching (emparejamientos) con el paréntesis de apertura, para ver cuál es. Esto es muy útil, ya que cada lista que se escribe en Lisp debe tener sus paréntesis emparejados con sus paréntesis de apertura. (Ver Sección "emacs" in `El Manual de GNU Emacs`, para más información acerca de modos de Emacs.)

1.2 Ejecutar un programa

Una lista en Lisp —cualquier lista— es un programa listo para ser ejecutado. Si lo ejecutas (lo que la jerga Lisp llama *evaluar*), el ordenador hará una de las tres cosas: nada excepto devolverte la lista misma; enviar un mensaje de error; o, tratar el primer símbolo en la lista como un comando para hacer alguna cosa. (¡Normalmente, es el último de estas tres cosas de lo que realmente se quiere!).

El apóstrofe, `'`, que se pone enfrente de algún ejemplo de listas en secciones precedentes se llama *quote*; (comilla); cuando precede una lista, Lisp no tiene que hacer nada con la lista, otra que la toma como está escrita. Pero si no hay una cita precediendo la lista, el primer ítem de la lista es especial: es un comando para que el ordenador obedezca. (En Lisp, estos comandos son llamados *funciones*.) La lista `(+ 2 2)` muestra debajo que no tuvo un quote en frente, así Lisp comprende que `+` es una instrucción para hacer alguna cosa con el resto de la lista: añadir los números que siguen.

Si estás leyendo esto dentro de GNU Emacs en Info, aquí está como puedes evaluar una lista: posiciona tu cursor de manera inmediata después de la siguiente lista y escribe *C-x C-e*:

```
(+ 2 2)
```

Verás que el número `4` aparece en el área echo. (La jerga que se usa es "evaluar la lista." El área echo es la línea arriba de la pantalla que muestra o hace "echo" del texto.) Ahora intenta la misma cosa con una lista entre comillas: posiciona el cursor correcto después de la siguiente lista y escribe *C-x C-e*:

```
'(esto es una lista comilla)
```

Tu verás `(esto es una lista citada)` aparece en el área echo.

En ambos casos, lo que estás haciendo es dar un comando al programa dentro de GNU Emacs llamado *intérprete Lisp* — dando al intérprete un comando para evaluar la expresión. El nombre del intérprete Lisp viene de la palabra para la tarea hecha por un humano que viene con el significado de una expresión — quien lo "interpreta".

También se puede evaluar un átomo que no es parte de una lista — uno que no está rodeado por paréntesis; de nuevo, el intérprete Lisp traduce desde la expresión humanamente legible al lenguaje del ordenador. Pero antes de discutir esto (ver Sección 1.7 "Variables", página 9), nosotros discutiremos lo que el intérprete de Lisp hace cuando tu creas el error.

1.3 Generar un mensaje de error

No se preocupe si genera un mensaje de error de manera accidental, se dará un comando para que el intérprete de Lisp lo genere. Esto es una actividad sin daño; y en efecto, con frecuencia se intenta generar mensajes de error de manera intencional. Una vez se comprende la jerga, los mensajes de error pueden ser informativos. En vez de ser llamados mensajes de "error", deberían ser llamados mensajes de "ayuda". Esto son como signos para un viajero en un país extraño; descifrarlos puede ser duro, pero una vez comprendidos, pueden encontrar el camino.

El mensaje de error está generado por un depurador de GNU Emacs. Se 'introduce el depurador'. Se obtiene el depurador escribiendo `q`.

Lo que se hace es evaluar una lista que no está citada y no tiene un comando con significado como su primer elemento. Aquí hay una lista casi exacta a la usada, pero sin la cita simple en vez de eso. Posicione el cursor derecho después y escribe *C-x C-e*:

```
(esto es una lista sin cita)
```

Una ventana `*Backtrace*` se abrirá y se verá lo siguiente:

```
---------- Buffer: *Backtrace* ----------
Debugger entered--Lisp error: (void-function this)
  (this is an unquoted list)
  eval((this is an unquoted list))
  eval-last-sexp-1(nil)
  eval-last-sexp(nil)
  call-interactively(eval-last-sexp)
---------- Buffer: *Backtrace* ----------
```

Su cursor estará en esta ventana (puede tener que esperar unos pocos segundos antes de llegar a ser visible). Para salir del depurador y de su ventana, escribe:

```
q
```

Por favor, escriba *q* bien ahora, así llegará a ser confidente de que se pueda bajar el depurador. Entonces, escribe *C-x C-e* de nuevo y se reintroduce.

Basado en lo que ya se sabe, se puede leer este mensaje de error.

Si se lee el búffer `*Backtrace*` desde arriba; cuenta lo que Emacs hace. Cuando se escribe *C-x C-e*, se hizo una llamada interactiva el comando `eval-last-sexp`. `eval` es una abreviación para 'evaluar' y `sexp` es una abreviación para la 'expresión simbólica'. El comando significa 'evalúa la última expresión simbólica', que es la expresión ahora antes de tu cursor.

En cada línea de abajo se cuenta lo que evaluó el siguiente intérprete. La acción más reciente está alta. El búffer es llamado `*Backtrace*` porque te permite habilitar la traza de Emacs hacia atrás.

Arriba del búffer `*Backtrace*`, se verá la línea:

```
El depurador introdujo el error Lisp: (void-function this)
```

El intérprete Lisp intentó evaluar el primer átomo de la lista, la palabra '`this`'. Esta acción que generaba el mensaje de error '`void-function this`'.

El mensaje contiene las palabras '`void-function`' y '`this`'.

La palabra '`function`' fué mencionada antes. Es una palabra muy importante. Para nuestros propósitos, se puede definir eso diciendo que una *función* (*function*) es un conjunto de instrucciones para el ordenador que cuenta al ordenador hacer alguna cosa.

Ahora se puede empezar a comprender el mensaje de error: '`void-function this`'. La función (que es, la palabra '`this`') no tiene una definición de cualquier conjunto de instrucciones del ordenador para llevarlo.

La palabra ligeramente extraña, '`void-function`', está diseñada para cubrir el camino en que Emacs Lisp está implementado, que es cuando un símbolo no tiene una definición de función adjunta a eso, el lugar que contendría las instrucciones es 'vacío'.

Por otro lado, desde que fuimos capaces de añadir 2 más 2 de manera exitosa, evaluando `(+ 2 2)`, se puede inferir que el símbolo `+` debe tener un conjunto de instrucciones para que el ordenador obedezca y estas instrucciones deben añadir los números que siguen el `+`.

Es posible prevenir que Emacs entre en el depurador en casos como este. No se explicará cómo hacer esto aquí, pero se mencionará que el resultado se parece, porque se puede encontrar una situación similar si hay un error en algún código Emacs que se está usando. En tales casos, se verá una línea del mensaje de error; que aparecerá en el área echo y parecerá así:

```
La definición de la función de símbolos está
vacío: this
```

El mensaje aparece tan pronto se escribe una tecla, incluso para mover el cursor.

Conocemos el significado de la palabra '`Símbolo`'. Se refiere al primer átomo de la lista, la palabra '`este`'. La palabra '`función`' se refiere a las instrucciones que cuentan al ordenador que hacer. (Técnicamente, el símbolo cuenta al ordenador donde encontrar las instrucciones, pero esto es una complicación que podemos ignorar por el momento.)

El mensaje de error puede ser comprendido: '`La definición del símbolo está vacío: este`'. El símbolo (que es, la palabra '`este`') le faltan instrucciones para que el ordenador funcione.

1.4 Nombres de símbolos y definiciones de funciones

Se puede articular otra característica de Lisp basada en lo que se discutió hace tiempo—una característica importante: un símbolo, como `+`, no es en sí mismo el conjunto de instrucciones que el ordenador transmite. En vez de eso, el símbolo está usado, quizás temporalmente, como un camino de localizar la definición o conjunto de instrucciones. Lo que se ve es el nombre a través del que las instrucciones se pueden encontrar. Los nombres de personas trabajan por el mismo camino. Puede ser llamado '`Bob`'; sin embargo, no soy '`B`', '`o`', '`b`' pero soy, o fuí, la consciencia consistente asociada con una forma de línea particular. El nombre no soy yo, pero puedo ser usado para referirme a mí.

En Lisp, un conjunto de instrucciones puede adjuntarse a varios nombres. Por ejemplo, las instrucciones de ordenador para añadir números pueden ser enlazados al símbolo `más` tan bien como el símbolo `+` (y son en algunos dialectos de Lisp). Entre humanos, puede referirse a '`Robert`' tan bien como '`Bob`' y en otras palabras también.

Por otro lado, un símbolo puede tener solo una función adjunta en un momento. De otro modo, el ordenador estaría confundido acerca de qué definición usar. Si este fuera el caso, solo una persona en el mundo podría llamarse '`Bob`'. Sin embargo, la definición de función para la que el nombre se refiere puede ser cambiada de manera legible. (Ver Sección 3.2 "Instalar una Definición de Función", página 28.)

Puesto que Emacs Lisp es grande, es costumbre nombrar símbolos de un modo que identifique la parte de Emacs de la función que pertenece. De este modo, todos los nombres de funciones que tratan con Texinfo empiezan con '`texinfo-`' y estas funciones que tratan con la lectura de correo empiezan con '`rmail-`'.

1.5 El intérprete Lisp

Basado en lo que se ha visto, ahora se puede empezar a entender lo que el intéprete Lisp hace cuando ejecutamos un comando para evaluar una lista. Primero, examina si hay un símbolo quote (cita) antes de la lista; si el intérprete da la lista. Por otro lado, si no hay cita, el intéprete mira si el primer elemento en la lista tiene una definición de función. De otro modo, el intérprete imprime un mensaje de error.

Así es como Lisp trabaja. Simple. Hay complicaciones añadidas que tendremos en un minuto, pero estas son fundamentales. De acuerdo, para escribir programas Lisp, se necesita conocer como escribir definiciones de función y adjuntarlas a nombres, y como hacer esto sin confundirnos a nosotros mismos o al ordenador.

Ahora, una primera complicación. Además de las listas, el intérprete Lisp puede evaluar un símbolo sin el símbolo cita (quote) y no tiene paréntesis alrededor. El intérprete Lisp intentará determinar el valor del símbolo como una *variable*. Esta situación está descrita en la sección acerca de variable. (Ver Sección 1.7 "Variables", página 9.)

La segunda complicación ocurre debido a que algunas funciones son inusuales y no funcionan de la manera usual. Estas que no son llamadas *formas especiales*. Son usadas para trabajos especiales, como definir una función, y no son muchas de ellas. En los siguientes próximos capítulos, se introducirán varias de las formas especiales más importantes.

La tercera y final complicación es la siguiente: si la función que el intérprete Lisp está buscando no es una forma especial, y si eso es parte de una lista, el intérprete Lisp quiere ver si la lista tiene una lista dentro. Si hay una lista dentro, el intérprete Lisp primero mira qué hacer con la lista dentro, y entonces trabaja en la lista de fuera. Si todavía hay otra lista embebida dentro de la propia lista, eso funciona en esta primero, y así. Eso siempre funciona en la lista más interior. El intérprete funciona más interior primero, para evaluar el resultado de esta lista. El resultado puede ser usado por la expresión entre paréntesis.

De otra manera, el intérprete trabaja de izquierda a derecha, desde una expresión a la siguiente.

1.5.1 Compilación de bytes

Otro aspecto de interpretación: el intérprete Lisp es capaz de interpretar dos tipos de entidad: código humanemente legible, en el que focalizarse exclusivamente, y especialmente código procesado, llamado *byte compilado*, que no es humanamente legible. El código máquina compilado se ejecuta más rápido que el código humanamente legible.

Tu puedes transformar código legible por humanos dentro de código compilado ejecutando un de los comandos compilados tales como `byte-compile-file`. El código compilado es normamelte almacenado en un fichero que finaliza con una extensión `.elc` en vez de una extensión `.el`. Verás ambos tipos de ficheros en el directorio `emacs/lisp`; los ficheros para leer estos son con extensiones `.el`.

Como una cuestión práctica, para la mayoría de las cosas tu podrías personalizar o extender Emacs, no necesitas compilar byte; y no discutirás el asunto aquí. Ver

Sección "Compilación de Byte" in *El Manual de Referencia de GNU Emacs*, para una completa descripción de compilación byte.

1.6 Evaluación

Cuando el intérprete Lisp funciona en una expresión, el término para la actividad es llamada *evaluación*. Decimos que el intérprete 'evalúa la expresión'. Yo he usado este término varias veces antes. La palabra viene desde su uso en el lenguaje de cada día, 'para cierto valor o cantidad de; para estimar' de acuerdo a *Webster's New Collegiate Dictionary*.

Después de evaluar una expresión, el intérprete Lisp normalmente *devuelve* el valor que el ordenador produce trayendo las instrucciones encontradas en la definición de la función, o quizás dará esta función y producirá un mensaje de error. (El intérprete puede también quedarse colgado, así hablar, a una función diferente o puede intentar repetir continuamente que está haciendo para siempre y siempre en lo que está llamado como un 'bucle infinito'. Estas acciones son menos comunes; y pueden ignorarse). Más frecuentemente, el intérprete devuelve un valor.

Al mismo tiempo el intérprete devuelve un valor, puede hacer cualquier cosa más también, tal como mover un cursor o copiar un fichero; este otro tipo de acción es llamada *efecto lateral*. Acciones que los humanos pensamos que son importantes tales como imprimir resultados son, con frecuencia, "efectos laterales" al intérprete Lisp. La jerga puede sonar peculiar, pero cambiar que es fácilmente fácil aprender a usar efectos laterales.

En resumen, evaluando una expresión simbólica normalmente causa que el intérprete devuelva un valor y quizás trajo un efecto lateral; o al menos produce un error.

1.6.1 Evaluando listas propias

Si la evaluación se aplica a una lista que está dentro de una lista de fuera, se puede usar el valor devuelto por la primera evaluación como información cuando la lista de fuera está evaluada. Esto explica por qué las expresiones propias son evaluadas primero: los valores devueltos son usados por las expresiones de fuera.

Nosotros podemos investigar este proceso evaluando otro ejemplo adicional. Deja tu cursor después de la siguiente expresión y escribe `C-x C-e`:

```
(+ 2 (+ 3 3))
```

El número 8 aparecerá en el área echo.

Lo que ocurre es que el intérprete Lisp primero evalúa la expresión propia, `(+ 3 3)`, para que el valor 6 se devuelva; entonces evalúa la expresión de fuera como si fuera escrita `(+ 2 6)`, que devuelve el valor 8. Puesto que no hay más expresiones cerradas para evaluar el intérprete imprime este valor en el área echo.

Ahora es fácil comprender el nombre de los comandos invocados por atajos `C-x C-e`: el nombre es `eval-last-sexp`. Las letras `sexp` son una abreviación para la 'expresión simbólica', y `eval` es una abreviación para 'evaluar'. El comando significa 'evaluar la última expresión simbólica'.

Como un experimento, tu puedes intentar evaluar la expresión poniendo el cursor al principio de la siguiente línea inmediatamente siguiendo la expresión, o dentro de la expresión.

Aquí hay otra copia de la expresión:

```
(+ 2 (+ 3 3))
```

Si se posiciona el cursor al principio de la línea en blanco que inmediatamente sigue la expresión y escribes `C-x C-e`, todavía se obtendrá el valor 8 impreso en el área echo. Ahora intenta poner el cursor dentro de la expresión. Si se pone bien después del siguiente al último paréntesis (así aparece para situarse arriba del último paréntesis), ¡se obtendrá un 6 impreso en el área echo! Esto es porque el comando evalúa la expresión `(+ 3 3)`.

Ahora se pone el cursor inmediatamente después de un número. Escribe `C-x C-e` y se tendrá el número en sí. En Lisp, si evalúas un número, tu tienes el número en sí—esto es cómo los números difiere desde los símbolos. Si evalúas una lista empezando con un símbolo como `+`, tendrás un valor devuelto que es el resultado del ordenador trayendo las instrucciones en la definición de función adjunta a este nombre. Si un símbolo por sí mismo es evaluado, alguna cosa diferente ocurre, como veremos en la siguiente sección.

1.7 Variables

En Emacs Lisp, un símbolo puede tener un valor adjunto como puede tener una definición de función adjunta. Las dos son diferentes. La definición de función es un conjunto de instrucciones que un ordenador obedece. Un valor, por otro lado, es alguna cosa como un número o un nombre, que puede variar (que es porque tal símbolo es llamado variable). El valor de un símbolo puede ser una expresión en Lisp, tal como un símbolo, número, lista, o cadena. Un símbolo que tiene un valor es con frecuencia llamado una *variable*.

Un símbolo puede tener ambos una definición de función y un valor adjunto al mismo tiempo. O puede tener solo uno u otro. Los dos están separados. Esto es algo similar al camino, el nombre Cambridge puede referirse a la ciudad en Massachusetts y tener alguna información adjunta al nombre tan bien, tal como "gran centro de programación".

Otro camino para pensar acerca de esto es imaginar un símbolo como ser una caja de cajones. La definición de función es poner en el cajón que maneja el valor que puede ser cambiado sin afectar los contenidos del cajón que maneja la definición de función, y viceversa.

La variable `fill-column` ilustra un símbolo con un valor adjunto: en cada buffer de GNU Emacs, este símbolo establece algún valor, normalmente 72 o 70, pero algunas veces algún otro valor. Para encontrar el valor de este símbolo, evalúalo por sí mismo. Si estás leyendo esto en Info dentro de GNU Emacs, tu puedes hacer esto poniendo el cursor después del símbolo y escribiendo `C-x C-e`:

```
fill-column
```

Después de que yo escribiera `C-x C-e`, Emacs imprimió el número 72 en mi área echo. Este es el valor por el que `fill-column` es escogido para mi, porque yo lo escribo. Puede ser diferente para ti en tu búffer Info. Sepa que el valor devuelto como una

variable es impreso exactamente por el mismo camino que el valor devuelto por una función trayendo sus instrucciones. Puesto que el punto de vista del intérprete Lisp, es un valor devuelto. El tipo de expresión viene de ceder a la cuestión una vez el valor se conoce.

Un símbolo puede tener cualquier valor adjunto a ello o, usar la jerga, se puede *bind* (asociar) la variable a un valor: a un número, tal como 72; a una cadena, `\"tal como esta\"`; a una lista, tal como `(abeto pino roble)`; podemos incluso asociar una variable a una definición de función.

Un símbolo puede ser emparejado por un valor en varios caminos. Ver Sección 1.9 “Configurando el valor de una variable”, página 16, para información acerca de un camino para hacer esto.

1.7.1 Mensaje de error para un símbolo sin una función

Cuando se evalúa `fill-column` para encontrar el valor de una variable, no se ponen paréntesis alrededor de la palabra. Esto es porque no pretendemos usarlos como un nombre de función.

Si `fill-column` fuese el primer o único elemento de una lista, el intérprete de Lisp intentaría encontrar la definición de función adjunta. Pero `fill-column` no tiene definición de función. Prueba evaluando esto:

```
(fill-column)
```

Se creará un buffer `*Backtrace*` que dice:

```
---------- Buffer: *Backtrace* ----------
Debugger entered--Lisp error: (void-function fill-column)
  (fill-column)
  eval((fill-column))
  eval-last-sexp-1(nil)
  eval-last-sexp(nil)
  call-interactively(eval-last-sexp)
---------- Buffer: *Backtrace* ----------
```

(Recuerda, para salir del depurador y crear la ventana del depurador para ir fuera, escribe `q` en el `*Backtrace*` buffer.)

1.7.2 Mensaje de error para un símbolo sin un valor

Si intentas evaluar un símbolo que no tiene un valor asociado, recibirás un mensaje de error. Puedes ver esto experimentando con nuestra suma 2 más 2. En la siguiente expresión, pon tu cursor correcto después de `+`, antes del primer número 2, escribe *C-x C-e*:

```
(+ 2 2)
```

En GNU Emacs 22, se creará un buffer `*Backtrace*` que dice:

```
---------- Buffer: *Backtrace* ----------
Debugger entered--Lisp error: (void-variable +)
  eval(+)
  eval-last-sexp-1(nil)
  eval-last-sexp(nil)
  call-interactively(eval-last-sexp)
---------- Buffer: *Backtrace* ----------
```

(De nuevo, se puede salir del depurador escribiendo `q` en el búffer `*Backtrace*`.)

Esta traza es diferente desde los muy primeros mensajes de error que vimos, que dijimos, '`El depurador introdujo--errores Lisp: (esta función está vacía)`'. En este caso, la función no tiene una valor como una variable; mientras en el otro mensaje de error, la función (la palabra 'this') no tuvo una definición.

En este experimento con el `+`, lo que se hizo fué causar que el intérprete Lisp evalúe el `+` y busque el valor de la variable en vez de la definición de la función. Nosotros hicimos esto dejando el cursor correcto después del símbolo en vez de después de los paréntesis que cierran la lista como hicimos antes. Como consecuencia, el intérprete Lisp evaluó la s-expresión precedente, que en este caso fué el `+` en sí.

Desde que `+` no tiene un valor asociado, solo la definición de función, el mensaje de error reportado que el valor del símbolo como una variable estaba vacío.

1.8 Argumentos

Para ver cómo la información pasa a funciones, se permite mirar de nuevo a nuestro viejo standby, la adición de dos más dos. En Lisp, esto es escrito como sigue:

```
(+ 2 2)
```

Si se evalúa esta expresión, el número 4 aparecerá en tu área echo. Lo que el intérprete de Lisp hace es añadir los número que sigue el `+`.

Los números añadidos por `+` son llamados *argumentos* de la función `+`. Estos números son la información que es dada para o *pasada* a la función.

La palabra 'argumento' viene del ámbito de las matemáticas y no se refiere a una disputa entre 2 personas, sino que se refiere a la información presentada a la función, en este caso, al `+`. En Lisp, los argumentos a una función son los átomos o listas que siguen la función. Los valores devueltos por la evaluación de estos átomos o listas son pasados a la función. Funciones diferentes requieren diferentes números de argumentos; algunas funciones no requieren ninguno más.[2]

[2] Es curioso trazar la ruta por la que la palabra 'argumento' viene para tener dos significados diferentes, uno en matemáticas y el otro en el inglés de cada día. De acuerdo al *Oxford English Dictionary*, la palabra deriva del Latín para '`clarificar`', de este modo significa, por un hilo de derivación, viene a significar 'asertir de una manera contra otro que puede crear un contador de aserciones', que lidera el significado de la palabra como una disputa. (Nótese aquí que la palabra Inglés tiene dos definiciones diferentes adjuntas al mismo tiempo. En contraste, en Emacs Lisp, un símbolo no puede tener dos definiciones de funciones diferentes al mismo tiempo.)

1.8.1 Tipos de argumentos de datos

Los tipos de datos que deberían ser pasados a una función dependen de que tipo de información usan. Los argumentos a una función tales como `+` deben tener valores que son números, puesto que `+` añade números. Otras funciones usan diferentes tipos de datos para sus argumentos.

Por ejemplo, la función `concat` enlaza o une dos o más cadenas de texto para producir una cadena. Los argumentos son cadenas. La concatenación de los dos caracteres de cadenas `abc`, `def` producen la cadena simple `abcdef`. Esto puede ser visto evaluando lo siguiente:

```
(concat "abc" "def")
```

El valor producido para evaluar esta expresión es `"abcdef"`.

Una función tal como `substring` usa como argumentos tanto una cadena como números. La función devuelve una parte de la cadena, una subcadena del primer argumento. Esta función toma tres argumentos. Su primer argumento es la cadena de caracteres, el segundo y tercer argumento son números que indican el principio y el fin de la subcadena. Los números son un contaje del número de caracteres (incluyendo espacios y puntuaciones) desde el principio de la cadena.

Por ejemplo, si evalúa lo siguiente:

```
(substring "El rápido zorro marrón saltó." 12 17)
```

se verá `"zorro"` en el área echo. Los argumentos son la cadena y los dos números.

Nótese que la cadena pasada a `substring` es un átomo simple incluso aunque sea hecho de varias palabras separadas por espacios. Lisp cuenta cada cosa entre dos marcas de citas como parte de la cadena, incluyendo los espacios. Se puede pensar la función `substring` como una forma de 'despedazar átomos' ya que toma un átomo indivisible y extrae una parte. Sin embargo, `substring` es solo capaz de extraer una subcadena desde un argumento que es una cadena, no otro tipo de átomo tal como un número o símbolo.

1.8.2 Un argumento como el valor de una variable o lista

Un argumento puede ser un símbolo que devuelva un valor cuando es evaluado. Por ejemplo, cuando el símbolo `fill-column` por sí mismo es evaluado, devuelve un número. Este número puede ser usado en una adición.

Posicionar el cursor después la siguiente expresión y escribe *C-x C-e*:

```
(+ 2 fill-column)
```

El valor será un número dos más que tu tienes evaluando `fill-column` solo. Para mí, este es 74, porque mi valor de `fill-column` es 72.

Como nosotros hemos visto, un argumento puede ser un símbolo que devuelve un valor cuando se evalúa. Además, un argumento puede ser una lista que devuelve un valor cuando es evaluada. Por ejemplo, en la siguiente expresión, los argumentos para la función `concat` son las cadenas `"Los"` y `" zorros rojos."` y la lista `(number-to-string (+ 2 fill-column))`.

```
(concat "Los " (number-to-string (+ 2 fill-column)) " zorros rojos.")
```

Si evalúas esta expresión—y si, como con mi Emacs, `fill-column` evalúa a 72—`"Los 74 zorros rojos."` aparecerán en el área echo. (Nótese que deben ponerse espacio después de la palabra '`The`' y antes de la palabra '`red`' así aparecerán en

la cadena final. La función `number-to-string` convierte los enteros que la función de adición devuelve una cadena. `number-to-string` es también conocida como `int-to-string`.)

1.8.3 Número de variables de argumentos

Algunas funciones, tales como `concat`, `+`, o `*`, toman cualquier número de argumentos. (El `*` es el símbolo para multiplicar.) Esto puede ser visto evaluando cada uno de las siguientes expresiones en el camino usual. Que verás en el área echo que está impresa en este texto después de '⇒', que puedes leer como 'evaluar a'.

En el primer conjunto, las funciones no tienen argumentos:

```
(+)       ⇒ 0

(*)       ⇒ 1
```

En este conjunto, las funciones tienen un argumento cada una:

```
(+ 3)     ⇒ 3

(* 3)     ⇒ 3
```

En este conjunto, las funciones tienen tres argumentos cada una:

```
(+ 3 4 5) ⇒ 12

(* 3 4 5) ⇒ 60
```

1.8.4 Usando el tipo incorrecto de objeto como un argumento

Cuando a una función se le pasa un argumento del tipo incorrecto, el interpréte Lisp produce un mensaje de error. Por ejemplo, la función `+` espera los valores de sus argumentos para ser números. Como un experimento nosotros podemos pasar el símbolo citado `hola` en vez de un número. Posicionar el cursor después la siguiente expresión y escribir *C-x C-e*:

```
(+ 2 'hola)
```

Cuando se hace esto se generará un mensaje de error. Lo qué ha ocurrido es que `+` ha intentado añadir el 2 para el valor devuelto por `'hola`, pero el valor devuelto por `'hola` es el símbolo `hola`, no un número. Solo los números pueden ser añadidos. Así `+` podría no encarrilar su adición.

Se creará e introducirá un búffer `*Backtrace*` que dice:

```
---------- Buffer: *Backtrace* ----------
Debugger entered--Lisp error:
         (wrong-type-argument number-or-marker-p hello)
  +(2 hello)
  eval((+ 2 (quote hello)))
  eval-last-sexp-1(nil)
  eval-last-sexp(nil)
  call-interactively(eval-last-sexp)
---------- Buffer: *Backtrace* ----------
```

Como es normal, el mensaje de error intenta ser útil y tiene sentido después de que se aprenda cómo leerlo.[3]

La primera parte del mensaje de error es honesto; dice '`wrong type argument`' (mal tipo de argumento). Lo siguiente viene con la misteriosa palabra de jerga '`number-or-marker-p`'. Esta palabra está intentando contar qué tipo de argumento `+` se espera.

El símbolo `number-or-marker-p` dice que el intérprete Lisp está intentando determinar si la información presentada (el valor del argumento) es un número o una marca (un objeto especial representando una posición de buffer). Lo que hace es testear para ver si el `+` está siendo números dados para añadir. También chequea para ver si el argumento es algo llamado un marcador, que es una funcionalidad específica de Emacs Lisp. (En Emacs, las localizaciones en un búffer son grabadas como marcadores. Cuando la marca está asignado con el atajo *`C-@`* o *`C-SPC`*, su posición se guarda como un marcador. La marca puede ser considerada un número— el número de caracteres la localización está desde el principio del búffer.) En Emacs Lisp, `+` puede ser usado para añadir el valor numérico de posiciones de marca como números.

La '`p`' de `number-or-marker-p` es el cuerpo de una práctica iniciada en los primeros días de la programación Lisp. La '`p`' es para 'predicado'. En la jerga usada por los primeros investigadores de Lisp, un predicado se refiere a una función para determinar si alguna propiedad es verdadera o falsa. Así la '`p`' nos cuenta que `number-or-marker-p` es el nombre de una función que determina que el argumento dado es un número o una marca. Otros símbolos Lisp que finalizan en '`p`' incluyen `zerop`, una función que chequea si sus argumentos tienen el valor de cero, y `listp`, una función que prueba si sus argumentos son una lista.

Finalmente, la última parte del mensaje de error es el símbolo `hello`. Esto es el valor del argumento que pasaba a `+`. Si la adición había sido pasada al tipo correcto de objeto, el valor pasado habría sido un número, tal como 37, en vez de un símbolo como `hello`. Pero entonces tu no habrías obtenido el mensaje de error.

1.8.5 La función `message`

Como `+`, la función `message` toma un número variable de argumentos. Es usado para enviar mensajes para el usuario y es así tan útil que nosotros lo describiremos aquí.

Un mensaje está impreso en el área echo. Por ejemplo, se puede imprimir un mensaje en tu área echo evaluando la siguiente lista:

```
(message "¡Este mensaje aparece en el área echo!")
```

La cadena entera entre comillas dobles es un argumento simple y está impreso *en todo*. (Nótese que en este ejemplo, el mensaje en sí aparecerá en el área echo con comillas comillas dobles; que es porque tu ves el valor devuelto por la función `message`. En la mayoría de los usos de `message` en programas que se escribe, el texto será impreso en el área echo como un efecto lateral, sin las comillas. Ver Sección 3.3.1 "`multiply-by-seven` en detalle", página 30, para un ejemplo de esto.)

[3] `(quote hola)` es una expresión de la abreviación `'hola`.

Sin embargo, si hay un '`%s`' en la cadena citada de caracteres, la función `message` no imprime el '`%s`' como tal, pero mira al argumento que sigue la cadena. Eso evalúa el segundo argumento e imprime el valor en la posición en la cadena donde el '`%s`' está.

Puedes ver esto posicionando el cursor después de la siguiente expresión y escribiendo *C-x C-e*:

```
(message "El nombre de este búffer es: %s." (buffer-name))
```

En Info, `"El nombre de este búffer es: *info*."` aparecerá en el área echo. La función `buffer-name` devuelve el nombre del búffer como una cadena, que la función `message` inserta en lugar de `%s`.

Para imprimir un valor como un entero, usa '`%d`' en el mismo camino que '`%s`'. Por ejemplo, para imprimir un mensaje en el área echo que sitúa el valor de `fill-column`, evalúa lo siguiente:

```
(message "El valor de fill-column es %d." fill-column)
```

En mi sistema, cuando evalúo esta lista, `"El valor de fill-column es 72"` aparece en mi área echo[4]

Si hay más de una '`%s`' en la cadena citada, el valor del primer argumento siguiente la cadena citada es impresa en la posición del primer '`%s`' y el valor del segundo argunmento es impreso en la posición del segundo '`%s`', y así.

Por ejemplo, si se evalúa lo siguiente,

```
(message "¡Hay %d %s en la oficina!"
         (- fill-column 14) "elefantes rosas")
```

un mensaje característico aparecerá en el área echo. En mi sistema dice `"¡Hay 58 elefantes rosas en la oficina!"`

La expresión `(- fill-column 14)` está evaluado y el número resultante está insertado en lugar del '`%d`'; y la cadena entre dobles comillas, `"elefantes rosas"`, está tratado como un argumento simple e insertado en lugar del '`%s`'. (Esto es decir, una cadena entre comillas dobles se evalúa así mismo, como un número.)

Finalmente, aquí está un ejemplo algo complejo que no solo ilustra la computación de un número, pero también muestra como se puede usar una expresión que genere el texto que es sustituido por '`%s`':

```
(message "Él vió %d %s"
         (- fill-column 36)
         (concat (substring
                  "Los rápidos zorros marrones saltaron." 12 21)
                 "rojos trotando."))
```

En este ejemplo, `message` tiene tres argumentos: la cadena, `"Él vió %d %s"`, después la expresión, que empieza con la función `concat`. El valor resultante desde la evaluación de `(- fill-column 32)` está insertado en lugar de '`%d`'; y el valor devuelto por la expresión empieza con `concat` está insertado en lugar del '`%s`'.

Cuando se rellena la columna es 70 y se evalúa la expresión, el mensaje `"Se vieron 38 zorros rojos trotando."` aparece en tu área echo.

---

[4] Actualmente, puede usar `%s` para imprimir un número. Eso es no específico. `%d` imprime solo la parte de un número a la izquierda de un punto decimal, y no cualquier cosa que no es un número.

1.9 Configurando el valor de una variable

Hay varios caminos por el que una variable puede dar un valor. Uno de los caminos es usar la función `set` o la función `setq`. Otro camino es usar `let` (véase Sección 3.6 "let", página 33). (La jerga para este proceso es *bind* una variable para un valor.)

Las siguientes secciones no solo describen cómo `set` y `setq` funcionan pero también ilustran como los argumentos pasan.

1.9.1 Usando `set`

Para asignar el valor del símbolo `flores` a la lista `'(rosa violeta margarita mantequilla)`, evalúa la siguiente expresión posicionando el cursor después de la expresión y escribiendo *C-x C-e*.

```
(set 'flores '(rosa violeta margarita mantequilla))
```

La lista `(rosa violeta margarita mantequilla)` aparecerá en el área echo. Esto es lo que está *devuelto* por la función `set`. Como efecto lateral, el símbolo `flores` está asociado a la lista; esto es, el símbolo `flores`, que puede ser visto como una variable, que es dada por la lista como su valor. (Este proceso, ilustra un efecto lateral al intérprete Lisp: asignando el valor, que puede ser el efecto primario en el que los humanos estamos interesados. Esto es porque cada función Lisp debe devolver un valor si no se obtiene un error, pero solo tendrá un efecto colateral si está diseñado para tener uno.)

Después de evaluar la expresión `set`, se puede evaluar el símbolo `flores` y devolverá el valor asignado. Aquí está el símbolo. Emplaza tu cursor después y escribe *C-x C-e*.

```
flores
```

Cuando se evalúa `flores`, la lista `(rosa violeta margarita mantequilla)` aparece en el área echo.

Incidentalmente, si se evalúa `'flores`, la variable con una comilla en frente, se ve en el área echo y es el símbolo en sí: `flores`. Aquí está el símbolo citado, así se puede probar esto:

```
'flores
```

Nótese también, que cuando se usa `set`, se necesitan citar ambos argumentos a `set`, a menos que se quiera evaluar. Puesto que nosotros no queremos argumentos evaluados, ni la variable `flores`, ni la lista `(rosa violeta margarita mantequilla)`, se citan. (Cuando se usa `set` sin citar su primer argumento, el primer argumento se evalúa antes que cualquier cosa se haga. Si se hizo esto y `flores` no tenía un valor ya, se tendría un mensaje de error que el '`Valor de símbolo como variable esté vacío`'; por otro lado, si `flores` no devolvió un valor después de que fuera evaluado, el `set` intentaría establecer un valor después de que fuera devuelto. Hay situaciones donde esto es lo correcto para la función a hacer, pero tales situaciones son raras.)

1.9.2 Usando `setq`

Como materia práctica, casi siempre se cita el primer argumento a `set`. La combinación de `set` y un argumento citado primero es común que tiene su propio nombre:

la forma especial `setq`. Esta forma especial es solo como `set` excepto que el primer argumento se cita automáticamente, así no se necesita escribir la marca de cita por uno mismo. También, como una conveniencia añadida, `setq` permite asignar varias variables diferentes a diferentes valores, todo en una expresión.

Para asignar el valor de la variable `carnívoros` a la lista `'(leon tigre leopardo)` usando `setq`, la siguiente expresión que se usa es:

```
(setq carnivoros '(leon tigre leopardo))
```

Esto es exactamente lo mismo que usar `set` excepto que el primer argumento se cita automáticamente por `setq`. (El 'q' en `setq` significa `quote`.)

Con `set`, la expresión que se vería es:

```
(set 'carnivoros '(leon tigre leopardo))
```

También, `setq` puede ser usado para asignar diferentes valores a diferentes variables. El primer argumento está asociado al valor del segundo argumento, el tercer argumento se asocia al valor del cuarto argumento, y así. Por ejemplo, se podría usar el lo siguiente para asignar una lista de árboles al símbolo `trees` y una lista herbívoros al símbolo `herbivores`:

```
(setq arboles '(pino abeto encina roble)
      herbivoros '(gacela antilope cebra))
```

(La expresión podría también haber estado en una línea, pero podría no haberse ajustado en una página; y los humanos encuentran que es fácil leer listas formateadas.)

Aunque yo he estado usando el término 'assign', hay otro camino de pensar acerca de los trabajos de `set` y `setq`; y que es decir que `set` y `setq` creen el símbolo *point* a la lista. Este camino posterior de pensamiento es muy común y en los capítulos siguientes volveremos al menos a un símbolo que tiene un valor, específicamente una lista, adjunta; o, expresa otros caminos, el símbolo asigna a "apuntar" a la lista.

1.9.3 Contando

Aquí hay un ejemplo que muestra cómo usar `setq` en un contador. Se podría usar esto para contar cuantas veces una parte de un programa se repite por sí mismo. Primero asigna una variable a cero; entonces añade uno al número cada vez que el programa se repite así mismo. Para hacer esto, se necesita una variable que sirve como un contador, y dos expresiones: una expresión `setq` que asigna la variable contador a cero; y una segunda expresión `setq` que incrementa el contador cada vez que eso es evaluado.

```
(setq counter 0)                ; Se inicializa.

(setq counter (+ counter 1))    ; Se incrementa.

counter                         ; Contador.
```

(El siguiente texto el '`;`' son comentarios. Ver Véase Sección 3.2.1 "Cambiar una definición de función", página 29.)

Si evalúas la primera de estas expresiones, el inicializador, `(setq counter 0)`, y entonces evalúa la tercera expresión, `counter`, el número `0` aparecerá en el área echo. Si entonces se evalúa la segunda expresión, el incrementador, `(setq counter`

`(+ counter 1))`, el contador obtendrá el valor 1. Así si evalúas de nuevo `counter`, el número `1` aparecerá en el área echo. Cada vez que se evalúa la segunda expresión, el valor del contador será incrementado.

Cuando se evalúa el incrementador, `(setq counter (+ counter 1))`, el intérprete Lisp primero evalúa la lista interna; esta es la adición. En orden para evaluar esta lista, debe evaluar la variable `counter` y el número `1`. Cuando evalúa la variable `counter`, recibe su valor actual. Esto pasa su valor y el número `1` para el `+` que los añade juntos. La suma es devuelta como el valor de la list propia pasada al `setq` que asigna la variable `counter` a este nuevo valor. De este modo, el valor de la variable `counter`, se cambia.

1.10 Resumen

Aprender Lisp es como escalar una montaña en la que la primera parte es la empinada. Ahora has escalado la parte más difícil; y lo que queda es más fácil.

En resumen,

- Los programas Lisp son hechos de expresiones, que son listas o átomos simples.
- La listas son hechas de cero o más átomos o listas internas, separadas por espacios en blanco y rodeados de paréntesis. Una lista puede estar vacía.
- Los átomos son símbolos multi-carácter, como `forward-paragraph`, los símbolos de caracteres como `+`, cadenas de caracteres entre marcas de comillas dobles, o números.
- Un número se evalúa a sí mismo.
- Una cadena entre comillas dobles también se evalúa a sí mismo.
- Cuando se evalúa un símbolo a sí mismo, su valor se devuelve.
- Cuando se evalúa una lista, el intérprete mira el primer símbolo en la lista y en la definición de la función asociada a este símbolo. Así se traen las instrucciones en la definición función.
- Una marca de cita (quote) simple, `'` , cuenta al intérprete Lisp que devolvería la siguiente expresión como se escribe, y se evalúa como si la cita no estuviera.
- Los argumentos son la información pasada a una función. Los argumentos a una función son computados evaluando el resto de los elementos de la lista de la que la función es el primer elemento.
- Una función siempre devuelve un valor cuando se evalúa (a menos que obtenga un error); además, puede también traer alguna acción llamada un "efecto colateral". En muchos casos, un propósito de la función primaria es crear un efecto colateral.

1.11 Ejercicios

Unos pocos ejercicios simples:

- Genera un mensaje de error evaluando un símbolo apropiado que no está entre paréntesis.

- Genera un mensaje de error evaluando un símbolo apropiado que está entre paréntesis.
- Crea un contador que incremente por dos en vez de uno.
- Escribe una expresión que imprima un mensaje en el área cuando es evaluada.

2 Practicando evaluación

Antes de aprender como escribir una definición de función en Emacs Lisp, es útil gastar un poco de tiempo evaluando varias expresiones que ya han sido escritas. Estas expresiones serán listas con las funciones como su primer (y con frecuencia único) elemento. Desde alguna de las funciones asociadas con búffers son ambas simples e interesantes, nosotros empezaremos con estas. En esta sección, evaluaremos unas pocas de estas. En otra sección, estudiaremos el código de varios otros funciones relacionadas con búffers, para ver cómo fueron escritas.

Siempre y cuando se de un comando de edición a Emacs Lisp, tal como el comando para mover el cursor o el scroll de la pantalla, *se está evaluando una expresión,* el primer elemento del que es una función. *Así es cómo Emacs funciona.*

Cuando se escriben teclas, se causa que el intérprete Lisp evalúe una expresión que devuelve resultados. Incluso escribiendo texto plano se evalúa la función Emacs Lisp, en este caso, uno que usa `self-insert-command`, simplemente inserta el caracter que se escribió. Las funciones que se evalúan escribiendo atajos de teclado se llaman funciones *interactive*, o *commands*; como crear una fución interactive será ilustrada en el capítulo sobre cómo escribir definiciones de funciones. Ver Sección 3.3 “Creando una Función Interactive”, página 29.

Además de escribir comandos de teclado, hemos visto un segundo camino para evaluar una expresión: posicionar el cursor después de una lista y escribiendo `C-x C-e`. Esto es lo que nosotros haremos en el resto de esta sección. Hay otros caminos de evaluar una expresión también; que serán descritos como vienen.

Siendo usadas para evaluación práctica, las funciones mostradas en las siguientes pocas secciones son importantes en su propio derecho. Un estudio de estas funciones hace clara la distinción entre búffers y ficheros, cómo cambiar a un búffer, y como determinar una localización con ella.

2.1 Nombres de búffer

Las dos funciones, `buffer-name` y `buffer-file-name`, muestra la diferencia entre un fichero y un búffer. Cuando se evalúa la siguiente expresión, `(buffer-file-name)`, el nombre del fichero para el que el búffer se refiere aparece en el área echo. Normalmente, el nombre devuelto por `(buffer-name)` es lo mismo que el nombre del fichero para el que se refiere, y el nombre devuelto por `(buffer-file-name)` es la ruta completa del fichero.

Un fichero y un búffer son dos entidades diferentes. Un fichero es información grabada de manera permanente en el ordenador (a menos que tu lo borres). Un búffer, por otro lado, es información dentro de Emacs que evanecerá al final de la sesión de la edición (o cuando mates el búffer). Normalmente, un búffer contiene información que has copiado desde un fichero; nosotros decimos que el búffer está *visitando* este fichero. Esta copia es la que se trabaja y modifica. Los cambios al búffer no cambian el fichero, hasta ser guardados. Cuando se guarda el búffer, el búffer está copiado al fichero y está de este modo salvado de manera permanente.

Si estás leyendo esto en Info dentro de GNU Emacs, se puede evaluar cada una de las siguientes expresiones posicionando el cursor después de eso y escribiendo *C-x C-e*.

```
(buffer-name)

(buffer-file-name)
```

Cuando hago esto en Info, el valor devuelto para la evaluación de `(buffer-name)` es `"*info"`, y el valor devuelto por evaluar `(buffer-file-name)` es `nil`.

Por otro lado, mientras se está escribiendo este documento, el valor devuelto por la evaluación de `(buffer-name)` es `"introduction.texinfo"`, y el valor devuelto por la evaluación `(buffer-file-name)` es `"/gnu/work/intro/introduction.texinfo"`.

La forma es el nombre del búffer y lo posterior es el nombre del fichero. En Info, el nombre búffer es `"*info*"`. Info no apunta a cualquier fichero, así el resultado de evaluar `(buffer-file-name)`] es `nil`. El símbolo `nil` es desde la palabra latina para 'nada'; en este caso, significa que el búffer no está asociado con cualquier fichero. (En Lisp, `nil` también se usa con el significado de 'falso' y es sinómino de la lista vacía, `()`.)

Cuando estoy escribiendo, el nombre de mi búffer es `"introduction.texinfo"`. El nombre del fichero al que se apunta es `"/gnu/work/intro/introduction.texinfo"`.

(En las expresiones, los paréntesis hacen que el intérprete Lisp trate a `buffer-name` y `buffer-file-name` como funciones; sin los paréntesis, el intérprete intentaría evaluar los símbolos como variables. Ver Sección 1.7 "Variables", página 9.)

A pesar de la distinción entre ficheros y buffers, con frecuencia se encuentra que hay personas que se refieren a un fichero cuando quieren un búffer y al revés. En realidad, la mayoría de las personas dicen, "Yo estoy editando un fichero," en vez de decir, "Estoy editando un búfer que pronto se guardará en un fichero." Es casi siempre claro desde el contexto que las personas quieren decir. Al tratar con programas de ordenador, sin embargo, es importante tener la distinción en mente, ya que el ordenador no es una persona inteligente.

La palabra 'búffer', por el camino, viene desde el significado de la palabra como una cuña que mata la fuerza de una colisión. En los primeros ordenadores, un búffer acuñaba la interacción entre ficheros y la unidad de procesamiento central. Los tambores o cintas que manejan un fichero y la unidad de procesamiento central fueron piezas de equipamiento que fueron muy diferentes a cualquier otra, trabajando a sus propias velocidades. El búffer se hizo posible para ellos a trabajar juntos de manera efectiva. Finalmente, el búffer creció siendo uno intermedio, un lugar tomado temporalmente, para ser el lugar donde el trabajo está hecho. Esta transformación es como un pequeño puerto de mar que crece dentro de una gran ciudad: una vez fué el lugar donde el lugar donde la carga era depositada temporalmente antes de ser cargada dentro de los barcos; entonces eso llega a ser un negocio y centro cultural en su propio derecho.

No todos los búffers están asociados con ficheros. Por ejemplo, un búffer no visita cualquier fichero. De manera similar, un búffer `*Help*` no está asociado con cualquier fichero.

Antaño, cuando se perdía un fichero `~/.emacs` y se empezaba una sesión Emacs escribiendo el comando `emacs` solo, sin nombrar ficheros Emacs iniciaba con el búffer `*scratch*` visible. En estos días, se ve una página de inicio. Se puede seguir uno de los comandos sugeridos en dicha pantalla, visitar un fichero, o presionar la barra espaciadora para lograr el búffer `*scratch*`.

Si se cambia al búffer `*scratch*`, se escribe la posición del `(buffer-name)` del cursor después, y entonces escribe `C-x C-e` para evaluar la expresión. El nombre `*scratch*` será devuelto y aparecerá en el área echo. `*scratch*` es el nombre del búffer. Cuando escribes `(buffer-file-name)` en el búffer `*scratch*` y evalúa que, `nil` aparecerá en el área echo, solo se hace cuando evalúas `(buffer-file-name)` en Info.

Incidentalmente, si estás en el búffer `*scratch*` y quiere el valor devuelto por una expresión para aparecer en el búffer por sí mismo que en el área echo, escribe `C-u C-x C-e` en vez de `C-x C-e`. Esto causa el valor devuelto para aparecer después de la expresión. El búffer se verá así:

```
(buffer-name)"*scratch*"
```

No se puede hacer esto en Info desde que Info es de solo lectura y no se permitirá que se cambien los contenidos en el búffer. Pero puedes hacer esto en cualquier búffer que se puede editar; y cuando se escribe código de documentación (tal como este libro), esta funcionalidad es muy útil.

2.2 Obteniendo búffers

La función `buffer-name` devuelve el *name* del búffer; para obtener el búffer *en sí*, una función diferente es necesaria: la función `current-buffer`. Si usa esta función en el código, que tiene en el búffer en sí.

Un nombre y el objeto o entidad para el que el nombre se refiere son cosas diferentes. Tu no eres tu nombre, Tu eres una persona que otras personas se refieren por tu nombre. Si preguntas para hablar por Jorge y alguien escribe una carta con las letras '`J`', '`o`', '`r`', '`g`', y '`e`' escrito, tu podrías estar distraido, pero no estarías satisfecho. No quieres hablar al nombre, pero sí a la persona a la que el nombre se refiere. Un búffer es similar: el nombre del búffer scratch es `*scratch*`, pero el nombre no es el búffer. Para tener un búffer por sí mismo, se necesita usar una función tal como `current-buffer`.

Sin embargo, hay una ligera complicación: si evalúas `current-buffer` en una expresión por sí mismo, como haremos aquí, lo que ves es una representación impresa del nombre del búffer sin los contenidos del búffer. Emacs funciona en este camino por dos razones: el búffer puede contener miles de líneas—eso es demasiado largo para ser convenientemente mostrado; y, otro búffer puede tener los mismos contenidos pero un nombre diferente, y es importante distinguirlo.

Aquí hay una expresión conteniendo la función:

```
(current-buffer)
```

Si se evalúa esta expresión de Info en Emacs en el camino normal, `#<buffer *info*>` aparecerá en el área echo. El formato especial indica que el búffer por sí mismo está siendo devuelto, en vez de solo su nombre.

Incidentalmente, mientras se escribe un número o símbolo en un programa, no se puede hacer esto con la representación impresa del búffer: el único camino para tener un búffer por sí mismo es con una función tal como `current-buffer`.

Un función relacionada es `other-buffer`. Esto devuelve los buffers seleccionados más recientemente que los únicos en los que tu estás actualmente, no una representación impresa de su nombre. Si tu recientemente has cambiado fuera de `*scratch*` búffer, `other-buffer` devolverá este búffer.

Puedes ver esto evaluando la expresión:

```
(other-buffer)
```

Verás que `#<buffer *scratch*>` aparece en el área echo, o el nombre de cualquier otro búffer que se cambió recientemente[1]

2.3 Cambiando búffers

La función `other-buffer` actualmente proporciona un búffer cuando es usada como un argumento a una función que uno requiere. Podemos ver esto usando `other-buffer` y `switch-to-buffer` para cambiar al búffer diferente.

Pero primero, una breve introducción a la función `switch-to-buffer`. Cuando se cambia atrás y adelante desde Info al búffer para evaluar `(buffer-name)`, normalmente se escribe *`C-x b`* y se visualiza `*scratch*`[2] cuando se ve en el minibuffer el nombre del búffer al que se quiere cambiar. El atajo, *`C-x b`*, causa que el intérprete Lisp evalúe la función interactiva `switch-to-buffer`. Como nosotros dijimos antes, esto es como Emacs funciona: diferentes atajos de teclado llaman o ejecutan diferentes funciones. Por ejemplo, *`C-f`* llama `forward-char`, *`M-e`* llama a `forward-sentence` y así

Escribiendo `switch-to-buffer` en una expresión, y dándole un búffer para cambiar, se puede cambiar a búffers solo como *`C-x b`* hace.

```
(switch-to-buffer (other-buffer))
```

El símbolo `switch-to-buffer` es el primer elemento de la lista, así el intérprete Lisp tratará eso como una función y trae las instrucciones adjuntas a eso. Pero antes de hacer esto, el intérprete notará este `other-buffer` está dentro de paréntesis y trabaja en este símbolo primero. `other-buffer` es el primero (y en este caso, el único) elemento de esta lista, así el intérprete Lisp llama o ejecuta la función. Eso

---

[1] Actualmente, por defecto, si el búffer desde el que tu has cambiado es visible para tí en otra ventana, `other-buffer` elegirá el búffer más reciente que no puedes ver; esto es algo pequeño que he olvidado.

[2] O incluso, para cambiar, solo se necesita pulsar *`RET`* si el buffer por defecto era `*scratch*`, o si era diferente, entonces se puede escribir solo parte del nombre, tal como `*sc`, luego presiona la tecla *`TAB`* para causar expandir al nombre completo, y entonces escribe la tecla *`RET`*

devuelve otro búffer. Después, el intérprete ejecuta `switch-to-buffer`, pasando, como un argumento, el otro búffer, que es al que Emacs cambia. Si estás leyendo esto en Info, prueba esto ahora. Evalúa la expresión. (Para volver, escribe *C-x b RET*.)[3].

En los ejemplos de programación en secciones posteriores de este documento, tu verás la función `set-buffer` con más con frecuencia que `switch-to-buffer`. Esto es porque a diferencia entre los programas de ordenador y humanos: los humanos tienen ojos y esperan ver el búffer en el que ellos están trabajando en sus terminales de ordenador. Esto es así de obvio, casi va sin decirlo. Sin embargo, los programas no tienen ojos. Cuando un programa de ordenador trabaja en un búffer, que búffer no necesitan ser visibles en la pantalla.

`switch-to-buffer` está diseñado para humanos y hace dos cosas diferentes: cambia el búffer para el que la atención de Emacs está dirigida; y cambia el búffer mostrada en la ventana al nuevo búffer. `set-buffer`, por otro lado, hace solo una cosa: eso cambia la atención del programa del ordenador a un búffer. El búffer en la pantalla permanece sin cambios (de acuerdo, normalmente no ocurre nada hasta que el comando finaliza ejecutándose).

También, nosotros hemos introducido otro término de jerga, la palabra *llamada*. Cuando tu evalúas una lista en el que el primer símbolo es una función, tu estás llamando a esta función. El uso del término viene desde la noción de la función como una entidad que puede hacer alguna cosa para tí si tu la 'llamas' — es decir, es una entidad que puede arreglar un problema si le llamas a él o a ella.

2.4 Tamaño de búffer y la localización del punto

Finalmente, permítame en varias funciones simples, `buffer-size`, `point`, `point-min`, y `point-max`. Estas dan información acerca del tamaño de un búffer y la localización del punto con eso.

La función `buffer-size` te cuenta el tamaño del búffer actual; que es, la función un contaje del número de caracteres en el buffer.

```
(buffer-size)
```

Puedes evaluar esto en el camino usual, posicionando el cursor después de la expresión y escribiendo *C-x C-e*.

En Emacs, la posición actual del cursor es llamada *punto*. La expresión `(point)` devuelve un número que cuenta donde está localizado como un contaje del número de caracteres desde el principio del búffer al punto.

---

[3] Recuerda, esta expresión te permite cambiar a tus búffers más recientes y otros buffers que no puedes ver. Si realmente quieres ir a tus búffers seleccionados más recientemente, se necesita evaluar la siguiente expresión más compleja:

```
(switch-to-buffer (other-buffer (current-buffer)))
```

En este caso, el primer argumento a `other-buffer` cuenta de que búffer salir — el actual — y el segundo argumento cuenta al `other-buffer` es OK para cambiar a un búffer visible. En uso regular, `switch-to-buffer` toma a una ventana invisible desde usarías *C-x o* (`other-window`) para ir a otro búffer visible

Se puede ver el contaje de caracteres apuntar en este búffer evaluando la siguiente expresión en el camino normal:

```
(point)
```

Mientras escribo esto, el valor de `point` es 65724. La función `point` está frecuentemente usada en alguno de los ejemplos posteriores en este libro.

El valor del punto depende, de acuerdo, a la posición que tiene en el búffer. Si evalúas punto en este lugar, el número será largo:

```
(point)
```

Para mí, el valor del punto en esta posición es 66043, lo que significa que hay 319 caracteres (incluyendo espacios) entre las dos expresiones. (Sin duda, se verán diferentes números, puesto que se ha editado esto desde que se evaluó (point).)

La función `point-min` es similar a `point`, pero eso devuelve del valor mínimo permisible del punto en el búffer actual. Este es el número 1 a menos que *narrowing* esté en efecto. (Narrowing, *Encogiendo* es un mecanismo donde uno se puede encoger a uno mismo, o un programa, a operaciones en solo un parte de un búffer. Capítulo 6 “Encogiendo y extendiendo”, página 70.) Así, la función `point-max` devuelve el valor del valor máximo permisible del punto en el búffer actual.

2.5 Ejercicio

Encuentra un fichero con que tu estás trabajando y mueve hasta la mitad. Encuentra el nombre de búffer, el nombre del fichero, tamaño, y su posición en el fichero.

3 Cómo escribir definiciones de funciones

Cuando el intérprete evalúa una lista, parece ver si el primer símbolo en la lista tiene definición adjunta; o, poner otro camino, si el símbolo apunta a una definición de función. Si lo hace, el ordenador trae las instrucciones en la definición. Un símbolo que tiene una definición de función llamada, simplemente, una función (aunque apropiadamente hablando, la definición es la función y el símbolo se refiere a eso).

Todas las funciones están definidas en términos de otras funciones, excepto por unas nuevas funciones *primitivas* que son escritas en el lenguaje de programación C. Cuando se escriben definiciones de funciones, se escriben en Emacs Lisp y se usan otras funciones como bloques en construcción. Algunas de las funciones usadas en sí mismas están escritas en Emacs Lisp (quizás por tí) y algunas serán primitivas escritas en C. Las funciones primitivas están escritas en C así podemos fácilmente ejecutarlas en GNU Emacs en cualquier ordenador que tiene suficiente poder y puede ejecutar C.

Permíteme enfatizar esto: cuando se escribe código en Emacs Lisp, no se distinge entre el uso de funciones escritas en C y el uso de funciones escritas en Emacs Lisp. La diferencia es irrevelante. Yo menciono la distinción solo porque es interesante conocerla. A menos que se investigue, uno no se da cuenta si una función ya escrita es escrita en Emacs Lisp o C.

3.1 La forma especial `defun`

En Lisp, un símbolo tal como `mark-whole-buffer` tiene código adjunto que cuenta lo que el ordenador hace cuando la función es llamada. Este código es llamado la *definición de función* y es creado evaluando una expresión Lisp que empieza con el símbolo `defun` (que es una abreviación para *función define*). Porque `defun` no evalúa sus argumentos en el camino usual, eso se llama *forma especial*.

En secciones subsiguientes, miraremos en definiciones de función desde el código fuente Emacs, tales como `mark-whole-buffer`. En esta sección, describiremos una definición de función simple, así puedes ver como se ve. Esta definición de función usa aritmética porque es un ejemplo simple. Algunas personas no le gustan los ejemplos usando aritmética; sin embargo, si usted es tal persona, no se asuste. En realidad, cualquier código que se puede estudiar en esta introducción va a recordar a aritmética o matemáticas. Los ejemplos de manera mayoritaria involucran texto en un camino u otro.

Una definición de función tiene cinco partes siguiendo la palabra `defun`:

1. El nombre del símbolo para el que la definición de función sería adjunta.
2. Una lista de los argumentos que serán pasados a la función. Si no hay argumentos, tendremos una lista vacía, `()`.
3. Documentación describiendo la función. (Técnicamente opcional, pero fuertemente recomendada.)
4. Opcionalmente, una expresión para crear la función interactive así se puede usar escribiendo `M-x` y entonces el nombre de la función; o escribiendo una tecla apropiada o acorde.

5. El código que instruye el ordenador qué hacer: el *cuerpo* de la definición de función.

Es útil pensar las cinco partes de una definición de función siendo organizada en una plantilla, con slots para cada parte:

```
(defun function-name (arguments...)
  "documentacion-opcional..."
  (interactive argument-passing-info)      ; opcional
  body...)
```

Por ejemplo, aquí está el código para una función que multiplica sus argumentos por 7. (Este ejemplo no es interactivo. Ver Sección 3.3 "Creando una Función Interactive", página 29, para esta información.)

```
(defun multiply-by-seven (number)
  "Multiplicar NUMBER por siete."
  (* 7 number))
```

Esta definición empieza con un paréntesis y el símbolo `defun` seguido por el nombre de la función.

El nombre de la función está seguido por una lista que contiene los argumentos que serán pasados a la función. Esta lista es llamada por la *lista de argumentos*. En este ejemplo, la lista tiene solo un elemento, el símbolo `número`. Cuando la función es usada, el símbolo será asociado al valor que es usado como el argumento para la función

En vez de elegir la palabra `número` por el nombre del argumento, podría haber escogido cualquier otro nombre. Por ejemplo, podría haber elegido la palabra `multiplicando`. Yo escojo la palabra 'número' porque cuenta qué tipo de valor se pretende para este slot; pero yo podría haber elegido 'multiplicando' para indicar el rol que el valor emplaza en este slot jugará en los trabajos de la función. Yo podría haber llamado `foogle`, pero habría sido una mala elección porque no contaría qué significa. La elección del nombre es subir al programador y habría elegido crear el significado claro de la función.

En realidad, se puede elegir cualquier nombre que se desee para un símbolo en una lista de argumentos, incluso el nombre del símbolo usado en alguna otra función: el nombre a usar en una lista de argumentos es privado para esta definición particular. En esta definición, el nombre se refiere a una entidad diferente que cualquiera que usa el mismo nombre fuera de la definición de función. Supón que tienes un apodo 'corto' en tu familia; cuando tus miembros de familia se refieren a 'corto', significa el apodo. Pero fuera de tu familia, en una película, por ejemplo, el nombre 'corto' se refiere a alguien más. Porque un nombre en una lista de argumentos es privado para la definición de la función, se puede cambiar el valor de un símbolo dentro del cuerpo de una función sin cambiar su valor fuera de la función. El efecto es similar a este producido por una expresión `let`. (Ver sección Sección 3.6 "let", página 33.)

La lista de argumentos está seguida por la documentación que describe la función. Esto es lo que tu ves cuando tu escribes `C-h f` y el nombre de una función. Incidentalmente, cuando se escribe una documentación como esta, se haría la primera línea una frase completa desde algunos comandos, tal como `apropos`, imprime solo la primera línea de una documentación multi-línea. También, no indentaría la se-

gunda línea de una documentación, si tu tienes una, esto se ve cuando usas `C-h f` (`describe-function`). La documentación es opcional, pero es también útil, debería ser incluido en casi cualquier función que se escribe.

La tercera línea del ejemplo consiste en el cuerpo de la definición de función. (La mayoría de las definiciones de funciones, de acuerdo, son más largas que esto.) En esta función, el cuerpo es la lista, `(* 7 number)`, que dice multiplicar el valor de *número* por 7. (En Emacs Lisp, `*` es la función para la multiplicación, solo como `+` es la función de suma.

Cuando se usa la función `multiply-by-seven`, el argumento `number` evalúa para el número actual que quiere ser usada. Aquí hay un ejemplo que muestra como `multiply-by-seven` es usada; pero ¡no intentes evaluar esto primero!.

```
(multiply-by-seven 3)
```

El símbolo `número`, especificado en la definición de función en la siguiente sección, es dada o "emparejado a" el valor 3 en el uso actual de la función. Note que aunque `número` estaba dentro de paréntesis en la definición de función, el argumento pasado a la función `multiply-by-seven` no está entre paréntesis. Los paréntesis son escritos en la definición de función así el ordenador puede figurarse donde la lista de argumentos finaliza y el resto de la definición de función empieza.

Si se evalúa este ejemplo, se obtendrá un mensaje error. (¡Ve adelante, pruébalo!) Esto es porque hemos escrito la definición de función pero no le hemos contado todavía al ordenador la definición — no se ha instalado (o 'cargado') la definición de función en Emacs. Instalando una función es el proceso que cuenta al intérprete Lisp la definición de la función. La instalación se describe en la siguiente sección.

3.2 Instalar una definición de función

Si estás leyendo esto dentro de Info en Emacs, se puede probar la función `multiply-by-seven` evaluando primero la definición de función y entonces evaluando `(multiply-by-seven 3)`. Una copia de la definición sigue. Emplaza el cursor después del último paréntesis de la definición de función y escribe `C-x C-e`. Cuando se hace esto, `multiply-by-seven` aparecerá en el área echo. (Lo que significa es que cuando una definición de función es evaluada, el valor devuelto es el nombre de la función definida.) Al mismo tiempo, esta acción instala la definición de función.

```
(defun multiply-by-seven (number)
  "Multiplicar NUMBER por siete."
  (* 7 number))
```

Evaluando esta `defun`, se ha instalado `multiply-by-seven` en Emacs. La función es ahora solo una parte de Emacs como `forward-word` o cualquier otra editando la función que se usa. (`multiply-by-seven` estará instalada hasta que sales de Emacs. Para recargar código automáticamente siempre y cuando empieces Emacs, ver Sección 3.5 "Instalar Código Permanentemente", página 32.)

Se puede ver el efecto de instalar `multiply-by-seven` evaluando el siguiente ejemplo. Localiza el cursor después de la siguiente expresión y escribe `C-x C-e`. El número 21 aparacerá en el área echo.

```
(multiply-by-seven 3)
```

Si lo deseas, se puede leer la documentación para la función escribiendo *C-h f* (`describe-function`) y entonces el nombre de la función, `multiply-by-seven`. Cuando haces esto, una ventana `*Help*` aparecerá en tu pantalla que dice:

```
multiply-by-seven es una función Lisp.
(multiply-by-seven NUMBER)

Multiplicar NUMERO por siete.
```

(Para devolver a una ventana simple en tu pantalla, escribe *C-x 1*.)

3.2.1 Cambiar una definición de función

Si quieres cambiar a cambiar el código en `multiply-by-seven`, solo reescríbelo. Para instalar la nueva versión en lugar de la vieja, evalúa la definición de la función de nuevo. Así se cómo modifica el código en Emacs. Es muy simple,

Por ejemplo, se puede cambiar la función `multiply-by-seven` añade el número por sí mismo siete veces en vez de multiplicar el número por siete. Eso produce la misma respuesta, pero por una ruta diferente. Al mismo tiempo, añadiremos un comentario; un comentario es texto que el intérprete Lisp ignora, pero un lector humano puede encontrar útil o iluminante. El comentario es que esto es la "segunda versión".

```
(defun multiply-by-seven (number)       ; Segunda versión.
  "Multiplicar NUMERO por siete."
  (+ number number number number number number number))
```

El comentario sigue por un punto y coma, '`;`'. En Lisp cada cosa en una línea sigue un punto y coma que es un comentario. El final de la línea es el fin del comentario. Para estrechar un comentario a través de dos o más líneas, empieza cada línea con un punto y coma.

Véase Sección 16.3 "Empezando un Fichero `.emacs`", página 194, y Sección "Comentarios" in *El Manual de Referencia de GNU Emacs Lisp*, para más comentarios.

Se puede instalar esta versión de la función `multiply-by-seven` para evaluándolo en el mismo camino que se evaluó la primera función: deja el cursor después de los últimos paréntesis y escribe *C-x C-e*.

En resumen, esto es cómo se escribe código en Emacs Lisp: tu escribes una función; se instala; se testea; y entonces crea arreglos y mejoras e instálalas de nuevo.

3.3 Crear una función interactive

Se crea una función interactive emplazando una lista que empieza con la forma especial `interactive` inmediatamente después de la documentación. Un usuario puede invocar una función interactive escribiendo *M-x* y entonces el nombre de la función; o escribiendo las teclas para el que está emparejado, por ejemplo, escribiendo *C-n* para `next-line` o *C-x h* para `mark-whole-buffer`.

De manera interesante, cuando se llama a una función interactive interactivamente, el valor devuelto no está automáticamente mostrado en el área echo. Esto es porque con frecuencia se llama a una función interactive para sus efectos laterales,

tales como mover hacia adelante por una palabra o línea, y no para el valor devuelto. Si el valor devuelto fuera mostrado en el área echo cada vez que escribiste una tecla, distraería mucho

Tanto el uso de la forma especial `interactive` y un camino para mostrar un valor en el área echo puede ser ilustrada creando una versión interactiva de `multiply-by-seven`.

Aquí está el código:

```
(defun multiply-by-seven (number)       ; Versión Interactiva.
  "Multiplicar NUMERO por siete."
  (interactive "p")
  (message "El resultado es %d" (* 7 number)))
```

Se puede instalar este código emplazando tu cursor después y escribiendo *`C-x C-e`*. El nombre de la función aparecerá en tu área echo. Entonces, se puede usar este código escribiendo *`C-u`* y un número y entonces escribiendo *`M-x multiply-by-seven`* y presionando `RET`. La frase '`El resultado es ...`' seguido por el producto aparecerá en el área echo

Hablando más generalmente, invoca una función como esta dentro si de dos caminos:

1. Escribiendo un argumento prefijo que contienes el número para ser pasado, y entonces escribiendo *`M-x`* y el nombre de la función, como con *`C-u 3 M-x forward-sentence`*; o,
2. Escribe siempre la tecla/s de la función estén emparejadas, como con *`C-u 3 M-e`*.

Ambos ejemplos solo trabajan mencionados idénticamente para mover puntos hacia adelantes tres frases. (Desde `multiply-by-seven` no está emparejado a una techa, eso no podría ser usado como un ejemplo de emparejar la tecla.

(Véase Sección 16.7 "Algunos Atajos de Teclas", página 197, para aprender como emparejar un comando a una tecla.)

Un argumento prefijo está pasado para una función interactive escribiendo la tecla `META` seguido por un número, por ejemplo, *`M-3 M-e`*, o escribiendo *`C-u`* y entonces un número, por ejemplo, *`C-u 3 M-e`* (si se escribe *`C-u`* sin un número, por defecto a 4).

3.3.1 Un `multiply-by-seven` interactivo

Permite mirar el uso de la forma especial `interactive` y entonces en la función `message` en la versión interactiva de `multiply-by-seven`. Se rellamará que la definición función se ve como esto:

```
(defun multiply-by-seven (number)       ; Versión Interactiva.
  "Multiplicar NUMERO por siete."
  (interactive "p")
  (message "El resultado es %d" (* 7 number)))
```

En esta función, la expresión, `(interactive "p")`, es una lista de dos elementos. El `"p"` cuenta Emacs a pasar el argumento prefijo a la función y usar su valor para el argumento de la función.

El argumento será un número. Esto significa que el símbolo `number` será asociado a un número en la línea:

```
(message "El resultado es %d" (* 7 number))
```

Por ejemplo, si tu argumento prefijo es 5, el intérprete Lisp evaluará la línea como si fuera:

```
(message "El resultado es %d" (* 7 5))
```

(Si estás leyendo esto en GNU Emacs, se puede evaluar esta expresión por sí misma.) Primera, el intérprete evaluará la lista interna, que es `(* 7 5)`. Esto devuelve un valor de 35. Lo siguiente, evaluará la lista externa, pasando los valores de la segunda y subsiguientes elementos de la lista a la función `message`.

Como se ha visto, `message` es una función Emacs Lisp especialmente diseñada para enviar una línea de mensaje a un usuario. (Véase Sección 1.8.5 "La función `message`", página 14) En resumen, la función `message` imprime su primer argumento en el área echo como es, excepto para ocurrencia de '`%d`', o '`%s`' (y varios otras %-secuencias que no hemos mencionado). Cuando se ve una secuencia de control, la función mira al segundo argumento o subsiguiente e imprime el valor del argumento en la localización en la cadena donde la secuencia de control está localizada.

En la función interactiva `multiply-by-seven`, la cadena de control es '`%d`', que requiere un número, y el valor devuelto evaluando `(* 7 5)` es el número 35. Por consiguiente, el número 35 es impreso en lugar de '`%d`' y el mensaje es '`El resultado es 35`'.

(Nótese que cuando se llama a la función `multiply-by-seven`, el mensaje está impreso sin comillas, pero cuando se llama a `message`, el texto es impreso con dobles comillas. Esto es porque el valor devuelto por `message` aparece en el área echo cuando se evalúa una expresión cuyo primer elemente es `message`; pero cuando se embebió en una función, `message` se imprime el texto como un efecto lateral sin comillas.)

3.4 Opciones diferentes para `interactive`

En el ejemplo, `multipy-by-seven` usado `"p"` como el argumento a `interactive`. Este argumento contó a Emacs para interpretar tu escritura si `C-u` seguido por un número o `META` seguido por un número como un comando para pasar este número a la función como su argumento. Emacs tiene más de veinte caracteres predefinidos para usar con `interactive`. En casi cada caso, una de estas opciones te habilitará para pasar la información adecuada interactivamente a una función. (Véase Sección "Carácteres Código para `interactive`" in *El Manual de Referencia GNU Emacs Lisp*).

Considera la función `zap-to-char`. Su expresión interactiva es

```
(interactive "p\ncZap to char: ")
```

La primera parte del argumento para `interactive` es '`p`', con el que tu estás ya familiarizado. Este argumento cuenta a Emacs intepretar un 'prefijo', como un número para ser pasado a la función. Tu puedes especificar un prefijo si escribiendo `C-u` seguidos por un número o escribiendo `META` seguido por un número. El prefijo es el número de caracteres especificado. De este modo, si tu prefijo es tres y el caracter especificado es '`x`', entonces se borrará todo el texto e incluyendo el tercer

‘`x`’ siguiente. Si no se fija un prefijo, entonces borra todo el texto e incluye el carácter específico, pero no más.

El ‘`c`’ cuenta la función el nombre del carácter para que borre.

Más formalmente, una función con dos o más argumentos puede tener información pasado a cada argumento añadiendo partes para la cadena que sigue `interactive`. Cuando haces esto, la información está pasada para cada argumento en el mismo orden esto está especificado en la lista `interactive`. En la cadena, cada parte está separada desde la siguiente parte por un ‘`\n`’, que es una nueva línea. Por ejemplo, tu puedes seguir ‘`p`’ con un ‘`\n`’ y un ‘`cZap to char:`’. Esto causa que Emacs pase el valor del argumento prefijo (si hay uno) y el carácter.

En este caso, la definición de función mira como lo siguiente, donde `arg` y `char` son los símbolos para que `interactive` empareja el argumento y el caracter especificado:

```
(defun nombre-de-funcion (arg char)
  "documentacion..."
  (interactive "p\ncZap to char: ")
  cuerpo-de-funcion...)
```

(El espacio después del punto y coma en pantalla hace que se vea mejor. Véase Sección 5.1 “La Definición de `copy-to-buffer`”, página 57, por ejemplo.)

Cuando una función no tiene argumentos, `interactive` no requiere ninguno. Tal función cotiene la expresión simple `(interactive)`. La función `mark-whole-buffer` es como esto.

Alternativamente, si los códigos de letras no son correctos para tu aplicación, se pueden pasar tus propios argumentos a `interactive` como una lista.

Véase Sección 4.4 “La Definición de `append-to-buffer`”, página 50, para un ejemplo. Véase Sección “Usando `interactive`” in *El Manual de GNU Emacs Lisp*, para una explicación más completa acerca de esta técnica.

3.5 Instalar código permanentemente

Cuando tu instales una definición de función evaluándolo, estará instalado hasta que salgas de Emacs. La siguiente vez que tu empieces una nueva sesión de Emacs, la función no será instalado a menos que tu evalúes la definición de nuevo.

En algún punto, tu puedes querer tener código instalado automáticamente siempre y cuando tu empieces una nueva sesión de Emacs. Hay varios caminos de hacer esto:

- Si tienes código que es solo para tí mismo, se puede poner el código para la definición de función en tu fichero de inicialización de `.emacs`. Cuando tu empieces Emacs, tu fichero `.emacs` es automáticamente evaluado y todas las definiciones de función con ello está instalado. Véase Capítulo 16 “Tu Fichero `.emacs`”, página 191.
- Alternativamente, se pueden poner las definiciones de función que tu quieres instalado en uno o más ficheros de su propio y usar la función `load` para causar a Emacs evaluar y cada una de las funciones en los ficheros. Véase Sección 16.9 “Cargando ficheros”, página 200.

- Tercero, si tu tienes un código que tu sitio completo usará, es normal ponerlo en un fichero llamado `site-init.el` que es cargado cuando Emacs es construido. Esto hace que el código disponible a cualquiera quien usa tu máquina. (Mira el fichero `INSTALL` que es parte de la distribución Emacs.)

Finalmente, si tienes código que cualquiera que use Emacs puede querer, se puede enviar en una red de ordenadores o enviar una copia a la Free Software Foundation. (Cuando se hace esto, por favor, licencia el código y su documentación bajo una licencia que permita a otras personas ejecutar, copiar, estudiar, modificar, y redistribuir el código y que te protege desde quien tome tu trabajo.) Si tu envias una copia de tu código a la Free Software Foundation y, te protege apropiadamente a tí mismo y a otros, eso puede ser incluido en la siguiente entrega de Emacs. Si miramos la historia así, es cómo Emacs ha crecido a través de los años pasados, por donaciones.

3.6 `let`

La expresión `let` es una forma especial en Lisp que necesitarás para usar en la mayoría de las definiciones de función.

`let` se usa para adjuntar o emparejar un símbolo para un valor en tal camino que el intérprete no confundirá la variable con otra variable del mismo nombre que no es parte de la función.

Para comprender por qué la forma especial `let` es necesaria, considera la situación en el que tu propio hogar que generalmente se refiere como 'la casa', como en la frase, "La casa necesita pintura." Si tu estás visitando a un amigo y tu alojamiento se refiere a 'la casa', él es amistoso para estar refiriéndose a *su* casa, no la suya, que es, una casa diferente.

Si tu amigo está refiriéndose a su casa y tu piensas que él está refiriéndose a su casa, tu puedes estar dentro por alguna confusión. La misma cosa podría ocurrir en Lisp si una variable que es usada dentro de una función tiene el mismo que una variable que es usada dentro de otra función, y las dos no se pretende referirse al mismo valor. La forma especial `let` previene este tipo de confusión.

La forma especial `let` evita confusiones. `let` crea un nombre para una *variable local* que ensombrece cualquier uso del mismo nombre fuera de la expresión `let`. Esto es como comprender que siempre y cuando tu host se refiera a 'la casa', significa su casa, no la tuya. (Símbolos usados en listas de argumentos trabajan en el mismo camino. Véase Sección 3.1 "La Forma Especial `defun`", página 26.)

Las variable locales son creadas por una expresión `let` que retiene su valor *solo* con la expresión `let` por sí misma (y con expresiones llamadas con la expresión `let`); las variables locales no tiene efecto fuera de la expresión `let`.

Otro camino para pensar acerca de `let` es que es como un `setq` que es temporal y local. Los valores asignado por `let` son automáticamente deshechos cuando el `let` está finalizado. La configuración solo afecta a expresiones que están dentro de los emparejamientos de la expresión `let`. En jerga de ciencia de computación, diríamos que "el emparejamiento de un símbolo es visible solo en funciones llamadas en la forma `let`; en Emacs Lisp, el alcance es dinámico, no léxico."

`let` puede crear más de una variable a la vez. También, `let` da cada variable eso crea un valor inicial, si un valor especificado por tí, o `nil`. (En la jerga, eso se llama 'asociar la variable al valor'.) Después `let` ha creado y asociado las variables, ejecuta el código en el cuerpo del `let` y devuelve el valor de la última expresión en el cuerpo, como el valor de la expresión `let` completa. ('Ejecuta' es un término de jerga que significa evaluar una lista: viene desde el uso de la palabra significando 'dar efecto práctico a' (*Diccionario de Inglés de Oxford*). Desde que evalúas una expresión para ejecutar una acción, 'ejecuta' ha evolucionado como un sinónimo para 'evaluar'.)

3.6.1 Las partes de una expresión `let`

Una expresión `let` es una lista de tres partes. La primera parte es el símbolo `let`. La segunda parte es una lista, llamada una *varlist*, cada elemento de que es un símbolo por sí mismo o una lista de dos elementos, el primer elemento de que es un símbolo. La tercera parte de la expresión `let` es el cuerpo del `let`. El cuerpo normalmente consiste de una o más listas.

Una plantilla para una expresión `let` se parece a esto:

```
(let varlist body...)
```

Los símbolos en la varlist son las variables que son valores iniciales dados por la forma especial `let`. Los símbolos por sí mismos son dados por el valor inicial de `nil`; y cada símbolo que es el primer elemento de una lista de dos elementos es emparejado al valor que el devuelto cuando el intérprete Lisp evalúa el segundo elemento.

De este modo, una varlist podría verse como esto: `(thread (needles 3))`. En este caso, es una expresión `let`, Emacs asocia el símbolo `thread` a un valor inicial de `nil`, y empareja el símbolo `needles` a un valor inicial de 3.

Cuando escribes una expresión `let`, qué hacer es poner las expresiones apropiadas en las cajas de la plantilla de expresión `let`.

Si la lista de variables está compuesta de listas de 2 elementos, como es frecuente el caso, la plantillas para la expresión `let` mira como esto:

```
(let ((variable valor)
      (variable valor)
      ...)
  body...)
```

3.6.2 Expresión simple `let`

La expresión siguiente crea y da valores dados iniciales para las dos variables `zebra` y `tiger`. El cuerpo de la expresión `let` es una lista que llama a la función `message`.

```
(let ((zebra 'rayas)
      (tiger 'fiero))
  (message "Un tipo de animal tiene %s y otro es %s."
           zebra tiger))
```

Aquí, la varlist es `((zebra 'rayas) (tiger 'fiero))`.

Las dos variables son `zebra` y `tiger`. Cada variable es el primer elemento de una lista de dos elementos y cada valor es el segundo elemento de su lista de dos

elementos. En la varlist, Emacs asocia la variable `zebra` al valor `rayas`[1], y asocia la variable `tiger` al valor `fiero`. En este ejemplo, ambos valores son símbolos precedidos por una comilla. Los valores podrían ser precedidos por una comilla. Los valores podrían también haber sido otra lista o cadena. El cuerpo de `let` sigue después de la lista manejando las variables. En este ejemplo, el cuerpo es una lista que usa la función `message` para imprimir una cadena en el área echo.

Se puede evaluar el ejemplo en el modo usual, emplazando el cursor después de los últimos paréntesis y escribiendo *`C-x C-e`*. Cuando se hace esto lo siguiente aparecerá en el área echo:

```
"Un tipo de animal tiene rayas y otro es fiero"
```

Como se ha visto antes, la función `message` imprime su primer argumento, excepto por '`%s`'. En este ejemplo, el valor de la variable `zebra` es impreso en la posición del primer '`%s`' y el valor de la variable `tigre` es impreso en la posición del segundo '`%s`'.

3.6.3 Variables no inicializadas en un sentencia `let`

Si no asocia las variables en una frase `let` para valores específicos iniciales, ellos automáticamente emparejan a un valor inicial de `nil`, como en la siguiente expresión:

```
(let ((abedul 3)
      pino
      abeto
      (encina 'algo))
  (message
   "Aquí están %d variables con %s, %s, y el valor %s."
   abedul pino abeto encina))
```

Aquí, la varlist es `((abeto 3) pino roble (encina 'otro))`.

Si se evalúa esta expresión en el modo usual, aparecerá lo siguiente en el área echo:

```
"Aquí están 3 variables con nil, nil, y algún
valor".
```

En este ejemplo, Emacs empareja el símbolo `birch` al número 3, empareja los símbolos `pine` y `fir` a `nil`, y empareja el símbolo `oak` al valor `otro`.

Note que en la primera parte del `let`, las variables `pine` y `fir` se aloja solo como átomos que no están rodeados por paréntesis; esto es porque están siendo emparejados a `nil`, la lista vacía. Pero `oak` es emparejado a `otro` y así es una parte de la lista `(oak 'otro)`. De manera similar, `birch` se empareja al número 3 y así es una lista con este número. (Desde que un número se evalúa por sí mismo, el número no necesita ser citado. También, el número es impreso en el mensaje usando '`%d`' en vez de un '`%s`'.) Las cuatro variables como un grupo son puestas dentro de una lista para delimitarlos desde el cuerpo del `let`.

---

[1] De acuerdo a Jared Diamond en *Guns, Germs, y Steel*, "... las cebras llegan a ser muy peligrosas a medida que crecen" pero el clamor aquí son que ellos no llegan a ser fieros como un tigre. (1997, W. W. Norton and Co., ISBN 0-393-03894-2, page 171)

3.7 La forma especial `if`

Una tercera forma especial, además de `defun` y `let`, es el condicional `if`. Esta forma es usada para instruir el ordenador para crear decisiones. Se puede escribir definiciones de función usando `if`, pero eso es usado con suficiente frecuencia, y es suficientemente importante para ser incluido aquí. Eso es usado, por ejemplo, en el código para la función `beginning-of-buffer`.

La idea básica de un `if`, es que "*if* un test es verdad *then* una expresión es evaluado." Si el test no es verdad, la expresión no está evaluada. Por ejemplo, podría crear una decisión tal y como, "¡si es cálido y soleado, entonces a la playa!"

Una expresión `if` expresión escrita en Lisp no usa la palabra 'then'; el test y la acción son el segundo y tercer elementos de la lista cuyo primer elemento es `if` (*si*). Ninguno menos, la parte de test de una expresión `if` (*si*) es con frecuencia llamada la *if-part* (*parte-si*) y el segundo argumento es con frecuencia llamada la *then-part* (*parte-entonces*).

También, cuando una expresión `if` es escrita, el test-verdadero-o-falso es normalmente escrito en la misma línea como el símbolo `if`, pero la acción para traer si el test es verdadero, el "then-part" *parte-entonces*, es escrita en la segunda y subsiguientes líneas. Esto hace que la expresión `if` *si* sea fácil de leer.

```
(if test-verdadero-o-falso
    accion-a-realizar-si-el-test-es-cierto)
```

El test-verdadero-o-falso será una expresión que es evaluado por el intérprete Lisp.

Aquí hay un ejemplo que se puede evaluar en la manera normal. El test es si el número 5 es mayor que el número 4. Desde eso, el mensaje '`¡5 es más grande que 4!`' será impreso.

```
(if (> 5 4)                                ; parte-si
    (message "¡5 es mayor que 4!"))    ; parte-entonces
```

(La función `>` chequea si su primer argumento es mayor que su segundo argumento y devuelve cierto si lo es.)

De acuerdo, en uso actual, el test en una expresión `if` no será corregido para todo el tiempo como eso es por la expresión `(> 5 4)`. En vez, al menos una de las variables usadas en el test será asociada a un valor que no es conocido en frente del tiempo. (Si el valor fuera conocido en el tiempo, ¡no necesitaríamos ejecutar el test!)

Por ejemplo, el valor puede ser asociado a un argumento de una definición de función. En la siguiente definición de función, el carácter del animal es un valor que es pasado a la función. Si el valor asociado a `característico` es `fiero`, entonces el mensaje, '`¡Es un tigre!`' será impreso; de otro modo, `nil` será devuelto.

```
(defun tipo-de-animal (caracteristica)
  "Imprime el mensaje en el área echo dependiendo de CARACTERISTICA.
Si la CARACTERISTICA es el símbolo 'fiera',
entonces avisa de un tigre."
  (if (equal caracteristica 'fiera)
      (message "¡Es un tigre!")))
```

Si estás leyendo esto dentro de GNU Emacs, se puede evaluar la definición función en el modo usual para instalarlo en Emacs, y entonces se puede evaluar las siguientes dos expresiones para ver los resultados:

```
(tipo-de-animal 'fiera)
(tipo-de-animal 'cebra)
```

Cuando se evalúa `(tipo-de-animal 'fiero)`, se verá el siguiente mensaje impreso en el área echo: `"¡Es un tigre!"`; y cuando se evalúa `(tipo-de-animal 'cebra)` verás `nil` impreso en el área echo.

3.7.1 La función `tipo-de-animal` en detalle

Mira la función `tipo-de-animal` en detalle.

La definición de función para `tipo-de-animal` fué escrito para rellenar los slots de dos plantillas, uno para una definición de función como un todo, y un segundo para una expresión `if` (*si*).

La plantilla para cada función que no es interactiva es:

```
(defun nombre-de-funcion (lista-de-argumentos)
  "documentacion..."
  cuerpo...)
```

Las partes de la función asociada a esta plantilla es:

```
(defun tipo-de-animal (caracteristica)
  "Imprime el mensaje en el área echo dependiendo de CARACTERISTICA.
Si la CARACTERISTICA es el símbolo ‘fiera’,
entonces avisa de que es un tigre."
  body: the if expression)
```

El nombre de función es `tipo-de-animal`; se pasa al valor de un argumento. La lista de argumentos es seguida por una cadena de documentación multi-línea. La cadena de documentación es incluida en el ejemplo porque es un buen hábito para escribir documentación para cada definición de función. El cuerpo de la definición de función consiste de la expresión `if`.

La plantilla para una expresión `if` se ve así:

```
(if test-verdadero-o-falso
    accion-a-realizar-si-el-test-es-cierto)
```

En la función `tipo-de-animal`, el código para el `if` (*si*) se ve así:

```
(if (equal caracteristica 'fiero)
    (message "¡Es un tigre!")))
```

Aquí, está la expresión test-verdadero-o-falso

```
(equal caracteristica 'fiero)
```

En Lisp, `equal` es una función que determina si su primer argumento es igual para su segundo argumento. El segundo argumento es el símbolo citado `'fiero` y el primer argumento es el valor del símbolo `característico` — en otras palabras, el argumento pasado a esta función.

En el primer ejercicio de `tipo-de-animal`, el argumento `fiera` es pasado a `tipo-de-animal`. Desde que `fiera` es igual a `fiera`, la expresión, `(equal caracteristica 'fiera)`, devuelve un valor de verdad. Cuando esto ocurre, el

`if` (*si*) evalúa el segundo argumento o parte-entonces del `if` (*si*): `(message "¡Es un tigre!")`.

Por otro lado, en el segundo ejercicio de `tipo-de-animal`, el argumento `cebra` es pasado a `tipo-de-animal`. `cebra` no es igual a `fiera`, así la parte-entonces no está evaluada y se devuelve `nil` por la expresión `if` (*si*).

3.8 Expresiones Si–entonces–resto

Una expresión `if` *si* puede tener un tercer argumento opcional, llamado la *parte-resto*, para el caso en el que test-verdadero-o-falso devuelve falso. Cuando esto ocurre, el segundo argumento o la parte-entonces sobre todo la expresión `if` (*si*), *no* es evaluado, pero el tercero o la parte-resto *es* evaluado. Se podría pensar en esto como la alternativa del día nublado para la decisión "si eso es cálido y soleado, ve a la playa, sino ¡lee un libro!"

La palabra "else" *resto* no está escrita en el código Lisp; la parte else *resto* de una expresión `if` *si* viene después de la parte then *entonces*. En el Lisp escrito, la parte else *resto* normalmente se escribe para empezar en la línea siguiente y está menos indentada que la parte then *entonces*:

```
(if test-verdadero-o-falso
    accion-a-realizar-si-el-test-es-cierto
  accion-a-realizar-si-el-test-es-falso)
```

Por ejemplo, la siguiente expresión `if` imprime el mensaje '`¡4 no es mayor que 5!`' cuando se evalúa eso en el camino usual:

```
(if (> 4 5)                                  ; parte-si
    (message "¡4 no es más grande que 5!") ; parte-entonces
  (message "¡4 no es más grande que 5!"))    ; parte-resto
```

Nótese que los diferentes niveles de indentación hacen fácil distinguir la parte then desde la parte resto. (GNU Emacs tiene varios comandos que automáticamente indenta expresiones correctamente `if` *si*. Véase Sección 1.1.3 "GNU Emacs te ayuda a escribir listas", página 3.)

Podemos extender la función `tipo-de-animal` para incluir una parte else para simplemente incorporar una parte adicional para la expresión `if` *si*.

Se puede ver las consecuencias de hacer esto si se evalúa la siguiente versión de la definición de función `type-of-animal` (*tipo-de-animal*) para instalarlo y entonces evaluar las dos expresiones subsiguientes para pasar diferentes argumentos para la función.

```
(defun tipo-de-animal (caracteristica)  ; Segunda versión.
  "Imprime el mensaje en el área echo dependiendo de CARACTERISTICA.
Si la CARACTERISTICA es el símbolo 'fiera',
entonces avisa de un tigre; sino di que no es una fiera."
  (if (equal caracteristica 'fiera)
      (message "¡Es un tigre!")
    (message "¡No es una fiera!")))

(tipo-de-animal 'fiera)
(tipo-de-animal 'cebra)
```

Cuando se evalúa `(tipo-de-animal 'fiera)`, se verá el siguiente mensaje impreso en el área echo: `"¡Eso es un tigre!"`; pero cuando se evalúa `(tipo-de-animal 'cebra)`, se verá `"¡No es una fiera!"`.

(De acuerdo, si la *característica* fuera `feroz`, el mensaje `"¡No es una fiera!"` sería impreso; ¡y sería erróneo! Cuando se escribe código, se necesita tener en cuenta la posibilidad que algunos argumentos será probado por `if` *si* y escribir tu programa de acuerdo.

3.9 Verdad y falsedad en Emacs Lisp

Hay un aspecto importante para el test de verdad en una expresión `if` (*si*). Así, hemos hablado de 'verdad' y 'mentira' como valores de predicados como si fueran nuevos tipos de objetos Emacs Lisp. En efecto, 'falso' es solo nuestro viejo amigo `nil`. Cualquier otra es 'verdadero'.

La expresión chequea si verdad se interpreta como (*true*) *verdadero* si el resultado de evaluarlo es un valor que no es `nil`. En otras palabras, el resultado del test se considera cierto si el valor devuelto es un número como 47, una cadena tal como `"hola"`, o un símbolo (otro como `nil`) tal como `flores`, o una lista (tan larga como eso no está vacía) ¡o incluso un búffer!

Antes de ilustrar un test para verdad, se necesita una explicación de `nil`.

En Emacs Lisp, el símbolo `nil` tiene dos significados. Primero, está el significado de la lista vacía. Segundo, está el valor de falso y es el valor devuelto cuando el test test-verdadero-o-falso salga falso. `nil`. Tan lejos como el intérprete Lisp es concebido, `()` y `nil` son el mismo. Los humanos, sin embargo, tienden a usar `nil` para falso y `()` para la lista vacía.

En Emacs Lisp, cualquier valor que no es `nil` — no es una lista vacía — es considerado verdad. Esto significa que si una evaluación devuelve alguna cosa que no es una lista vacía, una expresión `if` devuelve verdad. Por ejemplo, si un número es puesto en el slot para el test, será evaluado y devolverá por sí mismo, desde lo que hacen los números cuando se evalúan. En este condicional, la expresión `if` devuelve verdad. La expresión se chequea como falso solo cuando `nil`, una lista vacía, es devuelta evaluando la expresión.

Se puede ver esto evaluando las dos expresiones en los siguientes ejemplos.

En el primer ejemplo, número 4 es evaluado como el test en la expresión `if` y se devuelve por sí mismo; por consiguiente, la then-part de la expresión es evaluada y devuelta: '`true`' aparece en el área echo. En el segundo ejemplo, `nil` indica falso; por consiguiente, el else-part de la expresión es evaluada y devuelta: '`false`' aparece en el área echo.

```
(if 4
    'true
  'false)

(if nil
    'true
  'false)
```

Incidentalmente, si algún otro valor útil no está disponible para un test que devuelve cierto, entonces el intérprete Lisp retornará el símbolo `t` para cierto. Por ejemplo, la expresión `(> 5 4)` devuelve `t` cuando se evalúa, como puedes ver evaluándolo en el camino usual:

```
(> 5 4)
```

Por otro lado, esta función devuelve `nil` si el test es falso.

```
(> 4 5)
```

3.10 `save-excursion`

La función `save-excursion` es la cuarta y última forma especial que se discutirá en este en este capítulo.

En Emacs Lisp hay programas usados para edición, la función `save-excursion` es muy común. Eso guarda la posición de punto y marca, ejecuta el cuerpo de la función, y entonces restaura el punto y marca a sus posiciones previas si sus posiciones fueran cambiadas. Su propósito primario es guardar que el usuario sea sorprendido y molesto por movimientos inesperado de punto y marca.

Antes de discutir `save-excursion`, sin embargo, puede ser útil primero revisar que punto y marca están en GNU Emacs. *Punto* es la posición actual del cursor. En cualquier lugar que el cursor se posicione hay un punto. De manera más precisa, en terminales donde el cursor parece estar en lo alto de un carácter, el punto está inmediatamente antes del carácter. En Emacs Lisp, punto es un entero. El primer carácter en un búffer es el número uno, el segundo es el número dos, y así. La función `punto` devuelve la posición actual del cursor como un número. Cada búffer tiene su propio valor para el punto.

La *marca* es otra posición en el búffer; su valor puede ser asignado con un comando tal como *`C-SPC`* (`set-mark-command`). Si una marca ha sido asignada, se puede usar el comando *`C-x C-x`* (`exchange-point-and-mark`) para hacer que el cursor salte a la marca y asignar la marca para la posición previa del punto. Además, si tu asignas otra marca, la posición puede ser guardada por este camino. Se puede saltar al cursor para una marca guardada escribiendo *`C-u C-SPC`* una o más veces.

La parte del búffer entre el punto y la marca es llamada *la región*. Numerosos comandos trabajan en la región, incluyendo `center-region`, `count-lines-region`, `kill-region` y `print-region`.

La forma especial `save-excursion` salva las posiciones del punto y la marca y restaura estas posiciones después del código con el cuerpo de la forma especial es evaluada por el intérprete Lisp. De este modo, si el punto fuera en el principio de una pieza de texto y algún código movido apunta al fin del búffer, el `save-excursion` no apuntaría a donde fué antes, después las expresiones en el cuerpo de la fueron evaluadas.

En Emacs, una función frecuentemente mueve el punto como parte de sus trabajos internos incluso aunque un usuario no espere esto. Por ejemplo, `count-lines-region` se mueve al punto. Para prevenir al usuario que se preocupe por salto que es inesperado y (desde el punto de vista del usuario) innecesario, `save-excursion` es con frecuencia usado para punto y marca en la posición esperada por el usuario. El uso de `save-excursion` es un buen guarda casas.

Para estar seguro que la casa está limpia, `save-excursion` restaura los valores de punto y marca incluso si alguna cosa va mal en el código dentro de eso (o, para ser más preciso y usar la jerga apropiada, "en caso de salida anormal"). Esta funcionalidad es muy útil.

Además grabando los valores de punto y marca, `save-excursion` guarda la traza del actual buffer, y lo restaura, también. Esto significa que puedes escribir código que cambiará el buffer y tener que `save-excursion` vuelva al buffer original. Esto es como `save-excursion` es usado en `append-to-buffer`. (Véase Sección 4.4 "La Definición de `append-to-buffer`", página 50.)

3.10.1 Plantilla para una expresión `save-excursion`

La plantilla para código usando `save-excursion` es simple:

```
(save-excursion
  body...)
```

El cuerpo de la función es una o más expresiones que serán evaluadas en secuencia por el intérprete Lisp. Si hay más de una expresión en el cuerpo, el valor de la última será devuelto como el valor de la función `save-excursion`. Las otras expresiones en el cuerpo son evaluadas solo por sus efectos laterales; y `save-excursion` en sí es usado solo por su efecto lateral (que está restaurando las posiciones de punto y marca).

Para más detalles, la siguiente plantilla explica `save-excursion`

```
(save-excursion
  primera-expresion-en-el-cuerpo
  segunda-expresion-en-el-cuerpo
  tercera-expresion-en-el-cuerpo
   ...
  ultima-expresion-en-el-cuerpo)
```

Una expresión, de acuerdo, puede ser un símbolo por sí mismo o una lista.

En código Emacs Lisp, una expresión `save-excursion` con frecuencia ocurre el cuerpo de una expresión `let`. Eso se ve como esto:

```
(let varlist
  (save-excursion
    body...))
```

3.11 Revisar

En los últimos pocos capítulos se han introducido un número limpio de funciones y formas especiales. Aquí se han descrito brevemente, con unas pocas funciones similares que no han sido mencionadas todavía.

`eval-last-sexp`
: Evalúa la última expresión simbólica antes de la posición actual del punto. El valor es impreso en el área echo a menos que la función sea invocada con un argumento; en este caso, la salida es impresa en el actual búffer. Este comando está normalmente asociado a `C-x C-e`.

`defun`
: Definir función. Esta forma especial ha subido a cinco partes: el nombre una plantilla para los argumentos que serán pasados a la doc-

umentación de la función, una declaración interactiva opcional, y el cuerpo de la definición.

Por ejemplo, en las primeras versiones de Emacs, la definición de función era como sigue. (Eso es ligeramente más complejo ahora que si busca el primer caracter de espacio no en blanco en vez del primer caracter visible.)

```
(defun volver-a-indentacion ()
  "Mover el punto al primer caracter visible en linea."
  (interactive)
  (beginning-of-line 1)
  (skip-chars-forward " \t"))
```

`interactive`

Declara al intérprete que la función puede ser usada interactivamente. Esta forma especial puede ser seguida por una cadena con una o más partes que pasan la información a los argumentos de la función, en secuencia. Estas partes pueden también contar al intérprete para mostrar la información. Parte de las cadenas son separadas por nuevas líneas, '`\n`'.

Caracteres de código común son:

`b` El nombre de un búffer existente.

`f` El nombre de un fichero existente

`p` El argumento prefijo numérico. (Nótese que esta 'p' es minúscula.)

`r` El Punto y la marca, como dos argumentos numéricos, el más pequeño primero. Esta es la única letra que especifica dos argumentos sucesivos en vez de uno.

Véase Sección "Caracteres de Código para '`interactive`'" in *El Manual de Referencia de GNU Emacs Lisp*, para una lista de caracteres de código.

`let` Declara que una lista de variables es para usarla con el cuerpo del `let` y darles un valor inicial, bien `nil` o un valor específico; entonces se evaluarán el resto de las expresiones en el cuerpo del `let` y devolver el valor de la última. Dentro del cuerpo del `let`, el intérprete Lisp no ve los valores de las variables de los mismos nombres que son asociados fuera del `let`.

Por ejemplo,

```
(let ((foo (nombre-de-buffer))
      (bar (tamagno-de-buffer)))
  (message
   "Este buffer es %s y tiene %d caracteres."
   foo bar))
```

`save-excursion`
: Graba los valores de punto y marca y el actual búffer antes de evaluar el cuerpo de esta forma especial. Restaura los valores de punto y marca y el búffer después de esto.

  Por ejemplo,

```
(message "Hay %d caracteres dentro de este buffer."
         (- (point)
            (save-excursion
              (goto-char (point-min)) (point))))
```

`if`
: Evalúa el primer argumento a la función; si es verdad, evalúa el segundo argumento; lo demás evalúa el tercer argumento, si hay uno.

  La forma especial `if` es llamada *condicional*. Hay otros condicionales en Emacs Lisp, pero `if` es quizás lo más comúnmente usado.

  Por ejemplo,

```
(if (= 22 version-de-emacs-mayor)
    (message "Esta es la versión 22 de Emacs")
  (message "Esta no es la versión 22 Emacs"))
```

`<`
`>`
`<=`
`>=`
: La función `<` chequea si su primer argumento es más pequeño que su segundo argumento. Una función correspondiente, `>`, chequea si el primer argumento es mayor que el segundo. De otro modo, `<=` chequea si el primer argumento es menor o igual al segundo y `>=` chequea si el primer argumento es mayor o igual al segundo. En todos los casos, ambos argumentos deben ser números o marcas (las marcas indican posiciones en búffers).

`=`
: La función `=` chequea si dos argumentos, ambos números o marcadores, son iguales.

`equal`
`eq`
: Chequea si dos objetos son el mismo. `equal` usa un signicado de la palabra 'mismo' y `eq` usa otro: `equal` devuelve cierto si los dos objetos tienen una estructura y contenidos similares, tal como dos copias del mismo libro. En una mano, `eq`, devuelve cierto si ambos argumentos son actualmente el mismo objeto.

`string<`
`string-lessp`
`string=`
`string-equal`
: La función `string-lessp` chequea si su primer argumento es más pequeño que el segundo. En resumen, el nombre alternativo para la misma función (un `defalias`) es `string`.

  Los argumentos para `string-lessp` deben ser cadenas o símbolos; la ordenación es lexicográfica, así el caso es significativo. Los nombres impresos de símbolos son usado en vez de símbolos por sí mismos.

Una cadena vacía, '`"\"`', una cadena sin caracteres dentro, es más pequeña que cualquier cadena de caracteres.

`string-equal` provee el test correpondiente para igualdad. Su corto nombre alternativo es `string=`. No hay funciones test que correponden a `>`, `>=`.

`message` Imprime un mensaje en el área echo. El primer argumento es una cadena que puede contener, '`%s`', '`%d`', o '`%c`' para imprimir el valor de argumentos que siguen la cadena. El argumento usado por '`%d`' debe ser un número. El argumento usado por '`%c`' debe ser un número de código ASCII; eso será impreso como el caracter con este código ASCII. (Otras varias %-secuencias no han sido mencionadas.)

`setq`
`set` La función `setq` asigna el valor de su primer argumento al valor del segundo argumento. El primer argumento está automáticamente citado por `setq`. Eso hace lo mismo para los pares de argumentos. Otra función, `set`, toma solo dos argumentos y evalúa ambos de ellos antes de configurar el valor devuelto por su primer argumento al valor devuelto por su segundo argumento.

`buffer-name`
Sin un argumento, devuelve el nombre del búffer, como una cadena.

`buffer-file-name`
Sin un argumento, devuelve el nombre del fichero si el búffer lo está visitando.

`current-buffer`
Devuelve el búffer en el que Emacs es activo; eso puede no ser el búffer que es visible en la pantalla.

`other-buffer`
Devuelve el búffer seleccionado más recientemente (otro que el búffer pasado a `other-buffer` como un argumento y otro en vez de el búffer actual).

`switch-to-buffer`
Selecciona un búffer para Emacs esté activo y lo muestre en la ventana actual y así los usuarios puedan mirarlo. Normalmente se empareja a *`C-x b`*.

`set-buffer`
Cambia la atención de Emacs a un búffer en el que los programas se ejecutarán. No altera lo que la ventana está mostrando.

`buffer-size`
Devuelve el número de caracteres en el búffer actual.

`punto` Devuelve el valor de la actual posición del cursor, como un entero contando el número de caracteres desde el principio del búffer.

`point-min` Devuelve el valor mínimo permisible del punto en el búffer actual. Esto es 1, a menos que la contracción esté en efecto

`point-max` Devuelve el valor del máximo valor permisible del punto en el búffer actual. Esto es el fin del búffer, a menos que la contracción esté en efecto

3.12 Ejercicios

- Escribe una función no interactiva que doble el valor de su argumento, un número. Luego hazla función interactiva.
- Escribe una función que chequee si el valor actual de `fill-column` es más grande que el argumento pasado a la función, y si es así, imprime un mensaje apropiado.

4 Unas pocas funciones de buffer relacionadas

En este capítulo estudiamos en detalle varias de las funciones usadas en GNU Emacs. Esto se llama un "paseo a través". Estas funciones son usadas como ejemplos de código Lisp, pero no son ejemplos imaginarios; con la excepción del primero, la definición de función simplificada, esta funciones muestran el actual código usado en GNU Emacs. Se puede aprender mucho desde estas definiciones. Las funciones descritas aquí están todas relacionadas a búffers. Después, estudiaremos otras funciones.

4.1 Encontrando más información

En este paseo, se describe cada nueva función como viene, algunas veces en detalle y algunas veces brevemente. Si está interesado, se puede obtener la documentación completa de cualquier función Emacs Lisp en cualquier momento escribiendo `C-h f` y entonces el nombre de la función (y entonces `RET`. De manera similar, se puede obtener la documentación completa para una variable escribiendo `C-h v`, después el nombre de la variable (y entonces `RET`).

También, en `describe-function`, se encontrará la localización de la definición de la función.

Poner el punto dentro del nombre del fichero que contiene la función y presiona la tecla `RET`. En este caso, `RET` significa `push-button` en vez de 'return' o 'enter'. Emacs tomará directamente a la definición de la función.

De manera más general, si quieres ver una función en su fichero fuente original, se puede usar la función `find-tag` para saltar dentro. `find-tag` funciona con una amplia variedad de lenguajes, no solo Lisp, y C, y funciona con texto de no programación también. Por ejemplo, `find-tag` saltará a los varios nodos en el fichero fuente Texinfo de este documento. La función `find-tag` depende de 'tablas de etiquetas' que graba las localizaciones de las funciones, variables, y otros ítems para los que `find-tag` salta.

Para usar el comando `find-tag`, escribe `M-.` (por ej., presionando la tecla `META`, o al menos escribe la tecla `ESC` y entonces escribe la tecla punto), entonces, en la pantalla, se escribe el nombre de la función cuyo código fuente se quiere ver, tal como `mark-whole-buffer`, y luego escribe `RET`. Emacs cambiará el búffer y mostrará el código fuente para la función en la pantalla. Para volver al búffer actual, escribe `C-x b RET`. (En algunos teclados, la tecla `META` es etiquetada con `ALT`.)

Dependiendo de cómo los valores iniciales por defecto de tu copia de Emacs son asignados se puede también necesitar especificar la posición de tu 'tabla de tags', que es un fichero llamado `TAGS`. Por ejemplo, si se está interesado en fuentes de Emacs, la tabla de tags que se desea, si ya ha sido creada para tí, estará en un subdirectorio del directorio `/usr/local/share/emacs`; de este modo se usaría el comando `M-x visit-tags-table` y especifica una ruta tal como `/usr/local/share/emacs/22.1.1/lisp/TAGS`. Si la tabla tags no ha sido creada, tendrás que crearla por tí mismo. Será un fichero tal como `/usr/local/src/emacs/src/TAGS`.

Para crear un fichero `TAGS` en un directorio específico, cambia a este directorio en Emacs usando un comando *`M-x cd`*, o lista el directorio como `etags *.el` como el comando para ejecutar:

```
M-x compile RET etags *.el RET
```

Para más información, ver Sección 12.5 "Crea tu propio fichero `TAGS`", página 146.

Después de llegar a estar más familiarizado con Emacs Lisp, se encontrará frecuentemente usar `find-tag` para navegar tu camino alrededor del código fuente; y se crearán tus propias tablas `TAGS`.

Incidentalmente, los ficheros que contienen código Lisp son convencionalmente llamadas *librerías*. La metáfora se deriva que una librería, tal como la librería de leyes o una librería de ingeniería, en vez de una librería general. Cada librería, o fichero, contiene funciones que se relacionan a un asunto particular o actividad, tal como `abbrev.el` para manejar abreviaciones y otros atajos, y `help.el` para la ayuda on-line. (Algunas veces varias librerías proporcionan código para una actividad simple, como varios `rmail...` ficheros proveen código para leer correos electrónicos.) En *El Manual de GNU Emacs*, tu verás varias frases tales como "El comando *`C-h p`* te permite buscar el estándar de las librerías Emacs Lisp por las palabras claves.

4.2 Una definición simplificada de `beginning-of-buffer`

El comando `beginning-of-buffer` es una buena función para empezar ya puedes tener cierta familiaridad con eso y es fácil de comprender. Usado como un comando interactivo, `beginning-of-buffer` mueve el cursor al principio del búffer, dejando la marca en la posición previa. Eso es generalmente asignados a *`M-<`*.

En esta sección, se discutirá una versión ordenada de la función que muestra como eso es lo usado más frecuentemente. Esta función ordenada trabaja como se escribe, pero no contiene el código para una función compleja. En otra sección, describiremos la función entera. (Véase Sección 5.3 "beginning-of-buffer", página 63.)

Antes de mirar en el código, permítenos considerar que la definición de función tiene que contener: eso debe incluir una expresión que crea la función interactiva así puede ser llamado escribiendo *`M-x beginning-of-buffer`* o escribiendo unos atajos tales como *`M-<`*; debe incluir código para dejar una marca en la posición original en el búffer; y debe incluir código el cursor al principio del búffer.

Aquí está el texto completo la versión ordenada de la función:

```
(defun simplified-beginning-of-buffer ()
  "Mover punto al principio del bufer; dejar marca en la posición previa."
  (interactive)
  (push-mark)
  (goto-char (point-min)))
```

Como todas las definiciones de función, esta definición tiene cinco partes siguiendo la forma especial `defun`:

1. El nombre: en este ejemplo, `simplified-beginning-of-buffer`.
2. Una lista de los argumentos: en este ejemplo, una lista vacía, `()`,
3. La cadena de documentación.

4. La expresión interactive.

5. El cuerpo.

En esta definición de función, la lista de argumentos está vacía; esto significa que esta función no requiere argumentos. (Cuando se busca la definición para la función completa, se verá que puede pasarse un argumento opcional.)

La expresión interactiva cuenta a Emacs que la función se pretende ser usada interactivamente. En este ejemplo, `interactive` no tiene un argumento porque `simplified-beginning-of-buffer` no se requiere.

El cuerpo de la función consiste de dos líneas:

```
(push-mark)
(goto-char (point-min))
```

La primera de estas líneas es la expresión, `(push-mark)`. Cuando esta expresión es evaluado por el intérprete Lisp, eso asigna una marca en la posición actual del cursor, siempre y cuando esto pueda ser. La posición de esta marca está guardada en el anillo de marcas.

La siguiente línea es `(goto-char (point-min))`. Esta expresión salta el cursor al punto mínimo en el búffer, esto es, para el comienzo del búffer (o al principio de la porción accesible del búffer si eso está encogido. Véase Capítulo 6 “Encogiendo y extendiendo”, página 70.)

El comando `push-mark` estable una marca en el lugar donde el cursor fué localizado antes que fuera movido al principio del búffer por la expresión `(goto-char (point-min))`. Consiguientemente, puedes, si lo deseas, volver donde estabas originalmente escribiendo *`C-x C-x`*.

¡Esto es todo lo que hay para la definición de función!

Cuando se lee código como este y vuelve a una función no familiar, tal como `goto-char`, se puede encontrar que se hace usando el comando `describe-function`. Para usar este comando, escribe *`C-h f`* y entonces escribe en el nombre de la función y presiona `RET`. El comando `describe-function` imprimirá la documentaciń de la cadena de la función en una ventana `*Help*`. Por ejemplo, la documentación para `goto-char` es:

```
Asignar punto a POSITION, un número o marca
Empezando el buffer es la posición (point-min), y el final es
(point-max).
```

La función es un argumento es la posición deseada.

(La consola para `describe-function` te ofrecerá el símbolo abajo o precediendo al cursor, así se puede guardar escribiendo al posiciona el cursor a la derecha o después de la función y entonces escribiendo *`C-h f RET`*.)

La definición de función `end-of-buffer` está escrito en el mismo modo que la definición `beginnig-of-buffer` excepto que el cuerpo de la función contenga la expresión `(goto-char (point-max))` en lugar de `(goto-char (point-min))`

4.3 La definición de mark-whole-buffer

La función mark-whole-buffer no es tan difícil de comprender que la función simplified-beginning-of-buffer. En este caso, sin embargo, se verá la función completa, no una versión ordenada.

La función mark-whole-buffer no está comúnmente usada como la función beginning-of-buffer, pero eso no es útil: eso marca un búffer completo como una región poniendo el punto al principio y una marca al fin del búffer. Eso está generalmente asociado a *C-x h*.

En GNU Emacs 22, el código para la función completa se parece a:

```
(defun mark-whole-buffer ()
  "Pon el punto al principio y marca el fin del búffer.
Probablemante no deberías usar esta función en
  programas Lisp; normalmente un error para una función Lisp usa
  cualquier subrrutina que usa o asigna la marca."
  (interactive)
  (push-mark (point))
  (push-mark (point-max) nil t)
  (goto-char (point-min)))
```

Como todas las otras funciones, la función mark-whole-buffer se ajusta dentro de la plantilla para una definición. La plantilla se parece a esta:

```
(defun name-of-function (argument-list)
  "documentation..."
  (interactive-expression...)
  body...)
```

Aquí está cómo la función trabaja: el nombre de la función es mark-whole-buffer; eso es seguida por un argumento de lista vacía, '()', que significa que la función no requiere argumentos. La documentación viene la siguiente.

La siguiente línea es una expresión (interactive) que cuenta a Emacs que la función será usada interactivamente. Estos detalles son similares a la función simplified-beginning-of-buffer descrita en la sección previa

4.3.1 Cuerpo de mark-whole-buffer

El cuerpo de la función mark-whole-buffer consiste en tres líneas de código:

```
(push-mark (point))
(push-mark (point-max) nil t)
(goto-char (point-min))
```

El primero de estas líneas es la expresión, (push-mark (point)).

Esta línea hace exactamente el mismo trabajo que la primera línea del cuerpo de la función simplified-beginning-of-buffer, que está escrita (push-mark). En ambos casos, el intérprete Lisp asigna una marca en la posición actual del cursor.

No sé por qué en la expresión mark-whole-buffer está escrito (push-mark (point)) y en la expresión beginning-of-buffer está escrito (push-mark). Quizás quien escribió el código no sabía que los argumentos para push-mark son opcionales y que si push-mark no se pasa como argumento, la función automáticamente asigna la marca en la localización del punto por defecto. O quizás la expresión fué

escrita así como para parelizar la estructura de la siguiente línea. En cualquier caso, la línea causa que Emacs determine la posición del punto y asigne una marca allí.

En las primeras versiones de GNU Emacs, la siguiente línea de `mark-whole-buffer` fué `(push-mark (point-max))`. Esta expresión asigna una marca en el punto en el búffer que tiene el número más alto. Esto será el fin del búffer (o, si el búffer es encogida, el fin de la porción accesible del búffer. Véase Capítulo 6 "Encogiendo y extendiendo", página 70, para más acerca encoger.) Después esta marca ha sido asignada, la marca previa, uno asigna un punto, no es se asigna largo, pero Emacs recuerda su posición, solo como todas las otras marcas recientes son siempre recordadas. Esto significa que tu puedes, si lo deseas, vuelve a esta posición escribiendo *C-u C-SPC* dos veces.

En GNU Emacs 22, el `(point-max)` es ligeramente más complicado. La línea lee

```
(push-mark (point-max) nil t)
```

La expresión funciona cerca de lo mismo que antes. Eso asigna una marca en el lugar numerado más alto que se puede en el búffer. Sin embargo, en esta versión, `push-mark` tiene dos argumentos adicionales. El segundo argumento para `push-mark` es `nil`. Esto cuenta la función que *mostraría* un mensaje que dice 'Marca asignada' cuando eso empuja la marca. El tercer argumento es `t`. Esto cuenta `push-mark` para activar la marca cuando el modo Transient Mark está activado. Transient Mark mode ilumina la región de marca activa. Con frecuencia desactivada

Finalmente, la última línea de la función es `(goto-char (point-min))`. Esto es escrito exactamente el mismo camino camino como está escrito `beginning-of-buffer`. La expresión mueve el cursor al mínimo punto en el búffer, que es, al principio del búfferr (o para el principio de la porción accesible del búffer). Como un resultado de esto, punto está emplazado al principio del búffer y marca está asignada al fin del búffer. El búffer completo es, más allá, la región.

4.4 La definición de `append-to-buffer`

El comando `append-to-buffer` es más complejo que el comando `mark-whole-buffer`. Lo que hace es copiar la región (que es, la parte del búffer entre punto y marca) desde el buffer actual a un búffer específico.

El comando `append-to-buffer` usa la función `insert-buffer-substring` para copiar la región. `insert-buffer-substring` es descrita por su nombre: eso toma una cadena de caracteres desde parte de un búffer, una "subcadena", y las inserta dentro de otro búffer.

La mayoría de `append-to-buffer` se refiere con la configuración de las condiciones para `insert-buffer-substring` para trabajar: el código debe especificar ambos el búffer para el que el texto irá, la ventana viene y va, y la región que será copiada.

Aquí está el texto completo de la función:

```
(defun append-to-buffer (buffer start end)
  "Introduce al búffer específico el texto de la
  región. Esto es insertado de este búffer antes de su punto.

Cuando se llama desde un programa, se dan tres argumentos:
BUFFER (o nombre del búffer), START y END.
START y END especifica la porción del búffer actual para ser copiado."
  (interactive
   (list (read-buffer "Append to buffer: " (other-buffer
                                            (current-buffer) t))
         (region-beginning) (region-end)))
  (let ((oldbuf (current-buffer)))
    (save-excursion
      (let* ((append-to (get-buffer-create buffer))
             (windows (get-buffer-window-list append-to t t))
             point)
        (set-buffer append-to)
        (setq point (point))
        (barf-if-buffer-read-only)
        (insert-buffer-substring oldbuf start end)
        (dolist (window windows)
          (when (= (window-point window) point)
            (set-window-point window (point)))))))))
```

La función puede ser comprendida buscando como series de plantillas rellenas

La plantilla de fuera es para la definición de la función. En esta función, se ve como esto (con varios slots rellenos):

```
(defun append-to-buffer (buffer start end)
  "documentacion..."
  (interactive ...)
  cuerpo...)
```

La primera línea de la función incluye su nombre y los tres argumentos. Los argumentos son el `búffer` que el texto será copiado, y el `start` y `end` de la región en el buffer actual que será copiado.

La siguiente parte de la función es la documentación, que es claro y completo. Como es convencional, los tres argumentos son escritos en mayúsculas así se notificarán fácilmente. Incluso mejor, son descritas en el mismo orden como en la lista de argumentos.

Nótese que la documentación distingue entre un búffer y su nombre. (La función puede manejar otro.)

4.4.1 La expresión interactiva `append-to-buffer`

Desde que la función `append-to-buffer` será usada interactivamente, la función debe tener una expresión `interactive`. (Para una revisión de `interactive`, ver Sección 3.3 “Creando una Función Interactive”, página 29.) La expresión se lee de la siguiente manera:

```
(interactive
 (list (read-buffer
        "Agrega al buffer: "
        (other-buffer (current-buffer) t))
       (region-beginning)
       (region-end)))
```

Esta expresión no es una con letras separadas por partes, como se describe antes. En vez de eso, empieza una lista con estas partes:

La primera parte de la lista es una expresión para leer el nombre de un búffer y lo devuelve como una cadena. Esto es `read-buffer`. La función requiere una consola como su primer argumento, '`"Asocia al buffer: "`'. Su segundo argumento cuenta el comando que valora para proporciona si no especifica cualquier cosa.

En este caso este segundo argumento es una expresión conteniendo la función `other-buffer`, una excepción, y una '`t`', para verdad.

El primer argumento para `other-buffer`, la excepción, es todavía otra función, `other-buffer`. Esto es no yendo a ser devuelto. El segundo argumento es el símbolo para verdad, `t`. Esto cuenta `other-buffer` que puede mostrar búffers visibles (excepto en este caso, eso no mostrará el búffer actual, que tiene sentido).

La expresión se ve como:

```
(other-buffer (current-buffer) t)
```

El segundo y tercer argumento a la expresión `list` son `(region-beginning)` y `(region-end)`. Estas dos funciones especifican el principio el y el fin del texto para ser adjunto.

Originalmente, el comando usaba las letras '`B`' y '`r`'. La expresión completa `interactive` es así:

```
(interactive "BAsociar al buffer: \nr")
```

Pero cuando esto fué hecho, el valor por defecto del búffer cambió a ser invisible. Esto no se quería.

(La consola estaba separada del segundo argumento con una nueva línea, '`\n`'. Estaba seguido por un '`r`' que contaba a Emacs emparejar los dos argumentos que siguen el símbolo `buffer` en la lista de argumentos de la función (que es, `start` y `end`) para los valores de punto y marca. Este argumento trabajó bien.)

4.4.2 El cuerpo de `append-to-buffer`

El cuerpo de la función `append-to-buffer` empieza con `let`.

Como se ha visto antes (véase Sección 3.6 "let", página 33), el propósito de una expresión `let` es crear y dar valores iniciales a una o más variable que solo serán usada con el cuerpo del `let`. Esto significa que tal variable no será confuso con cualquier variable del mismo nombre fuera de la expresión `let`.

Podemos ver como la expresión `let` se ajusta dentro de la función como un todo mostrando una plantilla para `append-to-buffer` con la expresión `let` en línea:

```
(defun append-to-buffer (buffer start end)
  "documentacion..."
  (interactive ...)
  (let ((variable valor))
       cuerpo...)
```

La expresión `let` tiene tres elementos:

1. El símbolo `let`;
2. Una varlist conteniendo, en este caso, una lista simple de dos elementos, `(variable value)`;
3. El cuerpo de la expresión `let`.

En la función `append-to-buffer`, la varlist se parece a esto:

```
(oldbuf (current-buffer))
```

En esta parte de la expresión `let`, una variable, `oldbuf` es emparejada al valor devuelto por la expresión `(current-buffer)`. La variable, `oldbuf`, es usada para guardar la traza del búffer en el que tu estás trabajando y desde el que se copiará.

El elemento o elementos de una varlist son rodeados por un conjunto de paréntesis así el intérprete Lisp puede distinguir la varlist desde el cuerpo del `let`. Como consecuencia, la lista de dos elementos con la varlist está rodeados por un circunscrito conjunto de paréntesis. Las líneas se ven así:

```
(let ((oldbuf (current-buffer)))
  ... )
```

Los dos paréntesis antes de `oldbuf` podrían sorprenderte si no fuera porque los primeros paréntesis antes de `oldbuf` marcan el límite de la varlist y el segundo paréntesis marca el principio de la lista de dos elementos, `(oldbuf (current-buffer))`.

4.4.3 `save-excursion` en `append-to-buffer`

El cuerpo de la expresión `let` en `append-to-buffer` consiste de una expresión `save-excursion`.

La función `save-excursion` guarda las localizaciones de punto y la marca, y las restaura a estas posiciones después de las expresiones en el cuerpo de la ejecución completa `save-excursion`. Además, `save-excursion` completa la ejecución. Además, `save-excurion` guarda la traza del búffer original, y lo restaura. Esto es cómo `save-excursion` es usado en `append-to-buffer`.

Incidentalmente, no se valora nada aquí que una función Lisp es normalmente formateada así que cada cosa que es encerrada en conjunto multilínea que es indentada más a la derecha que el primer símbolo. En esta definición de función, el `let` es indentado más que `defun`, y el `save-excursion` es indentado más que el `let`, como esto:

```
(defun ...
  ...
  ...
  (let...
    (save-excursion
      ...
```

Esta convención formatea que sea fácil de ver que las líneas en el cuerpo de `save-excursion`, solo como `save-excurion` por sí mismo están encerradas por los paréntesis asociados con el `let`:

```
(let ((oldbuf (current-buffer)))
  (save-excursion
    ...
    (set-buffer ...)
    (insert-buffer-substring oldbuf start end)
    ...))
```

El uso de la función `save-excursion` puede ser vista como un proceso de rellenar slots de una plantilla:

```
(save-excursion
  primera-expresion-en-cuerpo
  segunda-expresion-en-cuerpo
   ...
  ultima-expresion-en-cuerpo)
```

En esta función, el cuerpo de `save-excursion` contiene solo una expresión, la expresión `let*`. Se conoce una función `let`. La función `let*` es diferente. Eso tiene un '`*`' en su nombre. Eso permite a Emacs asignar cada variable en su varlist en secuencia, uno después de otro.

Su funcionalidad crítica es que las variable después en la varlist puedan hacer uso de los valores para los que Emacs asigna variables pronto en la varlist. Véase "fwd-para let", página 142.

Se escaparán funciones como `let*` y focaliza en dos: la función `set-buffer` y la función `insert-buffer-substring`.

En antaño, la expresión `set-buffer` era simple:

```
(set-buffer (get-buffer-create buffer))
```

pero ahora eso es

```
(set-buffer append-to)
```

`append-to` se asigna a `(get-buffer-create-buffer)` pronto en la expresión `let*`. Esta asociación extra no sería necesaria excepto para este `append-to` es usado después en la varlist como un argumento para `get-buffer-window-list`.

La definición de la función `append-to-buffer` inserta texto desde el búffer en el que estás actualmente a un buffer nombrado. Eso sucede que `insert-buffer-substring` copia texto desde otro búffer al búffer actual, solo el inverso — que es porque la definición `append-to-buffer` empieza con un `let` que asocia el símbolo local `oldbuf` al valor devuelto por `current-buffer`.

La expresión `insert-buffer-substring` se ve como esto:

```
(insert-buffer-substring oldbuf start end)
```

La función `insert-buffer-substring` copia una cadena *from* al búffer especificado como su primer argumento e inserta la cadena dentro del búffer presente. En este caso, el argumento para `insert-buffer-substring` es el valor de la variable creada y asociada por el `let`, llama al valor de `oldbuf`, que fué el búffer actual cuando tu diste el comando `append-to-buffer`.

Después de que `insert-buffer-substring` ha hecho su trabajo, `save-excursion` restaurará la acción al búffer original y `append-to-buffer` habrá hecho su trabajo.

Escrito en forma esquelética, los trabajos del cuerpo se ven como esto:

```
(let (bind-oldbuf-to-value-of-current-buffer)
  (save-excursion                       ; Guarda la traza del búffer.
    change-buffer
    insert-substring-from-oldbuf-into-buffer)

  change-back-to-original-buffer-when-finished
 let-the-local-meaning-of-oldbuf-disappear-when-finished
```

En resumen, `append-to-buffer` funciona como sigue: se guarda el valor del búffer actual en la variable llamada `oldbuf`. Se obtiene el nuevo búffer (creando uno si necesita ser) y cambia la atención de Emacs a eso. Usando el valor de `oldbuf`, inserta la región del texto desde el viejo búffer dentro del nuevo búffer; y entonces usando `save-excursion`, trae atrás a tu búffer original.

Buscando `append-to-buffer`, se ha explorado una función limpia compleja. Eso muestra como usar `let` y `save-excursion`, y como cambiar y volver desde otro buffer. Muchas definiciones de función usan `let`, `save-excursion`, y `set-buffer` de este modo.

4.5 Revisar

Aquí está un breve resumen de varias funciones discutidas en este capítulo.

`describe-function`
`describe-variable`
: Imprime la documentación para una función o variable. Convencionalmente asociada a *`C-h f`* y *`C-h v`*.

`find-tag`
: Encuentra el fichero que contiene la fuente para una función o variable y cambia buffer a él, posicionando el punto al principio del ítem. Convencionalmente emparejado a *`M-.`* (esto es un período seguiendo la tecla `META`).

`save-excursion`
: Guarda la localización de punto y marca y restaura sus valores después de los argumentos para `save-excursion` y han sido evaluados. También, recuerda el buffer actual y devuélvelo.

`push-mark`
: Asigna la marca en una localización y graba el valor de la marca previa en el anillo de la marca. La marca es una localización en el búffer que guarda su posición relativa incluso si el texto es añadido o borrado desde el búffer.

`goto-char`
: Asigna punto a la localización especificada por el valor del argumento, que puede ser un número, una marca, o una expresión que devuelve el número de una posición, tal como `(point-min)`.

`insert-buffer-substring`
: Copia una región de texto desde un búffer que es pasado a la función como un argumento e inserta la región dentro del búffer actual.

`mark-whole-buffer`
: Marca el búffer completo como una región. Normalmente asignado a *C-x h*.

`set-buffer`
: Cambia la atención de Emacs a otro búffer, pero no cambies la ventana siendo mostrada. Usado cuando el programa en vez de un humano trabaja en un búffer diferente.

`get-buffer-create`
`get-buffer`
: Encuentra un buffer nombrado o crea uno si un buffer de este nombre no existe. La función `get-buffer` devuelve `nil` si el nombre del búffer no existe.

4.6 Ejercicios

- Escribe tu propia definición de función `simplified-end-of-buffer`; entonces testea para ver si funciona.
- Usa `if` y `get-buffer` para escribir una función que imprime un mensaje contando si un buffer existe.
- Usando `find-tag`, encuentra la fuente para la función `copy-to-buffer`

5 Unas pocas funciones más complejas

En este capítulo, se construye lo que se aprendió en los capítulos previos mirando en funciones más complejas. La función `copy-to-buffer` ilustra el uso de expresiones `save-excursion` en una definición, mientras la función `insert-buffer` ilustra el uso de un asterisco en una expresión `interactive`, uso de `o`, y la importante distinción entre un nombre y el objeto para el que el nombre se refiere.

5.1 La definición de `copy-to-buffer`

Después de comprender cómo se trabaja `append-to-buffer`, es fácil para comprender `copy-to-buffer`. Esta función copia texto dentro de un búffer, pero en vez de añadir al segundo búffer, se reemplaza a todo el texto previo en el segundo búffer.

El cuerpo de `copy-to-buffer` se ve como esto,

```
...
(interactive "BCopy to buffer: \nr")
(let ((oldbuf (current-buffer)))
  (with-current-buffer (get-buffer-create buffer)
    (barf-if-buffer-read-only)
    (erase-buffer)
    (save-excursion
      (insert-buffer-substring oldbuf start end)))))
```

La función `copy-to-buffer` tiene una expresión simple `interactive` en vez de `append-to-buffer`.

La definición entonces dice:

```
(with-current-buffer (get-buffer-create buffer) ...
```

Primero, mira en la expresión interna más temprana; que es evaluada primero. Esta expresión empieza con `get-buffer-create buffer`. La función cuenta al ordenador para usar el búffer con el nombre específicado como uno para el que estás copiando, o si tal búffer no existe, créalo. Entonces, la función `with-current-buffer` evalúa su cuerpo con este búffer temporalmente al actual.

(Esto demuestra otro camino para cambiar la atención del ordenador pero no los usuarios. La función `append-to-buffer` muestra como hacer lo mismo con `save-excursion` y `set-buffer`. `with-current-buffer` es uno nuevo, y argumentablemente fácil, mecanismo.)

La función `barf-if-buffer-read-only` envía un mensaje de error diciendo al búffer es de solo lectura si no se puede modificar.

La siguiente línea tiene la función `erase-buffer` como sus únicos contenidos. Este función borra el búffer.

Finalmente, las últimas dos líneas contienen la expresión `save-excursion` con `insert-buffer-substring` como su cuerpo. La expresión `insert-buffer-substring` copia el texto desde el búffer en el que estás (y no se ha visto el ordenador puesta su atención, así no se sabe que este búffer es ahora llamado `oldbuf`).

De manera incidental, esto es lo que significa por 'reemplazo'. Para reemplazar texto Emacs borra el texto previo y entonces inserta el nuevo texto.

El código fuente, del cuerpo de `copy-to-buffer` se parece a esto:

```
(let (bind-oldbuf-to-value-of-current-buffer)
    (with-the-buffer-you-are-copying-to
      (but-do-not-erase-or-copy-to-a-read-only-buffer)
      (erase-buffer)
      (save-excursion
        insert-substring-from-oldbuf-into-buffer)))
```

5.2 La definición de `insert-buffer`

`insert-buffer` es todavía una función relacionada con el búffer. Este comando copia otro búffer *dentro* del búffer actual. Es lo inverso de `append-to-buffer` o `copy-to-buffer`, desde que se copia una región de texto *desde* el búffer actual a otro búffer.

Aquí hay una discusión basada en el código original. El código era simplificado en 2003 y es duro de comprender.

(Véase Sección 5.2.6 “Nuevo Cuerpo para `insert-buffer`”, página 62, para ver una discusión del nuevo cuerpo.)

Además, este código ilustra el uso de `interactive` con un búffer que podría ser *read-only* y la distinción entre el nombre de un objeto y el objeto actualmente referido.

Aquí está el primer código

```
(defun insert-buffer (buffer)
  "Inserta después del punto los contenidos del BUFFER.
Pon la marca después del texto insertado.
El BUFFER puede ser un buffer un nombre de buffer."
  (interactive "*bInsert buffer: ")
  (or (bufferp buffer)
      (setq buffer (get-buffer buffer)))
  (let (start end newmark)
    (save-excursion
      (save-excursion
        (set-buffer buffer)
        (setq start (point-min) end (point-max)))
      (insert-buffer-substring buffer start end)
      (setq newmark (point)))
    (push-mark newmark)))
```

Como con otras definiciones de función, se puede usar una plantilla para visión de la función:

```
(defun insert-buffer (buffer)
  "documentation..."
  (interactive "*bInsert buffer: ")
  body...)
```

5.2.1 La expresión interactiva en `insert-buffer`

En `insert-buffer`, el argumetno para la declaración `interactive` tiene dos partes, un asterisco, ‘`*`’, y ‘`bInserta un buffer: `’.

Un búffer de solo lectura

El asterisco es para la situación cuando el buffer actual es un buffer de solo lectura — un buffer que no puede ser modificado. Si `insert-buffer` es llamado cuando el buffer actual es de solo lectura, un mensaje a este efecto está impreso en el área echo y el terminal puede avisar; no se permite insertar cualquier cosa dentro del buffer. El asterisco no necesita ser seguido por una nueva línea para separarse desde el siguiente argumento.

'b' en una expresión interactiva

El siguiente argumento en la expresión interactiva empieza con una tecla minúscula '`b`'. (Esto es diferente desde el código para `append-to-buffer`, que usa una mayúscula '`B`'. Véase Sección 4.4 "La Definición de `append-to-buffer`", página 50.) La tecla minúscula cuenta al intérprete Lisp que el argumento para `insert-buffer` sería un buffer existente o sino su nombre. (La mayúscula '`B`' provee para la posibilidad que el búffer no existe.) Emacs te mostrará en pantalla el nombre del búffer, ofreciendo un búffer por defecto, con la compleción de nombre habilitado. Si el búffer no existe, se recibe un mensaje que dice "No concuerda"; tu terminal te avisa también.

El nuevo y simplificado código genera una lista `interactive`. Eso usa las funciones `barf-if-buffer-read-only` y `read-buffer` con las que estamos ya familiarizados y la forma especial `progn` con los que no. (Eso será descrito después).

5.2.2 El cuerpo de la función `insert-buffer`

El cuerpo de la función `insert-buffer` tiene dos partes principales: una expresión `or` y una expresión `let`. El propósito de la expresión `or` es asegura que el argumento `buffer` es emparejado a un buffer y no solo el nombre de un buffer. El cuerpo de la expresión `let` contiene el código que copia los otros buffer dentro del buffer.

En el "outline" (esquema), las dos expresiones se ajustan dentro de la función `insert-buffer` como esto:

```
(defun insert-buffer (buffer)
  "documentation..."
  (interactive "*bInsertar buffer: ")
  (or ...
      ...
  (let (varlist)
      body-of-let... )
```

Para comprender como la expresión `or` asegura que el argumento `buffer` es emparejado a un buffer y no al nombre de un búffer, es primero necesario comprender la función `or`.

Antes de hacer esto, permíteme reescribir esta parte de la función usando `if` así puedes ver que es hecho en una manera que será familiar.

5.2.3 `insert-buffer` con un `if` en vez de un `or`

El trabajo que debe ser hecho y asegura el valor de `búffer` es un buffer en sí mismo y no el nombre de un búffer. Si el valor es el nombre, entonces el buffer en sí debe ser obtenido.

Te puedes imaginar a tí mismo en una conferencia donde un acomodador está observando una lista con tu nombre dentro y mirándote: el acomodador sabe "asociar" tu nombre, pero no a tí; pero cuando el acomodador te encuentra y te toma el brazo, el acomodador llega a "asociarte" a tí.

En Lisp, se podría describir esta situación así:

```
(if (not (holding-on-to-guest))
    (find-and-take-arm-of-guest))
```

Se quiere hacer la misma cosa con un búffer — si no tenemos el búffer en sí, queremos tenerlo.

Usando un predicado llamado `bufferp` que nos cuenta si tenemos un búffer (en vez de su nombre), se puede escribir el código como este:

```
(if (not (bufferp buffer))              ; if-part
    (setq buffer (get-buffer buffer)))  ; then-part
```

Aquí, el true-or-false-test de la expresión `if` es `(not (bufferp buffer))`; y la then-part es la expresión `(setq buffer (get-buffer buffer))`.

En el test, la función `bufferp` devuelve cierto si su argumento es un búffer — pero falso si su argumento es el nombre del buffer. (El último carácter del nombre de la función `bufferp` es el carácter '`p`'; como se vió antes, tal uso de '`p`' es una convención que indica que la función es un predicado, que es un término que significa que la función determinará si alguna propiedad es verdadera o falsa. Véase Sección 1.8.4 "Usando el objeto de tipo incorrecto como un argumento", página 13.)

La función `not` precede la expresión `(bufferp buffer)`, así el true-or-false-test se ve como esto:

```
(not (bufferp buffer))
```

`no` es una fución que devuelve cierto si su argumento es falso y falso si su argumento es verdadero. Así si `(bufferp buffer)` devuelve cierto, la expresión `no` devuelve falso y vice-versa: que es "no cierto" es falso que es "no falso" es verdadero.

Usando este test, la expresión `if` trabaja como sigue: cuando el valor de la variable `buffer` está actualmente en un búffer en vez de su nombre, el test true-or-false-test devuelve false y la expresión `if` no evalúa la parte then-part. Esto está bien, desde que no necesita para hacer cualquier cosa para la variable `buffer` si es realmente un búffer.

Por otro lado, cuando el valor de `buffer` no es un buffer en sí, pero el nombre de un buffer, el true-or-false-test devuelve cierto y la then-part de la expresión es evaluada. En este caso, la then-part es `(setq buffer (get-buffer buffer))`. Esta expresión usa la función `get-buffer` para devolver un buffer actual en sí, dado su nombre. El `setq` entonces asigna la variable `buffer` al valor del buffer en sí, reemplazando su valor previo (que era el nombre del buffer).

5.2.4 El or en el cuerpo

El propósito de la expresión or en la función insert-buffer es asegurar que el argumento buffer está asociado a un búffer y no solo al nombre de un búffer. La sección previa muestra como el trabajo podría haber sido hecho usando una expresión if. Sin embargo, la función insert-buffer actualmente usa or. Para comprender este, es necesario comprender como or trabaja.

Una función or puede tener cualquier número de argumentos. Eso evalúa cada argumento en turno y devuelve el valor del primero de sus argumentos que no es nil. También, y esto es una funcionalidad crucial de or, eso no evalúa cualquier argumentos subsiguientes después devolviendo el primer valor no-nil.

La expresión or se ve como esto:

```
(or (bufferp buffer)
    (setq buffer (get-buffer buffer)))
```

El primer argumento a or es la expresión (bufferp buffer). Esta expresión devuelve cierto (un valor no-nil) si el buffer es actualmente un buffer, y no solo el nombre de un buffer. En la expresión or, si este es el caso, la expresión or devuelve esto el valor cierto y no evalúa la siguiente expresión — y esto es bueno para nosotros, desde que nosotros no queremos hacer cualquier cosa al valor de buffer si eso es realmente un buffer.

Por otro lado, si el valor de (bufferp buffer) es nil, que será si el valor de buffer es el nombre de un buffer, el intérprete Lisp evalúa el siguiente elemento de la expresión. Esta es la expresión (setq buffer (get-buffer buffer)). Esta expresión devuelve un valor no-nil, que es el valor para el que asigna la variable buffer — y este valor es un búffer en sí, no el nombre de un búffer.

El resultado de todo esto es que el símbolo buffer es siempre asociado a un búffer en sí en vez del nombre de un búffer. Toda es necesario porque la función set-buffer en una línea siguiente trabaja con un buffer en sí, no con el nombre de un búffer.

Incidentalmente, usando or, la situación con el acomodador se vería así:

```
(or (holding-on-to-guest) (find-and-take-arm-of-guest))
```

5.2.5 La expresión let en insert-buffer

Después asegurando que la variable buffer se refiere a un buffer en sí y no solo al nombre de un buffer, la función insert-buffer continúa con una expresión let. Esto especifica tres variables locales, start, end y newmark y los asocia al valor inicial nil. Estas variables son usadas dentro del resto de let y temporalmente se oculta con cualquier otra ocurrencia de variables del mismo nombre en Emacs hasta el fin del let.

El cuerpo del let contiene dos expresiones save-excursion. Primero, miraremos la expresión save-excursion en detalle. La expresión se parece a esto:

```
(save-excursion
  (set-buffer buffer)
  (setq start (point-min) end (point-max)))
```

La expresión (set-buffer buffer) cambia la atención de Emacs desde el búffer actual a uno desde el que el texto será copiado. En este búffer las variables start y

`end` se asignadan al principio y al fin del búffer, usando los comandos `point-min` y `point-max`. Note que tenemos aquí una ilustración de cómo `setq` es capaz de asignar dos variables en la misma expresión. El primer argumento de `setq` es asignar al valor del segundo, y su tercer argumento está asignado al valor del cuarto.

Después el cuerpo del `save-excursion` propio es evaluado, el `save-excursion` restaura el búffer original, pero `start` y `end` permanece asignado a los valores del principio y fin del búffer de el que el texto será copiado.

La expresión por fuera `save-excursion` se ve como:

```
(save-excursion
  (inner-save-excursion-expression
     (go-to-new-buffer-and-set-start-and-end)
  (insert-buffer-substring buffer start end)
  (setq newmark (point)))
```

La función `insert-buffer-substring` copia el texto *dento* del búffer *desde* la región indicada por `start` y `end` en el `búffer`. Desde el total del segundo búffer cae entre `start` y `end`, el todo del segundo búffer es copiado dentro del búffer que estás editando. Lo siguiente, el valor del punto, que será al fin del texto insertado, es grabado en la variable `newmark`.

Después el cuerpo de `save-excursion` es evaluado, punto y marca son recolocados a sus lugares originales.

Sin embargo, es conveniente localizar una marca al fin del texto nuevamente insertado y localizar el punto al principio. La variable `newmark` graba el fin del texto insertado. En la última línea de la expresión `let`, la expresión de la función `(push-mark newmark)` asigna una marca a esta posición. (La posición previa de la marca está todavía accesible; está grabado en la marca del anillo y se puede regresar a eso con *`C-u C-SPC`*.) Mientras tanto, el punto está localizado al principio del texto insertado, que está donde estaba antes de ser llamado la función que inserta, la posición de lo que estaba guardado por la primera `save-excursion`.

La expresión `let` se parece a esto:

```
(let (start end newmark)
  (save-excursion
    (save-excursion
      (set-buffer buffer)
      (setq start (point-min) end (point-max)))
    (insert-buffer-substring buffer start end)
    (setq newmark (point)))
  (push-mark newmark))
```

Como la función `append-to-buffer`, la función `insert-buffer` usa `let`, `save-excursion` y `set-buffer`. Además, la función ilustra un camino para usar `o`. Toda estas funciones están construyendo bloque que encontrarán y usarán una y otra vez.

5.2.6 Nuevo cuerpo para `insert-buffer`

El cuerpo en la versión de GNU Emacs 22 es más confuso que en el original.

Consiste de dos expresiones

```
(push-mark
 (save-excursion
   (insert-buffer-substring (get-buffer buffer))
   (point)))

 nil
```

excepto, y esto es lo que los novicios confunden, un trabajo muy importantes es hecho dentro de la expresión `push-mark`.

La función `get-buffer` devuelve un buffer con el nombre proporcionado. Tu tomarás nota de que la función *no* es llamado `get-buffer-create`; eso no crea un búffer si uno no existe ya. El búffer devuelto por `get-buffer`, un búffer existente, es pasado a `insert-buffer-substring`, que inserta el total del búffer (desde que no se especificón ninguna cosa más).

La posición dentro del buffer es insertado es grabado por `push-mark`. Entonces la función devuelve `nil`, el valor de su último comando. Pon otro camino, la función `insert-buffer` existe solo para producir un efecto lateral, insertando otro buffer, no para devolver cualquier valor.

5.3 Definición completa de `beginning-of-buffer`

La estructura básica de la función `beginning-of-buffer` ya ha sido discutida. (Véase Sección 4.2 “Una Definición Simplificada `beginning-of-buffer`”, página 47). Esta sección describe la parte compleja de la definición.

Como se describe previamente, cuando se invoca sin un argumento, `beginning-of-buffer` mueve el cursor al principio del búffer (en realidad, al principio de la porción accesible del búffer), dejando la marca en la posición previa. Sin embargo, cuando el comando es invocado con un número entre uno y diez, la función considera que número será una fracción del tamaño del búffer, medido en decenas, y Emacs mueve el cursor en esta fracción del camino desde el principio del búffer. De este modo, se puede bien llamar a esta función con la tecla comando `M-<`, que moverá el cursor al principio del búffer, o con una tecla tal como `C-u 7 M-<` que moverá el cursor a un punto 70% del camino a través del búffer. Si un número más grande que diez es usado para el argumento se mueve al fin del búffer.

La función `beginning-of-buffer` puede ser llamado con o sin argumentos. El uso del argumento es opcional.

5.3.1 Argumentos opcionales

A menos que se cuente de otro modo, Lisp espera que una función con un argumento en su definición de función se llame con un valor para este argumento. Si esto no ocurre, se obtiene un error y un mensaje que dice ‘`Número de argumentos erróneo`’.

Sin embargo, los argumentos opcionales son una funcionalidad de Lisp: una *palabra clave* particular es usada para contar al intérprete Lisp que un argumento es opcional. La palabra clave es `&optional`. (El ‘`&`’ en frente de ‘`opcional`’ es parte de la palabra clave.) En una definición de función, si un argumento sigue a la

palabra clave `&optional`, ningún valor necesita ser pasado a este argumento cuando la función se llama.

La primera línea de la definición de función de `beginning-of-buffer` tiene lo siguiente:

```
(defun beginning-of-buffer (&optional arg)
```

En el "outline" (esquema), la función completa se parece a esto:

```
(defun beginning-of-buffer (&optional arg)
  "documentation..."
  (interactive "P")
  (or (is-the-argument-a-cons-cell arg)
      (and are-both-transient-mark-mode-and-mark-active-true)
      (push-mark))
  (let (determine-size-and-set-it)
  (goto-char
    (if-there-is-an-argument
        figure-out-where-to-go
      else-go-to
      (point-min))))
   do-nicety
```

La función es similar a la función `simplified-beginning-of-buffer` excepto que la expresión `interactive` tiene `"P"` como un argumento y la función `goto-char` es seguida por una expresión if-then-else que figura donde poner el cursor si hay un argumento que no es un cons cell.

(Puesto que no se explica un cons cell en muchos capítulos, por favor, considere ignorar la función `consp`. Capítulo 9 “Cómo las Listas son Implementadas”, página 104, y Sección “Cons Cell y Tipos de Listas” in *El Manual de Referencia GNU Emacs Lisp*).

El `"P"` en la expresión `interactive` cuenta a Emacs cómo pasar un argumento prefijo, si hay uno, a la función en forma plana. Un argumento prefijo se crea escribiendo la tecla `META` seguida por un número, o escribiendo `C-u` y entonces un número. (Si no escribes un número, `C-u` por defecto a un cons cell con un 4. Una minúscula `"p"` en la expresión `interactive` causa a la función convertir un argumento prefijo a un número.)

El true-or-false-test de la expresión `if` se ve compleja, pero no lo es: se chequea si `arg` tiene un valor que no es `nil` y si es un cons cell. (Esto es lo que `consp` hace; chequea si su argumento es un cons cell.) Si `arg` tiene un valor que no es `nil` (y no es un cons cell.), que será el caso si `beginning-of-buffer` se llama con un argumento, entonces este true-or-false-test devolverá cierto y la then-part de la expresión `if` falsa. Por otro lado, si `beginning-of-bufer` no se llama con un argumento, el valor de `arg` será `nil` y la else-part de la expresión `if` se evaluará. La else-part es simple `point-min`, y esto es lo de fuera, la expresión `goto-char` es `(goto-char (point-min))`, que es cómo se vió la función `beginning-of-buffer` en su forma simplificada.

5.3.2 `beginning-of-buffer` con un argumento

Cuando `beginning-of-buffer` se llama con un argumento, una expresión es evaluada que calcula que valor pasa a `goto-char`. Esto es incluso complicado a primera vista. Eso incluye una expresión `if` propia y mucha aritmética. Se ve así:

```
(if (> (buffer-size) 10000)
    ;; ¡Evitar sobrecarga para grandes tamaños de búffer!
                           (* (prefix-numeric-value arg)
                              (/ size 10))
  (/
   (+ 10
      (*
       size (prefix-numeric-value arg))) 10)))
```

Como otras expresiones que se ven complejas, la expresión condicional con `beginning-of-buffer` puede ser desenredada mirándola por partes de una plantilla, en este caso, la plantilla par una expresión if-then-else. En forma esquelética, la expresión se ve así:

```
(if (buffer-is-large
    divide-buffer-size-by-10-and-multiply-by-arg
  else-use-alternate-calculation
```

El true-or-fase-test de esta expresión `if` propia chequea el tamaño del buffer. La razón para esto es que la versión vieja de Emacs 18 usaba números que no son más grandes que 8 millones o así y en la computación que seguía, el programador temía que Emacs podría intentar usar a través de largos números si el búffer fuera largo. El término 'sobrecarga', que se mencionó en el comentario, significa que los números son grandes. Las versiones más recientes de Emacs usan números largos, pero este código no ha sido tocado, solo porque la gente ahora mira en búffers que están lejos, tan lejos como antes.

Hay dos casos: si el búffer es largo, o si no.

Qué ocurre en búffer largo

En `beginning-of-buffer`, la expresión propia `if` chequea si el tamaño del búffer es mayor que 10000 caracteres. Para hacer esto, se usa la función `>` y la computación de `size` que viene desde la expresión let.

Hace tiempo, se usaba la función `buffer-size`. No solo esta función era llamada varias veces, eso daba el tamaño del búffer completo, no la parte accesible. La computación tiene mucho más sentido cuando se maneja solo la parte accesible. (Véase Capítulo 6 "Encogiendo y extendiendo", página 70, para más información en focalizar la atención para una parte 'accesible'.)

La línea se parece a esto:

```
(if (> size 10000)
```

Cuando el búffer es largo, el then-part de la expresión `if` se evalúa. Eso se lee así (después de ser formateado para una fácil lectura):

```
(*
  (prefix-numeric-value arg)
  (/ size 10))
```

Esta expresión es una multiplicación, con dos argumentos para la función `*`.

El primer argumento es `(prefix-numeric-value arg)`. Cuando `"P"` se usa como argumento para `interactive`, el valor pasado para la función como argumento es un "argumento prefijo crudo", y no un número. (Es un número en una lista). Para desarrollar la aritmética, una conversión es necesaria, y `prefix-numeric-value` hace el trabajo.

El segundo argumento es `(/ size 10)`. Esta expresión divide el valor numérico por diez — el valor numérico del tamaño de la porción accesible del búffer. Esto produce un número que cuenta cuántos caracteres crean una decena del tamaño del búffer. (En Lisp, `/` es usado para división, solo como `*` es usado para multiplicación.)

En la expresión de la multiplicación como un todo, esta cantidad se multiplica por el valor del argumento prefijo — la multiplicación se parece a:

```
(* numeric-value-of-prefix-arg
   number-of-characters-in-one-tenth-of-the-accessible-buffer)
```

Si, por ejemplo, el argumento prefijo es '7', el valor one-tenth será multiplicado por 7 para dar una posición del 70% del camino.

El resultado de todo esto es que si la porción accesible del búffer es largo, la expresión `goto-char` se lee esto:

```
(goto-char (* (prefix-numeric-value arg)
              (/ size 10)))
```

Esto pone el cursor donde se quiere.

Qué ocurre en un búffer pequeño

Si el búffer contiene poco más de 10000 caracteres, una computación ligeramente diferente es medida. Se podría pensar que esto no es necesario, desde que la primera computación podría hacer el trabajo. Sin embargo, en un búffer pequeño, el primer método puede no poner el cursor en la línea exactamente deseada; el segundo método hace un trabajo mejor.

El código se parece a esto:

```
(/ (+ 10 (* size (prefix-numeric-value arg))) 10))
```

Este es el código en el que se ve qué ocurre descubriendo como las funciones se embeben entre paréntesis. Eso es fácil de leer si se reformatea con cada expresión indentada más profundamente que la expresión que encierra:

```
(/
 (+ 10
    (*
     size
     (prefix-numeric-value arg)))
 10))
```

Mirando los paréntesis, se ve que la operación propia es `(prefix-numeric-value arg)`, que convierte el argumento plano para un número. En la siguiente expresión, este número es multiplicado por el tamaño de la porción accesible del búffer:

```
(* size (prefix-numeric-value arg))
```

Esta multiplicación crea un número que puede ser más largo que el tamaño del buffer — siete veces más larga si el argumento es 7, por ejemplo. Diez se aãnade a éste nũmero y finalmente el número es dividido por 10 para proporcionar un valor que es un carácter más largo que la posición de porcentaje en el búffer.

El número que resulta de todo esto se pasa a `goto-char` y el cursor se mueve a este punto.

5.3.3 El `beginning-of-buffer` completo

Aquí está el texto completo de la función `beginning-of-buffer`:

```
(defun beginning-of-buffer (&optional arg)
  "Mueve el punto al principio del buffer;
deja marca en la posición previa.
Con prefijo \\[universal-argument],
no se asigna una marca en la posición previa.
Con el argumento numérico N, pon el
punto N/10 del camino desde el principio.
Si el búffer está encogido
este comando usa el principio y tamaño
de la parte accesible del búffer.

¡No use este comando en programas Lisp!
\(goto-char (point-min)) es rápido
y evita poseer la marca."
  (interactive "P")
  (or (consp arg)
      (and transient-mark-mode mark-active)
      (push-mark))
  (let ((size (- (point-max) (point-min))))
    (goto-char (if (and arg (not (consp arg)))
                   (+ (point-min)
                      (if (> size 10000)
                          ;; ¡Evita sobrecarga para grandes tamaños de búffer!
                          (* (prefix-numeric-value arg)
                             (/ size 10))
                        (/ (+ 10 (* size (prefix-numeric-value arg)))
                           10)))
                 (point-min))))
  (if arg (forward-line 1)))
```

Excepto por dos pequeños puntos, la discusión previa muestra cómo esta función trabaja. El primer punto trata un detalle en la cadena de documentación, y el segundo concierne la última línea de la función.

En la cadena de documentación, hay referencia a una expresión:

```
\\[universal-argument]
```

Un '`\\`' es usado antes de la primera llave de esta expresión. Este '`\\`' le cuenta al intérprete Lisp sustituir qué clave está actualmente emparejada a los '`[...]`'. En el caso de `universal-argument`, que es normalmente `C-u`, pero eso podría ser diferente. (Véase Sección "Consejos para Cadenas de Documentación" in *El Manual de Referencia de GNU Emacs Lisp*, para más información.)

Finalmente, la última línea del comando `beginning-of-buffer` dice mover el punto al principio de la siguiente línea si el comando es invocado con un argumento:

```
(if arg (forward-line 1)))
```

Esto pone el cursor al principio de la primera línea después de las apropiadas decenas de posiciones en el búffer. Esto significa que el cursor está siempre localizado *al menos* las decenas solicitadas del camino a través del búffer, que es un bien que es, quizás, no necesario, pero que, si no ocurrió, estaría seguro de dibujar rumores.

Por otro lado, eso significa que si se especifica el comando con un `C-u`, sin un número, que es decir, si el 'prefijo de argumento plano' es simplemente un cons cell, entonces el comando te pone al principio de la segunda línea ... no sé si este se pretende ningún trato con el código para evitar que esto ocurre.

5.4 Revisar

Aquí hay un breve resumen de los asuntos cubierto en este capítulo.

`or` Evalúa cada argumento en secuencia, y devuelve el valor del primer argumento que no es `nil`, si ninguno devuelve un valor que no es `nil`, devuelve `nil`. En breve, devuelve el primer valor de verdad de los argumento; devuelve un valor cierto si un *or* cualquier de los otros son verdad.

`and` Evalúa cada argumento en secuencia, y si cualquiera es `nil`, devuelve `nil`; si ninguno es `nil`, devuelve el valor del último argumento. En breve, devuelve un valor cierto solo si todos los argumentos cierto; devuelve un valor cierto si un *and* cada uno de los otros son ciertos.

`&optional` Una palabra clave usaba para indicar que un argumento a una definición de función es opcional; esto significa que la función puede ser evaluado sin el argumento, si se desea.

`prefix-numeric-value`
Convierte el 'argumento prefijo plano' producido por `(interactive "P")` a un valor numérico.

`forward-line`
Mueve el punto hacia delante al principio de la siguiente línea, o si el argumento es más de uno, hacia delante varias líneas. Si eso no se puede mover tan lejos hacia delante como se puede, `forward-line` va hacia delante tan lejos como se puede y entonces devuelve un contaje del número de líneas adicionales que no pudo moverse.

`erase-buffer`
: Borra todos los contenidos del búffer actual.

`bufferp`
: Devuelve `t` si su argumento es un búffer; de otro modo devuelve `nil`.

5.5 Ejercicio de argumento `opcional`

Escribe una función interactiva con un argumento opcional que chequee si su argumento, un número, es mayor o igual, o al menos, menos que el valor de `fill-column`, y lo escribe, en un mensaje. Sin embargo, si no se pasa un argumento a la función, usa 56 como un valor por defecto.

6 Encogiendo y extendiendo

Encoger es una funcionalidad de Emacs que hace posible focalizar en una parte específica de un búffer, y funcionar sin cambiar accidentalmente otras partes. Encoger normalmente se deshabilita puesto que puede confundir a novatos.

Con encoger, el resto del búffer se hace invisible, como si no estuviera. Esto es una ventaja si, por ejemplo, se quiere reemplazar una palabra en una parte del búffer pero no otro: se encoge la parte que se quiere y el reemplazo se trae solo en esta sección, no en el resto del búffer. Las búsquedas solo funcionarán con una región encogida, no fuera de una, así si tu estás arreglando una parte de un documento, se puede guardar en sí desde encontrar partes accidentalmente que no necesitas arreglar encongiendo solo la región que quieres. (La tecla asociada a `narrow-to-region` es *C-x n n*.)

Sin embargo, encoger hace que el resto del búffer sea invisible, esto puede asustar a gente quien inadvertidamente invoca a encoger y pensar que se ha borrado una parte de su fichero. Más allá, el comando `undo` (que es normalmente emparejado a *C-x u*) no se deja encoger, así las personas pueden llegar a estar bastante desesperadas si no conocen que ellas pueden volver al resto de un búffer para visibilizarlo con el comando `widen`. (El emparejamiento de la tecla para `widen` es *C-x n w*.)

Encoger es solo tan útil al intérprete Lisp como para un humano. Con frecuencia, una función Emacs Lisp está diseñada para trabajar en solo parte de un búffer; o de manera conversa, una función Emacs Lisp necesita trabajar en todo un búffer que ha sido encogido. La función `what-line`, por ejemplo, borra el encogimiento desde un búffer, si eso tiene cualquier encogimiento y cuando eso ha finalizado su trabajo, restaura el encogimiento para el que fuera. Por otro lado, la función `count-lines`, que es llamada por `what-line`, usa el encogimiento para restringirse a sí misma solo a la porción del búffer en el que está interesada y entonces restaura la situación previa.

6.1 La forma especial `save-restriction`

En Emacs Lisp, se puede usar la forma especial `save-restriction` para guardar la traza de siempre que el encogimiento esté en efecto. Cuando el intérprete Lisp se encuentra con `save-restriction`, eso ejecuta el código en el cuerpo de la expresión `save-restriction`, y entonces deshace cualquier cambio para encoger lo que el código causó. Si, por ejemplo, el búffer está encogido y el código que sigue al comando `save-restriction` devuelve el búffer para su región encogida. En el comando `what-line`, cualquier encogimiento del búffer que se puede tener se deshace por el comando `widen` que inmediatamente sigue el comando `save-restriction`. Cualquier encogimiento original es restaurado solo antes de la compleción de la función.

La plantilla para una expresión save-restriction es simple:

```
(save-restriction
  body... )
```

El cuerpo del save-restriction es una o más expresiones que serán evaluadas en secuencia por el intérprete Lisp.

Finalmente, un punto a anotar: cuando se usa tanto save-excursion y save-restriction, uno correcto después del otro, deberías usar save-excursion fuera. Si se escribe en el orden inverso, se podría fallar para grabar el encogimiento en el búffer para el que Emacs cambia después de llamar a save-excursion. De este modo, cuando se escribe junto a save-excursion y save-restriction sería escrito así:

```
(save-excursion
  (save-restriction
    body...))
```

En otras circunstancias, cuando no se escribe junto, las formas especiales save-excursion y save-restriction deben ser escritas en el orden apropiado para la función.

Por ejemplo,

```
(save-restriction
  (widen)
  (save-excursion
  body...))
```

6.2 what-line

El comando what-line cuenta el número de la línea en la que el cursor se ha localizado. La función ilustra el uso de los comandos save-restriction y save-excursion. Aquí está el texto original de la función:

```
(defun what-line ()
  "Imprime el número de línea actual (en el búffer)
  del punto."
  (interactive)
  (save-restriction
    (widen)
    (save-excursion
      (beginning-of-line)
      (message "Línea %d"
               (1+ (count-lines 1 (point)))))))
```

(En versiones recientes de GNU Emacs, la función what-line se ha expandido para contarte el número de líneas en un búffer encogido tan bien como tu número de línea en un búffer ampliado. La versión reciente es más compleja que la versión que se muestra. Si se siente venturoso, podría querer mirarla después de figurarse como esta versión funciona. Probablemente se necesitará usar *C-h f* (describe-function). La versión nueva usa un condicional para determinar si el búffer se ha encogido.

(También, eso usa line-number-at-pos, que otras expresiones simples, tales como (goto-char (point-min)), mueve el punto al principio de la línea actual con (forward-line 0) en vez de beginning-of-line.)

La función `what-line` como se muestra aquí tiene una línea de documentación y es interactiva, como se esperaría. Las dos líneas siguientes usan las funciones `save-restriction` y `widen`.

La forma especial `save-restriction` nota que encogiendo es en efecto, si cualquiera, en el buffer actual y restaura que encogiendo después del código en el cuerpo del `save-restriction` ha sido evaluada.

La forma especial `save-restriction` es seguida por `widen`. Esta función deshace cualquier distancia del actual buffer que puede haber tenido cuando `what-line` se llame. (La distancia que había es la distancia que `save-restriction` recuerda.) Esta ampliación se hace posible para la línea contando comandos a contar desde el principio del búffer. De otro modo, habría sido limitado para contar con la región accesible. Cualquier distancia original es restaurada solo antes de la compleción de la función por la forma especial `save-restriction`.

La llamada a `widen` es seguida por `save-excursion`, que guarda la posición del cursor (por ej., el punto) y de la marca, y la restaura después el código en el cuerpo del `save-excursion` usa la función `beginning-of-line` para mover el punto.

(Nótese que la expresión `(widen)` viene entre las formas especiales `save-restriction` y `save-excursion`. Cuando se escribe las dos expresiones `save- ...` en secuencia, escribe `save-excursion` finalmente.)

Las últimas dos líneas de la función `what-line` son funciones para contar el número de líneas en el búffer y entonces imprimir el número en el área echo.

```
(message "Line %d"
         (1+ (count-lines 1 (point)))))))
```

La función `message` imprime un mensaje de una línea abajo de la pantalla Emacs. El primer argumento está dentro de marcas de cita y está impreso como una cadena de caracteres. Sin embargo, se puede contener una expresión '`%d`' para imprimir un argumento siguiente. '`%d`' imprime el argumento como un decimal, así el mensaje dirá alguna cosa tal como '`Línea 243`'.

El número que está impreso en lugar de '`%d`' está computada por la última línea de la función:

```
(1+ (count-lines 1 (point)))
```

Lo que esto hace es contar las líneas desde la primera posición del búffer indicada por el `1`, subir al `(point)`, y entonces añadir uno a este número. (La función `1+` añade uno a su argumento.) Nosotros aändir a eso porque la línea 2 tiene solo una línea antes, y `count-lines` cuenta solo las líneas *antes* de la línea actual.

Después de que `count-lines` ha hecho su trabajo, y el mensaje ha sido impreso en el área echo, la función `save-excursion` restaura punto y marca a sus posiciones originales; y `save-restriction` restaura la contracción original, si la hay.

6.3 Ejercicio de encoger

Escribe una función que mostrará los primeros 60 caracteres del buffer actual, incluso si se ha encogido el buffer a su mitad así que la primera línea es inaccesible. Restaura punto, marca y encogimiento. Para este ejercicio, se necesita usa un popurri entero de funciones, incluyendo `save-restriction`, `widen`, `goto-char`, `point-min`, `message`, y `buffer-substring`.

(`buffer-substring` es una función previamente no mencionada que tendrás que investigar por tí mismo; o quizás tendrás que usar `buffer-substring-no-properties` o `filter-buffer-substring` . . . , todavía otras funciones. Las propiedades de texto son una funcionalidad que de otro modo no serían discutidas aquí. Véase Sección "Propiedades de Texto" in *El Manual de Referencia de Emacs Lisp*.)

Además, ¿realmente se necesita `goto-char` o `point-min`? ¿O se puede escribir la función sin ellos?

7 car, cdr, cons: Funciones fundamentales

En Lisp, `car`, `cdr`, y `cons` son funciones fundamentales. La función `cons` es usada para construir listas, y las funciones `car` y `cdr` son usadas tomarlas aparte.

En el paseo a través de la función `copy-region-as-kill`, se verá `cons` tan bien como dos variantes de `cdr`, llamadas `setcdr` y `nthcdr`. (Véase Sección 8.3 “copy-region-as-kill”, página 90.)

El nombre de la función `cons` es razonable: es una abreviación de la palabra ‘constructo’. Los orígenes de los nombres `car` y `cdr`, por otro lado, son esotéricos: `car` es un acrónimo desde la frase ‘Contenidos de la parte de la Dirección del Registro’; y `cdr` (pronunciado ‘could-er’) es un acrónimo desde la frase ‘Contenidos del Decremento del Registro’. Estas frases se refieren a piezas específicas de hardware en el ordenador en el que el Lisp original fué desarrollado. El resto de frases han sido completamente irrelevantes por más de 25 años a cualquiera pensando acerca de Lisp. Ninguno, excepto unos pocos académicos valientes han empezado a usar nombres más razonables para estas funciones, los viejos términos están todavía en uso. En particular, los términos usados en el código fuente Emacs Lisp, que usaremos en esta introducción.

7.1 car y cdr

El CAR de una lista es, bastatante simple, el primer ítem en la lista. De este modo, el CAR de la lista `(roer violet daisy buttercup)` es `rose`.

Si estás leyendo en Info y en GNU Emacs, se puede ver esto evaluando lo siguiente:

```
(car '(rose violet daisy buttercup))
```

Después de evaluar la expresión, `rose` aparecerá en el área echo.

Claramente, un nombre más razonable para la función `car` sería `first` y esto es con frecuencia lo que se sugiere.

`car` no elimina el primer ítem desde la lista; eso solo informa que eso es. Después `car` ha sido aplicado a una lista, la lista es todavía la misma que era. En la jerga, `car` es ‘no-destructiva’. Esta funcionalidad deja de ser importante.

El CDR de una lista es el resto de la lista, que es, la función `cdr` que devuelve la parte de la lista que sigue el primer ítem. De este modo, mientra el CAR de la lista `'(rose violet daisy buttercup)` es `rose`, el resto de la lista, el valor devuelto por la función `cdr`, es `(violet daisy buttercup)`.

Se puede ver esto evaluando lo siguiente del modo usual:

```
(cdr '(rose violet daisy buttercup))
```

Cuando se evalúa esto, `(violet daisy buttercup)` aparecerá en el área echo.

Al igual que `car`, `cdr` no elimina los elementos desde la lista — solo devuelve un informe de lo que el segundo y subsiguientes elementos son.

En el ejemplo, la lista de flores se cita. Si no, el intérprete Lisp intentaría evaluar la lista llamando a `rose` como una función. En este ejemplo, no queremos hacer esto.

Claramente, un nombre más razonable para `cdr` sería `rest`.

(Hay una lección aquí: cuando se nombran nuevas funciones, considere muy cuidadosamente lo que se está haciendo, ya que puede estar pegadas con los nombres largos que se esperan. La razón es que este doucumento perpetúa estos nombres que el código fuente de Emacs Lisp usa, y si no los usaba, se estaría un duro rato leyendo el código; pero, por favor, intente evitar usar estos términos por sí mismo. Las personas quienes vienen después se lo agradecerán.

Cuando `car` y `cdr` se aplica a una lista hecha de símbolos, tal como la lista `(pine fir oak maple)`, el elemento de la lista devuelta por la función `car` es el símbolo `pine` sin cualquier paréntesis alrededor. `pine` es el primer elemento en la lista. Sin embargo, el CDR de la lista es una lista por sí misma, `(fir oak maple)`, como se puede ver evaluando las siguientes expresiones en el modo usual:

```
(car '(pine fir oak maple))

(cdr '(pine fir oak maple))
```

Por otro lado, en una lista de listas, el primer elemento es en sí mismo una lista. `char` devuelve este primer elemento como una lista. Por ejemplo, la siguiente lista contiene tres sublistas, una lista de carnívoros, una lista de herbívoros y una lista de mamíferos:

```
(car '((leon tigre leopardo)
       (gacela antilope cebra)
       (ballena delfin foca)))
```

En este ejemplo, el primer elemento de CAR de la lista es la lista de carnívoros, `(lion tiger cheetah)`, y el resto de la lista es `((gazelle antelope zebra) (whale dolphin seal))`.

```
(cdr '((leon tigre leopardo)
       (gacela antilope cebra)
       (ballena delfin)))
```

Es un valor decir de nuevo que `car` y `cdr` son no destructivos — esto es, no se modifican o cambian listas para las que ser aplicados. Esto es muy importante por cómo son usados.

También, en el primer capítulo, en la discusión acerca de átomos, yo dije que en Lisp, "ciertos tipos de átomos, tales como un array, pueden separarse en partes; pero el mecanismo de hacer esto es diferente del mecanismo de separar una lista. Desde que Lisp se conoce, los átomos de una lista son indivisibles." (Véase Sección 1.1.1 "Átomos Lisp", página 1.) El `car` y el `cdr` son funciones que son usadas para dividir listas y son consideradas fundamentales para Lisp. Puesto que no pueden dividir o ganar acceso a las partes de un array, un array es considerado un átomo. De otro modo, la otra función fundamental, `cons`, se puede poner cerca o construir una lista, pero no un array. (Los arrays son manejados por funciones de array específicas. Véase Sección "Arrays" in *El Manual de Referencia de GNU Emacs Lisp*.)

7.2 `cons`

La función `cons` construye listas; eso es lo inverso de `car` y `cdr`. Por ejemplo, `cons` puede usarse para crear una lista de cuatro elementos desde los tres elementos de la lista, `(fir oak maple)`:

```
(cons 'pino '(abeto roble arce))
```

Después de evaluar esto, se verá lo siguiente:

```
(pino abeto roble arce)
```

aparece en el área echo. `cons` causa la creación de una nueva lista en el que el elemento es seguido por los elementos de la lista original.

Con frecuencia decimos que '`cons` pone un nuevo elemento al principio de una lista; adjunta o empuja el elemento dentro de la lista', pero esta frase puede ser incorrecta, puesto que `cons` no cambia a una lista existente, pero crea una nueva.

Como `car` y `cdr`, `cons` es no destructivo.

`cons` debe tener una lista para adjuntar a[1] No se puede empezar desde absolutamente nada. Si se está construyendo una lista, se necesita proveer al menos una lista vacía al principio. Aquí hay una serie de expresiones `cons` que construyen una lista de flores. Si estás leyendo esto en Info en GNU Emacs, se puede evaluar cada una de las expresiones en el camino usual; el valor es impreso en este texto después de '⇒', que puede leer como 'evaluas para'.

```
(cons 'buttercup ())
     ⇒ (buttercup)

(cons 'daisy '(buttercup))
     ⇒ (daisy buttercup)

(cons 'violet '(daisy buttercup))
     ⇒ (violet daisy buttercup)

(cons 'rosa '(violeta margarita mantequilla))
     ⇒ (rosa violeta margarita mantequilla)
```

En el primer ejemplo, la lista vacía se muestra como `()` y después se construye una lista de `mantequilla` seguida por la lista vacía. Como se puede ver, la lista vacía no se muestra en la lista que fué construida. Todo lo que se ve es `(mantequilla)`. La lista vacía no cuenta como un elemento de una lista porque no hay nada en una lista vacía. Generalmente hablando, una lista vacía es invisible.

El segundo ejemplo, `(cons 'daisy '(buttercup))` construye una nueva lista de dos elemento poniendo `daisy` en frente de `buttercup`; y el tercer ejemplo construye una lista de tres elementos poniendo `violet` en frente de `daisy` y `buttercup`.

7.2.1 Encuentra el tamaño de una lista: `length`

Se pueden encontrar cuántos elementos hay en una lista usando la función Lisp `length`, como en los siguientes ejemplos:

```
(length '(buttercup))
     ⇒ 1

(length '(daisy buttercup))
     ⇒ 2
```

[1] Actualmente, se puede `cons` un elemento para un átomo para producir a para punteado. Los pares punteados no se discuten aquí; ver Sección "Notación de Para Punteado" in *El Manual de Referencia de GNU Emacs Lisp*.

```
(length (cons 'violet '(daisy buttercup)))
     ⇒ 3
```

En el tercer ejemplo, la función `cons` es usada para construir una lista de tres elementos que entonces se pasa a la función `length` como su argumento.

Se puede también usar `length` para contar el número de elementos en una lista vacía:

```
(length ())
     ⇒ 0
```

Como se esperaría, el número de elementos en una lista vacía es cero.

Un experimento interesante es encontrar qué ocurre si se intenta encontrar el tamaño de una no lista en todo; que es, si se intenta llamar a `length` sin darle un argumento, incluso una lista no vacía:

```
(length )
```

Lo que se ve, si se evalúa esto, es el mensaje de error

```
Lisp error: (wrong-number-of-arguments length 0)
```

Esto significa que la función recibe el número incorrecto de argumentos, cero, cuando se espera algún otro número de argumentos. En este caso, se espera un argumento, el argumento de una lista cuyo tamaño de la función se está midiendo. (Nótese que *una* lista es *un* argumento, incluso si la lista tiene muchos elementos dentro.)

La parte del mensaje de error que dice '`length`' es el nombre de la función.

7.3 nthcdr

La función `nthcdr` está asociada con la función `cdr`. Que lo que hace es tomar la CDR de una lista repetidamente.

Si se toma el CDR de la lista `(pine fir oak maple)`, será devuelta la lista `(fir oak maple)`. Si se repite esto en lo que se devolvió, se devolverá lista lista `(oak maple)`. (De acuerdo, se repite CDR en la lista original solo se dará CDR desde la función que no cambia la lista. Se necesita evaluar el CDR del CDR y así.) Si se contiúa esto, finalmente será devuelta una lista vacía, que en este caso, en vez de ser mostrada como `()` es mostrado como `nil`.

Para revisar, aquí hay una serie de CDRs repetidos, el siguiente '⇒' muestra que se devuelve.

```
(cdr '(pine fir oak maple))
     ⇒(fir oak maple)

(cdr '(fir oak maple))
     ⇒ (oak maple)

(cdr '(oak maple))
     ⇒(maple)

(cdr '(maple))
     ⇒ nil

(cdr 'nil)
     ⇒ nil
```

```
(cdr ())
     ⇒ nil
```

Tambiés se pueden hacer varios CDRs sin imprimir los valores así:

```
(cdr (cdr '(pine fir oak maple)))
     ⇒ (oak maple)
```

En este ejemplo, el intérprete Lisp evalúa la lista propia dentro. La lista de dentro está entre comillas, así solo pasa la lista del `cdr` más interno. Este `cdr` pasa una lista hecho del segundo y subsiguientes elementos de la lista al `cdr` más externo, que produce una lista compuesta del tercer y subsiguientes elementos de la lista original. En este ejemplo, la función `cdr` se repite y devuelve una lista que consiste de la lista original sin sus primeros dos elementos.

La función `nthcdr` hace lo mismo que repitiendo la llamada a `cdr`. En el siguiente ejemplo, el segundo argumento se pasa a la función `nthcdr`, a través con la lista, y el valor devuelto es la lista sin sus primeros dos ítems, que son exactamente los mismos dos `cdr` en la lista:

```
(nthcdr 2 '(pine fir oak maple))
     ⇒ (oak maple)
```

Usando la lista original de cuatro elementos, se puede ver qué ocurre cuando varios argumentos numéricos son pasados a `nthcdr`, incluyendo 0, 1, y 5:

```
;; Deja la lista como estaba.
(nthcdr 0 '(pine fir oak maple))
     ⇒ (pine fir oak maple)

;; Devuelve una copia sin el primer elemento.
(nthcdr 1 '(pine fir oak maple))
     ⇒ (fir oak maple)

;; Devuelve una copia de la lista sin tres elementos.
(nthcdr 3 '(pine fir oak maple))
     ⇒ (maple)

;; Devuelve la lista faltando los cuatro elementos.
(nthcdr 4 '(pine fir oak maple))
     ⇒ nil

;; Devuelve una copia sin los elementos.
(nthcdr 5 '(pine fir oak maple))
     ⇒ nil
```

7.4 nth

La función `nthcdr` toma el CDR de una lista repetidamente. La función `nth` toma el CAR del resultado devuelto por `nthcdr`. Devuelve el elemento Nth de la lista.

De este modo, si no fuera definido en C para velocidad, la definición de `nth` sería:

```
(defun nth (n list)
  "Devuelve el elemento Nth de Lista.
N cuenta desde cero. Si Lista no es larga, nil es devuelto".
  (car (nthcdr n list)))
```

(Originalmente, `nth` fué definido en Emacs Lisp en `subr.el`, pero su definición fué rehecha en C en los 1980s.)

La función `nth` devuelve un elemento simple de una lista. Esto puede ser muy conveniente.

Nótese que los elementos son numerados desde cero, no desde uno. Es decir, el primer elemento de una lista, su CAR es el elemento cero. Esto se llama contar en 'base cero' y con frecuencia las personas tienes la costumbre del que el primer elemento en una lista sea el número uno, eso es 'basado en uno'.

Por ejemplo:

```
(nth 0 '("uno" "dos" "tres"))
    ⇒ "uno"

(nth 1 '("uno" "dos" "tres"))
    ⇒ "dos"
```

Es valorable mencionar que `nth`, como `nthcdr` y `cdr`, no cambia la lista original — la función no es destructiva. En contraste con las funciones `setcar` y `setcdr`.

7.5 setcar

Como se podría adivinar desde sus nombres, las funciones `setcar` y `setcdr` asignan el CAR o la CDR de una lista a un nuevo valor. Ellos actualmente cambia la lista original, no como `car` y `cdr` que deja la lista original como estaba. Un camino para encontrar cómo esto funciona es experimentar. Se comenzará con la función `setcar`.

Primero, podemos crear una lista y entonces asignar el valor de una variable a la lista usando la función `setq`. Aquí hay una lista de animales:

```
(setq animales '(antilope jirafa leon tigre))
```

Si estás leyendo esto en Info dentro de GNU Emacs, se puede evaluar esta expresión del modo usual, posicionando el cursor después de la expresión y escribiendo *C-x C-e*. (Esto haciendo esto tan bien aquí como yo escribo esto. Esto es una de las ventajas de tener el intérprete construido dentro del entorno de computación. Incidentalmente, cuando no hay nada en la línea después del paréntesis final, tal como un comentario, el punto puede estar en la siguiente línea. De este modo, si tu cursor está en la primera columna de la siguiente línea, no necesitas moverlo. En realidad, Emacs permite cualquier cantidad de espacio en blanco después del paréntesis final.)

Cuando se evalúa la variable `animal`, vemos que está asociada a la lista `(antelope giraffer lion tiger)`:

```
animals
     ⇒ (antelope giraffe lion tiger)
```

Pon otro camino, la variable `animales` apunta a la lista `(antilope jirafa leon tigre)`.

Lo siguiente, es evaluar la función `setcar` mientras le pasa dos argumentos, la variable `animales` y el símbolo citado `hipopotamo`; esto es hecho escribiendo la lista de tres elementos `(setcar animales 'hipopotamo)` y entonces evaluando en el modo usual:

```
(setcar animales 'hipopotamo)
```

Después de evaluar esta expresión, evalúa la variable `animales` de nuevo. Se puede ver que la lista de animales ha cambiado:

```
animales
     ⇒ (hipopótamo jirafa leon tigre)
```

El primer elemento en la lista, `antilope` se reemplaza por `hipopotamo`.

Así se puede ver que `setcar` no añadió un nuevo elemento a la lista como `cons` tendría; eso reemplazó `antílope` con `hipopótamo`; eso *cambió* la lista.

7.6 `setcdr`

La función `setcdr` es similar a la función `setcar`, excepto que la función reemplaza el segundo y subsiguientes elementos de una lista en vez del primer elemento.

(Para ver cómo se cambia el último elemento de una lista, mira hacia delante a la "La función `kill-new`", página 94, que usa las funciones `nthcdr` y `setcdr`.)

Para ver cómo esto funciona, asigna el valor de la variable a una lista de animales domesticados evaluando la siguiente expresión:

```
(setq domesticated-animals '(horse cow sheep goat))
```

Si ahora se evalúa la lista, será devuelta la lista `(horse cow sheep goat)`:

```
domesticated-animals
     ⇒ (horse cow sheep goat)
```

Lo siguiente, evalúa `setcdr` con dos argumentos, el nombre de la variable que tiene una lista como su valor, y la lista para la que el CDR de la primera lista sea asignada;

```
(setcdr domesticated-animals '(cat dog))
```

Si se evalúa esta expresión, la lista `(gato perro)` aparecerá en el área echo. Este es el valor devuelto por la función. El resultado en el que estamos interesados es el "efecto lateral", que se puede ver evaluando la variable `domesticated-animals`:

```
domesticated-animals
     ⇒ (horse cat dog)
```

En realidad, la lista es cambiada desde `(horse cow sheep goat)` a `(horse cat dog)`. El CDR de la lista es cambiada desde `(cow sheep goat)` a `(cat dog)`.

7.7 Ejercicio

Construye una lista de cuatro pájaros evaluando varias expresiones con `cons`. Encuentra que ocurre cuando `cons` una lista dentro de sí. Reemplaza el primer elemento de la lista de cuatro pájaros con un pez. Reemplaza el resto de esta lista con una lista de otro pez.

8 Cortando y almacenando texto

Siempre y cuando se corte o pegue texto de un búffer con un comando 'kill' en GNU Emacs, se almacenará dentro de una lista que se puede traer con un comando 'yank'.

(El uso de la palabra 'kill' *matar, cortar* en Emacs para procesos que específicamente *no* destruyen los valores de las entidades es un accidente histórico desafortunado. Una palabra mucho más apropiada debería ser 'clip' *cortar* puesto que es lo que los comandos de corte hacen; ellos cortan texto fuera de un búffer y lo ponen dentro del almacenamiento desde el que puede traerse. Con frecuencia ha sido tentada de reemplazar globalmente todas las ocurrencia de 'kill' *matar, cortar* en las fuentes de Emacs con 'clip' *cortar* y todas las ocurrencias de 'killed' *cortado, muerto* con 'clipped' *cortado*.)

Cuando el texto se corta de un búffer, es almacenado en una lista. Piezas sucesivas de texto se almacenan en la lista sucesivamente, así la lista podría verse así:

```
("una pieza de texto" "pieza previa")
```

La función `cons` puede usarse para crear una nueva lista desde una pieza de texto (un 'átomo', para usar la jerga) y una lista existente, como esta:

```
(cons "otra pieza"
      '("una pieza de texto" "pieza previa"))
```

Si se evalúa esta expresión, una lista de tres elementos aparecerá en el área echo:

```
("otra pieza" "una pieza de texto" "pieza previa")
```

Con las funciones `car` y `nthcdr`, se puede recuperar siempre la pieza de texto que se quiere. Por ejemplo, en el siguiente código, `nthcdr 1 ...` se devuelve la lista con el primer ítem eliminado; y el `car` devuelve el primer elemento de este resto — el segundo elemento de la lista original:

```
(car (nthcdr 1 '("otra pieza"
                 "una pieza de texto"
                 "pieza previa")))
     ⇒ "una pieza de texto"
```

Las funciones actuales en Emacs son más complejas que esto, de acuerdo. El código para cortar y recuperar texto tiene que ser escrito de modo que Emacs pueda ver qué elemento en la lista se quiere — el primero, segundo, tercer o cualquier otro. Además, cuando tiene el fin de la lista, Emacs daría el primer elemento de la lista, en lugar de nada.

La lista que maneja las piezas de texto se llama *kill ring* (anillo de la muerte). Este capítulo lidera una descripción del anillo de la muerte y como eso es usado por la primera traza de cómo la función `zap-to-char` funciona. Esta función usa (o 'llama') a una función que invoca a otra función que manipula el anillo de la muerte. De este modo, antes de lograr las montañas, se escalan las colinas.

Un capítulo subsiguiente describe cómo el texto que se corta desde el búffer se recupera. Véase Capítulo 10 "Pegando Texto", página 108.

8.1 zap-to-char

La función zap-to-char cambió poco entre GNU Emacs versión 19 y GNU Emacs versión 22. Sin embargo, zap-to-char llama a otra función, kill-region, que se reescribió más.

La función kill-region en Emacs 19 es compleja, pero no usa código que es importante en este momento. Se escapará.

La función kill-region en Emacs 22 es más de fácil leer que la misma función en Emacs 19 e introduce un concepto muy importante, que el error maneja. Nosotros pasearemos a través de la función.

Pero primero, déjanos ver en la función interactive zap-to-char.

La función zap-to-char elimina el texto en la región entre la localización del curso (por ej. punto) para incluir la siguiente ocurrencia de un caracter específico. El texto que zap-to-char borra es puesto en el kill ring *anillo de la muerte*; y puede ser recuperado desde el kill ring *anillo de la muerte* escribiendo C-y (yank). Si el comando dado es un argumento, eso borra texto a través de este número de ocurrencias. De este modo, si el cursor estuviera al principio de esta frase y el carácter fuera 's', 'De este modo' sería borrado. Si el argumento fueran dos, 'De este modo, si el cursor' se borrase, y incluiría la 's' en el 'cursor'.

Si el carácter específico no encuentra zap-to-char dirá "Búsqueda fallida", eso cuenta el carácter que se escribió, y no eliminó cualquier texto.

En orden para determinar cuánto texto eliminar zap-to-char usa una función de búsqueda. Las búsquedas son usadas extensivamente en el código que manipula texto, y focalizará la atención en ellos tan bien como el comando de borrado.

Aquí está el texto completo de la versión 22 de la función:

```
(defun zap-to-char (arg char)
  "Corta e incluye ARG'th ocurrencia de CHAR
En caso de ser ignorada si `case-fold-search' es no nulo en el
  búffer actual.
Para ir atrás si ARG es negativo; error si CHAR no se encuentra."
  (interactive "p\ncZap to char: ")
  (if (char-table-p translation-table-for-input)
      (setq char (or (aref translation-table-for-input char) char)))
  (kill-region (point) (progn
                         (search-forward (char-to-string char)
                                         nil nil arg)
                         (point))))
```

La documentación es en línea. Se necesita conocer el significado de la jerga de la palabra 'kill'.

8.1.1 La expresión interactive

La expresión interactiva en el comando zap-to-char se ve así:

```
(interactive "p\ncZap to char: ")
```

La parte con comillas, "p\ncZap to char: " *Cortar a caracter*, especifica dos cosas diferentes. Primero, y más simple es el 'p'. Este parte se separa desde la siguiente parte por una nueva línea, '\n'. El 'p' significa que la parte del primero argumento a la función será pasando el valor de un 'prefijo procesado'. El argumento

prefijo es pasado escribiendo `C-u` y un número, o `M-` y un número. Si la función es llamada interactivamente sin un prefijo, el número que se pasa es 1.

La segunda parte de `"p\ncZap a caracter: "` es '`cZap to carácter:`'. En esta parte, la tecla baja '`c`' indica que `interactive` espera una consola que el argumento será un caracter. La consola sigue el '`c`' y es la cadena '`Zap to caracter: `' (con un espacio después del punto y coma para hacerlo bien).

Tod lo que hace es preparar los argumentos para `zap-to-char` así ellos están en el tipo correcto, y dan al usuario un prompt.

En un búffer de solo lectura, la función `zap-to-char` copia el texto al anillo de la muerte, no se elimina. El área echo muestra un mensaje diciendo que el búffer es de solo lectura. También, la terminal avisa con un pitido.

8.1.2 El cuerpo de `zap-to-char`

El cuerpo de la función `zap-to-char` contiene el código que mata (que se borra/corta) el texto en la región desde la posición actual del cursor e incluyendo el carácter especificado.

La primera parte del código se ve como:

```
(if (char-table-p translation-table-for-input)
    (setq char (or (aref translation-table-for-input char) char)))
(kill-region (point) (progn
                       (search-forward (char-to-string char) nil nil arg)
                       (point)))
```

`char-table-p` es una función prescrita no vista. Eso determina si sus argumentos son una tabla de caracteres. Así, se asigna el carácter pasado a `zap-to-char` a uno de ellos, si este carácter existe, o al carácter en sí. (Esto llega a ser importante para ciertos caracteres en lenguajes no Europeos. La función `aref` extrae un elemento desde un array. Eso es una función específica de array que no está descrita en este documento. Véase Sección "Arrays" in *El Manual de Referencia de GNU Emacs Lisp*.)

`(point)` es la posición actual del cursor.

La siguiente parte del código es una expresión usando `progn`. El cuerpo del `progn` se basa en llamadas a `search-forward` y `point`.

Es fácil comprender cómo `progn` funciona después de aprender acerca de `search-forward`, así se verá en `search-forward` y entonces en `progn`.

8.1.3 La función `search-forward`

La función `search-forward` es usada para localizar el zapped-for-character en `zap-to-char`. Si la búsqueda es exitosa, `search-forward` deja el punto inmediatamente después del último carácter en la cadena objetivo. (En `zap-to-char`, la cadena objetivo es solo un carácter largo. `zap-to-char` usa la función `char-to-string` para asegurar que el ordenador trata este carácter como una cadena.) Si la búsqueda es hacia atrás, `search-forward` deja el punto solo antes del primer carácter en el objetivo. También, `search-forward` devuelve `t` para verdad. (Moviendo el punto allí es un 'efecto lateral'.)

En zap-to-char, la función search-forward se ve así:

```
(search-forward (char-to-string char) nil nil arg)
```

La función search-forward toma cuatro argumentos:

1. El primer argumento es el objetivo, para el que es buscado. Esto debe ser una cadena, tal como '"z"'.

   Cuando eso ocurre, el argumento pasado a zap-to-char es un caracter simple. Debido a la forma en la que los ordenadores son construidos, el intérprete Lisp puede tratar un caracter simple siendo diferente desde una cadena de caracteres. Dentro del ordenador, un caracter simple tiene un formato electrónico en vez de una cadena de un caracter. (Un caracter simple puede con frecuencia ser grabado en el ordenador usando exactamente un byte; pero una cadena puede ser larga, y el ordenador necesita estar listo para esto.) Desde que la función search-forward busca una cadena, el caracter que la función zap-to-char recibe como argumento debe ser convertida dentro del ordenador de un formato a otro; de otro modo, la función search-forward fallará. La función char-to-string es usada para hacer esta conversión.
2. El segundo argumento asocia la búsqueda; eso se especifica como una posición en el búffer. En este caso, la búsqueda puede ir al fin del búffer, así no se asigna y el segundo argumento es nil.
3. El tercer argumento cuenta la función que haría si la búsqueda cae — eso puede señalar un error (e imprimir un mensaje) o puede devolver nil. Un nil como tercer argumento hace que la función señale un error cuando la búsqueda falla.
4. El cuarto argumento search-forward es el contaje repetido — cuántas ocurrencias de la cadena para buscar. Este argumento es opcional y si la función es llamada sin un contaje repetido, este argumento pasa el valor 1. Si este argumento es negativo, la búsqueda va hacia atrás.

En la forma de plantilla, una expresión search-forward se ve así:

```
(search-forward "target-string"
                limit-of-search
                what-to-do-if-search-fails
                repeat-count)
```

Lo siguiente es echar un vistazo a progn.

8.1.4 La forma especial progn

progn es una forma especial que causa que cada uno de sus argumentos puedan ser evaluados en secuencia y entonces devuelve el valor del último. Las expresiones precedentes son evaluadas solo por los efectos laterales que ellos desarrollan. Los valores producidos por ellos son descartados.

La plantilla para una expresión progn es muy simple:

```
(progn
  body...)
```

En zap-to-char, la expresión progn tiene que hacer dos cosas: poner el punto en la posición exacta; y devolver la posición del punto de modo que kill-region conoce cómo de lejos se copia.

El primer argumento de `progn` es `search-forward`. Cuando `search-forward` encuentra la cadena, la función deja el punto inmediatamente después del último caracter en la cadena objetivo. (En este caso la cadena objetivo es solo un caracter de largo.) Si la búsqueda es hacia atrás, `search-forward` deja el punto justo antes del primer carácter en el objetivo. El movimiento del punto es un efecto lateral.

El segundo y último argumento de `progn` es la expresión `(point)`. Esta expresión devuelve el valor del punto, que en este caso será la localización para la que se ha movido por `search-forward`. (En la fuente, una línea que cuenta la función para ir al carácter previo, si se está yendo hacia delante, se comentó en 1999; yo no recuerdo si esta funcionalidad o no funcionalidad era siempre parte de las fuentes distribuidas.) El valor de `point` se devuelve por la expresión `progn` y se pasa a `kill-region` como el segundo argumento de `kill-region` .

8.1.5 Resumiendo `zap-to-char`

Ahora que se ha visto cómo `search-forward` y `progn` trabajan, se puede ver cómo la función `zap-to-char` funciona como un todo.

El primer argumento de `kill-region` es la posición del cursor cuando el comando `zap-to-char` da — el valor del punto en este momento. Con el `progn`, la búsqueda de la función mueve el punto a solo después del zapped-to-character y `point` devuelve el valor de localización. La función `kill-region` pone junto a estos dos valores de punto, el primero como el principio de la región y el segundo como el fin de la región, y borra la región.

La forma especial `progn` se necesita porque el comando `kill-region` toma dos argumentos; y fallaría si `search-forward` y expresiones `point` fueran escritas en secuencia como dos argumentos adicionales. La expresión `progn` es un argumento simple para `kill-region` y devuelve un valor para que `kill-region` se necesite por su segundo argumento.

8.2 `kill-region`

La función `zap-to-char` usa la función `kill-region`. Esta función corta texto desde una región y copia este texto al kill ring *anillo de la muerte*, desde el que puede ser recuperado.

Tanto la versión de Emacs 22 de esta función usa `condition-case` y `copy-region-as-kill`, ambas se explicarán. `condition-case` es una forma especial importante.

En esencia, la función `kill-region` llama a `condition-case`, que toma tres argumentos. En esta función, el primer argumento no hace nada. El segundo argumento contiene el código que hace el trabajo cuando todo va bien. El tercer argumento contiene el código que se llama en el evento de un error.

Ahora se puede volver a través del código `condition-case` en un momento. Primero, se echa un vistazo a la definición de `kill-region`, con comentarios añadidos:

```
(defun kill-region (beg end)
  "Kill (\"corta\") texto entre punto y marca.
Esto borra el texto desde el búffer y lo guarda en anillo de la
  muerte kill ring.
El comando \\[yank] puede recuperarse desde allí. ...
 "

  ;; • Desde materias de orden, pasa el punto primero.
  (interactive (list (point) (mark)))
  ;; • Y cuéntanos si no podemos cortar el texto.
  ;; 'a menos que' sea un 'if' sin una then-part.
  (unless (and beg end)
    (error "La marca no está asignada ahora, así que
  no hay región"))

  ;; • 'condition-case' toma tres argumentos
  ;;    Si el primer argumento es nulo, como aquí
  ;;    la información acerca del error no está
  ;;    almacenada para ser usada por otra función
  (condition-case nil

      ;; • El segundo argumento a 'condition-case' cuenta lo que el
      ;;    intérprete que hace cuando todo va bien.

      ;;    Empieza con una función 'let' que extrae la cadena
      ;;    y chequea si existe. Si es así (esto es lo
      ;;    que 'when' chequea), eso llama a una función 'if' que
      ;;    determina si el comando previo que era otra llamada para
      ;;    si el comando previo fué otra llamada a 'kill-region';
      ;;    si lo era, entonces el siguiente texto se añade
      ;;    cuando se chequea), eso llama a una función 'if' que
      ;;    determina si el comando previo era otra llamada a
      ;;    'kill-region'; si era eso, entonces el nuevo texto es
      ;;    añadido al texto previo; si no, entonces una función
      ;;    diferente, 'kill-new' se llama.

      ;;    La función 'kill-append' concatena la nueva cadena y
      ;;    la vieja.  La función 'kill-new' inserta texto dentro de
      ;;    ítem en el kill ring anillo de la muerte.

      ;;    'when' es un 'if' sin una parte else.  El segundo 'when'
      ;;    de nuevo chequea si la cadena actual existe;
      ;;    por añadidura, eso chequea si el comando previo fuese
      ;;    otra llamada a 'kill-region'. Si una u otra condición
      ;;    es verdadero, entoncese eso configura el actual comando a
      ;;    ser 'kill-region'.
```

```
  (let ((string (filter-buffer-substring beg end t)))
    (when string                    ;STRING is nil if BEG = END
      ;; Add that string to the kill ring, one way or another.
      (if (eq last-command 'kill-region)
          ;;    — 'yank-handler' es un argumento opcional para
          ;;    'kill-region' que cuenta el 'kill-append' y funciones
          ;;    'kill-new' como tratan con propiedades añadidas
          ;;    al texto, tal como 'negrilla' o 'itálica'
          (kill-append string (< end beg) yank-handler)
        (kill-new string nil yank-handler)))
    (when (or string (eq last-command 'kill-region))
      (setq this-command 'kill-region))
    nil)

;;  • El tercer argumento a 'condition-case' cuenta el intérprete
;;  qué hacer con un error.
;;    El tercer argumento tiene una parte de condiciones y una
;;    parte del cuerpo.
;;    Si las condiciones se encuentra (en este caso,
;;              si el texto o búffer son de solo lectura)
;;    entonces el cuerpo es ejecutado.
;;    La primera parte del tercer es el siguiente:
((buffer-read-only text-read-only) ;; parte if
 ;; ...  parte then-part
 (copy-region-as-kill beg end)
 ;;    Lo siguiente, también como parte de la then-part, asigna this-command, así
 ;;    será asignado en un error
 (setq this-command 'kill-region)
 ;;    Finalmente, en la then-part, envía un mensaje
 ;;    si se puede copiar el texto al anillo de la muerte
 ;;    kill ring sin señalar un error, pero no si no se puede.

 (if kill-read-only-ok
     (progn (message "Lee solo texto copiado para el kill ring") nil)
   (barf-if-buffer-read-only)
   ;; If the buffer isn't read-only, the text is.
   (signal 'text-read-only (list (current-buffer)))))
```

8.2.1 `condition-case`

Como se ha visto antes (véase Sección 1.3 "Genera un Mensaje de Error", página 4), cuando el intérprete Emacs Lisp tiene problemas evaluando una expresión, se provee de una ayuda; en jerga, se dice "signaling an error" *señalando un error*. Normalmente, el ordenador para el programa y te muestra un mensaje.

Sin embargo, algunos programas garantizan acciones complicadas. Eso no pararía en un error. En la función `kill-region`, la mayoría parece un error que intentará cortar texto que es de solo lectura y no puede ser eliminado. Así la función `kill-region` contiene código para manejar esta circunstancia. Este código, hace que el cuerpo de la función `kill-region`, esté dentro de una forma especial `condition-case`.

La plantilla para `condition-case` se parece a esto:

```
(condition-case
  var
  bodyform
  error-handler...)
```

El segundo argumento, *bodyform* es sencillo. La forma especial `condition-case` causa que el intérprete Lisp evalúe el código en *bodyform*. Si ningún error ocurre, la forma especial devuelve el valor del código y produce efectos laterales, si hay.

En resumen, la parte *bodyform* de una expresión `condition-case` determina qué ocurre cuando cualquier cosa funciona correctamente.

Sin embargo, si un error ocurre, entre sus otras acciones, la función genera la señal de error que definirá uno o más errores de nombres de condición.

Un manejador de errores es el tercer argumento para `condition-case`. Un manejador de errores tiene dos partes, un *condition-name* y un *body*. Si la parte *condition-name* de un manejador de errores encuentra un nombre de condición generado por un error, entonces la parte del *body* del manejador de errores se ejecuta.

Como se esperaría, la parte *condition-name* de un manejador de errores puede ser así, un nombre de condición simple o una lista de nombres de condición.

También, una expresión completa `condition-case` puede contener más de un manejador de errores. Cuando un error ocurre, el primer manejador aplicable se ejecuta.

Finalmente, el primer argumento a la expresión `condition-case`, es el argumento *var*, que es algunas veces asignado a una variable que contiene información acerca del error. Sin embargo, si este argumento es nulo, como es el caso en `kill-region`, esta información se descarta.

En breve, en la función `kill-region`, el código `condition-case` funciona así:

```
Si no hay errores, ejecuta solo este código
    pero, si hay errores, ejecuta este otro código.
```

8.2.2 Macro Lisp

La parte de la expresión `condition-case` que se evalúa en la expectativa de que todo va bien si tiene un `when`. El código usa `when` para determinar si la variable `string` (*cadena*) apunta al texto que existe.

Una expresión `when` es simplemente una conveniencia de programadores. Eso es un `if` sin la posibilidad de una cláusula else. En tu mente, se puede reemplazar `when` con `if` y comprender de que va. Esto es lo que el intérprete Lisp hace.

Técnicamente hablando, `when` es una macro Lisp. Una *macro* Lisp permite definir una nueva construcción de control y otras funcionalidades del lenguaje. Eso cuenta al intérprete cómo computar otra expresión Lisp que dejará de computar el valor. En este caso, la 'otra expresión' es una expresión `if`.

La definición de función también tiene una macro `unless`; que acompaña a `when`. La macro `unless` es un `if` sin una cláusula then.

Para más acerca de macros Lisp, ver Sección "Macros" in *El Manual de Referencia de Emacs Lisp*. El lenguaje de programación C también provee macros. Estos son diferentes, pero también útiles.

Guardando la macro `when`, en la expresión `condition-case`, cuando la cadena tiene contenido, entonces otra expresión condicional se ejecuta. Esto es un `if` tanto con una then-part y como con una else-part.

```
(if (eq last-command 'kill-region)
    (kill-append string (< end beg) yank-handler)
  (kill-new string nil yank-handler))
```

La parte then (then-part) se evalúa si el comando previo fué otra llamada para `kill-region`; si no, se evalúa la parte else (else-part)

`yank-handler` es un argumento opcional para `kill-region` que cuenta cómo las funciones `kill-append` y `kill-new` se tratan con propiedades añadidas al texto, tal como 'negrilla' o 'itálica'.

`last-command` es una variable que viene con Emacs y que no se ha visto antes. Normalmente, siempre y cuando una función se ejecute, Emacs asigna el valor de `last-command` al comando previo.

En este segmento de la definición, la expresión `if` chequea si el comando previo era `kill-region`. Si era eso,

```
(kill-append string (< end beg) yank-handler)
```

Se concatena una copia del nuevo texto cortado al texto cortado previamente en el kill ring *anillo de la muerte*.

8.3 copy-region-as-kill

La función `copy-region-as-kill` copia una región de texto desde un búffer y (via `kill-append` o `kill-new`) lo guarda en el `kill-ring`.

Si se llama a `copy-region-as-kill` inmediatamente después de un comando `kill-region`, Emacs inserta el texto nuevamente copiado al texto copiado previamente. Esto significa que si se pega el texto, se obtiene todo, tanto esto, como la operación previa. Por otro lado, si algún otro comando precede la `copy-region-as-kill`, la función copia el texto dentro de una entrada separada el kill ring *anillo de la muerte*.

Aquí está el texto completo de la versión 22 de la función `copy-region-as-kill`:

```
(defun copy-region-as-kill (beg end)
  "Guarda la región como si estuviera cortada, pero no la cortes.
En el modo Marca de Tránsito Transient Mark, se desactiva la
  marca.
Si 'interprogram-cut-function' es no nulo, también se guarda el
  texto para una sistema de ventanas de cortar y pegar."
  (interactive "r")
  (if (eq last-command 'kill-region)
      (kill-append (filter-buffer-substring beg end) (< end beg))
    (kill-new (filter-buffer-substring beg end)))
  (if transient-mark-mode
      (setq deactivate-mark t))
  nil)
```

De normal, esta función puede ser dividida dentro sus componentes:

```
(defun copy-region-as-kill (argument-list)
  "documentation..."
  (interactive "r")
  body...)
```

Los argumentos son `beg` y `end` y la función es interactiva con `"r"`, así los dos argumentos deben referir al principio y el fin de la región. Si has estado leyendo a través de este documento desde el principio, compreendiendo estas partes de una función es casi llegando a ser rutina.

La documentación es algo confusa a menos que recuerdes que la palabra 'kill' tiene un significado diferente del usual. La 'Marca Transitoria' y `interprogram-cut-function` comenta explicar cientos efectos laterales.

Después de que se ha asignado una marca, un búffer siempre contiene una región. Si se desea se puede usar el modo Marca Transitoria para iluminar la región temporalmente. (Nadie quiere ilumihnar la región todo el rato, así el modo Marca Transitoria subrraya solo en el momento apropiado. Muchas personas desactivan el modo Marca Transitoria, así la región nunca se ilumina.)

También, un sistema de ventanas permite copiar, cortar y pegar entre programas diferentes. En el sistema de X windows, por ejemplo, la función `interprogram-cut-function` es `x-select-text`, que funciona con el sistema de ventanas equivalente del kill ring de Emacs.

El cuerpo de la función `copy-region-as-kill` empieza con una cláusula `if`. Lo que esta cláusula hace es distinguir entre dos situaciones diferentes: si este comando es ejecutado o no inmediatamente después de un comando previo `kill-region`. En el primer caso, la nueva región es concatenada al texto copiado previamente. De otro modo, eso se inserta al principio del anillo de la muerte *kill ring* como una pieza separada de texto desde la pieza previa.

Las dos líneas de la función previene la región desde la iluminación si el modo Transient Mark *Marca Transitoria* está activado.

El cuerpo de `copy-region-as-kill` merece discusión en detalle.

8.3.1 El cuerpo de `copy-region-as-kill`

La función `copy-region-as-kil` funciona de un modo parecido a la función `kill-region`. Ambas están escritas de un modo que dos o más textos cortados en una fila combinan su texto en una entrada simple. Si se pega el texto desde el anillo de la muerte *kill ring*, se tiene todo en una pieza. Más allá, los cortes de textos que se cortan hacia adelante desde la posición actual del cursor se añaden al fin del texto copiado previamente y comanda este texto copiado vaya hacia atrás al principio del texto copiado previamente. De este modo, las palabras en el texto están en el orden apropiado.

Como `kill-region`, la función `copy-region-as-kill` hace uso de la variable `last-command` que deja traza del comando de Emacs previo.

Normalmente, siempre y cuando una función se ejecuta, Emacs asigna el valor de `this-command` a la función que se ejecuta (que en este caso sería `copy-region-`

`as-kill`). Al mismo tiempo, Emacs asigna el valor de `last-command` al valor previo de `this-command`.

En la primera parte del cuerpo de la función `copy-region-as-kill`, una expresión `if` determina si el valor de `last-command` es `kill-region`. Si es así, la then-part de la expresión `if` se evalúa; eso usa la función `kill-append` para concatenar el texto copiado en esta llamada a la función con el texto ya en el primer elemento (el CAR del anillo de la muerte. Por otro lado, si el valor de `last-command` no es `kill-region`, entonces la función `copy-region-as-kill` adjunta un nuevo elemento al anillo de la muerte *kill ring* usando la función `kill-new`.

La expresión `or` se ve así; usa `eq`:

```
(if (eq last-command 'kill-region)
    ;; parte then
    (kill-append  (filter-buffer-substring beg end) (< end beg))
  ;; parte else
  (kill-new  (filter-buffer-substring beg end)))
```

(La función `filter-buffer-substring` devuelve una subcadena filtrada del búffer, cualquiera. Opcionalmente — los argumentos no están aquí, así nunca está hecho — la función puede borrar el texto inicial o devolver el texto sin sus propiedades; esta función es un reemplazo para la vieja función `buffer-substring`, que viene antes que las propiedades del texto fuesen implementadas.)

La función `eq` chequea si su primer argumento es el mismo objeto Lisp que su segundo argumento. La función `eq` es similar a la función `equal` en esto que es usado para chequear para igualdad, pero difiere en esto que determina si dos representaciones son actualmente el mismo objeto dentro del ordenador, pero con diferentes nombres. `equal` determina si la estructura y contenidos de dos expresiones son la misma.

Si el comando previo era `kill-region`, entonces el intérprete Emacs Lisp llama a la función `kill-append`

La función `kill-append`

La función `kill-new` se ve como así:

```
(defun kill-append (string before-p &optional yank-handler)
  "Inserta STRING al fin del último corte en el anillo de la muerte kill ring.
Si BEFORE-P es no nulo, inserta STRING.
... "
  (let* ((cur (car kill-ring)))
    (kill-new (if before-p (concat string cur) (concat cur string))
              (or (= (length cur) 0)
                  (equal yank-handler
                         (get-text-property 0 'yank-handler cur)))
              yank-handler)))
```

La función `kill-append` es limpia. Usa la función `kill-new`, que discutiremos en más detalle en un momento.

(También, la función provee un argumento opcional llamado `yank-handler`; cuando se invoque, este argumento cuenta a la función cómo tratar con la propiedades añadidas al texto, tales como 'negrilla' o 'itálicas'.)

Eso tiene una función `let*` para asignar el valor del primer elemento del kill ring a `cur`. (No se sabe por qué la función no usa `let`; solo un valor es asignado en la expresión. ¿Quizás esto es un error que no produce problemas?

Considera el condicional que es uno de los dos argumentos para `kill-new`. Eso usa `concat` para concatenar el nuevo texto al CAR del anillo de la muerte *kill ring*. Si eso se concatena atrás o delante depende de los resultados de una expresión `if`:

```
(if before-p                            ; if-part
    (concat string cur)                 ; then-part
  (concat cur string))                  ; else-part
```

Si la región cortada está antes que la región que se cortó en el último comando, entonces debería ser puesto antes que el material salvador en el anterior corte *kill*; y de manera contraria, si el texto cortado sigue lo que fué cortado, eso sería añadido después del texto previo. La expresión `if` depende del predicado `before-p` para decidir si el texto nuevamente salvado es puesto antes o después.

El símbolo `before-p` es el nombre de uno de los argumentos a `kill-append`. Cuando la función `kill-append` se evalúa, se asocia al valor devuelto evaluando el argumento actual. En este caso, esta es la expresión `(< end beg)`. Esta expresión no determina directamente si el texto cortado en este comando se localiza antes o después del texto cortado del último comando; lo que hace es determinar si el valor de la variable `end` es menor que el valor de la variable `beg`. Si es así, significa que el usuario se encara al principio del búffer. También, el resultado de evaluar la expresión del predicado. `(< end beg)`, será verdadero y el texto se concatena antes del texto previo. Por otro lado, si el valor de la variable `end` es mayor que el valor del la variable `beg`, el texto será concatenado después del texto previo.

Cuando el texto nuevamente guardado se concatena, entonces la cadena con el nuevo texto será concatenado antes del viejo texto:

```
(concat string cur)
```

Pero si el texto será añadido, eso será concatenado después del viejo texto:

```
(concat cur string))
```

Para comprender cómo funciona esto, primero se necesita revisar la función `concat`. La función `concat` enlaza junto o une dos cadenas de texto. El resultado es una cadena. Por ejemplo:

```
(concat "abc" "def")
     ⇒ "abcdef"

(concat "new "
        (car '("first element" "second element")))
     ⇒ "new first element"

(concat (car
        '("first element" "second element")) " modified")
     ⇒ "first element modified"
```

Ahora puede tener sentido `kill-append`: eso modifica los contenidos del anillo de la muerte *kill ring*. El anillo de la muerte *kill ring* es una lista, en la que cada elemento es texto guardado. La función `kill-append` usa la función `kill-new` que usa la función `setcar`.

La función `kill-new`

La función `kill-new` se ve de esta manera:

```
(defun kill-new (string &optional replace yank-handler)
  "Crea STRING el último corte en el anillo de la muerte kill
  ring.
Asigna ‘kill-ring-yank-pointer’ para apuntarlo.

Si ‘interprogram-cut-function’ es no nulo, aplícalo a su
 STRING.
Segundo argumento opcional REPLACE no-nulo significa que STRING
 reemplazará el frente del kill ring, en vez de ser aãndido a la lista.
..."
  (if (> (length string) 0)
      (if yank-handler
          (put-text-property 0 (length string)
                             'yank-handler yank-handler string))
    (if yank-handler
        (signal 'args-out-of-range
                (list string "yank-handler specified for empty string"))))
  (if (fboundp 'menu-bar-update-yank-menu)
      (menu-bar-update-yank-menu string (and replace (car kill-ring))))
  (if (and replace kill-ring)
      (setcar kill-ring string)
    (push string kill-ring)
    (if (> (length kill-ring) kill-ring-max)
        (setcdr (nthcdr (1- kill-ring-max) kill-ring) nil)))
  (setq kill-ring-yank-pointer kill-ring)
  (if interprogram-cut-function
      (funcall interprogram-cut-function string (not replace))))
```

(Vea que la función no es interactiva.)

Normalmente, se mira a esta función en partes.

La definición de la función tiene un argumento opcional `yank-handler`, que cuando se invoca cuenta la función de cómo tratar con propiedades añadidas al texto, tal como ‘negrilla’ o ‘itálica’. Nosotros evitaremos esto.

La primer línea de la documentación tiene sentido:

```
Crea la CADENA la última copia en el anillo de la muerte kill
ring.
```

Permite escapar a través del resto de la documentación por el momento.

También, permite salir de la expresión inicial `if` y estas líneas de código involucrando `menu-bar-update-yank-menu`. Nosotros explicaremos debajo.

Las líneas críticas son estas:

```
(if (and replace kill-ring)
    ;; then
    (setcar kill-ring string)
  ;; else
(push string kill-ring)
  (setq kill-ring (cons string kill-ring))
  (if (> (length kill-ring) kill-ring-max)
      ;; avoid overly long kill ring
      (setcdr (nthcdr (1- kill-ring-max) kill-ring) nil)))
```

```
(setq kill-ring-yank-pointer kill-ring)
(if interprogram-cut-function
    (funcall interprogram-cut-function string (not replace))))
```

El test condicional es `(and replace kill-ring)`. Esto será verdad cuando dos condiciones se encuentran: el anillo de la muerte *kill ring* tiene alguna cosa dentro, y la variable `replace` es verdad.

Cuando la función `kill-append` asigna `replace` para ser cierto y cuando el anillo de la muerte *kill ring* tiene al menos un ítem en eso, la expresión `setcar` es ejecutada.

```
(setcar kill-ring string)
```

La función `setcar` actualemten cambia el primer elemento del anillo de la muerte (`kill-ring` lista al valor de `string`. Eso reemplaza el primer elemento.

Por otro lado, si el kill ring está vacío, o reemplazar es falso, la else-part de la condición está ejecutado:

```
(push string kill-ring)
```

`push` pone su primer argumento dentro del segundo. Es similar al viejo.

```
(setq kill-ring (cons string kill-ring))
```

o el nuevo

```
(add-to-list kill-ring string)
```

Cuando eso es falso, la expresión primero construye una nueva versión del anillo de la muerte *kill ring* añadiendo `string` al anillo de la muerte *kill ring* como un nuevo elemento (que es lo que `push` hace). Entonces ejecuta un segundo `if` cláusula. Este segundo `if` cláusula guarada el anillo de la muerte *kill ring* desde el creciente demasiado largo.

Déjanos mirar estas dos expresiones en orden.

La línea `push` de la parte else asigna el nuevo valor del kill ring *anillo de la muerte* a que resultados añaden la cadena siendo cortada al viejo anillo de la muerte *kill ring*

Nosotros podemos ver cómo esto funciona con un ejemplo.

Primero,

```
(setq example-list '("aqui una clausula" "otra clausula"))
```

Después de evaluar esta expresión con *C-x C-e*, se puede evaluar `example-list` y mira lo que devuelve:

```
example-list
     ⇒ ("aquí hay una claúsula" "otra claúsula")
```

Ahora, se puede añadir un nuevo elemento en esta lista evaluando la siguiente expresión:

```
(push "a third clause" example-list)
```

Cuando se evalúa `example-list`, se encuentra su valor es:

```
example-list
     ⇒ ("una tercera claúsula" "aquí hay una
     claúsula" "otra claúsula")
```

De este modo, la tercera claúsula se añadide a la lista con `push`.

Ahora para la segunda parte de la claúsula `if`. Esta expresión deja el kill ring desde lo creciente demasiado largo. Eso se ve de la siguiente manera:

```
(if (> (length kill-ring) kill-ring-max)
    (setcdr (nthcdr (1- kill-ring-max) kill-ring) nil))
```

El código chequea si el tamaño del anillo de la muerte *kill ring* es más grande que el máximo tamaño permitido. Este es el valor de `kill-ring-max` (que es 60, por defecto). Si el tamaño del anillo de la muerte *kill ring* es demasiado largo, entonces este código asigna el último elemento del anillo de la muerte *kill ring* a `nil`. Eso hace esto usando dos funciones, `nthcdr` y `setcdr`.

Nosotros vemos que `setcdr` temprano (véase Sección 7.6 “`setcdr`”, página 80). Eso asigna el CDR de una lista, solo como `setcar` asigna el CAR de una lista. En este caso, sin embargo, `setcdr` no estará configurando el CDR del kill ring completo; la función `nthcdr` es usada para causarlo para asignar el CDR del siguiente al último elemento del kill ring — esto significa que desde el CDR del siguiente al último elemnto del kill ring *anillo de la muerte*, eso asignará el último elemento del kill ring *anillo de la muerte*.

La función `nthcdr` funciona repetidamente tomando el CDR de una lista — eso toma el CDR del CDR del CDR .... Eso hace esto N veces y devuelve los resultados. (Véase Sección 7.3 “`nthcdr`”, página 77.)

De este modo, si teniamos una lista de cuatro elemento que era supuestamente de tres elementos, se podría asignar el CDR del siguiente al último elemento a `nil`, y por eso se ordena la lista. (Si se asigna el último elemento a algún otro valor a `nil`, que se podría hacer, entonces no se habría ordenado la lista. Véase Sección 7.6 “`setcdr`”, página 80.)

Se puede ver ordenando la evaluación de las siguientes tres expresiones en turno. Primero asigna el valor de `arboles` a `(arce encina pino abedul)` entonces asigna el CDR de su segundo CDR y entonces encuentra el valor de `arboles`.

```
(setq arboles '(arce encina pino abedul))
     ⇒ (arce encina pino abedul)

(setcdr (nthcdr 2 arboles) nil)
     ⇒ nil

árboles
     ⇒ (arce encina pino)
```

(El valor devuelto por la expresión `setcdr` es `nil` desde que es que el CDR es asignado.)

Para repetir, en `kill-new`, la función `nthcdr` toma el CDR un número de veces que es uno menos que el tamaño máximo permitido del anillo de la muerte *kill ring* y `setcdr` asigna el CDR de este elemento (que será el resto de los elementos en el anillo muerte) para `nil`. Esto previene el anillo de la muerte *kill ring* desde lo que crece demasiado largo.

De la siguiente a la última expresión en la función `kill-new` es

```
(setq kill-ring-yank-pointer kill-ring)
```

El `kill-ring-yank-pointer` es una variable global que es asignado para ser el `kill-ring`.

Incluso aunque el `kill-ring-yank-pointer` es llamado un '`puntero`', eso es una variable solo como el anillo de la muerte *kill ring*. Sin embargo, el nombre que ha sido elegido para ayudar a humanos a comprender cómo la variable se usa.

Ahora, para devolver rápido una expresión en el cuerpo de la función:

```
(if (fboundp 'menu-bar-update-yank-menu)
    (menu-bar-update-yank-menu string (and replace (car kill-ring))))
```

Empieza con una expresión `if`

En este caso, la expresión chequea primero ve si `menu-bar-update-yank-menu` existe como una función, y si así, se llama. La función `fboundp` devuelve cierto si el símbolo que se chequea tiene una definición de función que 'no es vacía'. Si el símbolo de la definición de función fuera vacío, recibiría un mensaje de erro, como se hizo cuando se creó errores intencionalmente (véase Sección 1.3 "Genera un Mensaje de Error", página 4).

La then-part contiene una expresión cuyo primer elemento es la función `and`.

La forma especial `and` evalúa cada uno de sus argumentos hasta que uno de los argumento devuelva un valor de `nil`, en cuyo caso la expresión `and` devuelve `nil`; sin embargo, si ninguno de los argumentos devuelve una valor de `nil`, el valor resultante desde la evaluación el último argumento es devuelto. (Desde que tal valor no es `nil`, eso es considerado cierto en Emacs Lisp.) En otras palabras, una expresión `and` devuelve un valor cierto solo si todos sus argumentos son verdaderos. (Véase Sección 5.4 "Revisar el segundo búffer relacionado", página 68.)

La expresión determina si el segundo argumento `menu-bar-update-yank-menu` es verdader o no.

`menu-bar-update-yank-menu` es una de la funciones que lo hace posible para usar el menu 'Seleccionar y Pegar' en el ítem Edit de una barra de menu; usando un ratón, se puede mirar en varias piezas de texto que se guardado y selecciona una pieza para pegar.

La última expresión en la función `kill-new` añade las cadenas nuevamente copiadas a aquella facilidad que existe copiando y pegando entre diferentes programas ejecutando un sistema de ventanas. En el Sistema de Ventanas de X, por ejemplo, la función `x-select-text` toma la cadena y la almacena en memoria operada por X. Se puede pegar la cadena en otro programa, tal como un Xterm.

La expresión se ve como:

```
(if interprogram-cut-function
    (funcall interprogram-cut-function string (not replace))))
```

Si una `interprogram-cut-fuction` existe, entonces Emacs ejecuta `funcall`, que en vez llama su primer argumento como una función y pasan los argumentos que permanecen a eso. (Incidentalmente, tan lejos como se puede ver, esta expresión `if` podría ser reemplazado por una expresión `and` similar a uno en la primera parte de la función.)

Estamos yendo a discutir sistemas de ventanas y otros programas más allá pero meramente nota que este es un mecanismo que habilita GNU Emacs a trabajar fácilmente y bien con otros programas.

Este código para emplazar texot en el anillo de la muerte *kill ring*, concatenado con un elemento existente o como un nuevo elemento, nos lidera al código para traer texto que ha sido cortado del búffer — los comandos de corte. Sin embargo, antes de discutir los comandos de corte, es mejor aprender cómo las listas son implementadas en un ordenador. Esto hará claro tales misterios como el uso del término 'puntero'. Pero antes de esto, desviaremos dentro de C.

8.4 Disgresión dentro de C

La función `copy-region-as-kill` (véase Sección 8.3 "copy-region-as-kill", página 90) usa la función `filter-buffer-substring`, que en vez de eso usa la función `delete-and-extract-region`. Eso elimina los contenidos de una región y no se puede volverlos a tener.

Al contrario que el otro código discutido aquí, la función `delete-and-extract-region` no está escrita en Emacs Lisp; eso está escrito en C y es una de las primitivas del sistema GNU Emacs. Puesto que es muy simple, se hará la disgresión brevemente desde el Lisp y se describe aquí.

Como muchas de las otras primitivas Emacs, `delete-and-extract-region` se escribe como una instancia de una macro C, una macro es una plantilla para codificar. La macro completa se parece a esto:

```
DEFUN ("delete-and-extract-region", Fdelete_and_extract_region,
       Sdelete_and_extract_region, 2, 2, 0,
       doc: /* Borra el texto entre START y END y lo devuelve.  */)
       (Lisp_Object start, Lisp_Object end)
{
  validate_region (&start, &end);
  if (XINT (start) == XINT (end))
    return empty_unibyte_string;
  return del_range_1 (XINT (start), XINT (end), 1, 1);
}
```

Sin ir dentro de los detalles de la macro que escribe el proceso, se hará un apunte de esta macro que empieza con la palabra `DEFUN`. La palabra `DEFUN` fué elegida puesto que el código sirve para el mismo propósito que `defun` hace en Lisp. (La macro C `DEFUN` definida en `emacs/src/lisp.h`.)

El palabra `DEFUN` tiene siete partes dentro de los paréntesis:

- La primera parte es el nombre dado a la función en Lisp, `delete-and-extract-region`.
- La segunda parte es el nombre de la función en C, `Fdelete_and_extract_region`. Por convención, eso empieza con '`F`'. Puesto que C no usa guiones en nombres, los guiones bajos son usados a su vez.
- La tercera parte es el nombre para la estructura constante C que registra información en esta función para uso interno. Es el nombre de la función en C pero empieza con una '`S`' en vez de una '`F`'.

- Las partes cuarta y quinta especifican el número mínimo y máximo de argumentos que la función puede tener. Esta función demanda exactamente 2 argumentos.
- La sexta parte está cerca del argumento que sigue la declaración `interactive` en una función escrita en Lisp: una carta seguida, quizás, por una consola. La única diferencia desde el Lisp es cuando la macro se llama sin argumentos. Entonces escribe un `0` (que es una 'cadena nula'), como en esta macro.

  Si se fueran a especificar argumentos, se emplazarían entre marcas de comillas. La macro C para `goto-char` incluye `\"NGoto char \"` en esta posición se indica que la función espera un prefijo plano, en este caso, una localización numérica en un búffer, y provee una consola.
- La séptima parte es una cadena de documentación, solo como la única para una función escrita en Emacs Lisp. Esto es escrito como un comentario C. (Cuando se escribe Emacs, el programa `lib-src/make-docfile` extrae estos comentarios y los usa para crear la documentación "real".)

En una macro C, los parámetros son los siguientes, con una frase de este tipo de objeto se siguen por lo que podría ser llamado el 'cuerpo' de la macro. Para `delete-and-extract-region` el 'cuerpo' consiste de las siguientes cuatro líneas:

```
validate_region (&start, &end);
if (XINT (start) == XINT (end))
  return build_string ("");
return del_range_1 (XINT (start), XINT (end), 1, 1);
```

La función `validate_region` chequea si los valores pasados como el principio y fin de la región son el tipo apropiado y son del mismo rango. Si las posiciones del principio y fin son lo mismo, entonces devuelve una cadena vacía.

La función `del_range_1` actualmente borra el texto. Eso es una función compleja que no miraremos. Eso actualiza el búffer y hace otras cosas. Sin embargo, es el valorable mirar los dos argumentos pasados para `del_range`. Estos son `XINT (start)` y `XINT (end)`.

Tan lejos como el lenguaje C es concebido, `start` y `end` son dos enteros que marcan el principio y el fin de la región para ser borrada[1].

En las primeras versiones de Emacs, estos dos números fueron 32 bits de longitud, pero el código está lentamente siendo generalizado para manejar otras longitudes. Tres de los bits disponibles son usados para especificar el tipo de información; los bits permanecen ser usados como 'contenido'.

'`XINT`' es una macro C que extrae los números relevantes desde la colección larga de bits; los otro tres bits son descartados.

El comando en `delete-and-extract-region` se parece a esto:

```
del_range_1 (XINT (start), XINT (end), 1, 1);
```

Esto borra la región entre la posición del principio, `start`, y la posición final, `end`.

[1] Más precisamente, y requiriendo conocimiento más experto para comprender, los dos enteros son del tipo 'Lisp_Object', que puede tambiés ser una unión C en vez de un tipo de entero.

Desde el punto de vista de la persona que escribe Lisp, Emacs es muy simple; pero oculta en el fondo un gran trato de complejidad para hacer todo el trabajo.

8.5 Inicializando una variable con `defvar`

La función `copy-region-as-kill` es escrita en Emacs Lisp. Dos funciones con eso, `kill-append` y `kill-new`, copiar una región en un búffer y guardarlo en una variable llamada el `kill-ring`. Esta sección describe cómo la variable `kill-ring` es creada e inicializada usando la forma especial `defvar`.

(De nuevo se nota que el término `kill-ring` es un sin nombre. El texto que es cortado fuera del búffer puede ser traido; eso no es un corpus de anillo, pero un anillo de texto resucitable.)

En Emacs Lisp, una variable tal como `kill-ring` es creada y dada por un valor inicial usando la forma especial `defvar`. El nombre desde "define variable".

La forma especial `defvar` es similar a `setq` en este se configura el valor de una variable. Eso no es `setq` en dos modos; primero solo configura el valor de la variable si la variable no tiene ya un valor. Si la variable ya tiene un valor, `defvar` no sobreescribe el valor existente. Segundo, `defvar` tiene una cadena de documentación.

(Otra forma especial, `defcustom`, está diseñado para variables que la gente personaliza. Eso tiene más funcionalidades que `defvar`. (Véase Sección 16.2 "Configurando Variables con `defcustom`", página 192.)

Se puede ver el actual valor de una variable, cualquier variable, usando la función `describe-variable`, que es normalmente invocado escribiendo *`C-h v`*. Si se escribe *`C-h v`* y `kill-ring` (seguido por `RET`), se verá que hay en tu anillo de la muerte actual *kill ring* al ser pulsado — ¡esto puede ser bastante tranquilo! A la inversa, si no has estado haciendo nada esta sesión en Emacs, excepto leer este documento, se puede no tener nada dentro. También, se verá la documentación para `kill-ring`:

```
Documentación:
Lista de secuencias de texto muerto (guardado).
Desde que el (kill ring) se supone que interactua bien con
cut-and-paste facilita ofrecer por sistemas de ventanas,
debería usar esta variable
interactúa bin con ‘interprogram-cut-function’ y
‘interprogram-paste-function’. Las funciones ‘kill-new’,
‘kill-append’, y ‘current-kill’ se suponen para implementar esta
interacción; se puede querer usarlo en vez de manipular en anillo de
la muerte kill ring directamente.
```

El kill ring *anillo de la muerte* está definido por un `defvar` del siguiente modo:

```
(defvar kill-ring nil
  "Lista de secuencia de textos cortados.
...")
```

En esta definición de variable, la variable es dada un valor inicial de `nil`, que tiene sentido, desde que si no se ha guardado nada, no se quiere nada si se da un comando `yank`. La cadena de documentación es escrito solo como la cadena de documentación de un `defun`. Como con la cadena de documentación sería una frase completa, desde que algunos comandos, como `apropos`, imprime solo la primera

línea de documentación. Las líneas de exito no serían indentadas; de otro modo se mira cuando se usa `C-h v` (`describe-variable`).

8.5.1 `defvar` y un asterisco

En el pasado, Emacs usaba la forma especial `defvar` tanto para variables interna que no esperaría que un usuario cambie y para variables que espera un usuario cambie. Aunque se puede todavía usar `defvar` para variables personalizadas, por favor, usa `defcustom` en vez, desde que la forma especial provee una ruta dentro de los comando de Personalización. (Véase Sección 16.2 "Especificando Variables usando `defcustom`", página 192.)

Cuando tu especificaste una variable usando la forma especial `defvar`, tu podrías distinguir una variable que un usuario podría querer cambiar desde otros escribiendo, '`*`', en la primera columna de su cadena de documentación. Por ejemplo:

```
(defvar shell-command-default-error-buffer nil
  "*Nombre de buffer para ‘shell-command’ ... salir del error.
... ")
```

Tu podrías (y todavía puedes) usar el comando `set-variable` para cambiar el valor de `shell-command-default-error-buffer` temporalmente. Sin embargo, las opciones configuradas usando `set-variable` no están asignadas solo por la duración de tu sesión de edición. Los nuevos valores no están guardados entre sesiones. Cada vez que Emacs empieza, lee el valor original, a menos que tu cambia el valor con tu fichero `.emacs`, si configurándolo manualmente o usando `customize`. Véase Capítulo 16 "Tu Fichero `.emacs`", página 191.

Para mí, el mayor uso del comando `set-variable` es sugerir variables que se podrían querer asignar en mi fichero `.emacs`. Ahora hay más de 700 variables, demasiadas para recordarlas fácilmente. Afortunadamente, se puede presionar `TAB` después de llamar al comando `M-x set-variable` para ver la lista de variables. (Véase Sección "Examinando y Configurando Variables" in *El Manual de GNU Emacs*.)

8.6 Revisar

Aquí hay un breve resumen de algunas funciones introducidas recientemente.

`car`
`cdr` — `car` devuelve el primer elemento de una lista; `cdr` devuelve el segundo y subsiguientes elementos de una lista.

Por ejemplo:

```
(car '(1 2 3 4 5 6 7))
     ⇒ 1
(cdr '(1 2 3 4 5 6 7))
     ⇒ (2 3 4 5 6 7)
```

`cons` — `cons` construye una lista enlazando su primer argumento a su segundo argumento.

Por ejemplo:

```
(cons 1 '(2 3 4))
     ⇒ (1 2 3 4)
```

`funcall` `funcall` evalúa su primer argumento como una función. Así pasa los argumentos que permanecen a su primer argumento.

`nthcdr` Devuelve el resultado de tomar CDR 'n' veces en una lista. The n^{th} `cdr`. El 'resto del resto', como estaba

Por ejemplo:

```
(nthcdr 3 '(1 2 3 4 5 6 7))
     ⇒ (4 5 6 7)
```

`setcar`
`setcdr` `setcar` cambia el primer elemento de una lista; `setcdr` cambia el segundo y subsiguiente elementos de una lista.

Por ejemplo:

```
(setq triple '(1 2 3))

(setcar triple '37)

triple
     ⇒ (37 2 3)

(setcdr triple '("foo" "bar"))

triple
     ⇒ (37 "foo" "bar")
```

`progn` Evalúa cada argumento en secuencia y entonces devuelve el valor del último.

Por ejemplo:

```
(progn 1 2 3 4)
     ⇒ 4
```

`save-restriction`
Graba siempre que encoger esté en efecto en el búffer, si cualquiera, restaura este encogimiento después de evaluar los argumentos.

`search-forward`
Buscar una cadena, y si la cadena es encontrada, mueve el punto. Con una expresión regular, usa algo similar a `re-search-forward`. (Véase Capítulo 12 "Buscar regexp", página 135, para una explicación de expresiones regulares patrones y búsquedas.)

`search-forward` y `re-search-forward` tiene cuatro argumentos:

1. La cadena o la expresión regular para buscar.
2. Opcionalmente, el límite de la búsqueda.
3. Opcionalmente, que haces si la búsqueda falla, devuelve `nil` o un mensaje de error.
4. Opcionalmente, cuántas veces repetir la búsqueda; si negativa, la búsqueda va hacia atrás.

`kill-region`
`delete-and-extract-region`
`copy-region-as-kill`

`kill-region` corta el texto entre punto y marca desde el búffer y almacena ese texto en el anillo de la muerte *kill ring*, así se puede obtener pegándolo.

`copy-region-as-kill` copia el texto entre punto y marca dentro del anillo de la muerte *kill ring*, que se puede obtener pegándolo. La función no corta o borra el texto desde el búffer.

`delete-and-extract-region` elimina el texto entre el punto y marca desde el búffer y a través. No se puede volver. Esto no es un comando interactivo.)

8.7 Buscando ejercicios

- Escribe una función interactiva que busca una cadena. Si la búsqueda encuentra la cadena, deja el punto después y muestra un mensaje que dice "¡Encontrado!". (No use `search-forward` como nombre de esta función; si se hace, se sobreescribirá la versión existente `search-forward` que viene con Emacs. Use un nombre tal como `test-search` en vez de eso.
- Escribe una función que imprime el tercer elemento del kill ring *anillo de la muerte* en el área echo, si cualquiera; si el kill ring *anillo de la muerte* no contiene un tercer elemento, imprime un mensaje apropiado.

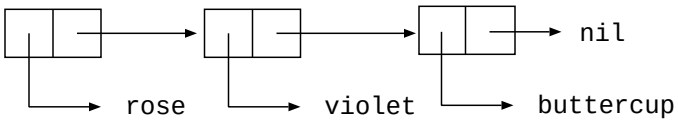
9 Cómo las listas se implementan

En Lisp, los átomos se graban de manera simple, si la implementación no es sencilla en la práctica, no es, nada sencilla en la teoría. El átomo '`rosa`', por ejemplo, se graba como las cuatro letras contiguas '`r`', '`o`', '`s`', '`a`'. Una lista, por otro lado, se guarda de manera diferente. El mecanismo es igualmente simple, pero toma un momento para tener usada la idea. Una lista se guarda usando una serie de pares de punteros. En las series, el primer puntero en cada par de puntos a un átomo o a otra lista, y el segundo puntero en cada par al siguiente par, o al símbolo `nil`, que marca el fin de la lista.

Un puntero por sí mismo es poco simple a la dirección electrónica de la que está apuntada. Aquí, una lista se guarda como una serie de direcciones electrónicas.

Por ejemplo, la lista `(rosa violeta mantequilla)` tiene tres elementos, '`rosa`', '`violeta`', y '`mantequilla`'. En el ordenador, la dirección electrónica de '`rosa`' se graba en un segmento de memoria del ordenador a través de la dirección que da la dirección electrónica de donde el átomo '`violeta`' está localizado; y esta dirección (la que cuenta donde '`violeta`' está se localiza) se guarda con una dirección que cuenta donde la dirección para el átomo '`mantequilla`' se localiza.

Esto parece más complicado de lo que es y es más fácil visto en un diagrama:

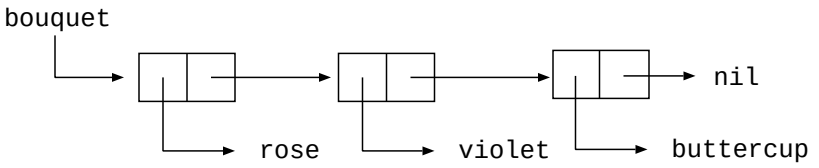


En el diagrama, cada caja representa una palabra de memoria del ordenador que maneja un objeto Lisp, normalmente en la forma de una dirección de memoria. Las cajas, por ej. las direcciones, están en pares. Cada flecha apunta a lo que la dirección es la dirección de, si un átomo u otro par de direcciones. La primera caja es la dirección electrónica de '`rosa`' y la flecha apunta a '`rosa`'; la segunda caja es la dirección del siguiente par de cajas, la primera parte de la que es la dirección de '`violeta`' y la segunda parte es la dirección del siguiente par. La última caja apunta al símbolo `nil`, que marca el fin de la lista.

Cuando una variable es configurado a una lista con una función tal como `setq`, almacena la dirección de la primera caja en la variable. De este modo, la evaluación de la expresión es:

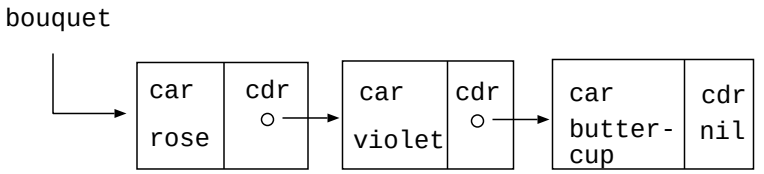
```
(setq ramo '(rosa violeta mantequilla))
```

crea una situación como esta:



En este ejemplo, el símbolo `ramo` maneja la dirección del primer par de cajas.

Esta misma lista puede ser ilustrada en un modo diferente de anotación de cajas como esta:

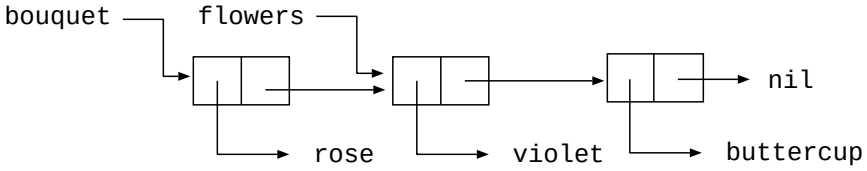


(Los símbolos consisten de más pares de direcciones, pero la estructura de un símbolo es hecha de direcciones. De manera profunda, el símbolo `ramo` consiste de un grupo de cajas-de-direcciones, una que es la dirección de la palabra impresa '`ramo`', una segunda de la que es la dirección de una definición de función adjunta al símbolo, si cualquiera, un tercero del que es la dirección del primer par de cajas-de-direccion para la lista `(rosa violeta mantequilla)`, y así. Aquí se está mostrando que la tercera caja de dirección del símbolo apunta al primer par de cajas-de-direccion para la lista.)

Si un símbolo se asigna al CDR de una lista, la lista en sí no cambia; el símbolo simplemente tiene una dirección abajo de la lista. (En la jerga, CAR y CDR son 'no destructivos'.) De este modo, se evalúa la siguiente expresión

```
(setq flores (cdr ramo))
```

produce esto:



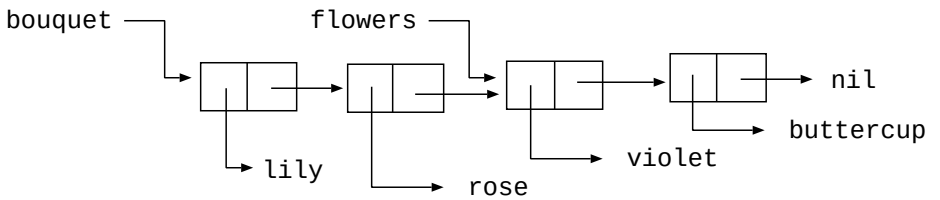
El valor de `flores` es `(violeta mantequilla)`, esto es decir que el símbolo `flores` maneja la dirección del par address-boxes el primero que maneja la dirección de `violeta`, y el segundo que maneja la dirección de `mantequilla`.

Un par de cajas-de-direcciones se llama una *cons cell* o *par de puntos*. Véase Sección "la Célula Cons y los Tipos Lista" in *El Manual de Referencia de Emacs Lisp*, y Sección "Notación de Pares de Puntos" in *El Manual de Referencia de GNU Emacs Lisp*, para más información acerca de células cons y pares de puntos.

La función `cons` añade un nuevo par de direcciones al frente de una serie de direcciones como son mostradas debajo. Por ejemplo, evaluando la expresión

```
(setq ramo (cons 'lila ramo))
```

produce:



Sin embargo, esto no cambia el valor del símbolo `flores`, así puedes ver evaluando lo siguiente,

```
(eq (cdr (cdr ramo)) flores)
```

que devuelve `t` para verdad.

Hasta que se resetea, `flores` todavía tiene el valor de `(violeta mantequilla)`; que es, eso tiene la dirección de la celula cons cuya primera dirección es `violeta`. También, esto no altera cualquier célula prexistente cons; ellas está todavía allí.

De este modo, en Lisp, tiene el CDR de una lista, se obtiene la dirección del siguiente cons en las serie; para tener el CAR de una lista, se obtiene la dirección del primer elemento de la lista; para `cons` un nuevo elemento en una lista, se añade una nueva célula cons al frente de la lista. ¡Esto es todo lo que hay así! ¡La estructura subyacente de Lisp es brillantemente simple!

¿Y qué hace la última dirección en una serie de células cons se refieren? Eso es la dirección de la lista vacía, de `nil`.

En resumen, cuando una variable Lisp es asignada a un valor, eso provee con la dirección de la lista a la que la variable se refiere.

9.1 Símbolos como una caja con cajones

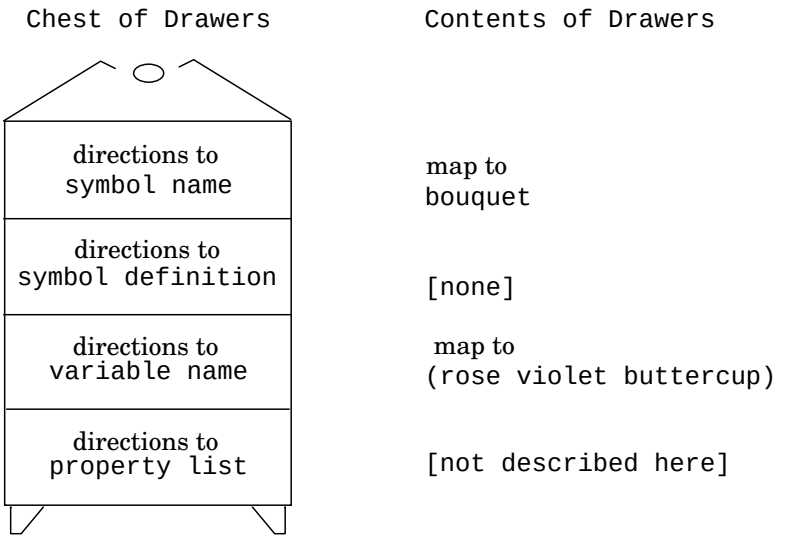
En una sección temprana, se sugería que se podría imaginar un símbolo siendo una caja con cajones. La definición de función se pone en un cajón, el valor en otro, y así. Lo que se pone en el cajón maneja el valor que puede cambiarse sin afectar a los contenidos del cajón manejando la definición de función, y viceversa.

Actualmente, lo que está puesto en cada cajón es la dirección del valor o definición de función. Eso es como si se encontrara un viejo cajón en el ático, y en uno

de sus cajones se encontrara un mapa dándote direcciones a donde está el tesoro escondido.

(Además de su nombre, la definición del símbolo, y un valor de la variable, un símbolo tiene un 'cajón' para una *lista de propiedades* que puede ser usada para grabar otra información. Las listas de propiedades no se discuten aquí; ver Sección "Listas de Propiedades" in *El Manual de Referencia de Emacs Lisp*.)

Aquí hay una representación visionaria:



9.2 Ejercicio

Asignar `flores` a `violeta` y `mantequilla`. Asigna dos flores más en esta lista y asigna esta nueva lista a `more-flowers`. Asigna el CAR de `flores` a un pez. ¿Qué lista contiene ahora `mas-flores`?

10 Pegando texto

Siempre y cuando se corta texto fuera de un búffer con un comando 'kill' en GNU Emacs, se puede traer con un comando 'copiar'. El texto cortado del búffer es puesto en el anillo de la muerte y en los comandos copiar, insertan los contenidos apropiados del kill ring detrás de un búffer (no necesariamente el búffer original).

Un simple comando `C-y` (`yank`) inserta el primer ítem desde el anillo de la muerte *kill ring* dentro del actual búffer. Si el comando `C-y` es seguido inmediatamente para `M-y`, el primer elemento se reemplaza por el segundo elemento. Los sucesivos comandos `M-y` reemplazan el segundo elemento con el tercer, cuarto, o quinto elemento, y así. Cuando se llega al último elemento en el anillo de la muerte *kill ring*, se reemplaza por el primer elemento y el ciclo se repite. (De este modo, el kill ring se llama un 'anillo' en vez de solo una 'lista'. Sin embargo, la estructura de de datos actual que maneja el texto es una lista. Véase ⟨undefined⟩ "Manejando el anillo de la muerte *kill ring*", página ⟨undefined⟩, para los detalles de cómo la lista es manejada como un anillo.)

10.1 Resumen del anillo de la muerte

El anillo de la muerte *kill ring* es una lista de cadenas textuales. Esto es lo que se ve:

```
("algun texto" "una pieza diferente pieza de texto"
"todavía más texto")
```

Si estos fueran los contenidos de mi anillo de la muerte *kill ring* y yo presionara `C-y`, la cadena de caracteres diciendo '`algún texto`' sería insertado en este búffer donde mi cursor está localizado.

El comando `yank` *pegar* es también usado para duplicar texto copiándolo. El texto copiado no es cortado desde el búffer, pero una copia de eso se pone en el anillo de la muerte *kill ring* y se inserta pegándolo.

Tres funciones se usan para atraer texto desde el anillo de la muerte *kill ring*: `yank` (*pegar*), que normalmente se asocian a `C-y`; `yank-pop`, que normalmente se asocia a `M-y`; y `rotate-yank-pointer`, que se usa por las otras dos funciones.

Estas funciones se refieren al kill ring *anillo de la muerte* a través de una variable llamada el `kill-ring-yank-pointer`. En vez de eso, la inserción del código para ambos son las funciones `yank` y `yank-pop`:

```
(insert (car kill-ring-yank-pointer))
```

(Bien, no más. En GNU Emacs 22, la función se ha reemplazado por `insert-for-yank` que llama a `insert-for-yank-1` repetitivamente para cada segmento `yank-handler`. En vez de eso, `insert-for-yank-1` destituye las propiedades de texto desde el texto insertado de acuerdo a `yank-excluded-properties`. De otro modo, eso es como `insert`. Nosotros lo pegamos con un `insert` plano puesto que sea fácil de comprender.)

Para empezar a comprender cómo `yank` y `yank-pop` funcionan, primero es necesario mirar en la variable `kill-ring-yank-pointer`.

10.2 La variable `kill-ring-yank-pointer`

`kill-ring-yank-pointer` es una variable, solo como `kill-ring` es una variable. Eso apunta a alguna cosa siendo asignada al valor de lo que apunta, como cualquier otra variable Lisp.

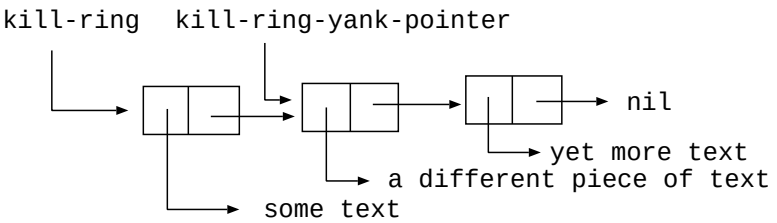
De este modo, si el valor del kill ring es:

```
("algun texto" "una pieza diferente pieza de texto"
"todavía más texto")
```

y el `kill-ring-yank-pointer` apunta a la segunda cláusula, el valor de `kill-ring-yank-pointer` es:

```
("una pieza diferente de texto" "todavía más texto")
```

Como se explica en el capítulo previo (Véase Capítulo 9 "Implementación de listas", página 104), el ordenador no guarda dos copias diferentes del texto siendo apuntado por ambos el `kill-ring` (el *anillo de la muerte*) y el `kill-ring-yank-pointer` (el *puntero de pegar el anillo de la muerte*). Las palabras "una pieza diferente de texto" y "todavía más texto" no están duplicadas. En vez de eso, las dos variables Lisp apuntan a las mismas piezas de texto. Aquí hay un diagrama:



Tanto la variable `kill-ring` y la variable `kill-ring-yank-pointer` son punteros. Pero el kill ring *anillo de la muerte* en sí es normalmente descrito como si fuera actualmente de lo que está compuesto. El `kill-ring` se refiere a lo que es la lista en vez de lo que apunta a la lista. Conversando, el `kill-ring-yank-pointer` se refiere a como se apunta a una lista.

Estas dos maneras hablar acerca de la misma cosa suena confuso al principio pero tiene sentido para reflexionar. El kill ring *anillo de la muerte* es generalmente pensado como la estructura completa de datos que manejan la información de lo que se ha cortado recientemente de los búffers de Emacs. El `kill-ring-yank-pointer` en la otra mano, sirve para indicar — que es, para 'apuntar a' — esta parte del anillo de la muerte del que el primer elemento (el CAR) será insertado.

10.3 Ejercicios con `yank` y `nthcdr`

- Usando `C-h v` (`describe-variable`), mira en el valor de tu kill ring *anillo de la muerte*. Añade varios ítems a tu anillo de la muerte *kill ring*; mira en su valor de nuevo. Usando `M-y` (`yank-pop`, mueve todo el camino alrededor del kill ring *anillo de la muerte*. ¿Cuántos ítems estaban en tu kill ring *anillo de la muerte*? Encuentra el valor de `kill-ring-max`. ¿Estaba tu anillo de la muerte *kill ring* completo, o podrías haber guardado más bloques de texto dentro?
- Usando `nthcrd` y `car`, construye una serie de expresiones para devolver, el primer, segundo, tercer y cuarto elemento de una lista.

11 Bucles y recursión

Emacs Lisp tiene dos caminos primarios para causar una expresión, o una serie de expresiones, para ser evaluado repetidamente: uno usa un bucle `while`, y el otro usa *recursión*.

La repetición puede ser valorable. Por ejemplo, para mover hacia delante cuatro frases, tu solo necesitas escribir un programa que moverá hacia delante una frase y entonces repite el proceso cuatro veces. Ya que un ordenador no está aburrido o cansado, tal acción repetitiva no tiene los efectos de borrado por equivocación o exceso que pueden tener los humanos.

La gente mayoritariamente escribe funciones de Emacs Lisp usando bucles `while`; pero se puede usar recursión, que provee un poderoso camino mental para resolver problemas[1].

11.1 `while`

La forma especial `while` chequea si el valor devuelto para evaluar el primer argumento es verdadero o falso. Esto es parecido a lo que el intérprete Lisp hace con un `if`; el intérprete hace lo siguiente, sin embargo, es diferente.

En una expresión `while`, si el valor devuelto por evaluar el primer argumento es falso, el intérprete Lisp escapa del resto de la expresión (el *cuerpo* de la expresión) y no la evalúa. Sin embargo, si el valor es cierto, el intérprete Lisp evalúa el cuerpo de la expresión y entonces de nuevo chequea si el primer argumento para `while` es cierto o falso. Si el valor devuelto de evaluar el primer argumento es cierto de nuevo, el intérprete Lisp evalúa el cuerpo de la expresión.

La plantilla para una expresión `while` se ve así:

```
(while true-or-false-test
  body...)
```

En el momento en el que el true-or-false-test de la expresión `while` devuelve un valor cierto cuando eso se evalúa, el cuerpo es repetidamente evaluado. Este proceso se llama bucle puesto que el intérprete Lisp repite la misma cosa una y otra vez, como un avión haciendo un loop. Cuando el resultado de evaluar el true-or-false-test es falso, el intérprete Lisp no evalúa el resto de la expresión `while` y 'existe el bucle'.

Claramente, si el valor devuelto evaluando el primer argumento para `while` es siempre cierto, el cuerpo siguiente será evaluado una y otra vez ... y ... para siempre. Recíprocamente, si el valor devuelto nunca es cierto, las expresiones en el cuerpo nunca serán evaluadas. La fortaleza de escribir un bucle `while` consiste de elegir un mecanismo tal que el true-or-false-test devuelva cierto solo el número de

[1] Se pueden escribir funciones recursivas para ser frugal o basura mental o recursos de ordenador; como eso ocurre, los métodos que la gente encuentra fáciles — son frugales de 'recursos mentales' — algunas veces usan recursos de ordenador considerables. Emacs fué diseñado para ejecutarse en máquinas que ahora se consideran limitadas y sus configuraciones por defecto son conservadoras. Se puede querer incrementar los valores de `max-specdl-size` y `max-lisp-eval-depth`. En mi fichero `.emacs`, yo los asigno a 15 o 30 veces su valor por defecto.

veces que se quieren las subsiguientes expresiones para ser evaluadas, y entonces tener el test devuelto a falso.

El valor devuelto evaluando `while` es el valor del true-or-false-test. Una consecuencia interesante de esto es que un bucle `while` que evalúa sin errores devolverá `nil` o falso sin dignidad de si eso ha girado 1 o 100 veces o ninguna. ¡Una expresión `while` que se evalúa de manera exitosa nunca devuelve un valor cierto! Lo que esto significa es que `while` es siempre evaluado por sus efectos laterales, que es decir, las consecuencias de evaluar las expresiones con el cuerpo del bucle `while`. Esto tiene sentido. Eso no es el mero acto del bucle que es deseado, pero las consecuencias de lo que ocurre cuando las expresiones en el bucle son repetidamente evaluadas.

11.1.1 Un bucle `while` y una lista

Un camino común para controlar un bucle `while` es chequear si una lista tiene cualquier elemento. Si eso se hace, el bucle se repite; pero si no, la repetición se finaliza. Desde esto es una técnica importante, creará un corto ejemplo para ilustrarlo.

Un camino simple para chequear si una lista tiene elementos es evaluar la lista: si eso no tiene elementos, si es una lista vacía y devolverá la lista vacía, `()`, que es un sinónimo para `nil` o falso. Por otro lado, una lista con elementos devolverá estos elementos cuando eso se evalúa. Puesto que Emacs Lisp considera como cierto cualquier valor que no es `nil`, una lista que devuelve elementos chequeará cierto en un bucle `while`.

Por ejemplo, se puede asignar la variable `empty-list` a `nil` para evaluar la siguiente expresión `setq`:

```
(setq empty-list ())
```

Después de evaluar la expresión `setq`, se puede evaluar la variable `empty-list` es el camino normal, posicionando el cursor después del símbolo y escribiendo *`C-x C-e`*; `nil` aparecerá en tu área echo:

```
empty-list
```

Por otro lado, si se asigna una variable para ser una lista con elementos, la lista aparecerá cuando se evalúe la variable, como se puede ver evaluando las siguientes dos expresiones:

```
(setq animales '(gacela jirafa leon tigre))

animales
```

De este modo, para un bucle `while` que chequea si hay cualquier ítem en la lista `animales`, la primera parte del bucle será escrito así:

```
(while animales
       ...
```

Cuando el `while` chequea su primer argumento, la variable `animales` se evalúa. Eso devuelve una lista. Mientras la lista tiene elementos, el `while` considera los resultados del test para ser verdadero; pero cuando la lista es vacía, eso considera los resultados del test para ser falso.

Para prevenir que el bucle `while` se ejecute siempre, se necesita proporcionar algún mecanismo. Una técnica usada con frecuencia es tener una de las subsiguientes

formas en la expresión `while` que asigna el valor de la lista para ser el CDR de la lista. Cada vez que la función `cdr` se evalúa, se va reduciendo, hasta que finalmente solo queda la lista vacía. En este punto, el test del bucle `while` devolverá falso, y los argumentos para el `while` no se evaluarán.

Por ejemplo, la lista de animales asociada a la variable `animals` se puede asignar a ser el CDR de la lista original con la siguiente expresión:

```
(setq animals (cdr animals))
```

Si se han evaluado las expresiones previas y entonces se evalúa esta expresión, se verá `(jirafa leon tigre)` que aparecerá en el área echo. Si se evalúa la expresiń de nuevo, `(leon tigre)` aparecerá en el área echo. Si se evalúa de nuevo, `(tigre)` y todavía de nuevo aparecerá la lista vacía y se mostrará como `nil`.

Una plantilla para un bucle `while` usa la función `cdr` repetidamente para causar el true-or-false-test finalmente para chequear la veracidad y se parece a esto:

```
(while test-whether-list-is-empty
  body...
  set-list-to-cdr-of-list)
```

Este chequeo y uso de `cdr` puede ser puesto junto a una función que va a través de una lista e imprime cada elemento de la lista en una línea de sí misma.

11.1.2 Un ejemplo: `print-elements-of-list`

La función `print-elements-of-list` ilustra un bucle `while` con una lista.

La función requiere varias líneas por su salida. Si estás leyendo esto en una instancia reciente de GNU Emacs, se puede evaluar la siguiente expresión dentro de Info, de normal.

Si se está usando una versión antigua de Emacs, es necesario copiar las expresiones necesarias para el búffer `*scratch*` y evaluarlas allí. Esto es porque el área echo tenía solo una línea en las versiones antiguas.

Se pueden copiar las expresiones marcando el principio de la región con *`C-SPC`* (`set-mark-command`), moviendo el cursor al fin de la región y entonces copiando la región usando *`M-w`* (`kill-ring-save`, que llama a `copy-region-as-kill` y entonces provee realimentación visual). En el búffer `*scratch*`, se pueden copiar las expresiones escribiendo *`C-y`* (`yank`).

Después de haber copiado las expresiones al búffer `*scratch*`, se evalúa cada expresión en orden. Asegúrese de evaluar la última expresión, `(print-elements-of-list animals)`, escribiendo *`C-u C-x C-e`*, que es, dando un argumento para `eval-last-sexp`. Esto causará el resultado de la evaluación para ser impreso en el búffer `*scratch*` en vez de siendo impreso en el área echo. (De otro modo se verá alguna cosa como esto en tu área echo: `^Jgacela^J^Jjirafa^J^Jleon^J^Jtigre^Jnulo`, en cada '`^J`' se estructura una 'nueva línea'.)

En una instancia de GNU Emacs reciente, se pueden evaluar estas expresiones directamente en el búffer Info, y el área echo crecerá para mostrar los resultados.

```
(setq animales '(gacela jirafa leon tigre))

(defun print-elements-of-list (list)
  "Imprime cada elemento de LIST en una línea."
  (while list
    (print (car list))
    (setq list (cdr list))))

(print-elements-of-list animales)
```

Cuando se evalúan las tres expresiones en secuencia, se verá esto:

```
gacela

jirafa

leon

tigre
nil
```

Cada elemento de la lista se imprime en una línea en sí (que es lo que la función `print` hace) y entonces el valor devuelto por la función se imprime. Desde que la última expresión en la función es el bucle `while`, y desde que el bucle `while` siempre devuelve `nil`, un `nil` se imprime después del último elemento de la lista.

11.1.3 Un bucle con un contaje incremental

Un bucle no es útil a menos que pare cuando debe. Bajo el control de un bucle con una lista, un camino común de parar un bucle es escribir el primer argumento como un test que devuelve falso cuando el número correcto de repeticiones es completo. Esto significa que el bucle debe tener un contador — una expresión que cuenta cuántas veces el bucle se repite a sí mismo.

El test para un bucle con un contador de incremento puede ser una expresión tal como `(< count desired-numberj)` que devuelve `t` para verdad si el valor de `count` es menor que el `desired-number` de repeticiones y `nil` para falso si el valor de `count` es igual a o es mayor que el `desired-number`. La expresión que incrementa el contador puede ser un simple `setq` tal como `(setq count (1+ count))`, donde `1+` es una función construida en Emacs Lisp que añade 1 a su argumento. (La expresión `(1+ count)` tiene el mismo resultado que `(+ count 1)`, que es fácil de leer para un humano.)

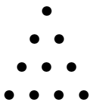
La plantilla para un bucle `while` controlado por un contador que se incrementa se parece a esto:

```
set-count-to-initial-value
(while (< count desired-number)         ; true-or-false-test
  body...
  (setq count (1+ count)))              ; incrementer
```

Note que se necesita asignar el valor inicial de `count`; normalmente asignado a 1.

Ejemplo con contador incremental

Supón que estás jugando en la playa y decides crear un triángulo de asteriscos, poniendo un asterisco en la primera fila, dos en la segunda fila, tres en la tercera fila y así:



(Hace 2500 años, Pitágoras y otras desarrollaron los principios de la teoría de números considerando preguntas como esta.)

Supón que quieres saber cuántos asteriscos necesitarás crear para un triángulo con 7 filas

Claramente, lo que necesitas hacer es añadir los números de 1 a 7. Hay dos caminos para hacer esto; se puede comenzar con los números más pequeños, uno, y añadir la lista en secuencia, 1, 2, 3, 4 y así; o empieza con el número más largo y añade la lista bajando: 7, 6, 5, 4 y así. Porque ambos mecanismos ilustran caminos comunes de escribir el bucle `while`, crearemos dos ejemplos, uno contando hacia arriba y el otro contando hacia abajo. En este primer ejemplo, empezaremos con 1 y añadimos 2, 3, 4 y así.

Si se quiere sumar toda una lista de números, el camino más fácil para hacer eso es sumar todos los números a la vez. Sin embargo, si no se sabe cuántos números tendrá la lista, o si se requiere estar preparado para una lista muy larga, entonces se necesita diseñar la adición, esto es, repetir un proceso simple muchas veces en vez de hacer un proceso más complejo.

Por ejemplo, en vez de añadir todas los asteriscos a la vez, lo que se puede hacer es añadir el número de asteriscos en la primera fila, 1, para el número en la segunda fila, 2, y entonces añadir el total de estas dos filas a la tercera fila, 3. Entonces se puede añadir el número en la cuarta fila, 4, al total de las primeras tres filas; y así.

La característica crítica del proceso es que cada acción repetitiva sea simple. En este caso, en cada paso nosotros añadimos solo dos números, el número de asteriscos en la fila y el total ya encontrado. Este proceso de añadir dos números es repetido de nuevo y de nuevo hasta la última fila que ha sido añadida al total de todas las filas precedentes. En un bucle más complejo la acción repetitiva podría no ser tan simple, pero será tan simple como hacer todo a la vez.

Las partes de la definición de función

El análisis precedente nos da los bonos de nuestra definición de función: primero, necesitaremos una variable que podemos llamar `total` que será el número total de asteriscos. Esto será el valor devuelto por la función.

Segundo, sabemos que la función requerirá un argumento: este argumento será el número de filas en el triángulo. Eso puede llamarse `number-of-rows`.

Finalmente, se necesita una variable para usarse como contador. Se podría llamar a esta variable `counter`, pero un nombre mejor es `row-number`. Debido a que lo que

el contador hace en esta función es contar filas, y un programa debería escribirse para ser comprendido en la medida de lo posible.

Cuando el intérprete Lisp primero empieza evaluando las expresiones de la función, el valor de `total` estaría asignado a cero, ya que no hemos añadido cualquier cosa a eso. Entonces la función añadiría el número de asteriscos en la primera fila al total, y entonces añade el número de asteriscos en la segunda al total, y entonces añade el número de asteriscos a la tercera fila al total, y así, hasta que no hay más filas a la izquierda para añadir.

Ambos `total` y `row-number` se usan solo dentro de la función, así ellos pueden ser declarados como variables locales con `let` y valores iniciales dados. Claramente, el valor inicial para total sería 0. El valor inicial de `row-number` sería 1, desde que se comienza con la primera fila. Esto significa que la frase `let` se parece a esto:

```
(let ((total 0)
      (row-number 1))
  body...)
```

Después de que las variables internas se declaran y se asignan a sus valores iniciales se podría empezar el bucle `while`. La expresión que sirve como el test devolvería un valor de `t` para la verdad tan grande como el `row-number` que es menor o igual al `number-of-rows`. (La expresión devuelve cierto solo si el número de fila es menor que el número de filas en el triángulo, la última fila nunca será añadida al total; aquí el número de fila tiene que ser menor o igual el número de filas.))

Lisp provee la función `<=` que devuelve cierto si el valor de su primer argumento es menor o igual al valor de su segundo argumento y falso de otro modo. Así la expresión que el `while` evaluará como si su test se vería como esto:

```
(<= row-number number-of-rows)
```

El número de asteriscos puede encontrarse repetidamente añadiendo el número de asteriscos en una fila al total ya encontrado. Puesto que el número de asteriscos en la fila es igual al número de la fila, el total puede encontrarse añadiendo el número de filas al total. (Claramente, en una situación más compleja, el número de asteriscos en la fila podría ser relacionada al número de la fila en un camino más complicado; si este fuera el caso, el número de fila sería reemplazado por la expresión apropiada.)

```
(setq total (+ total row-number))
```

Lo que esto hace es asignar el nuevo valor de `total` a ser igual a la suma de añadiendo el número de asteriscos en la fila al total previo.

Después de configurar el valor de `total`, las condiciones deben ser establecidas para la siguiente repetición del bucle, si hay alguna. Esto se hace incrementando el valor de la variable `row-number`, que sirve como un contador. Después que la variable `row-number` ha sido incrementada, el true-or-false-test al principio del bucle `while` chequea si su valor es todavía menor o igual al valor del `number-of-rows` y si eso es, añade el nuevo valor de la variable `row-number` al `total` de la repetición del bucle.

La función construida en Emacs Lisp `1+` añade 1 a un número, así la variable `row-number` puede ser incrementado con esta expresión:

```
(setq row-number (1+ row-number))
```

Poniendo la definición de la función junta

Nosotros hemos creado las partes para la definición de la función; ahora necesitamos ponerlas juntas.

Primero, los contenidos de la expresión `while`:

```
(while (<= row-number number-of-rows)    ; true-or-false-test
  (setq total (+ total row-number))
  (setq row-number (1+ row-number)))     ; incremento
```

Tener la expresión `let` de varlist, se acerca a completar el cuerpo de la definición de función. Sin embargo, eso requiere un elemento final, la necesidad para la que es alguna cosa pequeña.

El toque final es emplazar la variable `total` en una línea por sí misma después de la expresión `while`. De otro modo, el valor devuelto por la función completa es el valor de la última expresión que es evaluada en el cuerpo del `let`, y este es el valor devuelto por el `while` que es siempre `nil`.

Esto puede no ser evidente a primera vista. Eso casi se ve como si la expresión de incremento es la última expresión de la función completa. Pero esta expresión es parte del cuerpo del `while`; eso es el último elemento de la lista que empieza con el símbolo `while`. Más allá, el bucle `while` completo es una lista con el cuerpo del `let`.

En línea (*outline*), la función se parece a esto:

```
(defun name-of-function (argument-list)
  "documentation..."
  (let (varlist)
    (while (true-or-false-test)
      body-of-while... )
    ... ))                       ; Necesita la expresión final aquí.
```

El resultado de evaluar el `let` es que lo que está yendo para devolver el `defun` desde el `let` que no está embebido con cualquier lista que contiene, excepto para la `defun` como un todo. Sin embargo, si el `while` es el último elemento de la expresión `let`, la función siempre devolverá `nil`. ¡Esto no es lo que quiero! En vez de eso, lo que queremos es el valor de la variable `total`. Eso devuelve simplemente emplazando el símbolo como el último elemento de la lista empezando con `let`. Eso se evalúa después de los elementos precedentes de la lista evaluada, que significa que eso se evaluó después de haber sido asignado el valor correcto para el total.

Eso puede ser fácil de ver imprimiendo la lista empezando con `let` todo en una línea. Este formato hace evidente que las expresiones *varlist* y `while` son el segundo el tercer elementos de la lista empezando con `let`, y el `total` es el último elemento:

```
(let (varlist) (while (true-or-false-test)
body-of-while... ) total)
```

Poniendo cualquier cosa junta, la definición de función `triangle` se parece a esto:

```
(defun triangle (number-of-rows)     ; Versión con
                                     ;  contador de incremento.
  "Añade el número de asteriscos en un triángulo.
La primera fila tiene un asterisco, la segunda fila dos asteriscos,
la tercera fila tres asteriscos, y así.
El argumento es NUMBER-OF-ROWS."
  (let ((total 0)
        (row-number 1))
    (while (<= row-number number-of-rows)
      (setq total (+ total row-number))
      (setq row-number (1+ row-number)))
    total))
```

Después de haber instalado `triangle` para evaluar la función, se puede probar. Aquí hay dos ejemplos:

```
(triangle 4)

(triangle 7)
```

La suma del primero de cuatro números es 10 y la suma de los primeros siete números es 28.

11.1.4 Bucle que decrementa

Otro camino común para escribir un bucle `while` es escribir el test así que determina si un contador es mayor que cero. Así tan largo es el contador mayor que cero, el bucle se repite. Pero cuando el contador es igual o menor que cero, el bucle se para. Para este trabajo, el contador tiene que empezar mayor que cero y entonces se hace más pequeño y pequeño por una forma que es evaluada repetidamente.

El test será una expresión tal como `(> counter 0)` que devuelve `t` para cierto si el valor de `counter` es mayor que cero, y `nil` para falso si el valor de `counter` es igual a o menor que cero. La expresión hace que el número menor y menor puede ser un simple `setq` tal como `(setq counter (1- counter))`, donde `1-` es una función construida en Emacs Lisp que sustrae 1 de su argumento.

La plantilla para decrementar el bucle `while` se ve así:

```
(while (> counter 0)                    ; true-or-false-test
  body...
  (setq counter (1- counter)))          ; decremento
```

Ejemplo con el contador que se decrementa

Para ilustrar un bucle con un contador de decremento, reescribirá la función `triangle` así como el contador se decrementa a cero.

Esto es lo inverso de la versión temprana de la función. En este caso, para encontrar cuántos asteriscos son necesarios para crear un triángulo con 3 filas, añade el número de asteriscos en la tercera fila, 3, para el número en la fila precedente, 2, y entonces añade el total de estas dos filas a la fila que lo precede, que es 1.

Más allá, para encontrar el número de asteriscos en un triángulo con 7 filas, añade el número de asteriscos en la fila siete, 7, al número en la fila precedente, que

es 6, y entonces añade el total de estas dos filas a la fila esta que lo precede, que es 5, y así. Como en el ejemplo previo, cada adición solo involucra la adición de dos números, el total de las filas ya se añadió y el número de asteriscos en la fila que está siendo añadida al total. Este proceso de añadir dos números se repite de nuevo y de nuevo hasta que no haya más asteriscos que añadir.

Sabemos con cuántos asteriscos empezar: el número de asteriscos en la última fila es igual al número de filas. Si el triángulo tiene siete filas, el número de asteriscos en la última fila es 7. Más allá, sabemos cuántos asteriscos están en la fila precedente: eso es uno menos que el número en la fila.

Las partes de la definición de función

Empezamos con tres variables: el número total de filas en el triángulo; el número de asteriscos en una fila; y el número total de asteriscos, que es lo que queremos calcular. Estas variables pueden llamarse `number-of-rows`, `number-of-pebbles-in-row`, y `total`, respectivamente.

Ambos `total` y `number-of-pebbles-in-row` se usan solo dentro de la función y se declaran con `let`. El valor inicial de `total` sería cero. Sin embargo, el valor inicial de `number-of-pebbles-in-row` sería igual al número de filas en el triángulo, desde la adición empezará con la fila más larga.

Esto significa que el principio de la expresión `let` se verá como esto:

```
(let ((total 0)
      (number-of-pebbles-in-row number-of-rows))
  body...)
```

El número total de asteriscos puede encontrarse repetidamente añadiendo el número de asteriscos en una fila para el total ya encontrado, que, se evalúa repetidamente en la siguiente expresión:

```
(setq total (+ total number-of-pebbles-in-row))
```

Después el `number-of-pebbles-in-row` se añade al `total`, el `number-of-pebbles-in-row` sería decrementado por uno, desde que la siguiente vez el bucle repite, la fila precedente será añadida al total.

El número de asteriscos en una fila precedente es uno menos que el número de asteriscos en una fila, así la función Emacs Lisp construida `1-` puede usarse para computar el número de asteriscos de la fila precedente. Esto puede ser hecho con la siguiente expresión:

```
(setq number-of-pebbles-in-row
      (1- number-of-pebbles-in-row))
```

Finalmente, sabemos que el bucle `while` pararía creando repetidas adiciones cuando no hay asteriscos en una fila. Así el test para el bucle `while` es simple:

```
(while (> number-of-pebbles-in-row 0)
```

Poniendo la definición de la función junta

Se pueden poner estas expresiones juntas para crear una definición de función que funcione. Sin embargo, al examinarlas, encontraremos que una de la variables locales ¡es innecesaria!

La definición de función se ve como esto:

```
;;; Primero la versión substractiva.
(defun triangle (number-of-rows)
  "Add up the number of pebbles in a triangle."
  (let ((total 0)
        (number-of-pebbles-in-row number-of-rows))
    (while (> number-of-pebbles-in-row 0)
      (setq total (+ total number-of-pebbles-in-row))
      (setq number-of-pebbles-in-row
            (1- number-of-pebbles-in-row)))
    total))
```

Como se dijo, esta función funciona.

Sin embargo, no se necesita `number-of-pebbles-in-row`.

Cuando la función `triangle` se evalúa, el símbolo `number-of-rows` será asociado al número, dando un valor inicial. Este número puede ser cambiado en el cuerpo de la función si hubiera una variable local, sin miedo de que tal cambio se efectuará el valor de la variable fuera de la función. Esto es una característica muy útil de Lisp; eso significa que la variable `number-of-rows` puede ser usada en cualquier lugar en la función donde `number-of-pebbles-in-row` se usa.

Aquí hay una segunda versión de la función escrita un poco más limpiamente:

```
(defun triangle (number)                ; Second version.
  "Return sum of numbers 1 through NUMBER inclusive."
  (let ((total 0))
    (while (> number 0)
      (setq total (+ total number))
      (setq number (1- number)))
    total))
```

En breve, un bucle `while` apropiadamente escrito consistirá de tres partes:

1. Un test que devuelva falso después de que el bucle ha repetido por sí mismo el número de veces correcto.
2. Una expresión de la evaluación de que devolverá el valor deseado después de ser repetidamente evaluado.
3. Una expresión para cambiar el valor pasado al true-or-false-test así el test devuelve falso después de que el bucle se ha repetido por sí mismo el número de veces correcto.

11.2 Ahorra tiempo: `dolist` y `dotimes`

Además de `while`, tanto `dolist` como `dotimes` proveen un bucle. Algunas veces estos son rápidos para escribir el bucle equivalente `while`. Ambos son macros Lisp. (Véase Sección “Macros” in *El Manual de Referencia GNU Emacs Lisp*.)

`dolist` funciona como un bucle `while` con ‘CDRs que bajan la lista’: `dolist` automáticamente ordena la lista cada vez que la lista hace bucles — toma la CDR de la lista — y asocia el CAR de cada versión ordenada de la lista al primero de sus argumentos.

`dotimes` repite el bucle un número específico de veces: tu especificas el número.

La macro `dolist`

Supón, por ejemplo, que quieres invertir una lista, así que "primero", "segundo", "tercero" llega a ser "tercero", "segundo", "primero".

En la práctica, usarías la función `reverse`, como esta:

```
(setq animales '(gacela jirafa leon tigre))

(reverse animales)
```

Aquí se ve cómo se podría invertir la lista usando un bucle `while`:

```
(setq animales '(gacela jirafa leon tigre))

(defun reverse-list-with-while (list)
  "Usando while, invierte el orden de LIST."
  (let (value)  ; asegura que la lista comienza vacía
    (while list
      (setq value (cons (car list) value))
      (setq list (cdr list)))
    value))

(reverse-list-with-while animales)
```

Y aquí se ve cómo podría usarse la macro `dolist`:

```
(setq animales '(gacela jirafa leon tigre))

(defun reverse-list-with-dolist (list)
  "Usando dolist, reverse, la orden de la LISTA."
  (let (value)  ; asegura que la lista empieza vacía
    (dolist (element list value)
      (setq value (cons element value)))))

(reverse-list-with-dolist animals)
```

En Info, se puede localizar su cursor después de cerrar paréntesis de cada expresión y escribir *`C-x C-e`*; en cada caso, se vería

```
(tigre leon jirafa gacela)
```

en el área echo.

Para este ejemplo, la función `reverse` existente es obviamente la mejor. El bucle `while` es solo como nuestro primer ejemplo (véase Sección 11.1.1 "Un bucle `while` y una lista", página 112). El `while` primero chequea si la lista tiene elementos; si es así, eso construye una nueva lista añadiendo el primer elemento de la lista a la lista existente (que en la primera iteración del bucle es `nil`). Puesto que el segundo elemento está asignado en frente del segundo elemento, la lista es inversa.

En la expresión que usa el bucle `while`, la expresión `(setq list (cdr list))` ordena la lista, así el bucle `while` finalmente para. Además, se proporciona la expresión `cons` con un nuevo primer elemento creando una nueva lista y se ordena en cada repetición del bucle.

La expresión `dolist` hace lo mismo que la expresión `while`, excepto que la macro `dolist` hace algo del trabajo que se tiene que hacer cuando se escribe una expresión `while`.

Al igual que el bucle `while`, tenemos el bucle `dolist`. Lo que es diferente es que automáticamente ordena la lista cada vez que se repite — eso es 'recorrer los CDRs de la lista' en sí — y eso automáticamente asocia el CAR de cada versión ordenada de la lista al primero de sus argumentos.

En el ejemplo, el CAR de cada versión ordenada de la lista se refiere a usar el símbolo '`element`', la lista en sí se llama '`list`', y el valor devuelto se llama '`value`'. El resto de la expresión `dolist` es el cuerpo.

La expresión `dolist` asocia el CAR de cada versión ordenada de la lista al `element` y entonces evalúa el cuerpo de la expresión y repite el bucle. El resultado es devuelto en `value`.

La macro `dotimes`

La macro `dotimes` es similar a `dolist`, excepto que el bucle se repite un número específico de veces.

El primer argumento `dotimes` se asigna a los números 0, 1, 2 y así vuelve al bucle, y el valor del tercer argumento se devuelve. Se necesita proveer el valor del segundo argumento, que es cuántas veces la macro hace el bucle.

Por ejemplo, lo siguiente asocia los números desde 0 a, pero no incluyendo, el número 3 al primer argumento, *número*, y entonces construye una lista de los tres números. (El primer número es 0, el segundo número es 1, y el tercer número es 2; esto crea un total de tres números en todo, empezando con cero como el primer número.)

```
(let (value)      ; de otro modo un valor es una variable vacía
  (dotimes (number 3 value)
    (setq value (cons number value))))

⇒ (2 1 0)
```

`dotimes` devuelve `value`, así el camino para usar `dotimes` es para operar en alguna expresión el número de veces *number* y entonces devolver el resultado, como una lista o un átomo.

Aquí hay un ejemplo de una `defun` que usa `dotimes` para añadir el número de asteriscos en un triángulo.

```
(defun triangle-using-dotimes (number-of-rows)
  "Usando dotimes, añade el número de asteriscos en un triángulo."
(let ((total 0))  ; de otro modo un total es una variable vacía
  (dotimes (number number-of-rows total)
    (setq total (+ total (1+ number))))))

(triangle-using-dotimes 4)
```

11.3 Recursión

Una función recursiva contiene código que hace que el intérprete Lisp llame a un programa que ejecute el código en sí, pero con argumentos ligeramente diferentes. El código ejecuta exactamente lo mismo porque eso tiene el mismo nombre. Sin embargo, incluso aunque el programa tenga el mismo nombre, no es la misma entidad. Eso es diferente. En la jerga, se dice es una 'instancia' diferente.

Finalmente, si el programa es escrito correctamente, los 'argumentos ligeramente diferentes' llegan a ser suficientemente diferentes puesto los primeros argumentos de la instancia final se pararán.

11.3.1 Construyendo robots: Extendiendo la metáfora

Algunas veces es útil pensar en un programa en ejecución como un robot que hace un trabajo. Haciendo su trabajo, una función recursiva llama a un segundo robot para que le ayude. El segundo robot es idéntico al primero en cada paso, excepto que el segundo robot ayuda al primero y ha sido pasado diferentes argumentos que el primero.

En una función recursiva, el segundo robot puede llamar a un tercero; y el tercero puede llamar a un cuarto, y así. Cada una de estos es una entidad diferente; pero todos son clones.

Desde que cada robot tiene instrucciones ligeramente diferentes — los argumentos diferirán desde un robot al siguiente — el último robot conocería cuando pare.

Permite expandir la metáfora en el que un programa de ordenador es un robot.

Una definición de función provee impresiones para un robot. Cuando se instala una definición de función, que es, cuando se evalúa una forma especial `defun`, se instala el equipamiento para construir robots. Eso es como si tu estuvieras en una fábrica, configurando una línea de ensamblaje. Los robots con el mismo nombre son construidos de acuerdo a las mismas impresiones. Así ellos tienen, como estaban, el mismo 'número de modelo', pero un diferente 'número de serie'.

Nosotros con frecuencia decimos que una función recursiva 'se llama así misma'. Esto significa que las instrucciones en una función recursiva causa el intérprete de Lisp para ejecutar una función diferente que tiene el mismo nombre y hace el mismo trabajo como el primer, pero con diferentes argumentos.

Eso es importante que los argumentos difieren desde una instancia a la siguiente; de otro modo, el proceso nunca parará.

11.3.2 Las partes de una definición recursiva

Una función recursiva típicamente contiene una expresión condicional que tiene tres partes:

1. Un true-or-false-test que determina si la función se llama de nuevo, aquí se llama el *do-again-test*.
2. El nombre de la función. Cuando este nombre se llama, es una nueva instancia de la función — un nuevo robot, así — se crea y se dice qué hacer.
3. Una expresión que devuelve un valor diferente cada vez que la función se llama, aquí llamada la *next-step-expression*. Consecuentemente, el argumento (o argumentos) pasados a la nueva instancia de la función serán diferentes desde que se pasa a la instancia previa. Esto causa la expresión condicional, que el *do-again-test*, para devuelva falso después del número correcto de repeticiones.

Las funciones recursivas pueden ser más simples que cualquier otro tipo de funciones. De manera profunda, cuando la gente empieza a usarlas, con frecuencia se

miran así misteriosamente de manera tan simple como incompresible. Como montar en bicicleta, leer una función recursiva es duro al principio, pero después es simple.

Hay varios patrones recursivos diferentes. Un patrón muy simple se parece a:

```
(defun name-of-recursive-function (argument-list)
  "documentation..."
  (if do-again-test
    body...
    (name-of-recursive-function
         next-step-expression)))
```

Cada vez que una función recursiva es evaluada, una nueva instancia se crea y se dice qué hacer. Los argumentos le dicen a la instancia qué hacer.

Un argumento se empareja al valor de la next-step-expresion. Cada instancia se ejecuta con un valor diferente de la next-step-expression.

El valor en la next-step-expression es usado en la do-again-test.

El valor devuelto por la next-step-expression es pasada a las nuevas instancias de la función, que lo evalúa (o alguna transformación de eso) para determinar si continuar o parar. El next-step-expression está diseñado así que el do-again-test devuelve falso cuando la función no sería largamente repetida.

El do-again-test es algunas veces llamado la *condición de parar*, puesto que sirve para parar las repeticiones cuando devuelve falso.

11.3.3 Recursión con una lista

El ejemplo de un bucle `while` que imprimió los elementos de una lista de números puede ser escrito recursivamente. Aquí está el código, incluyendo una expresión para asignar el valor de la variable `animales` a una lista.

Si estás leyendo esto en el Info de Emacs, se puede evaluar esta expresión directamente en Info. De otro modo, se debe copiar el ejemplo al búffer `*scratch*` y evalúa cada expresión aquí. Usa `C-u C-x C-e` para evaluar la expresión `(print-elements-recursively animals)` así que los resultado se imprimen en el búffer; de otro modo el intérprete Lisp intentará presionar los resultados dentro de una línea del área echo.

También, posiciona tu cursor inmediatamente después del último paréntesis que cierra la función `print-elements-recursively`, antes del comentario. De otro modo, el intérprete Lisp intentará evaluar el comentario.

```
(setq animales '(gacela jirafa leon tigre))

(defun print-elements-recursively (list)
  "Imprime cada elemento de LISTA en una línea de
  sí.
Usa recursión."
  (when list                            ; do-again-test
        (print (car list))              ; body
        (print-elements-recursively     ; recursive call
         (cdr list))))                  ; next-step-expression

(print-elements-recursively animales)
```

La función `print-elements-recursively` primero chequea si hay cualquier contenido en la lista; si lo hay, la función imprime el primer elemento de la lista, el

CAR de la lista. Entonces la función se 'invoca en sí', pero da a sí mismo como su argumento, no la lista completa, pero el segundo y subsiguientes elementos de la lista, el CDR de la lista.

Pon otro camino, si la lista no está vacía, la función invoca otra instancia de código que es similar al código inicial, pero es un hilo diferente de ejecución, con diferentes argumentos a la primera instancia.

Veamos una manera más, si la lista no está vacía, el primer robot ensambla un segundo robot que cuenta qué hacer; el segundo robot es un individuo diferente desde el principio, pero es el mismo modelo.

Cuando la segunda evaluación ocurre, la expresión `when` se evalúa y si es verdad, se imprime el primer elemento de la lista que recibe como su argumento (que es el segundo elemento de la lista original). Entonces la función 'llamarse a sí mismo' con la CDR del CDR de la lista original.

Note que aunque nosotros decimos que la función 'se llama a sí misma', lo que significa es que el intérprete Lisp ensambla e instruye una nueva instancia del programa. La nueva instancia es un clon del primero, pero es un individuo separado.

Cada vez que la función 'se invoca a sí misma', se invoca a sí misma en una versión de la lista original. Eso crea una nueva instancia que funciona como una lista ordenada.

Finalmente, la función se invoca a sí misma en una lista vacía. Eso crea una nueva instancia cuyo argumento es `nil`. La expresión condicional chequea el valor de `lista`. Desde el valor de `lista` a `nil`, la expresión `when` devuelve falso así la then-part no está evaluada. La función es como un todo que entonces devuelve `nil`.

Cuando se evalúa la expresión `(print-elements-recursively animals)` en el búffer `*scratch*`, se verá este resultado:

```
gacela

jirafa

leon

tigre
nil
```

11.3.4 Recursión en lugar de un contador

La función `triangle` describe en una sección previa si puede ser escrita recursivamente. Se ve así:

```
(defun triangle-recursively (number)
  "Return the sum of the numbers 1 through NUMBER inclusive.
Uses recursion."
  (if (= number 1)                    ; do-again-test
      1                               ; then-part
    (+ number                         ; else-part
       (triangle-recursively          ; recursive call
        (1- number)))))               ; next-step-expression

(triangle-recursively 7)
```

Se puede instalar esta función evaluando y entonces se intenta evaluar `(triangle-recursively 7)`. (Recuerda poner tu cursor inmediatamente después de los últimos paréntesis de la definición de la función, antes del comentario.) La función se evalúa a 28.

Para comprender cómo funciona la función, hay que considerar qué ocurre en los varios casos cuando la función se le pasa 1, 2, 3, o 4 como el valor a su argumento.

Primero, veamos qué ocurre si el valor del argumento es 1

La función tiene una expresión `if` después de la cadena de documentación. Esto chequea si el valor de `number` es igual a 1; si es así, Emacs evalúa la then-part de la expresión `if`, que devuelve el número 1 como el valor de la función. (Un triángulo con una fila tiene un asterisco dentro.)

Supón, sin embargo, que el valor del argumento es 2. En este caso, Emacs evalúa la parte else de la expresión `if`.

La parte else consiste de una adición, la llamada recursiva para `triangle-recursively` y una acción de decremento; y se ve así:

```
(+ number (triangle-recursively (1- number)))
```

Cuando Emacs evalúa esta expresión, la expresión interna es evaluada primero; entonces las otras partes en secuencia. Aquí están los pasos en detalle:

Paso 1 *Evalúa la expresión interna.*
La expresión interna es `(1- number)` así Emacs decrementa el valor de `number` desde 2 a 1.

Paso 2 *Evalúa la función* `triangle-recursively`.
El intérprete Lisp crea una instancia individual de `triangle-recursively`. Eso no importa que esta función está contenida con sí misma. Emacs pasa el resultado Paso 1 como el argumento usado por esta instancia de la función `triangle-recursively`

En este caso, Emacs evalúa `triangle-recusrively` con un argumento de 1. Esto significa que esta evaluación de `triangle-recursively` devuelve 1.

Paso 3 *Evalúa el valor de* `number`.
La variable `number` es el segundo elemento de la lista que empieza con `+`; su valor es 2.

Paso 4 *Evalúa la expresión* `+`.
La expresión `+` recibe dos argumentos, el primero desde la evaluación de `number` (Paso 3) y el segundo desde la evaluación de `triangle-recursively` (Paso 2).

El resultado de la adición es la suma de 2 `+` 1, y el número 3 es devuelto, que es correcto. Un triángulo con dos filas tiene tres asteriscos ahí.

Un argumento de 3 o 4

Supón que `triangle-recursively` es llamado con un argumento de 3.

Paso 1 *Evalúa la do-again-test.*
La expresión `if` se evalúa primero. Esto es el test do-again y devuelve falso, así la parte else de la expresión `if` es evaluada. (Note que en este ejemplo, el do-again-test causa la función para llamarse a sí misma cuando eso se chequea como falso, no cuando eso se chequea como verdadero.)

Paso 2 *Evalúa la expresión propia de la parte else.*
La expresión propia de la parte es evaluada, decrementa 3 a 2. Esta la next-step-expression.

Paso 3 *Evalúa la función* `triangle-recursively`.
El número 2 es pasado a la función `triangle-recursively`.

Nosotros ya sabemos qué ocurre cuando Emacs evalúa `triangle-recursively` con un argumento de 2. Después de ir a través de la secuencia de acciones descritas temprano, eso devuelve un valor de 3. Así que es lo que ocurrirá aquí.

Paso 4 *Evalúa la adición.*
3 será pasado como un argumento para la adición y será añadido al número con el que la función se llamó, que es 3.

El valor devuelto por la función como un todo será 6.

Ahora que sabemos qué ocurrirá cuando `triangle-recursively` llama con un argumento 3, es evidente lo que ocurrirá si se llamado con el argumento 4:

En la llamada recursiva, la evaluación de

```
(triangle-recursively (1- 4))
```

devuelve el valor de evaluar

```
(triangle-recursively 3)
```

que es 6 este valor será añadido a 4 por la adición en la tercera línea.

El valor devuelto por la función como un todo será 10.

Cada vez que `triangle-recursively` se evalúa, se interpreta una versión de sí misma — una instancia diferente en sí — con un pequeño argumento, hasta que el argumento es suficientemente pequeño así que no se evalúa por sí.

Note que este particular diseño para una función recursiva requiere que las operaciones sean diferidas.

Antes de que `(triangle-recursively 7)` pueda calcular su respuesta, debe llamarse a `(triangle-recursively 6)`; y antes a `(triangle-recursively 5)`; y así. Esto es decir, que el cálculo de `(triangle-recursively 7)` a crear debe ser diferido hasta que `(triangle-recursively 6)` haga su cálculo; y `(triangle-recursively 5)` lo complete; y así.

Si cada una de estas instancias de `triangle-recursively` son pensadas como diferentes robots, el primer robot debe esperar por el segundo para completar su trabajo, que debe esperar hasta los terceros completos, y así.

Hay un camino alrededor de este tipo de espera, que se discutirá en Sección 11.3.7 “Recursión sin Defermentos.”, página 131

11.3.5 Ejemplo de recursión usando `cond`

La versión de `triangle-recursively` antes descrita se escribió con la forma especial `if`. Eso puede también ser escrita usando otra forma especial llamada `cond`. El nombre de la forma especial `cond` es una abreviación de la palabra '`conditional`'.

Aunque la forma especial `cond` no se usa con frecuencia en las fuentes de Emacs como `if`, se usa con suficiente frecuencia para justificarse explicando.

La plantilla para una expresión `cond` se parece a:

```
(cond
 body...)
```

donde el *body* es una serie de listas.

Escrito de manera más completa, la plantilla se parece a esto:

```
(cond
 (first-true-or-false-test first-consequent)
 (second-true-or-false-test second-consequent)
 (third-true-or-false-test third-consequent)
  ...)
```

Cuando el intérprete Lisp evalúa la expresión `cond`, evalúa el primer elemento (el CAR o true-or-false-test) de la primera expresión en una serie de expresiones con el cuerpo del `cond`.

Si el true-or-false-test devuelve `nil` el resto de esta expresión, el consecuente, se escapa y el true-or-false-test de la siguiente expresión se evalúa. Cuando una expresión encuentra un true-or-false-test cuyo valor no es `nil`, el consecuente de esta expresión se evalúa. El consecuente puede ser una o más expresiones. Si el consecuente consiste de más de una expresión, las expresiones son evaluadas en secuencia y el valor del último es devuelto. Si la expresión no tiene un consecuente, se devuelve el valor del true-or-false-test.

Si ninguno del test true-or-false-tests es cierto, la expresión `cond` devuelve `nil`.

Escrito usando `cond`, la función `triangle` se parece a esto:

```
(defun triangle-using-cond (number)
  (cond ((<= number 0) 0)
        ((= number 1) 1)
        ((> number 1)
         (+ number (triangle-using-cond (1- number))))))
```

En este ejemplo, el `cond` devuelve 0 si el número es menor o igual a 0, eso devuelve 1 si el número es 1 y eso evalúa `(+ number (triangle-using-cond (1- number)))` si el número es más grandes que 1.

11.3.6 Patrones recursivos

Aquí hay tres patrones recursivos. Cada uno involucra una lista. La recursión no se necesita para involucrar listas, pero Lisp se diseña para listas y esto provee un sentido de sus capacidades primarias.

Patrón recursivo: *every*

En el patrón recursivo `every`, se desarrolla una acción por cada elemento de una lista.

El patrón básico es:

- Si una lista es vacía, devuelve `nil`.
- Si no, actúa al principio de la lista (el CAR de la lista)
  - a través de una llamada recursiva por la función en el resto (el CDR) de la lista,
  - y, opcionalmente, combina el elemento sobre el que actúa, usando `cons`, con los resultados de actuar en el resto.

Aquí está el ejemplo:

```
(defun square-each (numbers-list)
  "Square each of a NUMBERS LIST, recursively."
  (if (not numbers-list)                ; do-again-test
      nil
    (cons
     (* (car numbers-list) (car numbers-list))
     (square-each (cdr numbers-list))))) ; next-step-expression

(square-each '(1 2 3))
    ⇒ (1 4 9)
```

Si `numbers-list` está vacío, no hagas nada. Pero si eso tiene contenido, construye una lista combinando el cuadrado del primer número en la lista con el resultado de la llamada recursiva.

(El ejemplo sigue el patrón exactamente: `nil` es devuelto si la lista de números es vacía. En la práctica, se escribiría el condicional, así se ejecuta la acción cuando la lista de números no es vacía.)

La función `print-elements-recursively` (véase Sección 11.3.3 "Recursión con una Lista", página 124) es otro ejemplo de un patrón `every`, excepto en este caso, en vez de traer los resultados juntos usando `cons`, se imprime cada elemento de salida.

La función `print-elements-recursively` se parece a esto:

```
(setq animales '(gacela jirafa leon tigre))

(defun print-elements-recursively (list)
  "Imprime cada elemento de LISTA en una línea de
  sí.
Usa recursión."
  (when list                            ; do-again-test
        (print (car list))              ; body
        (print-elements-recursively     ; recursive call
         (cdr list))))                  ; next-step-expression

(print-elements-recursively animales)
```

El patrón para `print-elements-recursively` es:

- Cuando la lista está vacía, no hacer nada.
- Pero cuando la lista tiene al menos un elemento,
  - actúa al principio de la lista (el CAR de la lista),
  - y crea una llamada recursiva en el resto (el CDR de la lista.

Patrón recursivo: *accumulate*

Otro patrón recursivo es llamado el patrón `accumulate`. En el patrón recursivo `accumulate`, una acción se realiza en cada elemento de una lista y el resultado de esta acción se acumula con los resultados de desarrollar la acción en otros elementos.

Esto es como 'cada' patrón usando `cons`, excepto que este `cons` no se esté usando, pero algún otro combine.

El patrón es:

- Si una lista está vacía, devuelve cero o alguna otra constante.
- Lo demás, actúa al principio de la lista (el CAR de la lista),
  - y combinar este elemento que actúa, usando `+` o alguna otra función de combinación, con
  - una llamada recursiva por la función en el resto (el CDR) de la lista.

Aquí hay un ejemplo:

```
(defun add-elements (numbers-list)
  "Añade los elementos de NUMBERS-LIST juntos."
  (if (not numbers-list)
      0
    (+ (car numbers-list) (add-elements (cdr numbers-list)))))

(add-elements '(1 2 3 4))
    ⇒ 10
```

Véase Sección 14.9.2 "Creando una lista de ficheros", página 174, para un ejemplo del patrón acumulado.

Patrón recursivo: *keep*

Un tercer patrón es llamado el patrón `keep`. En el patrón recursivo `keep`, cada elemento de una lista es testeado; el elemento actúa y los resultados son guardados solo si el elemento encuentra un criterio.

De nuevo, esto se parece a 'cada' patrón, excepto si el elemento se escapa a menos que encuentre un criterio.

El patrón tiene tres partes:

- Si una lista es vacía, devuelve `nil`.
- Lo demás, si el principio de la lista (el CAR de la lista) pasa un test
  - actua en este elemento y lo combina, usando `cons` con
  - una llamada recursiva por la función en el resto (el CDR de la lista.
- De otro modo, si el principio de la lista (el CAR de la lista) falla el test
  - escapa en este elemento,
  - y, recursivamente llama la función al resto (el CDR) de la lista.

Aquí hay un ejemplo que usa `cond`:

```
(defun keep-three-letter-words (word-list)
  "Guarda 3 palabras en WORD-LIST."
  (cond
   ;; Primero do-again-test: stop-condition
   ((not word-list) nil)

   ;; Segundo do-again-test: cuando actuar
   ((eq 3 (length (symbol-name (car word-list))))
    ;; combina el elemento que actúa con la llamada recursiva en la
   lista ordenada
    (cons (car word-list) (keep-three-letter-words (cdr word-list))))

   ;; Tercero do-again-test: cuando se escape el elemento;
   ;; recursivamente llama a la lista ordenada con la next-step expression
   (t (keep-three-letter-words (cdr word-list)))))

(keep-three-letter-words '(uno dos tres cuatro cinco seis))
    ⇒ (uno dos seis)
```

Eso va sin decir que no se necesita usar `nil` como el test para cuando para; y se puede, de acuerdo, combinar estos patrones.

11.3.7 Recursión sin diferir

Permita considerar de nuevo qué ocurre con la función `triangle-recursively`. Nosotros encontraremos que los cálculos se difieran hasta que todo pueda ser hecho.

Aquí está la definición de función:

```
(defun triangle-recursively (number)
  "Devuelve la suma de los números 1 a través de NUMBER inclusive
Usa recursión."
  (if (= number 1)                    ; do-again-test
      1                               ; then-part
    (+ number                         ; else-part
       (triangle-recursively          ; recursive call
        (1- number)))))               ; next-step-expression
```

¿Qué ocurre cuando se llama a esta función con un argumento de 7?

La primera instancia de la función `triangle-recursively` añade el número 7 al valor devuelto por una segunda instancia de `triangle-recursively`, una instancia que ha pasado un argumento de 6. Así, el primer cálculo es:

```
(+ 7 (triangle-recursively 6))
```

La primera instancia de `triangle-recursively` — se puede querer pensar como un pequeño robot — no puede completar su trabajo. Eso debe manejar el cálculo para `(triangle-recursively 6)` a una segunda instancia del programa, a un segundo robot. Este segundo individuo es completamente diferente desde el primero; eso es, en la jerga, una 'diferente instanciación'. O, poner otro camino, eso es un diferente robot. Eso es el mismo modelo como el primero; eso calcula números de triángulo recursivamente; pero eso tiene un número de serie diferente.

¿Y qué hace `(triangle-recursively 6)` devuelve? Eso devuelve el número 6 añadido al valor devuelto para evaluar `triangle-recursively` con un argumento de 5. Usando la metáfora del robot, eso cuestiona todavía otro robot para ayudarle.

Ahora el total es:

```
(+ 7 6 (triangle-recursively 5))
```

¿Y qué ocurre después?

```
(+ 7 6 5 (triangle-recursively 4))
```

Cada vez que `triangle-recursively` es llamado, excepto por la última vez, eso crea otra instancia del programa — otro robot — y pregunta para crear un cálculo.

Finalmente, la adición completa es de la siguiente manera:

```
(+ 7 6 5 4 3 2 1)
```

Este diseño para la función difiere el cálculo del primer paso hasta el segundo puede ser hecho, y difiere esto hasta que el tercero puede ser hecho, y así. Cada defermento significa el ordenador debe recordar que está siendo esperado dentro. Esto no es un problema cuando hay solo unos pocos pasos, como en este ejemplo. Pero eso puede ser un problema cuando hay más pasos.

11.3.8 No hay solución pospuesta

La solución al problema de operaciones pospuestas es para escribir en una manera que no posponga operaciones[2]. Esto requiere escribir a un patrón diferente, con frecuencia uno que involucra escribiendo dos definiciones de función, una función de 'inicialización' y una función 'ayuda'.

La función 'inicializactión' configura el trabajo; la función 'ayudante' hace el trabajo.

Aquí hay dos definiciones para añadir números. Son así de simple, aunque se encuentre duro de comprender.

```
(defun triangle-initialization (number)
  "Devuelve la suma de los número 1 a través de NUMBER inclusive.
Este es el componenete de 'inicialización' de una función duo que
usa recursión"
  (triangle-recursive-helper 0 0 number))
```

[2] La frase *cola recursiva* es usado para describir tal proceso, uno que usa 'espacio constante'.

```
(defun triangle-recursive-helper (sum counter number)
  "Devuelve SUM, usando COUNTER, a través de NUMBER inclusive.
Este es el componente 'helper' de unas dos funciones
que usan recursión."
  (if (> counter number)
      sum
    (triangle-recursive-helper (+ sum counter)  ; suma
                               (1+ counter)     ; contador
                               number)))        ; número
```

Instalar ambas definiciones de función por evaluarlo, entonces llama a `triangle-initialization` con 2 filas:

```
(triangle-initialization 2)
    ⇒ 3
```

La función 'inicialización' llama la primera instancia de la función 'ayudante' con tres argumentos: cero, cero, y un número que es el número de filas en el triángulo.

Los primeros dos argumentos pasaron a la función 'ayuda' son valores de inicialización. Estos valores son cambiados cuando `triangle-recursive-helper` invocan nuevas instancias.[3]

Permítase ver que ocurre cuando tenemos un triángulo que tiene una fila. (¡Este triángulo tendrá un asterisco dentro!)

`triangle-initialization` llamará su ayudante con los argumentos `0 0 1`. Esta función ejecutará el test condicional si `(> counter number)`:

```
(> 0 1)
```

y encuentra que el resultado es falso, así invocará la else-part de la claúsula `if`:

```
(triangle-recursive-helper
 (+ sum counter)  ; sum más counter ⇒ sum
 (1+ counter)     ; incrementa counter ⇒ counter
 number)          ; number parece lo mismo
```

que computará primero:

```
(triangle-recursive-helper (+ 0 0)  ; sum
                           (1+ 0)   ; counter
                           1)       ; number
```

que es:

```
(triangle-recursive-helper 0 1 1)
```

De nuevo, `(> counter number)` será falso, así de nuevo, el intérprete Lisp evaluará `triangle-recursive-helper`, creando una nueva instancia con nuevos argumentos.

[3] La jerga es medianamente confusa: `triangle-recursive-helper` usa un proceso que es iterativo en un procedimiento que es recursivo. El proceso se llama iterativo porque el ordenador necesita solo grabar los tres valores, `suma`, `contador`, y `número`: el procedimiento es recursivo porque la función 'llama a sí mismo'. Por otro lado, ambos el proceso y el procedimiento usado por `triangle-recursively` son llamados recursivos. La palabra 'recursivo' tiene diferentes significados en los dos contextos.

Esta nueva instancia será;

```
(triangle-recursive-helper
 (+ sum counter)  ; suma más contador ⇒ sum
 (1+ counter)     ; incrementar contador ⇒ contador
 number)          ; número empieza lo mismo
```

que es:

```
(triangle-recursive-helper 1 2 1)
```

En este caso, el test `(> counter number)` ¡será cierto! Así la instancia devolverá el valor de la suma, que será 1, como se espera.

Ahora, permite pasar `triangle-initialization` un argumento de 2, para encontrar cuántos asterisco hay en un triángulo con dos filas.

Esta función llama `(triangle-recursive-helper 0 0 2)`.

En fases, las instancias llamadas serán:

```
                              suma contador número
(triangle-recursive-helper 0    1       2)

(triangle-recursive-helper 1    2       2)

(triangle-recursive-helper 3    3       2)
```

Cuando la última instancia se llama, el `(> counter number)` se chequea si será cierto, así la instancia devolverá el valor de `sum`, que será 3.

Este tipo de patrón ayuda cuando estás escribiendo funciones que puede usar recursos en un ordenador.

11.4 Ejercicio de bucles

- Escribe una función similar a `triangle` en el que cada fila tiene un valor que es la raíz del número de la fila. Usa un bucle `while`.
- Escribe una función similar para `triangle` multiplica en vez de añadir los valores.
- Reescribe estas dos funciones recursivamente. Reescribe estas funciones usando `cond`.
- Escribe una función para el modo Texinfo que crea una entrada índice al principio de un párrafo para cada '`@dfn`' con el párrafo. (En un fichero Texinfo, '`@dfn`' marca una definición. El libro es escrito en Texinfo.)

  Muchas de las funciones necesitarán ser descritas en dos de los capítulos, Capítulo 8 "Cortando y almacenando texto", página 82 y Capítulo 10 "Pegando texto", página 108. Si usas `forward-paragraph` para poner la entrada índice al principio del párrafo, tendrá que usar `C-h f` (`describe-function`) para encontrar cómo conseguir que el comando vaya hacia atrás.

  Para más información, ver "Indicando Definiciones, Comandos, etc." en *Texinfo, el Formato de Documentación de GNU*.

12 Búsquedas de expresiones regulares

Las búsquedas expresiones regulares son usadas extensivamente en GNU Emacs. Las dos funciones `forward-sentence` y `forward-paragraph`, ilustran estas búsquedas bien. Usan expresiones regulares para encontrar donde mover el punto. La frase 'expresión regular' es con frecuencia escrita como 'regexp'.

Las búsquedas de expresiones regulares son descritas en Sección "Búsqueda de Expresión Regular" in *El Manual de GNU Emacs*, tan bien como en Sección "Expresiones Regulares" in *El Manual de Referencia de GNU Emacs Lisp*. Escribiendo este capítulo, estoy presumiendo que tienes al menos una intimidad con ellos. El mayor punto para recordar es que las expresiones regulares te permiten buscar patrones tan bien como para cadenas literales de caracteres. Por ejemplo, el código en `forward-sentence` busca para el patrón de posibles caracteres que podrían marcar el fin de una frase, y mueve el punto al otro lado.

Antes de mirar en el código la función `forward-sentence`, es valorable considerar que el patrón que marca el fin de una frase debe estar. El patrón se discute en la siguiente sección; siguiendo que es una descripción de la expresión regular de búsqueda, `re-search-forward`. La función `forward-sentence` es descrito en la sección siguiente. Finalmente, la función `forward-paragraph` es descrito en la última sección de este capítulo. `forward-paragraph` es una función compleja que introduce varias funcionalidades.

12.1 La expresión regular para `sentence-end`

El símbolo `sentence-end` se asocia al patrón que marca el fin de una frase. ¿Cuál sería esta expresión regular?

Claramente, una frase puede ser finalizada por un periodo, una marca de inicio de interrogación, o una marca de exclamación. Puesto que viene del inglés, solo las claúsulas que finalizan con uno de estos tres caracteres deberían ser consideradas el fin de una frase. Esto significa que el patrón incluiría el conjunto de caracteres:

```
[.?!]
```

Sin embargo, no queremos que `forward-sentence` salte a un periodo, una marca de pregunta, o una marca de exclamación, porque tal carácter podría ser usada en el medio de una frase. Un periodo, por ejemplo, se usa después de abreviaciones. Así otra información es necesaria.

De acuerdo a la convención, escribe dos espacios después de cada frase, pero solo un espacio después de un periodo, una marca de pregunta, o una marca de exclamación seguida por dos espacios es un buen indicador de un fin de frase. Sin embargo, en un fichero, los dos espacios puede en vez de ser un tabulador o el fin de una línea. Esto significa que la expresión regular incluiría estos tres ítems como alternativas.

Este grupo de alternativas se parece a esto:

```
\\($\\| \\|  \\)
       ^   ^^
      TAB  SPC
```

Aquí, '$' indica el fin de la línea, y yo he apuntado donde el tab y dos espacios están insertados en la expresión. Ambos están insertados poniendo los caracteres actuales dentro de la expresión.

Dos barras invertidas, '\\', se requiere antes de los paréntesis y barras verticales: la primera barra invertida cita la siguiente barra invertida en Emacs; y el segundo indica que el siguiente caracter, el paréntesis o la barra vertical, es especial.

También, una frase puede ser seguida por uno o más retornos de carro, como este:

```
[
]*
```

Como en los tabuladores y espacios, un retorno de carro se inserta dentro de una expresión regular insertándolo literalmente. El asterisco indica que el `RET` se repite cero o más veces.

Pero una frase no consiste solo en un periodo, una marca de pregunta o una marca de exclamación seguida por espacios apropiados: una marca de cerrar comillas o cerrar un paréntesis de algún tipo puede preceder el espacio. En realidad más de una marca o paréntesis pueden preceder el espacio. Estas requieren una expresión que se parezca a:

```
[]\"')}]*
```

En esta expresión, el primer '`]`' es el primer caracter en la expresión; el segundo caracter es '`"`', que está precedido por un '`\`' para contar Emacs el '`"`' *no* es especial. Los últimos tres caracteres son '`'`', '`)`', y '`}`'.

Todo esto sugiere que el patrón de la expresión regular para asociar el fin de una frase sería; y, profundamente, si se evalúa `sentence-end` y encuentra que se devuelve el valor siguiente:

```
sentence-end
     ⇒ "[.?!][]\"')}]*\\($\\|     \\|  \\)[
]*"
```

(Bien, no en GNU Emacs 22; esto es porque un esfuerzo para crear el proceso simple y manejar más símbolos y lenguajes. Cuando el valor de `sentence-end` es `nil`, entonces usa el valor definido por la función `sentence-end` es `nil`, entonces usa el valor definido por la función `sentence-end`. (Aquí se usa la diferencia entre un valor y una función en Emacs Lisp.) La función devuelve un valor construido desde las variables `sentence-end-base`, `sentence-end-double-space`, `sentence-end-without-period`, y `sentence-end-without-space`. La variable crítica es `sentence-end-base`; su valor global es similar a uno descrito debajo pero también contiene marcas de cita adicionales. Estas tienen diferentes grados de curvas. La variable `sentence-end-without-period`, cuando es verdad, dice a Emacs que una frase puede finalizar sin un periodo tal como texto en Thai.)

12.2 La función `re-search-forward`

La función `re-search-forward` es similar a la función `search-forward`. (Véase Sección 8.1.3 "La Función `search-forward`", página 84.)

`re-search-forward` busca una expresión regular. Si la búsqueda es exitosa, deja el punto inmediatamente después del último caracter en el objetivo. Si la búsqueda

es hacia atrás, deja el punto antes del primer caracter en el objetivo. Se puede contar `re-search-forward` para devolver `t` a cierto. (Moviendo el punto es por ello un 'efecto lateral'.)

Como `search-forward`, la función `re-search-forward` toma cuatro argumentos:

1. El primer argumento es la expresión regular que la función busca. La expresión regular será una cadena entre comillas.

2. El segundo argumento opcional limita cómo la función busca; es un emparejamiento, que se especifica como una posición en el búffer.

3. El tercer argumento opcional especifica cómo la función responde al fallo: `nil` como tercer argumento causa la función para señalar un error (e imprime un mensaje) cuando la búsqueda falla; cualquier otro valor causa devolver `nil` si la búsqueda falla y `t` si la búsqueda tiene éxito.

4. El cuarto argumento opcional es el contaje repetido. Un contaje negativo repetido causa `re-search-forward` para buscar hacia atrás.

La plantilla para `re-search-forward` se parece a esto:

```
(re-search-forward "regular-expression"
                   limit-of-search
                   what-to-do-if-search-fails
                   repeat-count)
```

El segundo, tercer, y cuarto argumentos son opcionales. Sin embargo, si se quiere pasar un valor a uno o ambos de los últimos dos argumentos, se debe también pasar un valor a todos los argumentos precedentes. De otro modo, el intérprete Lisp errará a qué argumento estás pasando el valor.

En la función `forward-sentence`, la expresión regular será el valor de la variable `sentence-end`. En forma simple, esto es:

```
"[.?!][]\"')}]*\\($\\|  \\|  \\)[
]*"
```

El límite de la búsqueda será el fin del párrafo (desde una frase no puede ir bajo un párrafo). Si la búsqueda falla, la función devuelve `nil`, y el contaje repite será provisto por el argumento para la función `forward-sentence`.

12.3 `forward-sentence`

El comando mueve el cursor hacia adelante una frase es una ilustración honesta de cómo usar búsquedas de expresiones regulares en Emacs Lisp. En realidad, la función parece más larga y más complicada de lo que es; esto es porque la función está diseñada para ir hacia atrás tan bien como hacia adelante; y, opcionalmente, a través de una frase. La función está normalmente asociada al comando `M-e`.

Aquí está la código para `forward-sentence`:

```
(defun forward-sentence (&optional arg)
  "Ve al siguiente ‘sentence-end’. Con argumento, repite.
Con argumento negativo, mueve atrás repetidamente a ‘sentence-beginning’.
La variable ‘sentence-end’ es una expresión regular que empareja el
fin de frases. También, cada párrafo asociado termina las frases
bien.
  (interactive "p")
  (or arg (setq arg 1))
  (let ((opoint (point))
        (sentence-end (sentence-end)))
    (while (< arg 0)
      (let ((pos (point))
            (par-beg (save-excursion (start-of-paragraph-text) (point))))
       (if (and (re-search-backward sentence-end par-beg t)
                (or (< (match-end 0) pos)
                    (re-search-backward sentence-end par-beg t)))
           (goto-char (match-end 0))
         (goto-char par-beg)))
      (setq arg (1+ arg)))
    (while (> arg 0)
      (let ((par-end (save-excursion (end-of-paragraph-text) (point))))
       (if (re-search-forward sentence-end par-end t)
           (skip-chars-backward " \t\n")
         (goto-char par-end)))
      (setq arg (1- arg)))
    (constrain-to-field nil opoint t)))
```

La función se ve larga a primera vista y es mejor mirar primero el esqueleto, y entonces su músculo. El camino para ver el esqueleto es mirar en las expresiones que empiezan las columnas más a la izquierda:

```
(defun forward-sentence (&optional arg)
  "documentation..."
  (interactive "p")
  (or arg (setq arg 1))
  (let ((opoint (point)) (sentence-end (sentence-end)))
    (while (< arg 0)
      (let ((pos (point))
            (par-beg (save-excursion (start-of-paragraph-text) (point))))
       rest-of-body-of-while-loop-when-going-backwards
    (while (> arg 0)
      (let ((par-end (save-excursion (end-of-paragraph-text) (point))))
       rest-of-body-of-while-loop-when-going-forwards
    handle-forms-and-equivalent
```

¡Esto parece bastante simple! La definición de la función consiste de documentación una expresión `interactive`, una expresión `or`, una expresión `let`, y bucles `while`.

Permite mirar cada una de estas partes.

Notamos que la documentación es profunda y comprensible.

La función tiene una declaración `interactive "p"`. Esto signifca que el argumento prefijo, si cualquiera es pasado a la función como su argumento. (Esto será

un número.) Si la función no pasa un argumento (eso es opcional) entonces el argumento `arg` será asociado a 1.

Cuando `forward-sentence` se llama no interactivamente sin un argumento, `arg` está asignado `nil`. La expresión `or` maneja esto. Lo que hace es dejar el valor de `arg` como eso es, pero solo si `arg` está asignado a un valor; o eso asigna el valor de `arg` a 1, en el caso de `arg` está asignado a `nil`.

Lo siguiente es un `let`. Que especifica los valores de dos variables locales `point` y `sentence-end`. El valor local de punto, desde antes de la búsqueda, es usada en la función `constrain-to-field` que maneja formularios y equivalentes. La variable `sentence-end` está asignado por la función `sentence-end`.

Los bucles `while`

Sigue dos bucles `while`. El primer `while` tiene un true-or-false-test que chequea cierto si el argumento prefijo para `forward-sentence` es un número negativo. Esto para volver hacia atrás. El cuerpo de este bucle es similar al cuerpo de la segunda cláusula `while`, pero eso no es exactamente el mismo. Se escapará este bucle `while` y concentra en el segundo bucle `while`.

El segundo bucle `while` está moviendo el punto hacia adelante. Su esqueleto se parece a esto:

```
(while (> arg 0)            ; true-or-false-test
  (let varlist
    (if (true-or-false-test)
        then-part
      else-part
    (setq arg (1- arg))))   ; while loop decrementer
```

El bucle `while` es el tipo de decremento. (Véase Sección 11.1.4 “Un Bucle con un Contador de Decremento”, página 118.) Eso tiene un true-or-false-test que chequea cierto tan largo con el contador (en este caso, la variable `arg`) es mayor que cero; y eso tiene un decremento que elimina 1 desde el valor del contador cada vez que el bucle se repite.

Si ningún argumento prefijo es dado para `forward-sentece`, que es el camino más común es usado, este bucle `while` ejecutará una vez, desde que el valor de `arg` será 1.

El cuerpo del cuerpo `while` consite de una expresión `let`, que crea y asocia una variable local, y tiene, su cuerpo, una expresión `if`.

El cuerpo del bucle `while` se parece a esto:

```
(let ((par-end
       (save-excursion (end-of-paragraph-text) (point))))
  (if (re-search-forward sentence-end par-end t)
      (skip-chars-backward " \t\n")
    (goto-char par-end)))
```

La expresión `let` crea y asocia la variable local `par-end`. Como se ve, esta variable local está diseñada para proporcionar una asociación o límite para la búsqueda de la expresión regular. Si la búsqueda falla para encontrar una frase apropiada finalizando en el párrafo, eso se parará logrando el fin del párrafo.

Pero primero, permítenos examinar cómo `par-end` se asocia a la variable del fin del párrafo. Qué ocurre es que el `let` asigna el valor de `par-end` al valor devuelto cuando el intérprete evalúa la expresión.

```
(save-excursion (end-of-paragraph-text) (point))
```

En esta expresión, `(end-of-paragraph-text)` mueve el punto al fin del párrafo, `(point)` devuelve el valor del punto, y entonces `save-excursion` restaura el punto a su posición original. De este modo, el `let` asocia `par-end` al valor devuelto por la expresión `save-excursion`, que es la posición del fin del párrafo. (La función `end-of-paragraph-text` usa `forward-paragraph`, que se discutirá pronto.)

Emacs evalúa el cuerpo del `let`, que es una expresión `if` que se parece a esto:

```
(if (re-search-forward sentence-end par-end t) ; if-part
    (skip-chars-backward " \t\n")              ; then-part
  (goto-char par-end)))                        ; else-part
```

El test `if` si su primer argumento es cierto y si así, evalúa su parte then; de otro modo, el intérprete Emacs Lisp evalúa la parte else. El true-or-false-test de la expresión `if` es la búsqueda de la expresión regular.

Puede estar mal tener que mirar como el 'trabajo real' de la función `forward-sentence` vista aquí, pero esto es un camino común de este tipo de operación traida en Lisp.

La búsqueda de expresiones regulares

La función `re-search-forward` busca el fin de la frase, que es, para el patrón definido por la expresión regular `sentence-end`. Si el patrón es encontrado — si el fin de la frase se encuentra — entonces la función `re-search-forward` hace dos cosas:

1. La función `re-search-forward` trae un efecto lateral, que es mover el punto al fin de la ocurrencia encontrada.
2. La función `re-search-forward` devuelve un valor de verdad. Esto es el valor recibido por el `if`, y significa que la búsqueda fué exitosa.

El efecto lateral, el movimiento del punto se completa antes de la función `if` y es manejado por el valor devuelto por la exitosa conclusión de la búsqueda.

Cuando la función `if` recibe el valor de verdad desde una llamada exitosa a `re-search-forward`, el `if` evalúa la parte then que es la expresión `(skip-chars-backward "\t\n")`. Esta expresión mueve atrás a través de espacios en blanco, los tabuladores o retornos de carro hasta un caracter impreso es encontrado y entonces deja el punto correcto después del caracter impreso cerrado de la frase, que es normalmente un periodo.

Por otro lado, si la función `re-search-forward` falla para encontrar un patrón marcando el fin de la frase, la función devuelve falso. Lo falso causa el `if` para evaluar su tercer argumento, que es `(goto-char par-end)`: eso mueve el punto al fin del párrafo.

(Y si el texto está en una forma o equivalente, y apunta a que no puede moverse completamente entonces la función `constrain-to-field` empieza a funcionar.)

Las búsquedas de expresiones regulares son excepcionalmente útiles y el patrón ilustrado por `re-search-forward`, en el que la búsqueda es el test de una expresión `if`, es manejable. Se verá o escribirá código incorporando este patrón con frecuencia.

12.4 `forward-paragraph`: una mina de oro de funciones

La función `forward-paragraph` mueve el punto al fin del párrafo. Eso está normalmente asociado a *`M-}`* y hace uso de un número de funciones que son importantes en sí, incluyendo `let*`, `match-beginning`, y `looking-at`.

La definición de función para `forward-paragraph` es considerablemente mayor que la definición de función para `forward-sentence` porque eso funciona como un párrafo, cada línea puede empezar con un prefijo de relleno *fill prefix*.

Un prefijo de relleno *fill prefix* consiste en una cadena de caracteres que se repite al principio de cada línea. Por ejemplo, en código Lisp, es una convención para empezar cada línea de un comentario de párrafo largo con '`;;; `'. En modo Texto, cuatro espacios en blanco crea otro prefijo de relleno *fill prefix* común, creando un párrafo indentado. (Véase Sección "Fill Prefix" in *The GNU Emacs Manual* para más información acerca de prefijos de relleno *fill prefix*.)

La existencia de un prefijo de relleno significa que además de ser capaz de encontrar el fin de un párrafo cuyas líneas empiezan más a la izquierda, la función `forward-paragraph` debe ser capaz de encontrar el fin de un párrafo cuando todas o muchas de las líneas en el búffer empiezan con el prefijo de relleno *fill prefix*.

Más allá, es algunas veces práctico ignorar un prefijo de relleno *fill prefix* que existe, especialmente cuando las líneas en blanco separen párrafos. Esto es una complicación añadida.

En vez de imprimir toda la función `forward-paragraph`, nosotros solo imprimiremos partes de la misma. ¡Lee sin preparación, la función puede estar para desanimar!

En esquema, la función se parece a esto:

```
(defun forward-paragraph (&optional arg)
  "documentation..."
  (interactive "p")
  (or arg (setq arg 1))
  (let*
      varlist
    (while (and (< arg 0) (not (bobp)))      ; backward-moving-code
      ...
    (while (and (> arg 0) (not (eobp)))      ; forward-moving-code
      ...
```

Las primeras partes de la función son rutinas: la función lista argumentos que consisten de un argumento opcional. La documentación sigue.

La letra minúscula '`p`' en la declaración `interactive` significa que el argumento prefijo se procesa, si se pasa a la función. Eso será un número, y es el contaje repetido de cuántos párrafos se moverá. La expresión `or` en la siguiente línea maneja el caso común cuando no hay argumentos que se pasan a la función, esto ocurre si la función se llama desde otro código en vez de interactivamente. Este caso se describe pronto.

(Véase Sección 12.3 "forward-sentence", página 137.) Ahora se logra el fin de la parte familiar de esta función.

La expresión `let*`

La siguiente línea de la función `forward-paragraph` empieza una expresión `let*`. Esto es tan diferente como `let`. El símbolo es `let*` no `let`.

La forma especial `let*` es como `let` excepto que Emacs asigna cada variable en secuencia, una después de otra, y las variables en la última parte de la varlist hacen uso de los valores para los que Emacs asignó variable al principio la varlist.

(Sección 4.4.3 "`save-excursion` en `append-to-buffer`", página 53.)

En la expresión `let*` en esta función, Emacs asigna un total de siete variables: `opoint`, `fill-prefix-regexp`, `parstart`, `parsep`, `sp-parstart`, `start`, y `found-start`.

La variable `parsep` aparece dos veces, primero, para borrar instancias de '^', y segundo, para manejar prefijos rellenos.

La variable `opoint` es solo el valor de `point`. Como se puede adivinar, eso se usa en una expresión `constrain-to-field`, solo como en `forward-sentence`.

La variable `fill-prefix-regexp` se asigna al valor devuelto para evaluar la siguiente lista:

```
(and fill-prefix
     (not (equal fill-prefix ""))
     (not paragraph-ignore-fill-prefix)
     (regexp-quote fill-prefix))
```

Esta es una expresión cuyo primer elemento es la forma especial `and`.

Como se aprendió antes la (véase "La función `kill-new`", página 94), la forma especial `and` evalúa cada uno de sus argumentos hasta uno de los argumentos y devuelve un valor de `nil` en el que el caso de la expresión `and` devuelve `nil`; sin embargo, si ninguno de los argumentos devuelve un valor de `nil`, el valor resultante de evaluar el último argumento es devuelto. (Puesto que tal valor no es `nil`, eso es considerado verdad en Lisp.) En otras palabras, una expresión `and` devuelve un valor de verdad solo si todos sus argumentos son verdad.

En este caso, la variable `fill-prefix-regexp` está asociado a un valor no `nil` solo si el las siguientes cuatro expresiones producen un valor true (por ej., un no `nil`) cuando son evaluados; de otro modo, `fill-prefix-regexp` está asociado a `nil`.

`fill-prefix`
: Cuando esta variable se evalúa, el valor del prefijo de relleno *fill prefix*, si cualquiera, está devuelto. Si no hay prefijo relleno, la variable devuelve `nil`.

`(not (equal fill-prefix "")`
: Esta expresión chequea si un prefijo lleno es una cadena vacía, que es, una cadena sin caracteres en eso. Una cadena vacía no es útil un prefijo relleno *fill prefix*.

`(not paragraph-ignore-fill-prefix)`
: Esta expresión devuelve `nil` si la variable `paragraph-ignore-fill-prefix` ha sido cambiado siendo asignado un valor de verdad tal como `t`.

`(regexp-quote fill-prefix)`
: Este es el último argumento para la forma especial `and`. Si todos los argumentos de `and` son verdaderos, el valor resultante de evaluar esta expresión será devuelto por la expresión `and` y asociado a la variable `fill-prefix-regexp`,

El resultado de evaluar esta expresión `and` con éxito es que `fill-prefix-regexp` se asociará al valor de `fill-prefix` como fué modificado por la función `regexp-quote`. Lo que `regexp-quote` hace es leer una cadena y devolver la expresión regular que asociará exactamente la cadena y nada más. Esto significa que `fill-prefix-regexp` será asignada a un valor que asociará el prefijo si el prefijo existe. De otro modo, la variable será asignada a `nil`.

Las dos variables locales siguientes en la expresión `let*` están diseñadas para eliminar instancias de '`^`' desde `parstart` y `parsep`, las variables locales indican que el párrafo empieza como separador de párrafo. La siguiente expresión asigna `parsep` de nuevo. Esto es manejar prefijos rellenos.

Esta es la configuración que requiere la llamada de la definición `let*` en vez de `let`. El true-or-false-test para el `if` depende de si la variable `fill-prefix-regexp` evalúa a `nil` o algún otro valor.

Si `fill-prefix-regexp` no tiene un valor, Emacs evalúa la parte else de la expresión `if` y asocia `parsep` a su valor local. (`parsep` es una expresión regular que asocia lo que los párrafos separan.)

Pero si `fill-prefix-regexp` tiene un valor, Emacs evalúa la parte then de la expresión `if` y asocia `parsep` a una expresión regular que incluye el `fill-prefix-regexp` como parte del patrón.

Específicamente, `parsep` está asignado al valor original del párrafo que separa la expresión regular concatenada con una expresión alternativa que consiste del `fill-prefix-regexp` seguido por espacios en blanco opcionales para el fin de la línea. El espacio en blanco está definido por `"[ \t]*$"`.) El '`\\|`' define esta porción del regexp como una alternativa a `parsep`.

De acuerdo a un comentario en el código, la siguiente variable local, `sp-parstart`, se usa para buscar, y entonces los dos finales, `start` y `found-start`, se asignan a `nil`.

Ahora tenemos dentro el cuerpo del `let*`. La primera parte del cuerpo del `let*` trata con el caso cuando la función es dada a un argumento negativo y consiguientemente moviéndose hacia atrás. Nosotros saldremos de esta sección yendo hacia atrás.

El bucle `while` hacia adelante

La segunda parte del cuerpo del `let*` trata con el proceso hacia adelante. Eso es un bucle `while` que se repite si el valor de `arg` es mayor que cero. En el uso más

común de la función el valor del argumento es 1, así el cuerpo del bucle `while` se evalúa exactamente una vez, y el cursor se mueve hacia adelante un párrafo.

Esta parte maneja tres situaciones: cuando el punto está entre párrafos, cuando hay un prefijo de relleno y cuando no hay prefijo de relleno *fill prefix*.

El bucle `while` se parece a esto:

```
;; yendo hacia adelante y no al fin del búffer
(while (and (> arg 0) (not (eobp)))

  ;; entre párrafos
  ;; Mueve hacia adelante a través de líneas de
  ;; separación...
  (while (and (not (eobp))
              (progn (move-to-left-margin) (not (eobp)))
              (looking-at parsep))
    (forward-line 1))
  ;;  Esto decremento el bucle
  (unless (eobp) (setq arg (1- arg)))
  ;; ... y una línea más
  (forward-line 1)

  (if fill-prefix-regexp
      ;; Hay un prefijo lleno; que sobreescribe parstart;
      ;; vamos adelante línea por línea
      (while (and (not (eobp))
                  (progn (move-to-left-margin) (not (eobp)))
                  (not (looking-at parsep))
                  (looking-at fill-prefix-regexp))
        (forward-line 1))

    ;; No hay prefijo;
    ;; vamos hacia adelante caracter por caracter
    (while (and (re-search-forward sp-parstart nil 1)
                (progn (setq start (match-beginning 0))
                       (goto-char start)
                       (not (eobp)))
                (progn (move-to-left-margin)
                       (not (looking-at parsep)))
                (or (not (looking-at parstart))
                    (and use-hard-newlines
                         (not (get-text-property (1- start) 'hard)))))
      (forward-char 1))

    ;; y si no hay prefijo y si no estamos al final
    ;; ir a lo que fué encontrado en la búsqueda de expresiones regulares
    ;; para sp-parstart
    (if (< (point) (point-max))
        (goto-char start))))
```

Se puede ver que esto es un contador de decremento `while`, usando la expresión `(setq arg (1- arg))` como lo que se decrementa. Esta expresión no está lejos desde el `while`, pero está oculta en otra macro Lisp, una macro `unless`. A menos que estemos al final del búffer — esto es lo que la función `eobp` determina; eso es una

abreviación de '`Fin del Buffer P`' — nosotros decrementamos el valor de `arg` por uno.

(Si estamos al fin del búffer, no podemos ir más hacia adelante y el siguiente bucle de la expresión `while` chequeará falso desde que el test es un `and` con `(not (eobp))`. La función `not` significa exactamente como se esperaba; eso es otro nombre de `null`, una función que devuelve cierto cuando su argumento es falso.)

De manera interesante, el bucle cuenta que no está decrementado hasta que deje el espacio entre párrafos, a menos que vuelva al fin del búffer o pare viendo el valor local del separador del párrafo.

El segundo `while` también tiene una expresión `(move-to-left-margin)`. La función es autoexplicativa. Eso está dentro de una expresión `progn` y no el último elemento de su cuerpo, así es solo invocado para su efecto lateral, que es mover el punto al margen izquierdo de la línea actual.

La función `looking-at` es tambié auto-explicativo; eso devuelve cierto si el texto después del punto asocia la expresión regular dada como su argumento.

El resto del cuerpo del bucle se ve difícil al principio, pero tiene sentido cuando se comprende.

Primero considera que ocurre si hay un prefijo de relleno *fill prefix*:

```
(if fill-prefix-regexp
    ;; Hay un prefijo lleno; que sobreescribe parstart;
    ;; vamos adelante línea por línea
    (while (and (not (eobp))
                (progn (move-to-left-margin) (not (eobp)))
                (not (looking-at parsep))
                (looking-at fill-prefix-regexp))
      (forward-line 1))
```

Esta expresión mueve el punto hacia adelante línea por línea tan lejos como que las cuatro condiciones son ciertas:

1. Punto no está al final del búffer.
2. Podemos mover al margen izquierdo del texto y no estar al fin del búffer.
3. El siguiente punto no separa párrafos.
4. El patrón que sigue el punto es la expresión regular prefija rellena.

La última condición puede ser un puzzle, hasta que recuerdes que punto fué movido al principio de la línea temprana en la función `forward-paragraph`. Esto significa que si el texto tiene el prefijo relleno, la función `looking-at` se verá.

Considera qué ocurre cuando no hay un prefijo lleno.

```
(while (and (re-search-forward sp-parstart nil 1)
            (progn (setq start (match-beginning 0))
                   (goto-char start)
                   (not (eobp)))
            (progn (move-to-left-margin)
                   (not (looking-at parsep)))
            (or (not (looking-at parstart))
                (and use-hard-newlines
                     (not (get-text-property (1- start) 'hard)))))
  (forward-char 1))
```

El bucle `while` nos tiene buscando hacia adelante para `sp-parstart`, que es la combinación de posibles espacios en blanco con un valor local del comienzo de un párrafo o de un párrafo separador. (Las últimas dos son con una expresión empezando con `(?:)` así que no están referenciadas por la función `match-beginning`.)

Las dos expresiones,

```
(setq start (match-beginning 0))
(goto-char start)
```

significa ir al comienzo del siguiente texto localizado por la expresión regular.

La expresión `(match-beginning 0)` es nueva. Eso devuelve un número especificando la posición del comienzo del texto fuese asociado a la última búsqueda.

La función `match-beginning` es usado aquí porque una característica de una búsqueda hacia adelante: una búsqueda hacia adelante, sin dignidad si eso es una búsqueda plana o una expresión regular, mueve el punto al fin del texto que es encontrado. En este caso, una búsqueda exitosa mueve el punto al fin del patrón para `sp-parstart`.

Sin embargo, se quiere poner el punto al fin del actual párrafo, no en algún lugar más. En vez de eso, desde que la búsqueda posiblemente incluye el separador del párrafo, el punto puede finalizar al principio de lo siguiente a menos que se use una expresión que incluya `match-beginning`.

Cuando un argumento de 0, `match-beginning` devuelve la posición que es el comienzo del texto asociado por la búsqueda más reciente. En este caso, la búsqueda más reciente parece `sp-parstart`. La expresión `(match-beginning 0)` devuelve la posición del comienzo de este patrón, en vez de la posición final de este patrón.

(Incidentalmente, cuando se pasa un número positivo como un argumento, la función `match-beginning` devuelve la localización de punto en el que la expresión con paréntesis en la última búsqueda a menos que la expresión con paréntesis empiece con `\(?:`. No sé porque `\(?:` aparece aquí desde que el argumento es 0.)

La última expresión cuando no hay prefijos es

```
(if (< (point) (point-max))
    (goto-char start))))
```

Esto dice que si no hay prefijo lleno y no estamos al punto final movería al principio de lo que fué encontrado por la búsqueda de la expresión regular para `sp-parstart`.

La definición completa para la función `forward-paragraph` no solo incluye código para avanzar, también código para retroceder.

Si estás leyendo esto dentro de GNU Emacs y quieres ver la función completa, se puede escribir *`C-h f`* (`describe-function`) y el nombre de la función. Esto da la documentación de función y el nombre de la librería conteniendo las fuentes de la función. Posiciona el punto a través del nombre de la librería y presionar la tecla RET; será tomado directamente a las fuentes. (¡Asegúrate de instalar las fuentes! ¡Sin eso, estarás como una persona que intenta conducir un coche con los ojos cerrados!)

12.5 Crea tu propio fichero `TAGS`

Bajo *`C-h f`* (`describe-function`), otro camino para ver la fuente de una función es escribir *`M-.`* (`find-tag`) y el nombre de la función se asigna para eso. Esto es un

buen hábito para obtenerlo. El comando *M-.* (find-tag) toma directamente a las fuentes de una función, variable, o nodo. La función depende de tablas de etiquetas para saber donde ir.

Si la función find-tag pregunta primero por el nombre de una tabla TAGS, dado el nombre de un fichero TAGS tal como /usr/local/src/emacs/src/TAGS. (La ruta exacta a tu fichero TAGS depende de cómo tu copia de Emacs fué instalada. Yo te cuento la localización que provee tanto mi C y mis fuentes de Emacs Lisp.)

Puedes también crear tu propio fichero TAGS para los directorios que faltan.

Con frecuencia se necesita construir e instalar etiquetas de tablas por uno mismo. Esas no son construidas automáticamente. Una tabla de etiquetas llama a un fichero TAGS; el nombre es letras mayúsculas.

Se puede crear un fichero TAGS llamando el programa etags que viene como parte de la distribución Emacs. Normalmente, etags está compilado e instalado cuando Emacs se construye. (etags no es una función Lisp o una parte de Emacs; eso es un programa C.)

Para crear el fichero TAGS, primero camibia el directorio en el que se quiere crear el fichero. En Emacs se puede hacer esto con el comando *M-x cd*, o visitando un fichero en el directorio, o listando el directorio etags *.el como el comando a ejecutar

```
M-x compile RET etags *.el RET
```

crear un fichero de TAGS para Emacs Lisp.

Por ejemplo, si tu tienes un número largo de ficheros en tu directorio ~/emacs, como se hace — Yo tengo 137 .el dentro, de que se carguen 12 — se puede crear un fichero TAGS para los ficheros Emacs Lisp en este directorio.

El programa etags toma en toda la consola usual 'comodines'. Por ejemplo, si tienes dos directorios para el que quieres un fichero TAGS simple, escribe etags *.el ../elisp/*.el, donde ../elisp/ es el segundo directorio:

```
M-x compile RET etags *.el ../elisp/*.el RET
```

Tipo

```
M-x compile RET etags --help RET
```

para ver una lista de las opciones aceptadas por etags tan bien como una lista de lenguajes soportados.

El programa etags maneja más de 20 lenguajes, incluyendo Emacs Lisp, Common Lisp, Scheme, C, C++, Ada, Fortran, HTML, Java, LaTeX, Pascal, Perl, Postscript, Python, TeX, Texinfo, makefiles, y la mayoría de ensambladores. El programa no cambia para especificar el lenguaje; eso reconoce el lenguaje como una entrada de fichero de acuerdo a su nombre de fichero y contenidos.

etags es muy útil cuando se escribe código por tí mismo y quiere referirse a funciones que ya se han escrito. Ahora ejecuta etags de nuevo en intervalos como se escriben nuevas funciones, así llegan a ser parte del fichero TAGS.

Si piensa que un fichero TAGS apropiado que ya existe para lo que quieres, pero no conoces donde está, se puede usar el programa locate para intentar encontrarlo.

Escribe *M-x locate RET TAGS RET* y Emacs listará para ti las rutas nombres completas de todos tus ficheros TAGS. En mi sistema, este comando lista 34 fichero

`TAGS`. Por otro lado, un sistema 'vanilla plano' que recientemente no contenía fichero `TAGS`.

Si la tabla de etiquetas que se quiere ha sido creada, se puede usar el comando `M-x visit-tags-table` para especificarlo. De otro modo, se necesitará la tabla de etiquetas por tí mismo y entonces usar `M-x visit-tags-table`.

Construyendo Etiquetas en las fuentes Emacs

Las fuentes GNU Emacs vienen con un `Makefile` que contiene un comando sofisticado `etags` que crea, recoge, y asocia tablas de etiquetas de todas las fuentes de Emacs y pone la información dentro de un fichero `TAGS` en el directorio `src/`. (El directorio `src/` está debajo del alto nivel de tu directorio Emacs.)

Para construir este fichero `TAGS`, se puede ir al alto nivel de directorio de fuentes Emacs y ejecutar el comando de compilar `make tags`:

```
M-x compile RET make tags RET
```

(El comando `make tags` trabaja bien con las fuentes de GNU Emacs, tan bien como con otros paquetes fuentes.)

Para más información, mira Sección "Tablas de Etiquetas" in *El Manual GNU Emacs*.

12.6 Revisar

Aquí hay un breve resumen de algunas funciones introducidas recientemente.

`while` Repetidamente evalúa el cuerpo de la expresión tan larga como el primer elemento del cuerpo chequea cierto. Entonces devuelve `nil`. (La expresión es evaluado solo por sus efectos laterales.)

Por ejemplo:

```
(let ((foo 2))
  (while (> foo 0)
    (insert (format "foo is %d.\n" foo))
    (setq foo (1- foo))))

     ⇒      foo is 2.
            foo is 1.
            nil
```

(La función `insert` inserta sus argumentos en el punto; la función `format` devuelve una cadena formateada desde sus argumentos el camino `message` formatea sus argumentos; `\n` produce una nueva línea.)

`re-search-forward`
Busca un patrón, y si el patrón se encuentra, mueve el punto al resto solo después de eso.

Toma cuatro argumentos, como `search-forward`:

1. Una expresión regular que especifica el patrón para buscarlo. (¡Recuerda por marcas de comillas alrededor de este argumento!)

2. Opcionalmente, el límite de la búsqueda.
3. Opcionalmente, que haces si la búsqueda falla, devuelve `nil` o un mensaje de error.
4. Opcionalmente, cuántas veces repetir la búsqueda; si negativa, la búsqueda va hacia atrás.

`let*` Asocia algunas variables localmente a valores particulares, y entonces evalúa los argumentos que permanencen, devolviendo el valor del último. Mientras se asocian las variables locales, se usan los valores locales de variables asociadas pronto, si acaso.

Por ejemplo:

```
(let* ((foo 7)
       (bar (* 3 foo)))
  (message "‘bar’ is %d." bar))
     ⇒ ‘bar’ is 21.
```

`match-beginning`
Devuelve la posición del principio del texto encontrado por la última búsqueda de la expresión regular.

`looking-at`
Devuelve `t` para verdadero si el texto después del punto se asocia al argumento, que debería ser una expresión.

`eobp` Devuelve `t` para cierto si el punto está en el fin de la parte accesible de un búffer. El fin de la parte accesible es el fin del búffer no está encogido; eso es el fin de la parte encogida si el búffer está encogido.

12.7 Ejercicios con `re-search-forward`

- Escribe una función para buscar para una expresión que detecte dos o más líneas blancas en secuencia.
- Escribe una función para buscar palabras duplicadas, tales como ‘el el’. Véase Sección “Sintaxis para Expresiones Regulares” in *El Manual de GNU Emacs*, para información de cómo escribir un regexp (una expresión regular) para asociar una cadena que es compuesto de dos mitades idénticas. Se puede disponer de varios regexps; algunos son mejores que otros. La función que se usa es descrito en un apéndice, a lo largo de varios regexps. Véase Apéndice A “Función de Palabras Duplicadas `the-the`”, página 217.

13 Contando: repetición y regexps

La repetición y búsqueda de expresiones regulares son herramientas poderosas que con frecuencia se usan cuando se escribe código en Emacs Lisp. Este capítulo ilustra el uso de búsqueda de expresiones regulares a través de la construcción de comandos de contaje de palabras usando bucles `while` y recursión.

La distribución de Emacs estándar contiene una función para contar el número de líneas en una región.

Hay cierto tipo de pregunta escrita para contar palabras. De este modo, si se escribe un ensayo, puede limitarse a 800 palabras; si se escribe una novela, te puedes disciplinar a ti mismo a escribir 1000 palabras al día. Parece raro, pero durante mucho tiempo, a Emacs le faltó un comando para contar palabras. Quizás la gente usaba Emacs mayoritariamente para codificar o documentar cosas que no requieren contar palabras, o quizás se restringían al sistema operativo el comando de contar palabras, `wc`. De manera alternativa, la gente puede seguir la convención de las editoriales y computaban un contaje de palabras dividiendo el número de caracteres en un documento por cinco.

Hay mucho caminos para implementar un comando para contar palabras. Aquí hay algunos ejemplos, que pueden desear compara con el comando de Emacs estándar, `count-words-region`.

13.1 La función `count-words-example`

Un comando de contar palabras podría contar palabras en una línea, párrafo, región, o búffer. ¿Qué comando funcionaría? Se podría diseñar el comando para contar el número de palabras en un búffer completo. Sin embargo, la tradición Emacs anima a la flexibilidad — se puede querer contar palabras solo en una sección, en vez de en todo un buffer. Así, tiene más sentido diseñar el comando para contar el número de palabras en una región. Una vez tienes un comando `count-words-region`, se puede, si lo deseas, contar palabras en un buffer completo marcándolo con *`C-x h`* (`mark-whole-buffer`).

Claramente, contar palabras es un acto repetitivo: empezando desde el principio de la región, se cuenta la primera palabra, entonces la segunda palabra, entonces la tercera palabra, y así, hasta que logres el fin de la región. Esto significa que contar palabras se ajusta idealmente a recursión o a un bucle `while`.

Primero, implementaremos el comando de contar palabras con un bucle `while`, entonces con la recursión. El comando, de acuerdo, será interactivo.

La plantilla para una definición de función interactiva es, como siempre:

```
(defun name-of-function (argument-list)
  "documentation..."
  (interactive-expression...)
  body...)
```

Lo que necesitamos hacer es rellenar los slots.

El nombre de la función sería auto-explicativo y similar al nombre del `count-lines-region` existente. Esto hace que el nombre sea fácil de recordar. `count-words-region` es una buena elección. Puesto que el nombre se usa ahora

para el comando de Emacs estándar para contar palabras, nosotros nombraremos nuestra implementación como `count-words-example`.

La función cuenta palabras con una región. Esto significa que el argumento lista debe contener símbolos que son asociados a las dos posiciones, el principio y fin de la región. Estas dos posiciones puede ser llamadas '`beginning`' y '`end`' respectivamente. La primera línea de la documentación sería una frase simple, desde que esto es todo lo que está impreso como documentación por un comando tal como `apropos`. La expresión interactiva será de la forma '`(interactive "r")`', puesto que causará que Emacs pase al principio y fin de la región a la lista de argumentos de función. Todo esto es rutina.

El cuerpo de la función necesita ser escrita para hacer tres tareas: primero, configurar condiciones bajo las que el bucle `while` pueda contar palabras, segundo, ejecutar el bucle `while`, y tercero, enviar un mensaje al usuario.

Cuando un usuario llama a `count-words-example`, apunta a que puede estar al principio o fin de la región. Sin embargo, el proceso de conteo debe empezar al principio de la región. Esto significa que querremos poner punto si eso no está allí. Esto significa que querremos poner el punto que hay si eso no está allí. Ejecutando `(goto-char beginning)` asegura esto. De acuerdo, querremos devolver el punto a su posición esperada cuando la función finalice su trabajo. Por esta razón, el cuerpo debe ser encerrado en una expresión `save-excursion`.

La parte central del cuerpo de la función consiste en un bucle `while` en el que una expresión salta el punto hacia delante palabra por palabra, y otra expresión cuenta estos saltos. El true-or-false-test del bucle `while` si es verdadero, el punto saltaría hacia adelante, y si es falso el punto estaría al fin de la región.

Nosotros podríamos usar `(forward-word 1)` como la expresión para mover el punto hacia adelante palabra por palabra, pero eso es fácil de ver que Emacs identifica como una 'palabra' si se usa una búsqueda de expresión regular.

Una expresión regular busca lo que encuentra el patrón que se está buscando deja el punto después del último carácter emparejado. Esto significa que una sucesión de palabras exitosas busquen que moverá el punto adelante palabra por palabra.

Como materia práctica, se quiere que la expresión regular se busque para saltar a través de un espacio en blanco y puntúe entre palabras tan bien a través de las palabras en sí. Una expresión regexp que rechaza para saltar a través de espacios en blanco entre palabras ¡nunca saltaría más de una palabra!. Esto significa que el regexp incluiría el espacio en blanco y la puntuación que sigue a una palabra, si cualquiera, como la palabra en sí. (Una palabra puede finalizar un búffer y no tiene cualquier espacio en blanco o puntuación, así esta parte del regexp debe ser opcional.)

De este modo, queremos para el regexp es un patrón definiendo una o más palabras caracteres que constituyen caracteres seguidos, opcionalmente, por uno o más caracteres que no son palabras consituyentes. La expresión regular para esto es:

```
\w+\W*
```

La tabla de sintaxis del búffer determina qué caracteres son y no son palabras constituyentes. Para más información acerca de sintaxis, véase Sección "Tablas de Sintaxis" in *El Manual de Referencia de GNU Emacs Lisp*.

La expresión se parece a esto:

```
(re-search-forward "\\w+\\W*")
```

(Note que las barras invertidas que preceden el '`w`' y '`W`'. Una barra invertida tiene significado especial al intérprete Emacs Lisp. Eso indica que el caracter siguiente es interpretado de manera diferente que la normal. Por ejemplo, los dos caracteres, '`\n`', son una '`nueva línea`', en vez de una barra invertida seguida por '`\n`'. Dos barras invertidas en una fila para una 'barra invertida no especial', así Emacs Lisp interpreta el fin de mirar una barra invertida simple seguida por una letra. Así descubre la letra que es especial.)

Se necesita un contador para contar cuántas palabras hay; esta variables debe primero ser asignado a 0 y entonces incrementados cada vez que Emacs va alrededor del bucle `while`. La expresión de incremento es simple:

```
(setq count (1+ count))
```

Finalmente, se quiere contar al usuario cuántas palabras hay en la región. La función `message` presenta este tipo de información al usuario. El mensaje tiene que ser fraseado de manera que se lea apropiadamente sin cuidado de cuantas palabras hay en la región: no queremos decir que "hay una palabra en la regiń". El conflicto entre singular y plural es no gramatical. Se puede resolver este problema usando una expresión condicional que evalúa diferentes mensajes dependiendo en el número de palabras en la región. Hay tres posibilidades: no palabras en la región, una palabra en la región, y más de una palabra. Esto significa que la forma especial `cond` es apropiada.

Todo esto lidera a la siguiente definición de función:

```
;;; ¡La Primera versión; tiene errores!
(defun count-words-region (beginning end)
  "Imprime el número de palabras en la región.
Las palabras están definidas al menos una palabra
constituida de caracteres seguido por al menos un
caracter que no constituye palabra. La tabla de
sintaxis del búffer determina qué caracteres hay."
  (interactive "r")
  (message "Contando palaras en la región ... ")

;;; 1. Configurar condiciones apropiadas.
  (save-excursion
    (goto-char beginning)
    (let ((count 0))

;;; 2. Ejecutar el bucle while.
      (while (< (point) end)
        (re-search-forward "\\w+\\W*")
        (setq count (1+ count)))
```

```
;;; 3. Enviar un mensaje al usuario.
      (cond ((zerop count)
             (message
              "La región no tiene palabras."))
            ((= 1 count)
             (message
              "The región tiene 1 palabra."))
            (t
             (message
              "The región tiene %d palabras." count))))))
```

Como se escribe, la función funciona, pero no en todas las circunstancias.

13.1.1 El error de espacio en blanco en `count-words-example`

El comando `count-words-example` descrito en las sección precedente tiene dos errores, o incluso, un error con dos manifestaciones. Primero, si se marca una región conteniendo solo espacio en blanco en el medio de algún texto el comando ¡`count-words-example` cuenta que la región contiene una palabra!. Segundo, si se marca una región conteniendo solo espacios en blanco al final del búffer o la porción accesible de un búffer encogido, el comando muestra un mensaje de error que se parece a esto:

```
Búsqueda fallida: "\\w+\\W*"
```

Si estás leyendo esto en Info en GNU Emacs, se puede testear para estos errores por sí mismo.

Primero, evalúa la función en la manera usual para instalarlo.

Si se desea, se puede también instalar este atajo para ser evaluado:

```
(global-set-key "\C-c=" 'count-words-example)
```

Para conducir el primer test, asigna marca y punto al principio y fin de la siguiente línea y entonces escribe *`C-c =`* (o *`M-x count-words-example`* si no se ha asignado *`C-c =`*):

```
uno   dos  tres
```

Emacs te contará, correctamente, que la región tiene tres palabras.

Repite el test, pero marca el lugar al principio de la línea y emplaza el punto justo *antes* de la palabra '`uno`'. De nuevo escribe el comando *`C-c =`* (o *`M-x count-words-example`*). Emacs cuenta que la región no tiene palabras, puesto que eso está compuesto solo por espacios en blanco al principio de la línea. ¡Pero en vez de que Emacs cuente que la región tiene una palabra!

Para el tercer test, copia la línea de ejemplo al fin del búffer `*scratch*` y entonces escribe varios espacios al fin de la línea. Posiciona la marca correcta después de la palabra '`tres`' y apunta al fin de la línea. (El fin de la línea será el fin del búffer.) Escribe *`C-c =`* (o *`M-x count-words-example`*) como se hizo antes. De nuevo, Emacs te contaría que la región no tiene palabras, puesto que eso está compuesto solo de los espacios en blanco al fin de la línea. En vez de eso, Emacs muestra un mensaje de error diciendo '`Búsqueda fallida`'.

Los dos errores queman el mismo problema.

Considere la primera manifestación del error, en el que el comando te cuenta que el espacio en blanco al principio de la línea contiene una palabra. Lo que ocurre

es esto: El comando `M-x count-words-example` mueve el punto al principio de la región. El test `while` si el valor del punto es más pequeño que el valor de `end`, que es. Por consiguiente, en la expresión regular se busca y encuentra la primera palabra. Eso deja el punto después de la palabra. `count` se establece a uno. El bucle `while` repite; pero esta vez el valor del punto es más largo que el valor de `end`, el bucle sale; y la función muestra un mensaje diciendo el número de palabras en la región es uno. En breve, la expresión regular busca y encuentra la palabra incluso aunque eso esté fuera de la región marcada.

En la segunda manifestación del error, la región es un espacio en blanco al fin del búffer. Emacs dice '`Búsqueda fallida`'. Lo que ocurre es que true-or-false-test en el bucle `while` chequea verdad, así la expresión de búsqueda es ejecutada. Pero desde que no hay más palabras en el buffer, la búsqueda falla.

En ambas manifestaciones del error, la búsqueda extiende o intenta extenderse fuera de la región.

La solución es limitar la búsqueda a la región — esto es una acción simple y limpia, pero como tu puedes tener que llegar a esperar, eso no es tan simple como se podría pensar.

Como se ha visto, la función `re-search-forward` toma un patrón de búsqueda como su primer argumento. Pero además de este primer, argumento obligatorio, se aceptan tres argumentos opcionales. El segundo argumento opcional asocia la búsqueda. El tercer argumento opcional, si `t`, causa la función a devolver `nil` en vez de la señal un error si la búsqueda falla. El cuarto argumento opcional es un contador repetido. (En Emacs, se puede ver una documentación de la función escribiendo *`C-h f`*, el nombre de la función, y entonces `RET`.)

En la definición `count-words-example`, el valor del fin de la región es tomada por la variable `end` que es pasada como un argumento para la función. De este modo, se puede añadir `end` como un argumento para la búsqueda de la expresión de búsqueda:

```
(re-search-forward "\\w+\\W*" fin)
```

Sin embargo, si se crea solo este cambio a la definición `count-words-example` y entonces se chequea la nueva versión de la definición en una extensión de espacio en blanco, se recibirá un mensaje de error diciendo '`Búsqueda fallida`'.

Lo que ocurre es esto: la búsqueda se limita a la región, y falla como se espera porque no hay caracteres de palabras constituyentes en la región. Puesto que eso falla, se recibe un mensaje de error. Pero no queremos recibir un mensaje de error en este caso; se quiere recibir el mensaje que "La región no tiene palabras".

La solución a este problema es proveer `re-search-forward` con un tercer argumento de `t`, que causa la función para devolver `nil` en vez la señalar un error si la búsqueda falla.

Sin embargo, si se crea este cambio y se intenta, se verá el mensaje "Contando palabras en la región ..." y . . . se guardará viendo que mensaje . . . , hasta se escribe *`C-g`* (`keyboard-quit`).

Aquí está lo que ocurre: la búsqueda está limitada a la región, como antes, y eso falla porque no hay caracteres no constituyentes de palabras en la región, como se espera. Consiguientemente, la expresión `re-search-forward` devuelve `nil`. Eso no

hace nada más. En particular, no mueve el punto, que hace como un efecto lateral si eso encuentra la búsqueda objetiva. Después la expresión `re-search-forward` devuelve `nil`, la siguiente expresión en el bucle `while` está evaluado. Esta expresión incrementa el contador. Entonces el bucle repite. El test true-or-false-test chequea cierto porque el valor del punto es todavía menor que el valor final, desde que la expresión `re-search-forward` no movería el punto. . . . y el ciclo repite . . .

La definición `count-words-example` requiere todavía de otra modificación para causar el true-or-false-test del bucle `while` para chequear falso si la búsqueda falla. Pon otro camino, hay dos condiciones que deben ser satisfechas en el true-or-false-test antes que el contador de palabras variable se incremente: el punto debe todavía estar con la región y la expresiń de búsqueda debe haber encontrado una palabra para contar.

Por ambas la primera condición y la segunda condición deben ser ciertas juntas, las dos expresiones, la región chequea y la expresión de búsqueda, puede estar unido con una forma especial `and` y embebido en el bucle `while` como el true-or-false-test, como esto:

```
(and (< (point) end) (re-search-forward "\\w+\\W*" end t))
```

(Para información acerca de `and`, ver "La función `kill-new`", página 94.)

La expresión `re-search-forward` devuelve `t` si la búsqueda es exitosa y como efecto lateral se mueve el punto. Consiguientemente, como las palabras se encuentran, el punto es movido a través de la región. Cuando la búsqueda de la expresión falla para encontrar otra palabra, o cuando el punto logra el fin de la región, el test true-or-false-test es falso, el bucle `while` existe, y la función `count-words-example` muestra uno u otro de sus mensajes.

Después de incorporar estos cambios finales, el `count-words-example` funciona sin errores (¡o al menos, sin los errores que yo haya encontrado!. Aquí está lo que parece:

```
;;; Versión final: while
(defun count-words-example (beginning end)
  "Imprime número de palabras en la región."
  (interactive "r")
  (message "Contando palabras en la región ... ")

;;; 1. Configura condiciones apropiadas.
  (save-excursion
    (let ((count 0))
      (goto-char beginning)

;;; 2. Ejecuta el bucle while
      (while (and (< (point) end)
                  (re-search-forward "\\w+\\W*" end t))
        (setq count (1+ count)))
```

```
;;; 3. Enviar un mensaje al usuario.
      (cond ((zerop count)
             (message
              "La región no tiene palabras."))
            ((= 1 count)
             (message
              "The región tiene 1 palabra."))
            (t
             (message
              "The región tiene %d palabras." count))))))
```

13.2 Cuenta palabras recursivamente

Se puede escribir la función para contar palabras tanto de manera recursiva como con un bucle `while`. Permita ver cómo se hace.

Primero, se necesita reconocer que la función `count-words-example` tiene tres trabajos: eso configura las condiciones apropiadas para contar lo que ocurre; eso cuenta las palabras en la región; y envía un mensaje al usuario contando cuántas palabras hay.

Si se escribe una función recursiva simple para hacer cualquier cosa se recibirá un mensaje para cada llamada recursiva. Si la región contiene 13 palabras, se recibirán trece mensajes, uno correcto después del otro. ¡No queremos esto!. En vez de eso, se deben escribir dos funciones para hacer el trabajo, una (la función recursiva) será usada dentro de la otra. Una función configurará las condiciones y muestra el mensaje; la otra devolverá el contador de palabras.

Permítase comenzar con la función que causa el mensaje que se muestra. Se puede continuar por llamarse `count-words-example`.

Esta es la función que el usuario llama. Será interactiva. En realidad, será similar a nuestras versiones previas de esta función, excepto que llamará `recursive-count-words` para determinar cuántas palabras hay en la región.

Se puede construir una plantilla legible para esta función, basada en versiones previas:

```
;; Versión Recursiva; usa la búsqueda de la expresión regular
(defun count-words-example (beginning end)
  "documentation..."
  (interactive-expression...)

;;; 1. Configura condiciones apropiadas.
  (explanatory message)
  (set-up functions...

;;; 2. Contar las palabras.
     recursive call

;;; 3. Envía un mensaje al usuario.
     message providing word count))
```

La definición parece sencilla, excepto que como el contador devuelve la llamada recursiva que debe ser pasada al mensaje mostrando el contaje de palabras. Un

pequeño pensamiento sugiere que esto puede ser hecho haciendo uso de una expresión `let` al número de palabras en la región, como se devuelve por la llamada recursiva; y entonces la expresión `cond`, que usa la asociación, puede mostrar el valor al usuario.

Con frecuencia, uno piensa que se puede asociar una expresión `let` como algo secundario al trabajo 'primario' de una función. Pero en este caso, se podría considerar el trabajo 'primario' de la función, contando palabras, es hecho con la expresión `let`.

Usando `let`, la definición de función se parece a esto:

```
(defun count-words-example (beginning end)
  "Imprime el número de palabras en la región."
  (interactive "r")

;;; 1. Configura condiciones apropiadas.
  (message "Contando palabras en la región ... ")
  (save-excursion
    (goto-char beginning)

;;; 2. Contar las palabras.
    (let ((count (recursive-count-words end)))

;;; 3. Enviar un mensaje al usuario.
      (cond ((zerop count)
             (message
              "La región no tiene palabras."))
            ((= 1 count)
             (message
              "The región tiene 1 palabra."))
            (t
             (message
              "The región tiene %d palabras." count))))))
```

Lo siguiente, que se necesita es escribir la función de contaje recursivo.

Una función recursiva tiene al menos tres partes: el 'do-again-test', la 'next-step-expression', y la llamada recursiva.

El do-again-test determina si la función será o no llamada de nuevo. Puesto que estamos contando palabras en una región y puede causar que una función se mueva el punto hacia delante por cada palabra, el do-again-test puede chequear si el punto está todavía con la región. El do-again-test encontraría el valor del punto y determina si el punto está antes, en, o después del valor del fin de la región. Se puede usar la función `point` para localizar el punto. Claramente, se debe pasar el valor del fin de la región a la función de contaje recursivo como un argumento.

Además, el do-again-test también chequearía si la búsqueda encuentra una palabra. Si no, la función no se llamaría de nuevo.

La next-step-expression cambia un valor así que cuando la función recursiva se supone que debe parar de llamarse así misma, se para. Más precisamente, los cambios de next-step-expression cambia un valor así que en el momento adecuado, el do-again-test para la función recursiva de la llamada en sí de nuevo. En este caso,

la next-step-expression puede ser la expresión que mueve el punto hacia adelante, palabra por palabra.

La tercera parte de una función recursiva es la llamada recursiva.

En algún lugar, también, se necesita una parte que hace el 'trabajo' de la función, una parte que es el contaje. ¡Una parte vital!

Pero ya, tenemos un guión de la función recursiva de contaje:

```
(defun recursive-count-words (region-end)
  "documentation..."
   do-again-test
   next-step-expression
   recursive call)
```

Ahora se necesita rellenar los slots. Permite comenzar con el caso más simple primero: si se apunta debajo del fin de la región, no puede haber palabras en la región, así la función devuelve cero. De otro modo, si la búsqueda falla no hay palabras para contar, así la función devolvería cero.

Por otro lado, si se apunta con la región y la búsqueda tiene éxito, la función se llamaría de nuevo.

De este modo, do-again-test se vería así:

```
(and (< (point) region-end)
     (re-search-forward "\\w+\\W*" region-end t))
```

Note que la expresión de búsqueda es parte del do-again-test — la función devuelve `t` si su búsqueda tiene éxito y `nil` si falla. (Véase Sección 13.1.1 "El Error de Espacio en Blanco en `count-words-example`", página 153), para una explicación de cómo `re-search-forward` funciona.)

El do-again-test es el test true-or-false de una cláusula `if`. Claramente si el do-again-test tiene éxito, la then-part de la cláusula `if` llamaría a la función; pero si eso falla, la else-part devolvería cero desde que el punto está fuera de la región o la búsqueda falló porque no había palabras a encontrar.

Pero antes de considerar la llamada recursiva, se necesita considerar la next-step-expression. ¿Qué es eso? De manera interesante, eso es la parte de la búsqueda del do-again-test.

Además para devolver `t` o `nil` para el do-again-test, `re-search-forward` mueve el punto hacia adelante como un efecto lateral de una búsqueda exitosa. Esta es la acción que cambia el valor de punto así que la función recursiva para de llamarse a sí misma cuando el punto complete su movimiento a través de la región. Por consiguiente, la expresión `re-search-forward` es la next-step-expression.

En esquema, entonces, el cuerpo de la función `recursive-count-words` se parece a esto:

```
(if do-again-test-and-next-step-combined
    ;; then
    recursive-call-returning-count
  ;; else
  return-zero)
```

¿Cómo incorporar el mecanismo que cuenta?

Si no estás acostumbrado a escribir funciones recursivas, una pregunta como esta puede ser un problema. Pero eso puede y sería enfocado sistemáticamente.

Se sabe que el mecanismo de contaje sería asociado en algún lugar con la llamada recursiva. En vez de eso, desde que la next-step-expression mueve el punto hacia adelante por una palabra, y desde que una llamada recursiva es hecha para cada palabra, el mecanismo de contaje debe ser una expresión que añade uno al valor devuelto por una llamada para `recursive-count-words`

Considera varias casos:

- Si hay dos palabras en la región, la función devolverá un valor resultante de añadir uno al valor devuelto cuando eso cuenta la primera palabra, más el número devuelto cuando eso cuenta las palabras que permanecen en la región, que en este caso es una.
- Si hay una palabra en la región, la función devolvería un valor resultante de añadir uno al valor devuelto cuando eso cuenta esta palabra más el número devuelto cuando eso cuenta las palabras que permanecen en la región, que en este caso es cero.
- Si no hay palabras en la región, la función devolvería cero.

Desde el esquema podemos ver que la parte else del `if` devuelve cero para el caso en el que no hay palabras. Esto significa que la parte then del `if` debe devolver un valor resultante de añadir uno al valor devuelto desde el contaje de las palabras que permanecen.

La expresión se parece a esto, donde `1+` es una función que añade uno a su argumento.

```
(1+ (recursive-count-words region-end))
```

La función completa `recursive-count-words` entonces se parecerá e esto:

```
(defun recursive-count-words (region-end)
  "documentation..."

  ;;; 1. do-again-test
  (if (and (< (point) region-end)
           (re-search-forward "\\w+\\W*" region-end t))

  ;;; 2. then-part: la llamada recursiva
      (1+ (recursive-count-words region-end))

  ;;; 3. else-part
    0))
```

Permíteme examinar cómo esto funciona:

Si no hay palabras en la región, la parte else de la expresión `if` es evaluada y consecuentemente la función devuelve cero.

Si hay una palabra en la región, el valor del punto es menor que el valor de `region-end` y la búsqueda tiene éxito. En este caso, el true-or-false-test de la expresión `if` chequea cierto, y la then-part de la expresión `if` es evaluada. La expresión de contaje es evaluada. Esta expresión devuelve un valor (que será el valor devuelto por la función completa) que es la suma de uno añadida al valor devuelto por una llamada recursiva.

Mientras tanto, la next-step-expression ha causado el punto para saltar a través de la primera (y en este caso única) palabra en la región. Esto significa que cuando

`(recursive-count-words region-end)` está evaluada una segunda vez, como un resultado de la llamada recursiva, el valor del punto será igual o mayor que el valor de la región final. Así esta vez, `recursive-count-words` devolverá cero. El cero será añadido a uno, y la evaluación original de `recursive-count-words` devolverá uno más cero (uno) que es la cantidad correcta.

Claramente, si hay dos palabras en la región, la primera llamada a `recursive-count-words` devuelve uno añadido al valor devuelto llamando `recursive-count-words` en una región contiendo la palabra que permanece — que es, eso añadir uno a uno, produciendo dos, que es la cantidad correcta.

Similarmente, si hay tres palabras en la región, la primera llamada `recursive-count-words` devuelve uno añadido al valor devuelto llamado `recursive-count-words` en una región conteniendo las dos palabras que permanecen — y así y así.

Con documentación completa las dos funciones se parecen a esto:

La función recursiva:

```
(defun recursive-count-words (region-end)
  "Número de palabras entre punto y REGION-END."

;;; 1. do-again-test
  (if (and (< (point) region-end)
           (re-search-forward "\\w+\\W*" region-end t))

;;; 2. then-part: la llamada recursiva
      (1+ (recursive-count-words region-end))

;;; 3. else-part
    0))
```

El envoltorio:

```
;;; Versión Recursiva
(defun count-words-example (beginning end)
  "Imprime el número de palabras en la región.

Las palabras son definidas como al menos una palabra constituyente
seguida por al menos un caracter que es una palabra constituyente. La
tabla de sintaxis del buffer determina qué caracter hay.
  (interactive "r")
  (message "Contando palabras en la región ... ")
  (save-excursion
    (goto-char beginning)
    (let ((count (recursive-count-words end)))
      (cond ((zerop count)
             (message
              "La región no tiene palabras."))
            ((= 1 count)
             (message "La región tiene 1 palabra."))
            (t
             (message
              "La región tiene %d palabras." count))))))
```

13.3 Ejercicio: contando puntuación

Usando un bucle `while`, escriba una función para contar el número de marcas de puntuación en una región — periodo, coma, punto y coma, dos puntos, exclamación, marca y marca de pregunta. Haga lo mismo usando recursión.

14 Contando palabras en una `defun`

Nuestro siguiente proyecto es contar el número de palabras en una definición de función. Claramente, esto puede ser hecho usando alguna variante de `count-words-example`. Véase Capítulo 13 "Contando palabras: repetición y regexps", página 150. Si estamos ahora yendo a contar las palabras en una definición, es suficientemente fácil marcar la definición con el comando *`C-M-h`* (`mark-defun`), y entonces llamar a `count-words-example`.

Sin embargo, soy más ambicioso: Yo quiero contar las palabras y símbolos en cada definición en las fuentes de Emacs y entonces imprimir un grafo que muestre cuántas funciones hay de cada tamaño: cuántas contienen de 40 a 49 palabras o símbolos, cuántas contienen de 50 a 59 palabras o símbolos, y así. Yo he sido con frecuencia curioso de cómo es una función típica, y esto se contará.

Descrito en una frase, el proyecto desanima; pero dividido dentro de numerosos pequeños pasos, cada uno de los que podemos tomar en un momento, el proyecto llegar ser menos atemorizante. Permítenos considerar qué pasos deben ser:

- Primero, escribe una función para contar las palabras en una definición. Esto incluye el problema de manejar símbolos tan bien como palabras.
- Segundo, escribe una función para listar los números de palabras en cada función en un fichero. Esta función puede usar la función `count-words-in-defun`.
- Tercero, escribe una función para listar los números de palabras en cada función en cada uno de varios ficheros. Esto encola automáticamente encontrando varios ficheros, cambiándolos, y contando las palabras en las definiciones con ellos.
- Cuarto, escribe una función para convertir la lista de números que nosotros creamos en tres pasos para un formulario que se ajustará para imprimir un grafo.
- Quinto, escribe una función para imprimir los resultados como un grafo.

¡Esto es un proyecto! Pero si tomamos cada paso lentamente, eso no será difícil.

14.1 ¿Qué contar?

Cuando nosotros primero empezamos pensando acerca del contaje de palabras en una definición de función, la primera pregunta es (o podría ser) ¿qué se va a contar? Cuando se habla de 'palabras' con repecto a una definición de función Lisp, estamos actualmente hablando, en parte, de 'símbolos'. Por ejemplo, la siguiente función `multiply-by-seven` contiene los cinco símbolos `defun`, `multipy-by-seven`, `number`, `*`, y `7`. Además, en la cadena de documentación, contiene cuatro palabras '`Multiplicar`', '`NUMBER`', '`por`', y '`siete`'. El símbolo '`número`' es repetido, así la definición contiene un total de diez palabras y símbolos.

```
(defun multiply-by-seven (number)
  "Multiplicar NUMBER por siete."
  (* 7 number))
```

Sin embargo, si se marca la definición `multiply-by-seven` con *`C-M-h`* (`mark-defun`), y entonces se llama a `count-words-example` dentro, se encontrará

que `count-words-example` ¡reclama la definicón tiene once palabras, no diez! ¡Alguna cosa está mal!

El problema es doble: `count-words-example` no cuenta el '`*`' como una palabra, y eso cuenta el símbolo simple, `multiply-by-seven`, conteniendo tres palabras. Las conexiones son tratadas como si fueran espacios entre palabras en vez de conectores entre palabras '`multiply-by-seven`' se cuenta como si fuese escrito '`multiply-by-seven`'.

La causa de esta confusión es la expresión regular que busca la definición `count-words-example` que mueve el punto hacia delante palabra por palabra. En la versión canónica de `count-words-example`, el regexp es:

```
"\\w+\\W*"
```

Esta expresión regular es un patrón definiendo una o más palabras constituyendo caracteres posiblemente seguidos por uno o más caracteres que no son palabras constituyentes. Esto significa que los 'caracteres que constituyen palabras' nos traen la cuestión de la sintaxis, que es el valor de una sección en sí.

14.2 ¿Qué constituye una palabra o símbolo?

Emacs trata diferentes caracteres perteneciendo a diferentes *categorías de sintaxis*. Por ejemplo, la expresión regular, '`\\w+`', es un patrón especificando uno o más caracteres de *palabras constituyentes*. Los caracteres de palabras constituyentes son miembros de una categoría de sintaxis. Otras categoría de sintaxis incluye la clase de caracteres de puntuación, tales como el espacio en blanco o el caracter de tabulación. (Para más información, ver Sección "La Tabla de Sintaxis" in *El Manual GNU Emacs*, y Sección "Tablas de Sintaxis" in *El Manual de Referencia GNU Emacs Lisp*.)

Las tablas de sintaxis especifican qué caracteres pertenecen a qué categorías. Normalmente un guión no está especificado como un 'caracter constituido por una palabra'. En vez de eso, se especificó como estando en la 'clase de caracteres que son parte de los nombres de símbolos, pero no las palabras.' Esto significa que la función `count-words-example` lo trata del mismo modo que trata un espacio en blanco entre palabras, que es el por qué `count-words-example` cuenta '`multiply-by-seven`' como tres palabras.

Hay dos caminos para causar que Emacs cuente '`multiply-by-seven`' como un símbolo: modificar la tabla de sintaxis o modificar la expresión regular.

Se podría redefinir un guión (*hyphen*) como un caracter que constituye una palabra modificando la tabla de sintaxis que Emacs guarda por cada modo. Esta acción serviría nuestro propósito, excepto que una conexión es meramente el caracter más común con símbolos que no son típicamente un caracter de palabra constituyente; hay otros, también.

Alternativamente, se puede redefinir la expresión regular *regexp* usada en la definición `count-words` así como incluir símbolos. Este procedimiento tiene el mérito de la claridad, pero la tarea es un pequeño truco.

La primera parte es suficientemente simple: el patrón debe asignarse "al menos un carácter que es una palabra o símbolo constituyente". De este modo:

```
"\\(\\w\\|\\s_\\)+"
```

El '`\\(`' es la primera parte del constructo que agrupa esto que incluye el '`\\w`' y el '`\\s_`' como alternativas, separadas por los '`\\|`'. El '`\\w`' asocia cualquier caracter de palabra constituyente y el '`\\s_`' asocia cualquier caracter que es parte de un nombre de símbolo pero no una palabra de caracteres constituyente. El '`+`' sigue al grupo que indica que la palabra o símbolo constituyan caracteres que deben ser asociados al menos por uno.

Sin embargo, la segunda parte de regexp es más difícil de diseñar. Lo que queremos es seguir la primera parte con "opcionalmente uno o más caracteres que no constituyen una palabra o símbolo". Primero, se pensaba que se podría definir esto con lo siguiente:

```
"\\(\\W\\|\\S_\\)*"
```

Las mayúsculas '`W`' y '`S`' asocian caracteres que *no* son constituyente de palabra o símbolo. Desafortunadamente, esta expresión asocia cualquier caracter que sea o no una palabra constituyente no un símbolo constituyente. ¡Esto asocia cualquier caracter!

Entonces se notificó que cada palabra o símbolo en mi región test fué seguida por algún espacio (espacio en blanco, tabulador, o nueva línea). Así yo intenté emplazar un patrón para asociar uno o más espacios en blanco después del patrón para una o más palabras o símbolos constituyentes. Esto falló, también. Palabras y símbolos son con frecuenca separados por espacios en blanco, pero en el código actual los paréntesis pueden seguir símbolos y puntuación puede seguir las palabras. Así finalmente, se diseño un patrón en el que la palabra o símbolo constituyente es seguido opcionalmente por caracteres que no son espacios en blanco y entonces son seguidos opcionalmente por espacios en blanco.

Aquí está la expresión regular completa:

```
"\\(\\w\\|\\s_\\)+[^ \t\n]*[ \t\n]*"
```

14.3 La función `count-words-in-defun`

Se ha visto que hay varios caminos para escribir una función `count-word-region`. Para escribir un `count-words-in-defun`, se necesita solamente adaptar una de estas versiones.

La versión que usa un bucle `while` es fácil de comprender, así estoy yendo a adaptar esto. Porque `count-words-in-defun` será parte de un programa más complejo, eso no necesita ser interactivo y no necesita mostrar un mensaje pero solo devuelve el contaje. Estas consideraciones simplifican la definición un poco.

Por otro lado, `count-words-in-defun` será usado con un buffer que contiene definiciones de función. Consiguientemente, es razonable preguntar que la función determina si se llamó cuando el punto está con una definición de función, y eso es, para devolver el contaje para esta definición. Esto añade complejidad a la definición, pero nos guarda desde la necesidad de pasar argumentos a la función.

Estas consideraciones nos llevan a preparar la siguiente plantilla:

```
(defun count-words-in-defun ()
  "documentation..."
  (set up...
     (while loop...)
   return count)
```

Así, el trabajo es rellenar los slots.

Primero, la configuración.

Estamos presuponiendo que esta función será llamada con un búffer conteniendo definiciones de función. Apunta si será con una definición de función o no. Para que `count-words-in-defun` funcione, el punto debe moverse al principio de la definición, un contador debe empezar a cero, y el bucle contando debe parar cuando el punto logre el fin de la definición.

La función `beginning-of-defun` busca atrás para un delimitador de apertura tal como '`(`' al principio de una línea, y mueve el punto a esta posición, o sino al límite de la búsqueda. En la práctica, esto significa que `beginning-of-defun` mueve el punto al principio de un cierre o definición de función precedente, o sino al principio del buffer.

El bucle `while` requiere un contador para guardar la traza de las palabras o símbolos siendo contados. Una expresión `let` puede ser usado para crear una variable local para este propósito, y lo asocia a un valor inicial de cero.

La función `end-of-defun` funciona como `beginning-of-defun` excepto que mueve el punto al fin de la definición. `end-of-defun` puede ser usado como parte de una expresión que determina la posición del fin de la definición.

La configuración para `count-words-in-defun` toma forma rápidamente: primero movemos el punto al principio de la definición, entonces se crea una variable local para manejar el conteo, y finalmente, se graba la posición del fin de la definición así el bucle `while` conocerá cuando parar el bucle.

El código se parece a esto:

```
(beginning-of-defun)
(let ((count 0)
      (end (save-excursion (end-of-defun) (point))))
```

El código es simple. La única ligera complicación es probablemente ir al `end`: eso está asociado a la posición del fin de la definición por una expresión `save-excursion` que devuelve el valor del punto después de `end-of-defun` temporalmente se mueve al fin de la definición.

La segunda parte del `count-words-in-defun`, después de la configuración, es el bucle `while`.

El bucle loop debe contener una expresión que mueve el punto hacia adelante palabra por palabra y símbolo por símbolo, y otra expresión que cuenta los saltos. El true-or-false-test para el bucle `while` chequearía verdadero tan largo como el punto debería saltar hacia adelante, y falso cuando apunta al fin de la definición. Ya se ha redefinido la expresión regular para esto, así el bucle es sencillo:

```
(while (and (< (point) end)
            (re-search-forward
             "\\(\\w\\|\\s_\\)+[^ \t\n]*[ \t\n]*" end t)
  (setq count (1+ count)))
```

La tercera parte de la definición devuelve el contaje de palabras y símbolos. Esta parte es la última expresión con el cuerpo de la expresión `let`, y puede ser, muy la variable local `count`, que cuando se evalúa devuelve el contaje.

Puesto junto, la definición `count-words-in-defun` se ve así:

```
(defun count-words-in-defun ()
  "Devuelve el número de palabras y símbolos en una defun."
  (beginning-of-defun)
  (let ((count 0)
        (end (save-excursion (end-of-defun) (point))))
    (while
        (and (< (point) end)
             (re-search-forward
              "\\(\\w\\|\\s_\\)+[^ \t\n]*[ \t\n]*"
              end t))
      (setq count (1+ count)))
    count))
```

¿Cómo se chequea esto? La función no es interactiva, pero es fácil poner un envoltorio alrededor de la función para hacerla interactiva; se puede usar casi el mismo código como la versión recursiva de `count-words-example`:

```
;;; Versión Interactiva.
(defun count-words-defun ()
  "Número de palabras y símbolos en una definición
  de función."
  (interactive)
  (message
   "Contando palabras y símbolos en la definición de función ... ")
  (let ((count (count-words-in-defun)))
    (cond
     ((zerop count)
      (message
       "La definición NO tiene palabras o símbolos."))
     ((= 1 count)
      (message
       "La definición tiene 1 palabra o símbolo."))
     (t
      (message
       "La definición tiene %d palabras o símbolos." count)))))
```

Permite reutilizar *C-c* = como un atajo conveniente:

```
(global-set-key "\C-c=" 'count-words-defun)
```

Ahora se puede intentar `count-words-defun`: instala ambas funciones `count-words-in-defun` y `count-words-defun`, y asigna el atajo, y entonces emplaza el cursor con la siguiente definición:

```
(defun multiply-by-seven (number)
  "Multiplicar NUMBER por siete."
  (* 7 number))
    ⇒ 10
```

¡Éxito! La definición tiene 10 palabras y símbolos.

El siguiente problema es contar los números de palabras y símbolos en varias definiciones con un fichero simple.

14.4 Contar varias `defuns` en un fichero

Un fichero tal como `simple.el` puede tener un centenar o más definiciones de función dentro. Nuestro objetivo es recoger estadísticas en muchos ficheros, pero en un primer paso, nuestro objetivo inmediato es recoger estadísticas en un fichero.

La información será una serie de números, cada número siendo el tamaño de una definición de función. Se puede almacenar los número en una lista.

Se sabe que se querrá incorporar la información considerando un fichero con información acerca de muchos otros ficheros; esto significa que la función para contar el tamaño de contaje con un fichero solo necesita devolver la lista de tamaños. Eso no necesita y no mostraría mensajes.

Los comando de contar palabras contienen una expresión para saltar el punto hacia adelante palabra por palabra y otra expresión para contar los saltos. La función devuelve los tamaños de definiciones que pueden ser diseñadas para trabajar del mismo modo, con una expresión para saltar el punto hacia la definición por definición y otra expresión para construir el tamaño de la lista.

Esta frase del problema hace elemental escribir la definición de función. Claramente, empezaremos el conteo al principio del fichero, así el primer comando será `(goto-char (point-min))`. Lo siguiente, es empezar el bucle `while`; y este true-or-false del bucle puede ser una búsqueda de expresión regular para la siguiente definición de función — así en el momento que la búsqueda tiene éxito, el punto se mueve hacia adelante y entonces el cuerpo del bucle es evaluado. El cuerpo necesita una expresión que construye la lista de tamaños. `cons`, la lista de construcción del comando, puede ser usado para crear la lista. Esto es casi todo lo que hay.

Aquí está este fragmento de código que se ve así:

```
(goto-char (point-min))
(while (re-search-forward "^(defun" nil t)
  (setq lengths-list
        (cons (count-words-in-defun) lengths-list)))
```

Dejamos fuera el mecanismo para encontrar el fichero que contiene las definiciones de función.

En ejemplos previos, nosotros habíamos usado esto, el fichero Info, o cambiamos atrás y adelante a algún otro búffer, tal como el búffer `*scratch*`.

Encontrando un fichero es un nuevo proceso que no tenemos todavía discutido.

14.5 Encontrar un fichero

Para encontrar un fichero en Emacs, se usa el comando *`C-x C-f`* (`find-file`). Este comando es casi, pero no bastante correcto para el problema de tamaños.

Permita mirar el fuente para `find-file`:

```
(defun find-file (filename)
  "Edita el fichero FILENAME.
Cambia a un búffer visitando el fichero FILENAME,
creando uno si no existe ya."
  (interactive "FFind file: ")
  (switch-to-buffer (find-file-noselect filename)))
```

(La versión más reciente de la definición de función `find-file` permite comodines especiales para visitar múltiples ficheros; que hacen la definiciń más compleja y no se discutirá aquí, ya que no es relevante. Se pueden ver sus fuentes usando *`M-.`* (`find-tag`) o *`C-h f`* (`describe-function`).)

La definición que se está mostrando posee una documentación corta, pero completa y una especificación interactiva que muestra un nombre de fichero cuando se usa el comando interactivamente. El cuerpo de la definición contiene dos funciones, `find-file-noselect` y `switch-to-buffer`.

De acuerdo a su documentación como muestra por *`C-h f`* (el comando `describe-function`), la función `find-file-noselect` lee el fichero nombrado dentro de un búffer y devuelve el búffer. (Su versión más reciente incluye un argumento comodín, también, así como otro para leer un fichero literalmente y otro que suprime mensajes de aviso. Estos argumentos opcionales son irrelevantes.)

Sin embargo, la función `find-file-noselect` no selecciona el búffer en el que se pone el fichero. Emacs no cambia su atención (o la tuya si estás usando `find-file-noselect`) al búffer seleccionado. Esto es lo que `switch-to-buffer` hace: eso cambia el búffer al que se dirige la atención de Emacs; y eso cambia el búffer mostrado en la ventana al nuevo búffer. Se ha discutido el búffer cambiando a otro lugar. (Véase Sección 2.3 “Cambiando búffers”, página 23.)

En este proyecto de histograma, no se necesita mostrar cada fichero en la pantalla como el programa determina el tamaño de cada definición con eso. En vez de emplear `switch-to-buffer`, se puede trabajar con `set-buffer`, que redirige la atención del programa de ordenador para un buffer diferente pero no lo muestra en pantalla. Así en vez llamar a `find-file` para hacer el trabajo, debe escribir nuestra expresión.

La tarea es fácil: usar `find-file-noselect` y `set-buffer`.

14.6 `lengths-list-file` en detalle

El núcleo de la función `lengths-list-file` es un bucle `while` conteniendo una función para mover el punto hacia delante ‘función a función’ y una función para contar el número de palabras y símbolos en cada función. Este núcleo debe ser rodeado por funciones que hacen otras tareas varias, incluyendo encontrar el fichero, y asegurando que el punto empieza al principio del fichero. La definición de la función se parece a:

```
(defun lengths-list-file (filename)
  "Devuelve la lista de tamaños de definiciones con FILE.
La lista devuelta es una lista de números.
Cada número es el número de palabras o
símbolos en una definición."
```

```
(message "Trabajando en `%s' ... " filename)
(save-excursion
  (let ((buffer (find-file-noselect filename))
        (lengths-list))
    (set-buffer buffer)
    (setq buffer-read-only t)
    (widen)
    (goto-char (point-min))
    (while (re-search-forward "^(defun" nil t)
      (setq lengths-list
            (cons (count-words-in-defun) lengths-list)))
    (kill-buffer buffer)
    lengths-list)))
```

La función pasa un argumento, el nombre del fichero en el que se trabajará. Eso tiene cuatro líneas de documentación, pero sin especificación interactiva. Para evitar la preocupación de si se ha estropeado el programa o no, la primera línea del cuerpo es un mensaje de aviso.

La siguiente línea contiene un `save-excursion` que devuelve a Emacs la atención al actual búffer cuando la función se completa. Esto es útil en caso de embeber esta función en otra función que presume que el punto restaura el búffer original.

En la varlist de la expresión `let`, Emacs encuentra el fichero y ajusta la variable local `buffer` al búffer conteniendo el fichero. Al mismo tiempo, Emacs crea `lengths-list` como una variable local.

Lo siguiente, Emacs cambia su atención al búffer.

En la siguiente línea, Emacs crea el búffer de solo lectura. Idealmente, esta línea no es necesaria. Ninguna de las funciones para contar palabras y símbolos en una definición de función cambiaría el búfer. Debajo, el búffer no está yendo para ser guardado, incluso si eso fuese cambiado. Esta línea es enteramente la consecuencia grata, quizás excesiva precaución. La razón para la precaución es que esta función y esta se llama a trabajar en las fuentes para Emacs y eso es un inconveniente si ellos están inadvertidamente modificadas. Eso va sin decir que no se realizó una necesidad para esta línea hasta que un experimento fué cambiado hacia un lado y empezó a modificar mis ficheros de fuentes Emacs . . .

Lo siguiente llama a alargar el búffer si eso está encogido. Esta función es normalmente innecesaria — Emacs crea un búffer fresco si ninguno ya existe; pero si un búffer está visitando el fichero que ya existe que Emacs devuelve uno. En este caso, el búffer puede ser encogido y debe ser amplio. Si se quiere ser completamente 'amigo del usuario', se pondría en orden para guardar la restricción y la localización del punto, pero no.

La expresión `(goto-char (point-min))` mueve el punto al principio del búffer.

Entonces llega un bucle `while` en el que se devuelve el 'trabajo' de la función. En el bucle, Emacs determina el tamaño de cada definición y construye una lista de tamaños conteniendo la información.

Emacs corta el búffer después de trabajar a través de eso. Esto es guardar espacio dentro de Emacs. Mi versión de GNU Emacs 19 contenía 300 ficheros fuente

de interés; GNU Emacs 22 contiene a través de un millar de ficheros fuente. Otra función aplicará `lengths-list-file` a cada uno de los ficheros.

Finalmente, la última expresión con la expresión `let` es la variable `lengths-list`; su valor es devuelto como el valor de la función completa.

Se puede probar esta función instalándolo en el modo usual. Entonces posiciona tu cursor después de la siguiente expresión y escribe *C-x C-e* (`eval-last-sexp`).

```
(lengths-list-file
 "/usr/local/share/emacs/22.1.1/lisp/emacs-lisp/debug.el")
```

(Se puede necesitar cambiar la ruta del fichero; el único aquí es para GNU Emacs versión 22.1.1. Para cambiar la expresión, cópialo al búffer `*scratch*` y edítalo.

(También, para ver el tamaño completo de la lista, en vez de una versión truncada se puede tener que evaluar lo siguiente:

```
(custom-set-variables '(eval-expression-print-length nil))
```

(Véase Sección 16.2 "Especificando Variables usando `defcustom`", página 192. Entonces evalúa la expresión `lengths-list-file`.)

La lista de tamaños para `debug.el` toma menos de un segundo para producirse esto en GNU Emacs 22:

```
(83 113 105 144 289 22 30 97 48 89 25 52 52 88 28 29 77 49 43 290 232 587)
```

(Usando mi vieja máquina, la versión 19 lista tamaños para `debug.el` que en siete segundos para producen esto:

```
(75 41 80 62 20 45 44 68 45 12 34 235)
```

(La versión nueva de `debug.el` contiene más defuns que la temprana; y mi nueva máquina es más rápida que la vieja.

Nótese que el tamaño de la última definición en el fichero es el primero en la lista.

14.7 Contar palabras en `defuns` en diferentes ficheros

En la sección previa, se creaba una función que devuelve una lista de los tamaños de cada definición en un fichero. Ahora, se quiere definir una función para devolver una lista maestra de los tamaños de las definiciones en una lista de ficheros.

Trabajar en cada elemento de una lista de ficheros es un acto repetitivo, así se puede usar un bucle `while` o recursión.

El diseño usando un bucle `while` es rutina. El argumento que se pasaba a la función es una lista de ficheros. Como se vió pronto (véase Sección 11.1.1 "Ejemplo de bucle", página 112), se puede escribir un bucle `while` de un modo que el cuerpo del bucle es evaluado si tal lista contiene elementos, pero sale del bucle si la lista está vacía. Para que este diseño funcione, el cuerpo del bucle debe contener una expresión que ordene la lista cada vez que el cuerpo es evaluado, de modo que finalmente la lista esté vacía. La técnica normal es asignar el valor de la lista para el valor del CDR de la lista cada vez que el cuerpo es evaluado.

La plantilla se ve así:

```
(while test-whether-list-is-empty
  body...
  set-list-to-cdr-of-list)
```

También, recuérdanos que un bucle `while` devuelve `nil` (el resultado de evaluar el true-or-false-test), no el resultado de cualquier evaluación con su cuerpo. (Las evaluaciones con el cuerpo del bucle son hechas para sus efectos laterales.) Sin embargo, la expresión que asigna la lista de tamaños es parte del cuerpo — y que es el valor que queremos devuelto por la función como un todo. Para hacer esto cerramos el bucle `while` con una expresión `let`, y pone en orden que el último elemento de la expresión `let` contiene el valor de lista de tamaños. (Véase "El Ejemplo del Bucle con un Contador de Incremento", página 115.)

Estas consideraciones lideran directamente a la función en sí:

```
;;; Usar bucle while.
(defun lengths-list-many-files (list-of-files)
  "Devuelve la lista de tamaños de funciones en LIST-OF-FILES."
  (let (lengths-list)

;;; true-or-false-test
    (while list-of-files
      (setq lengths-list
            (append
             lengths-list

;;; Genera una lista de tamaños.
              (lengths-list-file
               (expand-file-name (car list-of-files)))))

;;; Crea una lista ordenada de ficheros.
      (setq list-of-files (cdr list-of-files)))

;;; Devuelve la lista final de valores de tamaños.
    lengths-list))
```

`expand-file-name` es una función construida que convierte un nombre de fichero al absoluto, forma de nombre de ruta. La función emplea el nombre del directorio en el que la función se llama.

De este modo, si `expand-file-name` es llamado en `debug.el` cuando Emacs está visitando el directorio `/usr/local/share/emacs/22.1.1/lisp/emacs-lisp/debug.el` llega a ser

```
/usr/local/share/emacs/22.1.1/lisp/emacs-lisp/debug.el
```

El único nuevo elemento de esta definición de función es la todavía no estudiada función `append`, que merece una corta sección en sí.

14.7.1 La función `append`

La función `append` adjunta una lista a otra. De este modo,

```
(append '(1 2 3 4) '(5 6 7 8))
```

produce la lista

```
(1 2 3 4 5 6 7 8)
```

Esto es exactamente cómo queremos adjuntar dos listas de tamaños producidas por `lengths-list-file` a cualquier otra. Los resultados contrastan con `cons`,

```
(cons '(1 2 3 4) '(5 6 7 8))
```

que construye una nueva lista en el que el primer argumento para `cons` llega a ser el primer elemento de la nueva lista:

```
((1 2 3 4) 5 6 7 8)
```

14.8 Recursivamente cuenta palabras en diferentes ficheros

Bajo un bucle `while`, se puede trabajar cada lista de ficheros con recursión. Una versión recursiva de `lengths-list-many-files` es corta y simple.

La función recursiva tiene las partes normales: el 'do-again-test', la 'next-step-expression', y la llamada recursiva. El 'do-again-test' determina si la función se llamaría, que hará si la `list-of-files` contiene los elementos que permanecen; la 'next-step-expression' resetea el `list-of-files` al CDR en sí, así finalmente la lista será vacía; y la llamada recursiva llama en sí a la lista ordenada. ¡La función completa está ordenada por esta descripción!

```
(defun recursive-lengths-list-many-files (list-of-files)
  "Devuelve la lista de tamaños de cada defun en LIST-OF-FILES."
  (if list-of-files                     ; do-again-test
      (append
       (lengths-list-file
        (expand-file-name (car list-of-files)))
       (recursive-lengths-list-many-files
        (cdr list-of-files)))))
```

En una frase, la función devuelve de tamaños de la lista para la el primero de la `list-of-files` al resultado de llamarse así mismo al resto de `list-of-files`.

Aquí hay un test `recursive-lengths-list-many-files`, a lo largo de los resultados de ejecutar `lengths-list-file` en cada uno de los ficheros individualmente.

Instala `recursive-lengths-list-many-files` y `lengths-list-file`, si es necesario, y entonces evalúa las siguientes expresiones. Se puede necesitar cambiar las rutas de ficheros; aquí se trabaja cuando este fichero Info y las fuentes de Emacs están localizadas en sus lugares personales. Para cambiar las expresiones, cópialas al búffer `*scratch*`, edítalos y entonces evalúalos.

Los resultados son mostrados después del '⇒'. (Estos resultados son para ficheros de Emacs versión 22.1.1; ficheros desde otras versiones de Emacs puede producir diferentes resultados.)

```
(cd "/usr/local/share/emacs/22.1.1/")

(lengths-list-file "./lisp/macros.el")
     ⇒ (283 263 480 90)

(lengths-list-file "./lisp/mail/mailalias.el")
     ⇒ (38 32 29 95 178 180 321 218 324)

(lengths-list-file "./lisp/makesum.el")
     ⇒ (85 181)
```

```
(recursive-lengths-list-many-files
 '("./lisp/macros.el"
   "./lisp/mail/mailalias.el"
   "./lisp/makesum.el"))
     ⇒ (283 263 480 90 38 32 29 95 178 180 321 218 324 85 181)
```

La función `recursive-lengths-list-many-files` produce la salida que queremos.

El siguiente paso es preparar el dato en la lista para mostrarlo en un grafo.

14.9 Preparar los datos para mostrarlos en un grafo

La función `recursive-lengths-list-many-files` devuelve una lista de números. Cada número graba el tamaño de una definición de función. Lo que se necesita hacer ahora es transformar estos datos dentro de una lista de números ajustado para generar un grafo. La nueva lista contará cuántas definiciones de funciones contienen menos de 10 palabras y símbolos, cuantas contienen entre 10 y 19 palabras y símbolos, cuántas contienen entre 20 y 29 palabras y símbolos, y así.

En breve, se necesita ir a través el tamaño de la lista producida por la función `recursive-lengths-list-many-files` y contar el número de defuns con cada rango de tamaños, y produce una lista de esto números.

Basado en lo que hemos hecho antes, se prevee que no sería difícil escribir una función que 'CDRs' bajo la lista de tamaños, parece en cada elemento, determina que rango de tamaños está dentro, e incrementa un contador para este rango.

Sin embargo, antes de empezar a escribir tal función, nosotros consideraríamos las ventajas de ordenar los tamaños de la lista primero, así los números son ordenados desde el más pequeño al más largo. Primero, ordenando hará fácil contar los números en cada rango, desde dos números adyacentes será el mismo rango del tamaño en rangos adyacentes. Segundo, inspeccionando una lista ordenada, se puede descubrir el número mayor y menor, y esto significa determinar el rango de tamaño mayor y menor que necesitará.

14.9.1 Ordenando listas

Emacs contiene una función para listas ordenadas, llamadas (como se podría adivinar) `sort`. La función `sort` toma dos argumentos, la lista es ordenada, y un predicado que determina si la primera de dos elementos de lista es "menor" que la segunda.

Como se vió antes (véase Sección 1.8.4 "Usando el Tipo Incorrecto de Objeto como un Argumento", página 13), un predicado es una función que determina si alguna propiedad es verdadera o falsa. La función `sort` reordenará una lista de acuerdo a lo que la propiedad del predicado usa; esto significa que `sort` puede ser usado para ordenar listas no numéricas por un criterio no numérico — eso puede, por ejemplo, alfabetizar una lista.

La función < es usado cuando ordenas una lista numérica. Por ejemplo,

```
(sort '(4 8 21 17 33 7 21 7) '<)
```

produce esto:

```
(4 7 7 8 17 21 21 33)
```

(Note que en este ejemplo, ambos argumentos son citados así que los símbolos no son evaluados antes de ser pasados para `sort` como argumentos.)

Ordenando la lista devuelta por la función `recursive-lengths-list-many-files` es honesta; eso usa la funciń <:

```
(sort
 (recursive-lengths-list-many-files
  '("./lisp/macros.el"
    "./lisp/mailalias.el"
    "./lisp/makesum.el"))
 '<)
```

que produce:

```
(29 32 38 85 90 95 178 180 181 218 263 283 321 324 480)
```

(Note que en este ejemplo, el primer argumento para `sort` no está citado, desde que la expresión debe ser evaluado así como producir la lista que es pasada para `sort`.)

14.9.2 Creando una lista de ficheros

La función `recursive-lengths-list-many-files` requiere una lista de fichero como su argumento. Para nuestros ejemplos de test, se construyeron tal como una lista a mano; pero el directorio fuente de Emacs Lisp es demasiado largo para nosotros para hacer esto. En vez, se escribirá una función para hacer el trabajo para nosotros. En esta función, usaremos ambos un bucle `while` y una llamada recursiva.

Nosotros no tuvimos que escribir una función como esta para viejas versiones de GNU Emacs, desde que todos los ficheros '`.el`' en un directorio. En vez deeso, seremos capaces de usar la función `directory-files`, que lista los nombres de fichero que asocian un patrón específicos con un directorio simple.

Sin embargo, las versiones reciente Emacs emplazan fichero de Emacs Lisp en subdirectorios del directorio de alto nivel `lisp`. Esto facilita la navegación. Por ejemplo, todo los ficheros de correo relacionados están en subdirectorio llamado `mail`. Pero al mismo tiempo, este facilidad nos fuerza a crear un fichero listando la función que desciende dentro de los subdirectorios.

Se puede crear esta función, llamada `files-en-below-directory`, usando funciones familiares tales como `car`, `nthcdr`, y `substring` en conjunción con una función existente llamada `directory-files-and-attributes`. Esta última función no solo listas de ficheros en un directorio, incluyendo los nombres de subdirectorios, pero también sus atributos.

Para empezar nuestro objetivo: crear una función que nos permita alimentas ficheros a `recursive-lengths-list-many-files` como una lista que se parece a esto (pero con más elementos):

```
("./lisp/macros.el"
 "./lisp/mail/rmail.el"
 "./lisp/makesum.el")
```

La función `directory-files-and-attributes` devuelve una lista de listas. Cada una de las listas con la lista principal consiste de 13 elementos. El primer elemento es una cadena que contiene el nombre del fichero – que, en GNU/Linux, puede ser un 'directorio fichero', que dice, un fichero con los atributos especiales de un directorio. El segundo elemento de la lista es `t` para un directorio, una cadena para el enlace simbólico (la cadena es el nombre enlazado), o `nil`.

Por ejemplo, el primer fichero '`.el`' en el directorio es `abbrev.el`. Su nombre es `/usr/local/share/emacs/22.1.1/lisp/abbrev.el` y no es un directorio o un enlace simbólico.

Esto es cómo `directory-files-and-attributes` lista este fichero y sus atributos:

```
("abbrev.el"
nil
1
1000
100
(20615 27034 579989 697000)
(17905 55681 0 0)
(20615 26327 734791 805000)
13188
"-rw-r--r--"
nil
2971624
773)
```

Por otro lado, `mail/` es un directorio con el directorio `lisp/`. El principio de su listado se parece a esto:

```
("mail"
t
...
)
```

(Para aprender acerca de los diferente atributos, mira en la documentacióon de `file-attributes`. Tener en mente que la función `file-attributes` no lista el nombre del fichero, así su primer elemento es `directory-files-and-attributes` es el segundo elemento.)

Se querrán nuestras nuevas funciones, `files-in-below-directory`, para listar los fichero '`.el`' en el directorio eso es contado para chequear, y en los directorios bajo este directorio.

Esto nos da una sugestión de como construir `files-in-below-directory`: con un directorio, la función añadir los nombres de ficheros '`.el`' a una lista; y si, con un directorio, la función viene con un subdirectorio, iría dentro de este subdirectorio y repite sus acciones.

Sin embargo, nosotros notaríamos que cada directorio contiene un nombre que se refiere a sí mismo, llamado `.`, ("dot") y un nombre que se refiere a su directorio padre, llamado `..` ("doble punto"). (En `/`, el directorio raíz, `..` se refiere así mismo, desde que `/` no tiene padre.) Claramente, no que se quiere nuestra función `files-in-below-directory` para introducir estos directorio, desde que ellos siempre nos lideran, directamente o indirectamente, al directorio actual.

Consecuentemente, nuestra función `files-in-below-directory` debe hacer varias tareas:

- Chequea para ver si eso está mirando en un nombre de fichero que finaliza en '`.el`' y si así, añade su nombre a una lista.

- Chequee para ver si está mirando en un nombre de fichero que es el nombre de un directorio; y si es así,

  - Chequee para ver si está mirnado en `.` o `..`; y si es así sal.

  - O además, ve dentro de este directorio y repite el proceso.

Permita escribir una definición de función para hacer estas tareas. Se usará un bucle `while` para mover de un nombre de fichero a otro con un directorio chequeando lo que necesita ser hecho; y usaremos una llamada recursiva para repetir las acciones en cada subdirectorio. El patrón recursivo es 'acumular' (véase "Patrón recursivo: *accumulate*", página 130) usando `append` para combinar.

Aquí está la función:

```
(defun files-in-below-directory (directory)
  "Lista los fichero .el en DIRECTORIO y en sus subdirectorios."
  ;; Aunque la función será usada no interactivamente,
  ;; será fácil chequear si lo hacemos interactivo.
  ;; El directorio tendrá un nombre tal como
  ;;  "/usr/local/share/emacs/22.1.1/lisp/"
  (interactive "DNombre del Directorio: ")
  (let (el-files-list
        (current-directory-list
         (directory-files-and-attributes directory t)))
    ;; mientras estamos en el directorio actual
    (while current-directory-list
      (cond
       ;; chequee para ver si el nombre del fichero finaliza en '.el'
       ;; y si es así, añade su nombre a una lista.
       ((equal ".el" (substring (car (car current-directory-list)) -3))
        (setq el-files-list
              (cons (car (car current-directory-list)) el-files-list)))
       ;; chequee si el nombre del fichero es un directorio
       ((eq t (car (cdr (car current-directory-list))))
        ;; decide si escapar o recurrir
        (if
            (equal "."
                   (substring (car (car current-directory-list)) -1))
            ;; entonces no hagas nada desde que el nombre del fichero es
            ;; actual directorio o padre, "." o ".."
            ()
```

```
        ;; else desciende dentro del directorio y repite el proceso
        (setq el-files-list
              (append
               (files-in-below-directory
                (car (car current-directory-list)))
               el-files-list)))))
      ;; mueve al siguiente fichero en la lista; esto también
      ;; ordena la lista así mientras el bucle
      ;; eventualmente llega a un fin
      (setq current-directory-list (cdr current-directory-list)))
    ;; devuelve los ficheros
    el-files-list))
```

Las función `files-in-below-directory` `directory-files` toma un argumento, el nombre de un directorio.

De este modo, en mi sistema,

```
(length
 (files-in-below-directory "/usr/local/share/emacs/22.1.1/lisp/"))
```

cuéntame que dentro y debajo de mi directorio de fuentes Lisp hay 1031 ficheros '`.el`'

`files-in-below-directory` devuelve una lista en orden alfabético inverso. Una expresión para ordenar la lista en orden que parece como este:

```
(sort
 (files-in-below-directory "/usr/local/share/emacs/22.1.1/lisp/")
 'string-lessp)
```

14.9.3 Contando definiciones de función

Nuestro objetivo inmediato es generar una lista que cuenta, cuantas definiciones de funciones contienen menos de 10 palabras y símbolos, cuantas contienen entre 10 y 19 palabras y símbolos, cuantos contienen entre 20 y 29 palabras y símbolos, y así.

Con una lista ordenada de números, esto es fácil: cuentas cuantos elementos de la lista son más pequeños que 10, entonces, después de mover al pasado los números solo contados, cuentas cuantos son más pequeños de 20, entonces, después de mover pasado los números solo contados, cuenta cuantos son más pequeños de 30, y así. Cada uno de los números, 10, 20, 30, 40, y el como, es uno más largo que el tope de este rango. Se puede llamar a la lista de tales números la lista `top-of-ranges`.

Si se desea, se podría generar esta lista automáticamente, pero es más simple escribir una lista manualmente. Aquí está:

```
(defvar top-of-ranges
 '(10  20  30  40  50
   60  70  80  90 100
  110 120 130 140 150
  160 170 180 190 200
  210 220 230 240 250
  260 270 280 290 300)
 "Listar especificando rangos para 'defuns-per-range'.")
```

Para cambiar los rangos, se edita esta lista.

Lo siguiente, que se necesita es escribir la función que crea la lista del número de definiciones con cada rango. Claramente, esta función debe tomar el `sorted-lengths` y las listas `top-of-ranges` listas como argumentos.

La función `defuns-per-range` debe hacer dos cosas una y otra vez: eso debe contar el número de definiciones con un rango específico por el actual valor top-of-range; y eso debe dividir al siguiente gran valor en la lista `top-of-ranges` después de contar el número de definiciones en el rango actual. Desde que cada una de estas acciones es repetitiva, se puede usar los bucles `while` para el trabajo. Un bucle cuenta el número de definiciones en el rango definido por el valor actual top-of-range, y el otro bucle selecciona cada uno de los valores top-of-range en turno.

Varias entradas de la lista `sorted-lengths` son contadas para cada rango; esto significa que el bucle para la lista `sorted-lengths` será dentro del bucle para la lista `top-of-ranges`, como un pequeño adorno dentro de un gran adorno.

El bucle interno cuenta el número de definiciones con el rango. Eso es un simple contaje del tipo que nosotros hemos visto antes. (Véase Sección 11.1.3 “Un bucle con un contador de incremento”, página 114.) El test true-or-false del bucle chequea si el valor desde la lista `sorted-lengths` es más pequeña que el actual valor de lo alto del rango. Si eso es, la función incrementa el contador y chequea el siguiente valor desde la lista `sorted-lengths`.

El bucle interno se parece a esto:

```
(while length-element-smaller-than-top-of-range
  (setq number-within-range (1+ number-within-range))
  (setq sorted-lengths (cdr sorted-lengths)))
```

El bucle de fuera debe empezar con el valor más bajo de la lista `top-of-ranges`, y entonces asignares a cada uno de los valores superiores exitosos a su vez. Esto puede ser hecho con un bucle como este:

```
(while top-of-ranges
  body-of-loop...
  (setq top-of-ranges (cdr top-of-ranges)))
```

Pon junto, los dos bucles como este:

```
(while top-of-ranges

  ;; Contar el número de elementos con el actual rango.
  (while length-element-smaller-than-top-of-range
    (setq number-within-range (1+ number-within-range))
    (setq sorted-lengths (cdr sorted-lengths)))

  ;; Mover al siguiente rango.
  (setq top-of-ranges (cdr top-of-ranges)))
```

Además, en cada circuito del bucle exterior, Emacs grabaría el número de definiciones con este rango (el valor de `number-within-range`) en una lista. Se puede usar `cons` para este propósito. (Véase Sección 7.2 “`cons`”, página 75.)

La función `cons` trabaja bien, excepto que la lista que se construye contendrá el número de definiciones para el alto rango y su principio el número de definiciones para el más bajo rango a su fin. Esto es porque `cons` adjunta nuevos elementos de la lista al principio de la lista, y desde los dos bucles están trabajando su camino a través de la lista de tamaños desde lo bajo finaliza primero, el `defuns-per-range-`

`list` finalizará el primer número más largo. Pero nosotros querremos imprimir nuestro grafo con pequeños valores primer y el más largo después. La solución es invertir el orden del `defuns-per-range-list`. Nosotros podemos hacer esto usando la función `nreverse`, que invierte el orden de una lista.

Por ejemplo,

```
(nreverse '(1 2 3 4))
```

produce:

```
(4 3 2 1)
```

Note que la función `nreverse` es "destructiva" — que es, cambiar la lista a la que se aplica; esto contrasta con las funciones `car` y `cdr`, que no son destructivas. En este caso, no se quiere que el original `defuns-per-range-list`, de manera que no hay materia que se destruya. (La función `reverse` provee un copia inversa de una lista, dejando la lista original como es.)

Pon todo junto, el `defuns-per-range` se parece a esto:

```
(defun defuns-per-range (sorted-lengths top-of-ranges)
  "funciones de SORTED-LENGTHS en cada rango TOP-OF-RANGES."
  (let ((top-of-range (car top-of-ranges))
        (number-within-range 0)
        defuns-per-range-list)

    ;; Bucle Exterior.
    (while top-of-ranges

      ;; Bucle Interno.
      (while (and
              ;; Necesita el número para el test numérico.
              (car sorted-lengths)
              (< (car sorted-lengths) top-of-range))

        ;; Contar número de definiciones con el rango actual.
        (setq number-within-range (1+ number-within-range))
        (setq sorted-lengths (cdr sorted-lengths)))

      ;; Sal del bucle interno pero permanece con el bucle externo.

      (setq defuns-per-range-list
            (cons number-within-range defuns-per-range-list))
      (setq number-within-range 0)       ; Resetear el contaje a cero.

      ;; Mover al siguiente rango.
      (setq top-of-ranges (cdr top-of-ranges))
      ;; Especifica el siguiente mejor rango de valores.
      (setq top-of-range (car top-of-ranges)))
```

```
    ;; Salir del bucle externo y contar el número de defuns más
  ;; largas que
    ;;   el valor más largo del valor top-of-range.
    (setq defuns-per-range-list
          (cons
           (length sorted-lengths)
           defuns-per-range-list))

    ;; Devuelve una lista del número de definiciones con cada rango,
    ;;  del más pequeño al más largo.
    (nreverse defuns-per-range-list)))
```

La función es simple excepto para una pequeña funcionalidad. El test verdadero-o-falso para el bucle interno se parece a esto:

```
(and (car sorted-lengths)
     (< (car sorted-lengths) top-of-range))
```

en vez de algo como esto:

```
(< (car sorted-lengths) top-of-range)
```

El propósito del test es determinar si el primer ítem en la lista `sorted-lengths` es menor que el valor de lo mejor del rango.

La versión simple del test trabaja bien a menos que la lista `sorted-lengths` tiene un valor `nil`. En este caso, la expresión `(car sorted-lengths)` devuelve `nil`. La función `<` no se puede compara un número a `nil`, que es una lista vacía, así Emacs señala un error y para la función desde el intento de continuar la ejecución.

La lista `sorted-lengths` siempre llega a ser `nil` cuando el contador logra el fin de la lista. Esto significa que cualquier intento de usar la función `defuns-per-range` con la versión simple del test fallará.

Se resuelve el problema usando `(car sorted-lengths)` en conjunción con la expresión `and`. La expresión `(car sorted-lengths)` devuelve un valor no `nil` tan largo como la lista que tiene al menos un número con eso, pero devuelve `nil` si la lista está vacía. La expresión `and` primero evalúa el `(car sorted-lengths)`, y si eso es `nil`, devuelve falso *sin* evaluar la expresión `<` y devuelve este valor como el valor de la expresión `and`.

Este camino, evita un error. (Para información acerca de `and`, ver "La función `kill-new`", página 94.)

Aquí hay un pequeño test de la función `defuns-per-range`. Primero, evalúa la expresión que ajusta (una resumida) lista `top-of-ranges` a la lista de valores, entonces evalúa la expresión para ajustar la lista `sorted-lengths`, y entonces evalúa la función `defuns-per-range`.

```
;; (La lista ordenada que usará después.)
(setq top-of-ranges
 '(110 120 130 140 150
   160 170 180 190 200))

(setq sorted-lengths
      '(85 86 110 116 122 129 154 176 179 200 265 300 300))

(defuns-per-range sorted-lengths top-of-ranges)
```

La lista devuelta se parece a esto:

```
(2 2 2 0 0 1 0 2 0 0 4)
```

Dentro, hay dos elementos de la lista `sorted-lengths` menores de 110, dos elementos entre 110 y 119, dos elementos entre 120 y 129 y así. Hay cuatro elementos con un valor de 200 o superior.

15 Leyendo un grafo

Nuestro objetivo es construir un grafo mostrando los números de definiciones de función de varios tamaños en las fuentes de Emacs lisp.

Como una materia práctica, si se estuviese creando un grafo, probablemente usarías un programa tal como `gnuplot` para hacer el trabajo. (`gnuplot` está bien integrado dentro de GNU Emacs.) En este caso, sin embargo, creamos uno desde cero, y en el proceso lo reconstruimos nosotros mismos con algo de lo que aprendemos antes y así aprender más.

En este capítulo, primero escribiremos un grafo simple imprimiendo función. Esta primera definición será un *prototipo*, una función escrita rápidamente nos permite reconocer este territorio creando un grafo. Nosotros descubriremos dragones, o encontramos que son mitos. Después de olisquear el terreno, nos sentiremos más confidentes y mejoraremos la función para etiquetar las coordenadas automáticamente.

Puesto que Emacs está diseñado para ser flexible y trabajar con todo tipo de terminales, incluyendo los terminales de caracteres, el grafo necesitará ser hecho desde símbolos de 'escritura'. Un asterisco hará; como nosotros mejoramos la función de impresión del grafo, se puede crear la elección del símbolo una opción de usuario.

Se puede llamar a esta función `graph-body-print`; se tomará un `numbers-list` como su único argumento. En esta fase, no se etiquetará el grafo, pero se imprime su cuerpo.

La función `graph-body-print` inserta una columna vertical de asteriscos para cada elemento en la lista `numbers-list`. La altura de cada línea está determinada por el valor de este elemento de la `numbers-list`.

Insertar columnas es un acto repetitivo; que significa que esta función debe ser escrita con un bucle `while` o recursivamente.

Nuestro primer reto es descubrir como imprimir una columna de asteriscos. Normalmente, en Emacs, se imprimen caracteres dentro de una pantalla horizontalmente, línea a línea, escribiendo. Se tienen dos rutas que se pueden seguir: escribir nuestra función column-insertion o descubrir si una que exista en Emacs.

Para ver si hay uno en Emacs, se puede usar el comando *`M-x apropos`*. Este comando es como el comando *`C-h a`* (`command-apropos`), excepto que último encuentra solo estas funciones que son comandos. El comando *`M-x apropos`* lista todos los símbolos que se asocian a una expresión regular, incluyendo funciones que no son interactivas.

Lo que queremos buscar es algún comando que imprima o inserte columnas. Muy probablemente, el nombre de la función contendrá la palabra 'print' o la palabra 'insert' o la palabra 'column'. Por esta razón, podemos simplemente escribir *`M-x apropos RET print \|insert\|column RET`* y mira el resultado. En mi sistema, este comando toma todavía algún tiempo, y entonces produjo una lista de 79 funciones y variables. Ahora no toma mucho en todo y produce una lista de 211 funciones y variables. Escaneando la lista, la única función que mira como si pudiera hacer el trabajo que es `insert-rectangle`.

En realidad, esta es la función que queremos; su documentación dice:

```
insert-rectangle:
Insertar texto de RECTANGLE con la esquina izquierda a punto
La primera línea de RECTANGLE es insertada al punto
su segunda línea es insertada a un punto verticalmente
bajo el punto, etc.
El RECTANGLE debería ser una lista de cadenas.
Después de este comando, la marca está en la esquina izquierda
superior y el punto en la esquina derecha inferior.
```

Se puede ejecutar un test rápido, para asegurar que hace lo que se espera de eso.

Aquí está el resultado de emplazar el cursor después de la expresión `insert-rectangle` y escribiendo *C-u C-x C-e* (`eval-last-sexp`). La función inserta las cadenas '`"primero"`', '`"segundo"`', y '`"tercero"`' en el punto. Tambiés la función devuelve `nil`.

```
(insert-rectangle '("primero" "segundo" "tercer"))primero
                                                  segundo
                                                  terceronil
```

De acuerdo, nosotros no estaremos insertando el texto de la expresión `insert-rectangle` en sí dentro del búffer en el que estamos marcando el grafo, pero llamará la función de nuestro programa. Nosotros, sin embargo, tendremos que asegurar que el punto está en el búffer en el lugar donde la función `insert-rectangle` insertará la columna de cadenas.

Si se está leyendo esto en Info, se puede ver como este trabajo cambia a otro búffer, tal como el búffer `*scratch*`, emplazando el punto a algún lugar en el búffer, escribiendo *M-:*, escribiendo la expresión `insert-rectangle` dentro del minibuffer en la consola, y entonces escribiendo `RET`. Esto causa Emacs para evaluar la expresión en el minibuffer, pero usa como el valor del punto la posición del punto en el búffer `*scratch*`. (*M-:* es el atajo para `eval-expression`. También, `nil` no aparece en el búffer `*scratch*` desde que la expresión es evaluada en el minibuffer.)

Se encuentra cuando hacer esto que punto finaliza al fin de la última línea insertada — esto es decir, que esta función mueve el punto como un efecto lateral. Si nosotros estábamos para repetir el comando, con el punto en esta posición, la siguiente inserción sería debajo y a la derecha de la inserción previa. ¡Nosotros no queremos esto! Si estamos yendo a crear un gráfico de barras, las columnas necesitan estar debajo unas de otras.

Así se descubre que cada ciclo del bucle `while` de column-inserting debe reposicionar el punto al lugar que queremos, y este lugar estará arriba, no abajo, de la columna. Más allá, se recuerda que cuando se imprime un grafo, no esperan todas las columnas para estar a la misma altura. Esto significa que el alto de cada columna puede estar a una altura diferente desde una previa. Nosotros simplemente reposicionamos el punto a la misma línea cada vez, pero movido cubriendo a la derecha — o quizás se puede . . .

Estamos planeando crear las columnas de la barra gráfica sin asteriscos. El número de asteriscos en la columna es el número específico por el elemento actual del `numbers-list`. Necesitamos construir una lista de asteriscos del tamaño derecho para cada llamada a `insert-rectangle`. Si esta lista consiste únicamente del

número requerido de asteriscos, entonces tendremos la posición de punto el número correcto de líneas bajo la base del gráfico para imprimirse correctamente. Esto podría ser difícil.

Alternativamente, si podemos figurarnos algún camino para pasar `insert-rectangle` del mismo tamaño cada vez, entonces podemos posicionar el punto en la misma línea cada vez, pero se mueve a través de una columna a la derecha por cada nueva columna. Si hacemos esto, sin embargo, alguna de las entradas en la lista pasaba a `insert-rectangle` y deben ser espacios en blanco en vez de asteriscos. Por ejemplo, si la altura máxima del grafo es 5, pero la altura de la columna es 3, entonces `insert-rectangle` requiere un argumento que se parezca a esto:

```
(" " " " "*" "*" "*")
```

Esta última propuesta no es tan difícil, de hecho se puede determinar la altura de la columna. Hay dos caminos para nosotros especificar la altura de la columna: se puede arbitrariamente situar que será, lo que funcionaría bien para gráficas de esta altura; o podemos buscar a través de la lista de números y usar la altura máxima de la lista como la altura máxima del grafo. Si la última operación fuera difícil, entonces el procedimiento formal sería fácil, pero hay una función construida en Emacs para determinar el máximo de sus argumentos. Se puede usar esta función. La función se llamaba `max` y eso devuelve el mayor de sus argumentos, que deben ser números. De este modo, por ejemplo,

```
(max  3 4 6 5 7 3)
```

devuelve 7. (Una función correspondiente llamada `min` devuelve lo más pequeño de todos sus argumentos.)

Sin embargo, no podemos simplemente llama a `max` en el `numbers-list`; la función `max` espera números como sus argumentos, no una lista de números. De este modo, la siguiente expresión,

```
(max  '(3 4 6 5 7 3))
```

produce el siguiente mensaje error;

```
Mal tipo de argumento: number-or-marker-p, (3 4 6 5 7 3)
```

Se necesita una función que pasa una lista de argumentos a una función. Esta función es `apply`. Esta función 'aplica' su primer argumento (una función) para los argumentos que permanecen, el último puede ser una lista.

Por ejemplo,

```
(apply 'max 3 4 7 3 '(4 8 5))
```

devuelve 8

(Incidentalmente, yo no sabría cómo aprender acerca de esta función sin un libro tal como este. Eso es posible descubrir otras funciones, como `search-forward` o `insert-rectangle`, adivinando una parte de sus nombres y entonces usando `apropos`. Incluso aunque su base metafórica es clara — 'apply' su primer argumento al resto — dudo que un novicio vendría con esta palabra particular usando `apropos` u otra ayuda. De acuerdo, podría ser incorrecto; después de todo, la función fué primero llamada por alguien quien lo había inventado.

El segundo y subsiguientes argumentos para `apply` son opcionales, así se puede usar `apply` para llamar a una función y pasan los elementos de una lista, como este, que también devuelve 8:

```
(apply 'max '(4 8 5))
```

Este camino tardío es como se usará `apply`. La función `recurisive-lengths-list-many-files` devuelve una lista de números que se puede aplicar a `max` (nosotros podríamos también aplicar *(hacer apply)* a la lista de números ordenados; eso no importa si la lista está o no.)

Aquí, la operación para encontrar el tamaño máximo del grafo es este:

```
(setq max-graph-height (apply 'max numbers-list))
```

Ahora se puede devolver la pregunta de como crear una lista de cadenas para una columna del grafo. Cuenta la máxima altura del grafo y el número de asteriscos que aparecerían en la columna, la función devolverá una lista de cadenas para el comando a insertar `insert-rectangle`.

Cada columna es realiza con asteriscos o espacios en blanco. Puest que la función pasa el valor del alto de la columna y el número de asteriscos en el columna, el número de espacios en blanco puede ser encontrado sustrayendo el número de asteriscos desde lo alto de la columna. Dado el número de espacios en blanco y el número de asteriscos, dos bucles `while` puede ser usado para construir la lista:

```
;;; Primera versión.
(defun column-of-graph (max-graph-height actual-height)
  "Devuelve la lista de cadenas que una columna de un grafo."
  (let ((insert-list nil)
        (number-of-top-blanks
         (- max-graph-height actual-height)))

    ;; Llenar asteriscos.
    (while (> actual-height 0)
      (setq insert-list (cons "*" insert-list))
      (setq actual-height (1- actual-height)))

    ;; Rellena espacios en blanco.
    (while (> number-of-top-blanks 0)
      (setq insert-list (cons " " insert-list))
      (setq number-of-top-blanks
            (1- number-of-top-blanks)))

    ;; Devuelve la lista completa.
    insert-list))
```

Si se instala esta función y entonces evalúa la siguiente expresión se verá que devuelve la lista como se desea:

```
(column-of-graph 5 3)
```

devuelve

```
(" " " " "*" "*" "*")
```

Como está escrito, `column-of-graph` contiene una grieta mayor: los símbolos usados para el espacio en blanco para las entradas marcadas en la columna son 'codificadas duras' como un espacio y un asterisco. Esto está bien para un prototipo, pero tu, u otro usuario, puede desear usar otros símbolos. Por ejemplo,

chequeando la función grafo, tu quieres usar un periodo en vez del espacio, asegura el punto que está siendo reposicionando apropiadamente cada vez que la función `insert-rectangle` se llama; o se podría querer sustituir un signo '+' u otro símbolo para el asterisco. Se podría incluso querer hacer un graph-column que es más que un ancho de columna. El programa debería ser más flexible. El camino para hacer esto es reemplazar el espacio en blanco y el asterisco con dos variables que se puede llamar `graph-blank` y `graph-symbol` y define estas variables separadamente.

También la documentación no está escrita. Estas consideraciones nos llevan también a la segunda versión de la función:

```
(defvar graph-symbol "*"
  "Cadena usada como símbolo en grafo, normalmente un asterisco.")

(defvar graph-blank " "
  "La cadena como un espacio en blanco en grafo, normalmente un
  espacio en blanco.
graph-blank debe ser el mismo número de columnas amplio como graph-symbol.")
```

(Para una explicación de `defvar`, ver Sección 8.5 "Inicializando una Variable con `defvar`", página 100.)

```
;;; Segunda versión.
(defun column-of-graph (max-graph-height actual-height)
  "Devuelve cadenas MAX-GRAPH-HEIGHT; ACTUAL-HEIGHT son
  símbolos de grafos.

Los graph-symbols son entradas contiguo al fin de la lista.
La lista será insertado como una columna de un grafo.
Las cadenas son tanto graph-blank o graph-symbol.

  (let ((insert-list nil)
        (number-of-top-blanks
         (- max-graph-height actual-height)))

    ;; Rellenar en graph-symbols.
    (while (> actual-height 0)
      (setq insert-list (cons graph-symbol insert-list))
      (setq actual-height (1- actual-height)))

    ;; Rellenar en graph-blanks.
    (while (> number-of-top-blanks 0)
      (setq insert-list (cons graph-blank insert-list))
      (setq number-of-top-blanks
            (1- number-of-top-blanks)))

    ;; Devuelve la lista completa.
    insert-list))
```

Si se desea, podríamos reescribir `column-of-graph` una tercera vez para proporcionar opcionalmente un gráfico de líneas, como gráfico de barras. Esto no sería duro de hacer. Un camino para pensar en un grafo de líneas es que no es más que un grafo de barras en el que la parte de cada barra que está debajo del alto es blanco. Para construir una columna para gráfico de líneas, la función primero construyen

una lista de espacios en blanco que es una más ordenada que el valor, entonces usa `cons` para adjuntar un símbolo gráfico a la lista; entonces eso usa `cons` de nuevo para adjuntar el 'alto de espacios en blanco' a la lista.

Es fácil ver como escribir tal función, pero desde que no se necesita eso, no se hará. Pero el trabajo podría ser hecho, y si eso fuera hecho, eso sería hecho con `column-of-graph`. Incluso más importante, no se valora nada más que pocos cambios que tendría que ser hecho de cualquier otra manera. La mejora, si nosotros siempre deseamos hacerla, es simple.

Ahora, finalmente, volvemos a nuestra primera función de grafo impresa. Esto imprime el cuerpo de un grafo, no las etiquetas para los ejes horizontal y vertical, así se puede llamar este `graph-body-print`.

15.1 La función `graph-body-print`

Después de nuestra preparación en la sección precedente, la función `graph-body-print` es simple. La función imprimirá la columna después de la columna de asteriscos y espacios en blanco, usando los elementos de la lista de números para especificar el número de asteriscos en cada columna. Esto es un acto repetitivo, que significa que se puede usar un bucle `while` que decrementa o una función recursiva para el trabajo. En esta sección, se escribirá la definición usando un bucle `while`.

La función `column-of-graph` requiere el alto del grafo como un argumento, así se asigna y guarda esto como una variable local.

Esto lidera a la siguiente plantilla para el bucle `while` versión de esta función:

```
(defun graph-body-print (numbers-list)
  "documentation..."
  (let ((height  ...
         ...))

    (while numbers-list
      insert-columns-and-reposition-point
      (setq numbers-list (cdr numbers-list)))))
```

Necesitamos rellenar los slots de la plantilla.

Claramente, se puede usar la expresión `(apply 'max numbers-list)` para determinar el alto del grafo.

El bucle `while` iterará a través de `numbers-list` un elemento a la vez. Como eso está ordenado por la expresión `(setq numbers-list (cdr numbers-list))`, el CAR de cada instancia de la lista es el valor del argumento para `column-of-graph`.

En cada ciclo del bucle `while`, la función `insert-rectangle` inserta la lista devuelta por `column-of-graph`. Desde que la función `insert-rectangle`, se necesita guardar la localización de punto al tiempo que el rectángulo se inserta, mueve atrás a esta posición después de que el rectángulo es insertado, y entonces se mueve horizontalmente al siguiente lugar desde el que `insert-rectangle` se llama.

Si las columnas se insertan en un carácter amplio, será si los espacios en blanco y asteriscos se usan, el comando de reposición es simple `(forward-char 1)`; sin embargo, el ancho de una columna puede ser más grande que uno. Esto significa que el comando de reposicionamiento sería escrito `(forward-char symbol-width)`. El

mejor lugar para asociar la variable `symbol-width` al valor del width de la columna grafo está en la varlist de la expresión `let`.

Estas consideraciones lideran a la siguiente definición de función:

```
(defun graph-body-print (numbers-list)
  "Imprime un gráfico de barras de la NUMBERS-LIST.
La numbers-list consiste en los valores del eje Y."

  (let ((height (apply 'max numbers-list))
        (symbol-width (length graph-blank))
        from-position)

    (while numbers-list
      (setq from-position (point))
      (insert-rectangle
       (column-of-graph height (car numbers-list)))
      (goto-char from-position)
      (forward-char symbol-width)
      ;; Dibuja la columna del grafo por columna.
      (sit-for 0)
      (setq numbers-list (cdr numbers-list)))
    ;; Emplaza el punto para etiquetas de ejes X.
    (forward-line height)
    (insert "\n")
))
```

La expresión inesperada en esta función es la expresión `(sit-for 0)` en el bucle `while`. Esta expresión hace que el grafo imprima la operación más interesante para vigilar lo que sería de otro modo. La expresión causa que Emacs pare (sit-for 0) para un momento cero y entonces redibuje la pantalla. Puesto aquí, eso causa que Emacs redibuje la pantalla columna por columna. Sin eso, Emacs no redibujaría la pantalla hasta que la función exista.

Se puede chequear `graph-body-print` con una corta lista de números.

1. Instala `graph-symbol`, `graph-blank`, `column-of-graph`, que están en Capítulo 15 "Leyendo un grafo", página 182, and `graph-body-print`.
2. Copia la siguiente expresión:

   ```
   (graph-body-print '(1 2 3 4 6 4 3 5 7 6 5 2 3))
   ```

3. Cambia al búffer `*scratch*` y emplaza el cursor donde quiere que el grafo empiece.
4. Escribe *M-:* (`eval-expression`).
5. Pega la expresión `graph-body-print` dentro del minibuffer con *C-y* (`yank`).
6. Presiona `RET` para evaluar la expresión `graph-body-print`

Emacs imprimirá un grafo como este:

```
                *
              *   **
              *  ****
             *** ****
            ********* *
           ************
          *************
```

15.2 La función `recursive-graph-body-print`

La función `graph-body-print` puede también ser escrito recursivamente. La solución recursiva es dividida dentro de dos partes: una fuera 'wrapper' *envoltorio* que usa una expresión `let` para determinar los valores varias variables que solo necesitan ser encontradas una vez, tal como la máxima altura del grafo, y una función dentro que es llamada recursivamente para imprimir el grafo.

El 'envoltorio' no es complicado:

```
(defun recursive-graph-body-print (numbers-list)
  "Imprime un gráfico de barras del NUMBERS-LIST.
El numbers-list consiste en los valores del eje Y."
  (let ((height (apply 'max numbers-list))
        (symbol-width (length graph-blank))
        from-position)
    (recursive-graph-body-print-internal
     numbers-list
     height
     symbol-width)))
```

La función recursiva es un poco más difícil. Eso tiene cuatro partes: el 'do-again-test', el código impreso, la llamada recursiva, y la 'next-step-expression'. El 'do-again-test' es una expresión `when` que determina si el `numbers-list` contiene cualquier elementos que permanecen; si hace eso, la función imprime una columna del grafo usando el código impreso y se llama así mismo de nuevo. La función llama así misma de nuevo de acuerco al valor producido por la 'next-steup-expression' que causa para llamar a actuar en una versión ordenada de la `numbers-list`.

```
(defun recursive-graph-body-print-internal
  (numbers-list height symbol-width)
  "Imprime un gráfico de barras.
Usado con la función recursive-graph-body-print."

  (when numbers-list
        (setq from-position (point))
        (insert-rectangle
         (column-of-graph height (car numbers-list)))
        (goto-char from-position)
        (forward-char symbol-width)
        (sit-for 0)     ; Dibuja un gráfico columna por columna.
        (recursive-graph-body-print-internal
         (cdr numbers-list) height symbol-width)))
```

Después la siguiente instalación, esta expresión puede ser testeado; aquí hay un ejemplo:

```
(recursive-graph-body-print '(3 2 5 6 7 5 3 4 6 4 3 2 1))
```

Aquí está lo que `recursive-graph-body-print` produce:

```
                *
               **   *
              ****  *
              **** ***
            * *********
            ************
            *************
```

Cada una de estas dos funciones, `graph-body-print` o `recursive-graph-body-print`, crea el cuerpo de un grafo.

15.3 Necesidad para ejes impresos

Un grafo necesita ejes impresos, así se puede orientar a tí mismo. Para un proyecto do-once, eso puede ser razonable dibujar los ejes a mano usando el modo de emacs Picture, pero un grafo dibuja la función que puede ser usada más de una vez.

Por esta razón, se han escrito mejoras a la función básica `print-graph-body` que automáticamente imprime etiquetas para los ejes horizontal y vertical. Desde la etiqueta de imprimir funciones no contiene mucho material nuevo, se ha emplazado su descripción en un apéndice Véase Apéndice C "Un Grafo con Ejes Etiquetados", página 227.

15.4 Ejercicio

Escribe una versión de línea de grafo de la funciones de impresión del grafo.

16 Tu fichero `.emacs`

"No te tiene que gustar Emacs para lo que te gusta" — esto que parece una frase paradójica es el secreto de GNU Emacs. En realidad, Emacs es una herramienta genérica. La mayoría de la gente que usa Emacs, lo personaliza para ajustarlo a sus necesidades.

GNU Emacs está mayoritariamente escrito en Emacs Lisp; este significa que escribiendo expresiones en Emacs Lisp se puede modificar o extender Emacs.

Hay quien aprecia la configuración por defecto de Emacs. Después de todo, Emacs empieza en modo C cuando se edita un fichero C, empieza en modo Fortran cuando se edita un fichero Fortran, y empieza en modo Fundamental cuando se edita un fichero no adornado. Esto tiene sentido, si no sabes quien está yendo a usar Emacs. ¿Quién sabe lo que una persona espera hacer con un fichero no adornado? El modo fundamental es el modor correcto por defecto para tal fichero, tal como el modo C es lo correcto para editar código C. (Suficientes lenguajes de programación tienen sintaxis que permiten compartir funcionalidades, tal como el modo C es ahora proporcionado por el modo CC, la 'Colección C'.)

Pero cuando se conoce quien está yendo a usar Emacs — tu, tu mismo – entonces eso tiene sentido para personalizar Emacs.

Por ejemplo, yo raramente quiero el modo Fundamental cuando edito un fichero de otro modo no distinguido; yo quiero el modo Texto. Esto es por lo que yo personalizo Emacs: así eso se ajusta a mí.

Se puede personalizar y extender Emacs escribiendo o adaptando un fichero `~/.emacs`. Esto es un fichero de inicialización personal; sus contenidos, escritos en Emacs Lisp, cuentan a Emacs qué hacer.[1]

Un fichero `~/.emacs` contiene código Emacs Lisp. Se puede escribir este código por tí mismo; o se puede usar la funcionalidad `customize` para escribir el código para ti. Se puede combinar tus propias expresiones y expresiones auto-escritas personalizadas en tu fichero `.emacs`.

(Yo prefiero por mí mismo escribir mis propias expresiones, excepto para estas, fuentes particularmente, que se encuentran fáciles de manipular usando el comando `customize`. Yo combino los dos métodos.)

La mayoría de este capítulo es acerca de escribir expresiones por tí mismo. Eso describe un fichero `.emacs` simple; para más información, mira Sección "El Fichero de Inicio" in *El Manual GNU Emacs*, y Sección "El Fichero de Inicio" in *El Manual de Referencia GNU Emacs Lisp*.

[1] Tu puedes también añadir `.el` para `~/.emacs` y llama a un fichero `~/.emacs`. En el pasado, fué prohibido escribir los atajos de teclado extra que el nombre `~/.emacs.el` requiere, pero ahora puedes. El nuevo formato es consistente con las conveniciones de nombre del fichero Emacs Lisp; el viejo formato guarda la escritura.

16.1 Fichero de inicialización site-wide

Además de tu fichero de inicialización personal, Emacs automáticamente carga varios ficheros de inicialización amplios, si existen. Estos tienen la misma forma que tu fichero `.emacs`, pero son cargados por cualquiera.

Dos ficheros de incialización, `site-load.el` y `site-init.el`, están cargados dentro de Emacs y volcados *dumped* sin una versión dumped de Emacs se creó, como es más común. (Las copias dumped de Emacs cargan más rápidamente. Sin embargo, desde que un fichero se carga y compila, un cambio no llega a ser un cambio en Emacs a menos que cargues por tí o recompiles Emacs. Véase Sección "Construyendo Emacs" in *El Manual de Referencia de GNU Emacs Lisp*, y el fichero `INSTALL`)

Los tres otros ficheros de inicialización son cargados automáticamente cada vez que se inicia Emacs, si existen. Esto son `site-start.el`, que es cargado *antes* que tu fichero `.emacs`, y `default.el`, y el tipo de fichero terminal, que son cargados *después* de tu fichero `.emacs`.

Las configuraciones y definiciones en tu fichero `.emacs` sobreescribirán las configuraciones en conflicto y definiciones en un fichero `site-start.el`, si eso existe; pero las configuraciones y definiciones en un `default.el` o tipo de fichero terminal sobreescribirá estos en tu fichero `.emacs`. (Se pueden prevenir interferencias desde un tipo de fichero terminal configurando `term-file-prefix` para `nil`. Véase Sección 16.11 "Una extensión simple", página 202.)

El fichero `INSTALL` que viene en la distribución contiene descripciones de los fichero `site-init.el` y `site-load.el`.

Los ficheros `loadup.el`, `startup.el`, y `loaddefs.el` controlan la carga. Estos ficheros están en el directorio `lisp` de la distribución Emacs y tiene valor de uso.

El fichero `loaddefs.el` contiene buenas sugerencias como las que poner dentro de tu propio fichero `.emacs`, o dentro de un fichero de inicialización amplio.

16.2 Especificar variables usando `defcustom`

Se pueden especificar variables usando `defcustom` así que tu y otros podéis usar la funcionalidad de Emacs `customize` para asignar sus valores. (No se puede usar `customize` para escribir definiciones de función; pero se pueden escribir `defuns` en tu fichero `.emacs`. En vez de eso, se puede escribir cualquier expresión Lisp en tu fichero `.emacs`)

La funcionalidad `customize` depende de la forma especial `defcustom`. Aunque se puede usar `defvar` o `setq` para las variables que los usuarios asignan, la forma especial `defcustom` está diseñada para el trabajo.

Se puede usar tu conocimiento de `defvar` para escribir los primeros tres argumentos para `defcustom`. El primer argumento para `defcustom` es el nombre de la variable. El segundo argumento es el valor inicial de la variable, cualquiera; y este valor es asignado solo si el valor no ha sido ya asignado. El tercer argumento es la documentación.

El cuarto y subsiguientes argumentos para `defcustom` especifican los tipos y opciones; estos no son funcionales en `defvar`. (Estos argumentos son opcionales.)

Cada uno de estos argumentos consiste de una palabra seguido de una palabra por un valor. Cada palabra clave empieza con los dos puntos ‘`:`’.

Por ejemplo, la variable de opciones personalizable `text-mode-hook` se parece a esto:

```
(defcustom text-mode-hook nil
  "El hook normal se ejecuta cuando se introduce en modo texto y
  muchos modos relacionados."
  :type 'hook
  :options '(turn-on-auto-fill flyspell-mode)
  :group 'data)
```

El nombre de la variable es `text-mode-hook`; no tiene valor por defecto; y su cadena de documentación cuenta lo que hace.

La palabra clave `:type` le cuenta a Emacs el tipo de datos para los que `text-mode-hook` sería asignado y como muestra el valor en un búffer de Personalización.

La palabra clave `:options` especifica una lista sugerida de valores para la variable. Normalmente, `:options` se asocia a un gancho (*hook*. La lista es solo una sugerencia; esa no es exclusiva; una persona quien asigna la variable puede asignarse a otros valores; la lista mostrada siguiendo la palabra clave `:options` se pretende ofrecer elecciones convenientes a un usuario.

Finalmente, la palabra clave `:group` cuenta el comando de Personalización de Emacs en el que el grupo de la variable está localizado. Esto cuenta dónde encontralo.

La función `defcustom` reconoce más de una docena de palabras clave. Para más información, mire Sección “Escribiendo las Definiciones de Personalización” in *El Manual de Referencia GNU Emacs Lisp*.

Considere `text-mode-hook` como un ejemplo.

Hay dos caminos para personalizar esta variable. Se puede usar el comando de personalización o escribir las expresiones apropiadas por tí mismo.

Usando el comando de personalización, se puede escribir:

```
M-x customize
```

y encuentre que el grupo para editar ficheros datos se llama ‘datos’. Introduzca este grupo. El Hook *Disparador* es el primer miembro. Se puede hacer click en sus opciones varias, tal como `turn-on-auto-fill`, para asignar los valores. Después de hacer click en el botón.

```
Guárdalo para Futuras Sesiones
```

Emacs escribirá una expresión en tu fichero `.emacs`. Se parecerá a esto:

```
(custom-set-variables
  ;; custom-set-variables fué añadido por Custom.
  ;; Si se edita a mano, tu podrías liartte,
  ;; así que ten cuidado.
  ;; Tu fichero init contendría solo esta instancia.
  ;; Si hay más de uno, ellos no quieren trabajar.
 '(text-mode-hook (quote (turn-on-auto-fill text-mode-hook-identify))))
```

(La función `text-mode-hook-identify` cuenta `toggle-text-mode-auto-fill` que buffers hay en modo Texto. Eso viene automáticamente)

La función `custom-set-variables` funciona de alguna manera diferente más de un `setq`. Mientras yo nunca he aprendido las diferencias, yo modifico las expresiones `custom-set-variable` en mi fichero `.emacs` a mano: yo creo los cambios en los que aparecen a mi para ser una manera razonable y no tener problemas. Otros prefieren usar el comando de Personalización y permitir a Emacs hacer el trabajo para ellos.

Otra función `custom-set-...` es `custom-set-faces`. Esta función asigna varios tipos de fuentes. A través del tiempo, yo he asignado un considerable número de tipos. Algo de tiempo, yo las reseteo usando `customize`; otras veces, simplemente edito la expresión `custom-set-faces` en mi fichero `.emacs` en sí.

El segundo modo de personalizar tu `text-mode-hook` es asignarte a tí mismo en tu fichero `.emacs` usando código que no tiene nada que hacer con las funciones `custom-set-...`.

Cuando se hace esto, y después usa `customize`, se verá un mensaje que dice:

```
CHANGED fuera de Personalizar; operando dentro aquí
puede ser no confiable.
```

Este mensaje es solo un aviso. Si se puede cliquear en el botón a

```
Guárdalo para Futuras Sesiones
```

Emacs escribirá una expresión `custom-set-...` cerca del fin de tu fichero `.emacs` que será evaluado después de que tu expresión sea escrita a mano. Por esta razón, se sobreescribirá tu expresión escrita a mano. Ningún daño será hecho. Cuando se haga esto, sin embargo, ten cuidado para recordar que expresión está activa; si olvidas, puedes confundirte por tí mismo.

Tan largo como se recuerda donde los valores son configurados, no habrá problemas. En cualquier eventos, los valores son siempre configurados en tu fichero de inicialización, que es normalmente llamado `.emacs`.

Yo mismo hago un `customize` para cualquier cosa. Mayoritariamente, escribo expresiones por mí mismo.

Incidentalmente, para ser una definición concerniente más completa: `defsubst` define una función inline. La sintaxis es solo como esta de `defun`. `defconst` define un símbolo como una constante. El intento es que ningún programa o usuario cambiarían un valor asignado por `defconst`. (Se puede cambiar; el valor asignado es una variable; pero por favor no lo haga.)

16.3 Empieza por un fichero `.emacs`

Cuando se abre Emacs, se carga tu fichero `.emacs` a menos que se cuente que no se especifique '`-q`' en la línea de comandos. (El comando `emacs -q` tu da un Emacs plano, fuera.)

Un fichero `.emacs` contiene expresiones Lisp. Con frecuencia, no hay más expresiones para configura valores; algunas veces esas son definiciones de funciones.

Véase Sección "El Fichero de Inicio `~/.emacs`" in *El Manual GNU Emacs*, para una corta descripción de fichero de inicialización.

Este capítulo cubre algo del mismo suelo, pero es un paseo entre extractos desde un completo, largamente usado fichero `.emacs` — por mí.

La primera parte del fichero consiste en comentario: me recuerdo a mí mismo. Por ahora, yo recuerdo estas cosas, pero cuando empecé, no.

```
;;;; fichero .emacs de Bob
; Robert J. Chassell
; 26 de Septiembre de 1985
```

¡Mira en esta fecha! Yo empecé este fichero hace mucho tiempo. Yo he estado añadiendo cosas desde siempre.

```
; Cada sección en este fichero es introducido por una
; línea empezando con cuatro puntos y comas y cada
; entrada es introducida por una línea empezando con
; tres puntos y comas.
```

Esto describe las convenciones usuales para comentarios en Emacs Lisp. Cada cosa en una línea que sigue un punto y coma es un comentario. Dos, tres, y cuatro puntos y coma son usados como subsección y marcas de sección. (Véase Sección "Comentarios" in *El Manual de Referencia GNU Emacs Lisp*, para más comentarios.)

```
;;;; La Tecla de Ayuda
; Control-h es la tecla de ayuda;
; después escribiendo control-h, escribe una letra a
; indica el asunto acerca del que quieres ayuda.
; Para una explicación de la facilidad de ayuda,
; escribe control-h dos veces en una fila.
```

Solo recuerda: escribe *C-h* dos veces para ayudar.

```
; Para informarse acerca de cualquier modo, escribe control-h m
; mientras esté en este modo. Por ejemplo, para encontrar
; acerca del modo correo, introduce el modo correo y entonces
; escribe control-h m.
```

'Modo ayuda', como yo llamo a esto, es muy útil. Usualmente, se cuenta todo lo que se necesita saber.

De acuerdo, no se necesitan incluir comentarios y ficheros como estos `.emacs`. Yo los incluí en el mío porque se olvida el Modo ayuda o las convenciones para comentarios — pero era capaz de recordar ver aquí recordármelo a mí mismo.

16.4 Modo texto y auto relleno

Ahora regresa a la parte que 'vuelve' al modo Texto y modo Auto Relleno.

```
;;; Modo texto modo Auto Fill
;; Las siguiente dos líneas puestas en Emacs dentro de
;; modo Texto y en el modo Auto Fill, son para escritores que
;; quieren empezar a escribir prosa en vez de código.
(setq-default major-mode 'text-mode)
(add-hook 'text-mode-hook 'turn-on-auto-fill)
```

¡Aquí está la primera parte de este fichero `.emacs` que hace alguna cosa bajo recuerdo de un humano olvidado!

La primera de las dos líneas en paréntesis cuentan a Emacs a cambiar al modo Texto que se encuentra un fichero, *a menos que* el fichero iría dentro de algún otro modo, tal como modo C.

Cuando Emacs lee un fichero, eso parece la extensión al nombre del fichero. (La extensión es la parte que viene después de un '.'.) Si el fichero finaliza con una

extensión '.c' o '.h' entonces Emacs cambia al modo C. También, Emacs parece al principio una línea no blanca del fichero; si la línea dice '-*- C -*-', Emacs cambia al modo C. Emacs posee una lista de extensiones y especificaciones que usa automáticamente. Además, Emacs se ve cerca de la última página por buffer, "lista variables locales".

Mira las secciones "Cómo los Modos Mayores son Elegidos" y "Variables Locales en Fichero" en *El Manual GNU Emacs*.

Ahora, regresa al fichero .emacs.

Aquí está la línea de nuevo; ¿cmo funciona?

```
(setq major-mode 'text-mode)
```

Esta línea es un resumen, pero completa la expresión Emacs Lisp.

Ya estamos familiarizados con setq. Eso asigna la siguiente variable, major-mode, al subsiguiente valor, que es text-mode. La marca de cita simple antes de text-mode cuenta a Emacs como tratar directamente con el símbolo, no con cualquier cosa que pudiera existir. Véase Sección 1.9 "Configurando el Valor de una Variable", página 16, por un recuerdo de como setq funciona. El principal punto es que no hay diferencia entre el procedimiento que se usa para asignar un valor en su fichero .emacs y el procedimiento que se usa en cualquier lugar más en Emacs.

Aquí está la siguiente línea:

```
(add-hook 'text-mode-hook 'turn-on-auto-fill)
```

En esta línea, el comando add-hook añade turn-on-auto-fill para la variable.

¡turn-on-auto-fill es el nombre de un programa, que se adivina!, cambia al modo Auto Fill.

Cada vez que Emacs cambia al modo texto, Emacs ejecuta el comando 'hooked' dentro de modo Texto. Así cada vez que Emacs cambia al modo Texto, Emacs también cambia al modo de autoajuste.

En breve, la primera línea causa a Emacs a entrar en modo Texto cuando se edite un fichero, a menos que la extensión del nombre del fichero, una línea no en blanco, variables locales para contar a Emacs de otro modo.

El modo texto entre otras acciones, asigna la tabla de sintaxis para trabajar adecuadamente a escritores. En modo texto, Emacs considera un apóstrofe como parte de una palabra como una letra; pero Emacs no considera un período o un espacio como parte una palabra. De este modo, *M-f* se mueve hacia tí a travé de 'eso es'. Por otro lado, en modo C, *M-f* para solo después del 't' de 'eso es'.

La segunda línea causa que Emacs active el modo Auto Fill cuando cambia al modo Texto. En modo Auto Fill, Emacs automáticamente rompe una línea que es demasiado amplio y trae la parte excesivamente amplia de la línea de debajo a la siguiente línea. Emacs rompe líneas entre palabras con ellas.

Cuando el modo Auto Fill está desactivado, las líneas continuan a la derecha como se escriben. Dependiendo de como configuras el valor de truncate-lines, las palabras que escribe si desaparecen al lado derecho de la pantalla, o lo demás son mostradas, en un modo feo e ilegible, como una línea de continuación en la pantalla.

Además, en esta parte de mi fichero `.emacs`, yo cuento a Emacs el ajuste comandos para insertar dos espacios después de dos puntos:

```
(setq colon-double-space t)
```

16.5 Alias de correo

Aquí hay un `setq` que 'activa' el alias de correo, para más ocasiones.

```
;;; Modo Correo
; Para entrar en el modo correo, escribe 'C-x m'
; Para introducir RMAIL (para leer el correo),
; escribe 'M-x rmail'
(setq mail-aliases t)
```

Este comando `setq` asigna el valor de la variable `mail-aliases` al `t`. Desde que `t` significa cierto, la línea dice, en efecto, "Sí uso alias de correo."

Los alias de correo son nombres cortos convenientes para largas direcciones de correo o para listas de direcciones de correo. El fichero donde guardar tus 'aliases' es `~/.mailrc`. Se escribe un alias como este:

```
alias geo george@foobar.wiz.edu
```

Cuando se escribe un mensaje a Jorge, la dirección a '`geo`'; el correo automáticamente expandirá '`geo`' a la dirección completa.

16.6 Indentar modo de tabulaciones

Por defecto, Emacs inserta tabulaciones en lugar en múltiples espacios cuando se formatea una región. (Por ejemplo, se podrían indentar muchas líneas de texto todo a la vez con el comando `indent-region`.) Los tabuladores se ven bien en un terminal o con impresión ordinaria, pero ellos producen mala salida de indentación cuando se usa TEX o Texinfo puesto que TEX ignora tabuladores.

Lo siguiente desactiva el modo de Indentar Tabulaciones:

```
;;; Prevenir Tabulaciones Extrañas
(setq-default indent-tabs-mode nil)
```

Note que esta línea usa `setq-default` en vez de el comando `setq` que hemos visto antes. El comando `setq-default` asigna valores solo en buffers que no tienen sus propios valores locales para la variable.

Ver secciones "Tabuladores versus Espacios" y "Variables Locales en Ficheros" en *El Manual de GNU Emacs*.

16.7 Algunos atajos

Ahora para algunos atajos personales:

```
;;; Compara ventanas
(global-set-key "\C-cw" 'compare-windows)
```

`compare-windows` es un comando excelente que compara el texto en tu ventana actual con texto en la siguiente ventana. Eso hace la comparación empezando al punto en cada ventana, moviendo a través del texto en cada ventana tan lejos como ellos asocian. Yo uso este comando todo el tiempo.

Esto también muestra como configurar una tecla globalmente, para todo los modos

El comando es `global-set-key`. Es seguido por el atajo. En un fichero `.emacs`, el atajo es escrito como se ve: `\C-c` que se asocia a 'control-c', que significa 'presionar la tecla de control y la tecla `c` al mismo tiempo'. La `w` significa 'presionar la tecla `w`'. El atajo es rodeador por dobles comillas. En la documentación, se escribiría esto como *`C-c w`*. (Si estuviera asociando una tecla `META`, tal como *`M-c`*, en vez de una tecla de `CTRL`, se escribiría `\M-c` en su fichero `.emacs`. Véase Sección "Reasociando Teclas en Su Fichero Init" in *El Manual de GNU Emacs*, para más detalles.)

El comando invocado por las teclas es `compare-windows`. Note que `compare-windows` es precedido por una comilla simple; de otro modo, Emacs primero intentaría evaluar el símbolo para determinar su valor.

Estas tres cosas, las marcas de dobles comillas, la barra invertida antes de la '`C`', y la marca de comilla simple son partes necesarias de atajos de teclado que tiendo a olvidar. Afortunadamente, he llegado a recordar que miraría mi fichero `.emacs` existente, y lo adaptaría a lo que hay.

Como para el atajo en sí: *`C-c w`*, combina la tecla prefija, *`C-c`*, con un caracter simple, en este caso, *`w`*. Este conjunto de teclas, *`C-c`* seguido por un caracter simple, es estrictamente reservado para un uso propio individual. (Esto se llama teclas 'propias', desde que estas son para su propio uso.) Siempre sería capaz de crear tal atajo para el uso propio sin pisar fuerte en algún atajo más. Si siempre se escribe una extensión a Emacs, por favor evite tomar cualquier de estas teclas para uso público. Se cree que una tecla como *`C-c C-w`* en vez de eso. De otra manera, ejecutará sin sus 'propias' teclas.

Aquí hay otro atajo, con un comentario:

```
;;; Atajo para 'occur'
; Yo uso mucho occur, así permite asignarlo a una tecla:
(global-set-key "\C-co" 'occur)
```

El comando `occur` muestra todas las líneas en el buffer actual que contiene un emparejamiento para una expresión regular. Asociar las líneas que se muestran en un búffer llamado `*Occur*`. Este buffer sirve como un menu para saltar a ocurrencias.

Aquí se muestra como desasignar una tecla, así no funciona:

```
;;; Desasociar 'C-x f'
(global-unset-key "\C-xf")
```

Hay una razón para esta no asociación: Yo encontré inadvertidamente escrito *`C-x f`* cuando significó escribir *`C-x C-f`*. En vez de encontrar un fichero, como se pretende, accidentalmente asigna el ancho para el fichero lleno, casi siempre a un tamaño que no quería. Desde que duramente se reseteó mi ancho por defecto, yo simplemente disocié la tecla.

Lo siguiente reasocia una tecla existente:

```
;;; Reasocia ‘C-x C-b’ al ‘buffer-menu’
(global-set-key "\C-x\C-b" 'buffer-menu)
```

Por defecto, *C-x C-b* ejecute el comando `list-buffers`. Este comando lista sus buffers en *otra* ventana. Desde que casi siempre se quiere hacer alguna cosa en esta ventana, se prefiere el comando `buffer-menu`, que no solo lista los buffers, pero mueve el punto dentro de esta ventana.

16.8 Mapas de teclado

Emacs usa *keymaps* para grabar qué teclas llaman a qué comandos. Cuando se use `global-set-key` para asignar los atajos de teclados a un simple comando en todo `current-global-map`.

Modos específicos, tales como modo C o modo Texto, tiene sus propios mapas de teclado; mapas de teclado de modo específico sobreescribe el mapa global que es compartido por todos los buffers.

La función `global-set-key` asocia, o reasocia, el mapa de teclado global. Por ejemplo, las siguientes asociaciones la tecla *C-x C-b* a la función `buffer-menu`:

```
(global-set-key "\C-x\C-b" 'buffer-menu)
```

Mapas de teclado específico de modo son asociados usando la función `define-key`, que toma un mapa de teclado específico como un argumento, tan bien como la tecla y el comando. Por ejemplo, mi fichero `.emacs` contiene la siguiente expresión asociada al comando `texinfo-insert-@group` comando a *C-c C-c g*:

```
(define-key texinfo-mode-map "\C-c\C-cg" 'texinfo-insert-@group)
```

La función `texinfo-insert-@group` en sí es una pequeña extensión del modo Texinfo que inserta ‘`@group`’ dentro de un fichero Texinfo. Se usa este comando todo el tiempo y se prefieren escribir los tres atajos *C-c C-c g* en vez de los seis atajos *@ g r o u p*. (‘`@group`’ y su asociación ‘`@end group`’ son comandos que guarda todo el texto cerrado junto a una página; muchos ejemplos multi-línea en este libro están rodeados por ‘`@group ... @end group`’.)

Aquí está la definición de función `texinfo-insert-@group`:

```
(defun texinfo-insert-@group ()
  "Inserta la cadena @group en un buffer Texinfo."
  (interactive)
  (beginning-of-line)
  (insert "@group\n"))
```

(De acuerdo, podría haber usado el modo Abbrev para dejar de escribir, en vez de escribir una función para insertar una palabra; pero prefiero atajos de teclado consitentes con otro modo Texinfo para atajos de teclado.)

Verá numerosas expresiones `define-key` en `loaddefs.el` tan bien como en varios modos de librerías, tal como `cc-mode.el` y `lisp-mode.el`.

Véase Sección “Personalizando Atajos de Teclado” in *El Manual GNU Emacs*, y Sección “Mapas de Teclado” in *El Manual de Referencia GNU Emacs Lisp*, para más información acerca de mapas de teclado.

16.9 Cargando ficheros

Muchas personas en la comunidad de GNU Emacs han escrito extensiones a Emacs. Hace tiempo, que estas extensiones son con frecuencia incluidas en las nuevas entregas *releases*. Por ejemplo, los paquetes Calendario y Diario son ahora parte del estándar GNU Emacs, como es Calc.

Se puede usar un comando `load` para evaluar un fichero completo que significa instalar todas las funciones y variables en el fichero Emacs. Por ejemplo:

```
(load "~/emacs/slowsplit")
```

Esto evalúa, por ej. carga, el fichero `slowsplit.el` o si eso existe, lo más rápido, el fichero compilado `slowsplit.elc` desde el subdirectorio `emacs` del directorio home. El fichero contiene la función `split-window-quietly`, que John Robinson escribió en 1989.

La función `split-window-quietly` divide una ventana con el mínimo de redisplay. Yo lo instalé en 1989 porque trabajó bien con los terminales de 1200 baudios que entonces estaba usando. Ahora, ocasionalmente vengo a través de una conexión lenta, pero continúa usando la función porque me gusta el camino que deja arriba del búffer en el bajo de las nuevas ventanas y arriba en la ventana superior.

Para reemplazar el atajo de teclado por defecto `split-window-vertically`, se debe también desasignar esta tecla y asociar las teclas a `split-window-quietly`, como este:

```
(global-unset-key "\C-x2")
(global-set-key "\C-x2" 'split-window-quietly)
```

Si se cargan muchas extensiones, como yo hago, entonces en vez de especificar la posición exacta del fichero, como se muestra arriba, se puede especificar que directorio como parte del `load-path` de Emacs. Entonces, cuando Emacs carga un fichero, buscará que directorio tan bien como su lista por defecto de directorios. (La lista por defecto es especificada en `paths.h` cuando Emacs se construye.)

El comando siguiente añade tu directorio `~/emacs` a la ruta existente:

```
;;; Ruta Emacs
(setq load-path (cons "~/emacs" load-path))
```

Incidentalmente, `load-library` es un interfaz interactivo a la función `load`. La función se parece a esto:

```
(defun load-library (library)
  "Carga la librería llamada LIBRARY.
Esto es una interfaz a la función `load'."
  (interactive
   (list (completing-read "Carga la librería: "
                          (apply-partially 'locate-file-completion-table
                                           load-path
                                           (get-load-suffixes)))))
  (load library))
```

El nombre de la función, `load-libray`, viene desde el uso de 'library' como un sinónimo para 'file'. La fuente para el comando `load-library` está en la librería `files.el`.

Otro comando interactivo que hace un trabajo ligeramente diferente es `load-file`. Véase Sección "Librerías de Código Lisp para Emacs" in *El Manual*

GNU Emacs, para información en la distinción entre `load-library` y este comando.

16.10 Autoloading

En vez de instalar una función cargando el fichero que lo contiene, o evaluando la definición de función, se puede hacer la función disponible pero actualmente no se instala hasta la primera vez llamada. Este proceso se llama *autocarga* (*autoloading*).

Cuando se ejecuta una función de autocarga, Emacs automáticamente evalúa el fichero que contiene la definición, y entonces llama a la función.

Emacs empieza rápido con funciones de autocarga, puesto que sus librerías son no cargadas bien; pero si necesita esperar un momento cuando su primer uso tal como una función, mientras el fichero que lo contiene es evaluado.

Raramente las funciones usadas son frecuentemente autocargadas. La librería `loaddefs.el` coniene cientos de funciones autocargadas, desde `bookmark-set` a `wordstar-mode`. Si se usa una función 'rara' frecuentemente, se debería cargar este fichero de función con una expresión de `load` en tu fichero `.emacs`.

En mi fichero `.emacs`, se cargan 14 librerías que contienen funciones que de otro modo serían autocargadas. (Actualmente, eso habría sido mejor para incluir estos ficheros en mi Emacs 'volcado', pero se olvida. Véase Sección "Construyendo Emacs" in *El Manual de Referencia GNU Emacs Lisp*, y el fichero `INSTALL` para más acerca de volcados.)

Se puede también querer incluir expresiones autocargadas en tu fichero `.emacs`. `autoload` es una función construida que toma cinco argumento, los tres finales de los que son opcionales. El primer argumento es el nombre de la función para ser autocargada. El segundo es el nombre del fichero para ser cargado. El tercer argumento es documentación para la función, y el cuarto cuenta si la función puede ser llamada interactivmente. El quinto argumento cuenta que tipo de objeto — `autoload` puede manejar un mapa de teclado o macro tan bien como una función (por defecto es una función).

Aquí hay un ejemplo típico:

```
(autoload 'html-helper-mode
  "html-helper-mode" "Editar documentos HTML" t)
```

(`html-helper-mode` es una vieja alternativa a `html-mode`, que es un parte estándar de la distribución.)

Esta expresión autocarga la función `html-helper-mode`. Esto lo toma desde el fichero `html-helper-mode-el` (o desde la versión compilada `html-helper-mode.elc`, si esto existe.) El fichero debe ser localizada en un directorio específico por `load-path`. La documentación dice que esto es un modo para ayudarte a editar documentos escritos en Lenguaje de Marcas de Hiper Texto. Se puede llamar este modo interactivamente escribiendo *`M-x html-helper-mode`*. (Se necesitan duplicar las funciones regulares de documentación en la expresión de autocarga porque la función regular no está todavía cargada, así su documentación no está disponible.)

Véase Sección "Autocarga" in *El Manual de Referencia de GNU Emacs Lisp*, para más información.

16.11 Una extensión simple: `line-to-top-of-window`

Aquí hay una simple extensión a Emacs que mueve el punto de línea arriba de la ventana. Yo uso esto todo el tiempo, para hacer fácil leer el texto.

Se puede poner el siguiente código dentro de un fichero separado y entonce cargarlo desde tu fichero `.emacs`, o se puede incluir con tu fichero `.emacs`.

Aquí es la definición

```
;;; Línea a lo alto de la ventana;
;;; reemplaza tres secuencias de atajos de teclado  C-u 0 C-l
(defun line-to-top-of-window ()
  "Mueve la línea que apunta a lo alto de la ventana."
  (interactive)
  (recenter 0))
```

Ahora el atajo.

En estos días, las teclas de función así como los eventos del ratón y caracteres no ASCII son escritos con corchetes, sin marcas de citas. (En Emacs versión 18 y antes, se tenía que escribir diferentes teclas de función asignadas por cada diferente creación del terminal.)

Yo asocio `line-to-top-of-window` a mi función `F6` como esta:

```
(global-set-key [f6] 'line-to-top-of-window)
```

Para más información, mira Sección "Reasociando Teclas en tu fichero init" in *El Manual GNU Emacs*.

Si ejecutas dos versiones de GNU Emacs, tal como las versiones 22 y 23, y usas un fichero `.emacs`, se puede seleccionar qué código evalúa el siguiente condicional:

```
(cond
 ((= 22 emacs-major-version)
  ;; evalúa la version 22
  ( ... ))
 ((= 23 emacs-major-version)
  ;; evalúa la version 23
  ( ... )))
```

Por ejemplo, en versiones más recientes se ocultan los cursores por defecto. Si se odia tal ocultación se escribe lo siguiente en mi fichero `.emacs`[2]:

```
(when (>= emacs-major-version 21)
  (blink-cursor-mode 0)
  ;; Inserta la nueva línea cuando se presiona 'C-n' (next-line)
  ;; al fin del búffer
  (setq next-line-add-newlines t)
  ;; Cambia la imagen viendo
  (auto-image-file-mode t)
  ;; Activa la barra de menu (esta barra tiene texto)
  ;; (Usa un argumento numérico para activarlo)
  (menu-bar-mode 1)
```

[2] Cuando se empiezan las instancias de Emacs que no cargan mi fichero `.emacs` o cualquier fichero, también se puede deshabilitar la ocultación:

```
emacs
-q --no-site-file -eval '(blink-cursor-mode nil)'
```

O ahora,

```
 ;; Desactiva la barra de herramientas (esta barra tiene iconos)
 ;; (Usa argumentos numéricos para activarlo)
 (tool-bar-mode nil)
;; Desactiva el modo tooltip para la tool bar
;; (Este modo causa explicaciones de iconos al pop up)
;; (Usa el argumento numérico para activarlo)
(tooltip-mode nil)
;; Si los tooltips activados, crea consejos aparecen en el prompt
(setq tooltip-delay 0.1)  ; por defecto es de 0.7 segundos
 )
```

16.12 Colores X11

Se pueden especificar colores cuando se usa Emacs con el Sistema de Ventanas X del MIT.

Me disgustan los colores de por defecto y especificar los míos.

Aquí están las expresiones en mi fichero `.emacs` que establecen valores:

```
;; Asigna el color del cursor
(set-cursor-color "white")

;; Asigna el color del ratón
(set-mouse-color "white")

;; Asigna foreground y background
(set-foreground-color "white")
(set-background-color "darkblue")

;;; Asigna colores para isearch y drag
(set-face-foreground 'highlight "white")
(set-face-background 'highlight "blue")

(set-face-foreground 'region "cyan")
(set-face-background 'region "blue")

(set-face-foreground 'secondary-selection "skyblue")
(set-face-background 'secondary-selection "darkblue")

;; Asigna colores al calendario
(setq calendar-load-hook
      '(lambda ()
         (set-face-foreground 'diary-face   "skyblue")
         (set-face-background 'holiday-face "slate blue")
         (set-face-foreground 'holiday-face "white")))
```

Las varias sombras de azul disparan mi ojo y me preveen de ver la ventana desplegada.

Alternativamente, se podrían haber configurado mis especificaciones en varios ficheros inicialización de X. Por ejemplo, se podría asignar el foreground, background, cursor y puntero (por ej., ratón) colores en mi fichero `~/.Xresources` como esto:

```
Emacs*foreground:   white
Emacs*background:   darkblue
Emacs*cursorColor:  white
Emacs*pointerColor: white
```

En cualquier evento que no es parte de Emacs, se asigna el color raíz de mi ventana X en mi fichero `~/.xinitrc`, como este[3]

```
xsetroot -solid Navy -fg white &
```

16.13 Configuraciones misceláneas para un fichero .emacs

Aquí hay unas pocas configuraciones misceláneas:

– Asigna la forma y color del ratón del cursor:

```
; Formas de Cursor están definidas en
; '/usr/include/X11/cursorfont.h';
; por ejemplo, el cursor 'objetivo' es número 128;
; el cursor 'top_left_arrow' es el número 132.

(let ((mpointer (x-get-resource "*mpointer"
                                "*emacs*mpointer")))
  ;; Si no se ha asignado tu puntero de ratón
  ;; entonces asignalo, de otro modo, déjalo así:
  (if (eq mpointer nil)
      (setq mpointer "132")) ; top_left_arrow
  (setq x-pointer-shape (string-to-int mpointer))
  (set-mouse-color "white"))
```

– O se pueden asignar los valores de una variedad de funcionalidades en una alist, como esta:

```
(setq-default
 default-frame-alist
 '((cursor-color . "white")
   (mouse-color . "white")
   (foreground-color . "white")
   (background-color . "DodgerBlue4")
   ;; (cursor-type . bar)
   (cursor-type . box)
   (tool-bar-lines . 0)
   (menu-bar-lines . 1)
   (width . 80)
   (height . 58)
   (font .
         "-Misc-Fixed-Medium-R-Normal--20-200-75-75-C-100-ISO8859-1")
   ))
```

– Convierte *CTRL-h* dentro DEL y DEL dentro de *CTRL-h*.
(Algunos viejos teclados lo necesitan, aunque yo no he visto el problema recientemente.)

[3] también se ejecutan gestores de ventanas más modernos, tales como Enlightenment, Gnome, o KDE; en estos casos, con frecuencia se especifica una imagen en vez de un color plano.

```
;; Traducir 'C-h' a <DEL>.
; (keyboard-translate ?\C-h ?\C-?)

;; Traducir <DEL> a 'C-h'.
(keyboard-translate ?\C-? ?\C-h)
```

– ¡Desactiva un cursor oculto!

```
(if (fboundp 'blink-cursor-mode)
    (blink-cursor-mode -1))
```

o empieza GNU Emacs con el comando `emacs -nbc`.

– Cuando se usa 'grep'
'`-i`' Ignore distinciones de letras
'`-n`' El prefijo de cada línea de la salida con el número de líneas
'`-H`' Imprime el nombre de fichero para cada cadena encontrada.
'`-e`' Protege patrones empezando con un caracter de guión, '`-`'

```
(setq grep-command "grep -i -nH -e ")
```

– Encuentra un búffer existente, incluso si eso tiene un nombre diferente
Esto evita problemas con enlaces simbólicos.

```
(setq find-file-existing-other-name t)
```

– Configura tu entorno de lenguaje y el método de entrada por defecto

```
(set-language-environment "latin-1")
;; Recuerda que se puede habilitar o deshabilitar el texto de lenguaje
;; multilingüe con el comando toggle-input-method' (C-\)
(setq default-input-method "latin-1-prefix")
```

Si se quiere escribir con el caracter Chino 'GB', asigna esto:

```
(set-language-environment "Chinese-GB")
(setq default-input-method "chinese-tonepy")
```

Arreglando Atajos de Teclados

Algunos sistemas asocian teclas de maneras no agradables. Algunas veces, por ejemplo, la tecla `CTRL` en un modo perverso en vez de la lejanía a la izquierda de la fila.

Normalmente, cuando las personas arreglan estos atajos de teclado, no se cambia su fichero `~/.emacs`. En vez de eso, se asocian las teclas apropiadas en sus consolas con los comandos `loadkeys` o `install-keymap` en su script de inicio y entonces incluyen comandos `xmodmap` en su fichero `.xinitrc` o `.Xsession` para X Windows. Para un script de inicio:

```
loadkeys /usr/share/keymaps/i386/qwerty/emacs2.kmap.gz
```

or

```
install-keymap emacs2
```

Para un fichero `.xinitrc` o un fichero `.Xsession` cuando la tecla `Caps Lock` es que tan lejos de la fila del home:

```
# Asocia la tecla etiquetada 'Caps Lock' a 'Control'
# (Tal como un interfaz de usuario roto sugiere que el teclado hecho
# piensa que los ordenadores son máquinas de escribir desde 1885.)

xmodmap -e "clear Lock"
xmodmap -e "add Control = Caps_Lock"
```

En un .xinitrc o .Xsession, para convertir una tecla ALT a una tecla META:

```
# Algunos teclados mal diseñados tienen una tecla etiquetada ALT y no Meta
xmodmap -e "keysym Alt_L = Meta_L Alt_L"
```

16.14 Una línea modificada

Finalmente, una funcionalidad que realmente me gusta: un mode line modificado.

Cuando se trabaja a través de una red, se olvida que máquina se está usando. También, se tiende a perder la traza de donde se está, y a qué línea se apunta.

Así se resetea mi mode line para que se parezca a esto:

```
-:-- foo.texi   rattlesnake:/home/bob/  Line 1  (Texinfo Fill) Top
```

Estoy visitando un fichero llamado foo.texi, en mi máquina rattlesnake en mi búffer /home/bob. Yo estoy en la línea 1, en modo Texinfo, y estoy arriba del búffer.

Mi fichero .emacs tiene una sección que se parece a esto:

```
;; Asigna un Mode Line que nos cuente que máquina, que directorio,
;; y que línea estoy on, más la información de client.
(setq-default mode-line-format
 (quote
  (#("-" 0 1
     (help-echo
      "mouse-1: select window, mouse-2: delete others ..."))
   mode-line-mule-info
   mode-line-modified
   mode-line-frame-identification
   "    "
   mode-line-buffer-identification
   "    "
   (:eval (substring
           (system-name) 0 (string-match "\\..+" (system-name))))
   ":"
   default-directory
   #(" " 0 1
     (help-echo
      "mouse-1: select window, mouse-2: delete others ..."))
   (line-number-mode " Line %l ")
   global-mode-string
   #("   %[(" 0 6
     (help-echo
      "mouse-1: select window, mouse-2: delete others ..."))
   (:eval (mode-line-mode-name))
   mode-line-process
   minor-mode-alist
   #("%n" 0 2 (help-echo "mouse-2: widen" local-map (keymap ...)))
   ")%] "
   (-3 . "%P")
   ;;   "-%-"
   )))
```

Aquí, se redefine el mode line por defecto. La mayoría de las partes son desde el original; pero yo creo unos pocos cambios. Yo asigno el formato de mode line *default* así como permitir varios modos, tales como Info, para sobreescribirlo.

Muchos elementos en la lista son auto-explicativos: `mode-line-modified` es una variable que cuenta si el búffer ha sido modificado, `mode-name` cuenta el nombre del modo, y así. Sin embargo, el formato parece complicado porque las dos funcionalidades no han sido discutidas.

La nueva cadena de formato tiene una sintaxis especial:

```
#("-" 0 1 (help-echo "mouse-1: select window, ..."))
```

El `#(` empieza una lista. El primer elemento de la lista es la cadena en sí, solo un '`-`'. El segundo y tercer elemento especifica el rango a través con el cuarto elemento aplicado. Un rango empieza *después* un carácter, así un cero significa el rango que empieza solo después del primer caracter; un 1 significa que el rango finaliza solo después del primer caracter. El tercer elemento es la propiedad para el rango. Eso consiste en una lista de propiedades, un nombre de propiedad, en este caso, '`help-echo`', seguido por un valor, en este caso, una cadena. El segundo, tercer y cuarto elementos de este nuevo formato de cadena puede ser repetido.

Véase Sección "Propiedades de Texto" in *El Manual de Referencia de GNU Emacs Lisp*, y ver Sección "Formato Mode Line" in *El Manual de Referencia de GNU Emacs Lisp*, para más información.

`mode-line-buffer-identification` muestra el nombre del buffer. Eso es una lista empezando por `(#("%12b" 0 4 ....` El `#(` empieza la lista.

El '`"%12b"`' muestra el nombre del actual búffer, usando la función `buffer-name` con la que estamos familiarizados; el '12' especifica el número máximo de caracteres que serán mostrados. Cuando un nombre tiene pocos caracteres, el espacio en blanco se añade para rellenar este número. (Los nombres del búffer puede y con frecuencia serán más largos de 12 caracteres; esta longitud funciona bien en la típica ventana de 80 columnas de ancho.)

`:eval` dice evaluar la siguiente forma y usa el resultado como una cadena para mostrarse. En este caso, la expresión muestra el primer componente del sistema completo. El fin del primer componente es un '`.`' ('periodo'), así se usa la función `string-match` para contar el tamaño del primer componente. La subcadena desde el caracter cero a este tamaño del primer componente. La subcadena desde el caracter cero a este tamaño es el nombre de la máquina.

Esta es la expresión:

```
(:eval (substring
        (system-name) 0 (string-match "\\..+" (system-name))))
```

'`%[`' y '`%]`' causa un par de corchetes que aparezcan por cada edición nivel de edición recursiva editando el nivel. '`%n`' dice 'Encoger' cuando esto puede hacerse. '`%P`' te cuenta el porcentaje del buffer que está debajo debajo de la ventana, o 'arriba', 'abajo', o 'todo'. (Una minúscula '`p`' cuenta el porcentaje bajo el alto de la ventana.) '`%-`' inserta suficientes guiones para rellenar la línea.

Recuerda, "No tiene que gustarte Emacs para que le gustes" — tu Emacs puede tener diferentes colores, diferentes comandos, y diferentes teclas que un Emacs por defecto.

Por otro lado, si se quiere traer un plano 'fuera de la caja' Emacs, sin personalización, escribe:

```
emacs -q
```

Esto inicializará un Emacs que *no* cargue tu `~/.emacs` fichero de inicialización. Uno plano, el que trae Emacs por defecto. Nada más.

17 Depurando

GNU Emacs tiene dos depuradores, `debug` y `edebug`. El primero es construido dentro de las tripas de Emacs y está siempre contigo; el segundo requiere que exista una función antes de que se pueda usar.

Ambos depuradores son descritos extensivamente en Sección "Depurando Programas Lisp" in *El Manual de Referencia GNU Emacs Lisp*. En este capítulo, se explicará un breve ejemplo de esto.

17.1 depurar

Supón que se ha escrito una definición de función que se pretende devolver la suma de los números 1 a través de un número dado. (Esta es la función `triangle` discutida pronto. Véase "Ejemplo con Contador de Decremento", página 118, para una discusión.)

Sin embargo, tu definición de función tiene un error. Se ha malescrito '`1=`' por '`1-`'. Aquí está la definición rota:

```
(defun triangle-bugged (number)
  "Devuelve suma de números 1 a través de NUMBER inclusive."
  (let ((total 0))
    (while (> number 0)
      (setq total (+ total number))
      (setq number (1= number)))      ; Error aquí.
    total))
```

Si se está leyendo esto en Info, se puede evaluar esta definición en el modo normal. Se verá `triangle-bugged` aparece en el área echo.

Ahora evalúa la función `triangle-bugged` con un argumento de 4:

```
(triangle-bugged 4)
```

En un GNU Emacs reciente, se creará e introducirá un búffer `*Backtrace*` que dice:

```
---------- Buffer: *Backtrace* ----------
Debugger entered--Lisp error: (void-function 1=)
  (1= number)
  (setq number (1= number))
  (while (> number 0) (setq total (+ total number))
         (setq number (1= number)))
  (let ((total 0)) (while (> number 0) (setq total ...)
    (setq number ...)) total)
  triangle-bugged(4)
  eval((triangle-bugged 4))
  eval-last-sexp-1(nil)
  eval-last-sexp(nil)
  call-interactively(eval-last-sexp)
---------- Buffer: *Backtrace* ----------
```

(Se ha reformateado este ejemplo ligeramente; el depurador no contiene muchas líneas. Así, se puede salir del depurador escribiendo `q` en el buffer `*Backtrace*`.)

En la práctica, debido a un error tan simple como este, la línea de 'error Lisp' explica lo que se necesita saber para corregir la definición. La función `1=` está 'vacía'.

Sin embargo, si no se conoce con bastante certeza lo que está pasando, se puede leer la traza completa.

En este caso, se necesita ejecutar una versión reciente de GNU Emacs, que automáticamente empieza el depurador que pone en el búffer `*Backtrace*`; o además, se necesita para empezar el depurador manualmente como se describe debajo.

Lee el búffer `*Backtrace*` de abajo a arriba; eso cuenta lo que le hizo a Emacs tener un error. Emacs hace una llamada interactiva a *C-x C-e* (`eval-last-sexp`), que lleva a la evaluación de la expresión `triangle-bugged`. Cada línea de debajo cuenta lo que el intérprete Lisp evaluó.

La tercera línea desde lo alto del búffer es

```
(setq number (1= number))
```

Emacs intentó evaluar esta expresión; para hacerlo así, se intentó evaluar la expresión interna para ser mostrada en la segunda línea desde arriba:

```
(1= number)
```

Aquí es donde el error ocurre; como se dice en la línea de arriba:

```
Debugger entered--Lisp error: (void-function 1=)
```

Se puede corregir el error, reevalúa la definición de función, y entonces se puede testear de nuevo.

17.2 `debug-on-entry`

Un GNU Emacs actual abre el depurador automáticamente cuando la función tenga un error.

Incidentalmente, se puede empezar el depurador manualmente para todas las versiones de Emacs; la ventaja es que el depurador se ejecuta incluso si no se tiene un error en su código. Algunas veces, ¡su código estará libre de errores!

Se puede introducir el depurador cuando se llama a la función llamando `debug-on-entry`.

Tipo:

```
M-x debug-on-entry RET triangle-bugged RET
```

Ahora, evalúa lo siguiente:

```
(triangle-bugged 5)
```

Todas las versiones de Emacs crearán un búffer `*Backtrace*` y cuenta tu que eso es el principio para evaluar la función `triangle-bugged`:

```
---------- Buffer: *Backtrace* ----------
Debugger entered--entering a function:
* triangle-bugged(5)
  eval((triangle-bugged 5))
  eval-last-sexp-1(nil)
  eval-last-sexp(nil)
  call-interactively(eval-last-sexp)
---------- Buffer: *Backtrace* ----------
```

En el búffer `*Backtrace*`, escribe *d*. Emacs evaluará la primera expresión en `triangle-bugged`; el búffer se parece a esto:

```
---------- Buffer: *Backtrace* ----------
Debugger entered--beginning evaluation of function call form:
* (let ((total 0)) (while (> number 0) (setq total ...)
        (setq number ...)) total)
* triangle-bugged(5)
  eval((triangle-bugged 5))
  eval-last-sexp-1(nil)
  eval-last-sexp(nil)
  call-interactively(eval-last-sexp)
---------- Buffer: *Backtrace* ----------
```

Ahora, escribe `d` de nuevo, ocho veces, lentamente. Cada vez que se escribe `d` Emacs evaluará otra expresión en la definición de función.

Eventualmente, el búffer se parece a esto:

```
---------- Buffer: *Backtrace* ----------
Debugger entered--beginning evaluation of function call form:
* (setq number (1= number))
* (while (> number 0) (setq total (+ total number))
        (setq number (1= number)))

* (let ((total 0)) (while (> number 0) (setq total ...)
        (setq number ...)) total)
* triangle-bugged(5)
  eval((triangle-bugged 5))

  eval-last-sexp-1(nil)
  eval-last-sexp(nil)
  call-interactively(eval-last-sexp)
---------- Buffer: *Backtrace* ----------
```

Finalmente, después se escribe `d` dos veces más, Emacs logrará el error y las dos líneas superiores del buffer `*Backtrace*` se ve así:

```
---------- Buffer: *Backtrace* ----------
Debugger entered--Lisp error: (void-function 1=)
* (1= number)
...
---------- Buffer: *Backtrace* ----------
```

Escribiendo `d`, sería capaz de pasear a través de la función.

Se puede salir de un buffer `*Backtrace*` escribiendo `q`; esto se sale de la traza, pero no cancela `debug-on-entry`.

Para cancelar el efecto de `debug-on-entry`, llama a `cancel-debug-on-entry` y el nombre de la función, como esto:

```
M-x cancel-debug-on-entry RET triangle-bugged RET
```

(Si estás leyendo esto en Info, cancela `debug-on-entry` ahora.)

17.3 debug-on-quit y (debug)

Adición a la configuración `debug-on-error` o llamando `debug-on-entry`, hay otros dos caminos para empezar `debug`.

Se puede empezar `debug` siempre y cuando se escribe `C-g` (`keyboard-quit`) se configura la variable `debug-on-quit` para `t`. Esto es útil para depurar bucles infinitos.

O, se puede insertar un línea que dice `(debug)` dentro de tu código donde se quiere que el depurador empiece, así:

```
(defun triangle-bugged (number)
  "Devuelve suma de números 1 a través de NUMERO inclusive."
  (let ((total 0))
    (while (> number 0)
      (setq total (+ total number))
      (debug)                          ; Empieza el depurador.
      (setq number (1= number)))       ; Error aquí.
    total))
```

La función `debug` se describe en detalle en Sección “El Depurador Lisp” in *El Manual de Referencia GNU Emacs Lisp*.

17.4 El depurador de nivel de fuentes `edebug`

Edebug es un depurador a nivel de fuentes Edebug normalmente muestra las fuentes del código que se está depurando, con una flecha a la izquierda que muestra que línea se está actualmente ejecutando.

Se puede pasear a través de la ejecución de una función, línea a línea, o ejecutarse rápidamente hasta lograr un *punto de ruptura* donde la ejecución pare.

Edebug se describe en Sección “Edebug” in *El Manual de Referencia de GNU Emacs Lisp*.

Aquí hay una función con errores para `triangle-recursively`. Véase Sección 11.3.4 “Recursión en lugar de un contador”, página 125, para una revisión de eso.

```
(defun triangle-recursively-bugged (number)
  "Devuelve la suma of números 1 a través de NUMBER inclusive.
Usa recursión."
  (if (= number 1)
      1
    (+ number
       (triangle-recursively-bugged
        (1= number)))))                ; Error aquí.
```

Normalmente, se instalaría esta definición posicionando su cursor después de la función cerrando paréntesis y escribiendo *`C-x C-e`* (`eval-last-sexp`) o lo demás posicionando tu cursor con la definición y escribiendo *`C-M-x`* (`eval-defun`). (Por defecto, el comando `eval-defun` funciona solo en modo Emacs Lisp o en el modo de interacción de Lisp.)

Sin embargo, para preparar esta definición de función para Edebug, se debe primero *instrumentar* el código usando un comando diferente. Se puede hacer esto posicionando el cursor dentro o después de la definición y escribiendo

```
M-x edebug-defun RET
```

Esto causará que Emacs cargue Edebug automáticamente si eso no está ya cargado y, apropiadamente prepara la función.

Después de preparar la función, emplaza tu cursor después de la siguiente expresión y escribe *`C-x C-e`* (`eval-last-sexp`):

```
(triangle-recursively-bugged 3)
```

Se volverá a las fuentes de `triangle-recursively-bugged` y el cursor posicionó al principio del `if` línea de la función. También, verá una flecha a la mano izquierda al lado de esta línea donde la función se está ejecutando. (En los siguientes ejemplos, se muestra la flecha con '`=>`'; en un sistema de ventanas, se puede ver la flecha como un triángulo sólido en el 'borde' de la ventana.)

```
=>*(if (= number 1)
```

En el ejemplo, la posición del punto es mostrado con una estrella, '*' (en Info, eso es mostrado como '`-!-`').

Si ahora se presiona `SPC`, el punto se moverá a la siguiente expresión para ejecutarse; la línea se parece a esto:

```
=>(if *(= number 1)
```

Como se continua presionando `SPC`, el puntero se moverá desde la expresión a la expresión. Al mismo tiempo, siempre y cuando una expresión devuelva un valor, este valor será mostrado en el área echo. Por ejemplo, después de mover el punto pasado `number`, se verá lo siguiente:

```
Resultado: 3 (#o3, #x3, ?\C-c)
```

Esto significa el valor de `number` es 3, que son tres octales, tres hexadecimales, y ASCII 'control-c' (la tercera letra del alfabeto, en caso de que necesites conocer esta información).

Uno puede continuar moviéndose a través del código hasta que logre la línea con el error. Antes de la evaluación, esta línea se parece a esto:

```
=>        *(1= number)))))                ; Error aquí.
```

Cuando se presiona `SPC` una vez de nuevo, se producirá un mensaje de error que dice:

```
La definición de la función está vacío: 1=
```

Este es el error.

Presiona *`q`* para salir de Edebug.

Para eliminar la instrumentación desde una definición de función, simplemente se reevalúa con un comando que no lo instrumente. Por ejemplo, se podría posicionar su cursor después de la definición cerrando paréntesis y escribiendo *`C-x C-e`*.

Edebug hace un gran trato más antes de entrar en una función. Se puede asignar así conducir a sí mismo, parando solo en un error o en punto específicos, se puede causar para mostrar los valores cambiantes de varias expresiones; se puede encontrar cuantas veces una función es llamada, y más.

Edebug se describe en Sección "Edebug" in *El Manual de Referencia de GNU Emacs Lisp*.

17.5 Ejercicios de depuración

- Instale la función `count-words-example` y provoque que se introduzca el depurador construido cuando se llame. Ejecute el comando en una región conteniendo dos palabras. Se necesitará presionar *`d`* un número remarcable de veces. En el sistema, es un 'hook' llamado después que el comando se finaliza. (Para información sobre hooks, mira Sección "Resumen del Comando Bucle" in *El Manual de Referencia GNU Emacs Lisp*.)

- Copie `count-words-example` dentro del búffer `*scratch*`, instrumente la función para Edebug, y navegue a través de su ejecución. La función no necesita tener un error, aunque se puede introducir uno si se desea. Si a la función le falta un error, el paseo se completa sin problemas.
- Mientras se ejecuta Edebug, escriba `?` para ver una lista de todos los comandos Edebug. (El `global-edebug-prefix` normalmente es *`C-x X`*, por ej *`CTRL-x`* seguido por una tecla mayúscula *`X`*; use este prefijo para comandos hechos fuera del búffer de depuración Edebug.)
- En el búffer de depuración Edebug, usa el comando *`p`* (`edebug-bounce-point`) para ver si `count-words-example` está funcionando.
- Mueve el punto a algún sitio debajo de la función y entonces escribe el comando *`h`* (`edebug-goto-here`) para saltar a esta localización.
- Usa el comando *`t`* (`edebug-trace-mode`) para causar que Edebug pasee a través de la función en sí; usa una letra mayúscula *`T`* para `edebug-Trace-fast-mode`.
- Asigna un punto de ruptura, entonces ejecuta Edebug en modo Traza hasta que logre el punto de parada.

18 Conclusión

Ahora se ha llegado al fin de esta Introducción. Se ha aprendido lo suficiente acerca de programación en Emacs Lisp para asignar valores, para escribir ficheros `.emacs` para tí mismo y tus amigos, y escribir personalizaciones simples y extensiones a Emacs.

Este es un lugar para parar. O, si se desea, se puede ir adelante, y aprender más por uno mismo.

Se han aprendido algunas bases de programación. Pero solo algunas. Todavía hay muchas cosas que son fáciles de usar que no se ha tocado.

Otra idea para seguir es leer las fuentes de Emacs o *El Manual de Referencia de GNU Emacs*.

Las fuentes de Emacs Lisp son una aventura. Cuando se leen las fuentes vienen a través de una función o expresión que nos es familiar, se necesita imaginar o encontrar qué se hace.

Ir al Manual de Referencia. Eso está a través del completo, limpio y fácil de leer descripción de Emacs Lisp. Está escrito no solo para expertos, pero sí para gente que conoce lo que usted conoce. (El *Manual de Referencia* viene con la distribución de GNU Emacs. Como esta introducción, viene como un fichero fuente Texinfo, así se puede leer on-line como un libro impreso.)

Ir a otra ayuda on-line que sea parte de GNU Emacs: la documentación on-line para todas las funciones y variables, y `find-tag`, el programa que va a las fuentes.

Aquí hay un ejemplo de cómo explorar las fuentes. Porque su nombre, `simple.el` es el fichero que se vió primero, hace tiempo. Como eso ocurre alguna de las funciones en `simple.el` son complicadas, o al menos parece complicado a primera vista. La función `open-line`, por ejemplo, parece complicada.

Se puede querer pasear a través de esta función lentamente, como nosotros hicimos la función `forward-sentence`. (Véase Sección 12.3 “forward-sentence”, página 137.) O se puede querer escapar de esta función y mira en otra, tal como `split-line`. No se necesita leer todas las funciones. De acuerdo a `count-words-in-defun`, la función `split-line` contiene 102 palabras y símbolos.

Incluso aunque sean pocas, `split-line` contiene expresiones que no se han estudiado: `skip-chars-forward`, `indent-to`, `current-column` y `insert-and-inherit`.

Considera la función `skip-chars-forward`. (Eso es parte de la definición de función para `back-to-indentation`, que es mostrado Sección 3.11 “Revisar”, página 41.)

En GNU Emacs, se puede encontrar más acerca de `skip-chars-forward` escribiendo *`C-h f`* (`describe-function`) y el nombre de la función. Esto te da la documentación de función.

Se puede ser capaz de adivinar que es hecho por una función bien llamada tal como `indent-to`; o se puede buscar, también. Incidentalmente, la función `describe-function` en sí está en `help.el`; eso es una de estos largas, pero descifrables funciones. ¡Se puede buscar `describe-function` usando el comando *`C-h f`*!

En esta instancia, desde el código es Lisp, el búffer `*Help*` contiene el nombre de la librería conteniendo las fuentes de la función. Se puede poner el punto a través del nombre de la librería y presiona la tecla RET, que está en esta situación está asociado a `help-follow`, y se toma directamente de las fuentes, en el mismo camino que *M-.* (`find-tag`).

La definición para `describe-function` ilustra como personalizar las expresiones `interactive` sin usar los códigos de caracter estándar y eso muestra como crear un búffer temporal.

(La función `indent-to` es escrita en C en vez de Emacs Lisp; eso es una función 'construida'. `help-follow` toma su fuente como `find-tag`, cuando se configura apropiadamente.)

Se puede mirar en las fuentes de la función usando `find-tag`, que está asociado a *M-.*. Finalmente, se puede encontrar que el Manual de Referencia tiene que decir visitando el manual en Info, y escribiendo *i* (`Info-index`) y el nombre de la función, o buscando la función en el índice a una copia impresa del manual.

Similarmente, se puede encontrar que significa por `insert-and-inherit`.

Otros ficheros fuente interesantes incluyen `paragraphs.el`, `loaddefs.el` y `loadup.el`. El fichero `paragraphs.el` incluye ordenar, funciones fácilmente comprendidas tan bien como las largas. El fichero `loaddefs.el` contiene muchos autoloads estándar y muchos mapas de teclado. Nunca se ha buscado en todo; solo en las partes. `loadup.el` es el fichero que carga las partes estándar de Emacs; eso cuenta un gran trato acerca de cómo Emacs está construido. (Véase Sección "Construyendo Emacs" in *El Manual de Referencia GNU Emacs Lisp*, para más acerca de construcción.)

Como dije, se han aprendido algunas cosas; sin embargo, y de manera muy importante, se han tocado fuertes aspectos de la programación; no se ha dicho nada acerca de como ordenar la información, excepto para usar la función predefinida `sort`; no se ha dicho nada acerca de cómo almacenar la información, excepto para usar variables y listas; no se ha dicho nada acerca de como escribir programas que escriben programas. Esto son asuntos para otro tipo diferente de libro, un diferente tipo de aprendizaje.

Lo que has hecho es aprender lo suficiente para hacer mucho trabajo práctico con GNU Emacs. Lo realizado es comenzar. Este es el fin del principio de una gran amistad.

Apéndice A La función `the-the`

Algunas veces cuando se se escribe texto, se duplican palabras — como con “se se” cerca del principio de esta frase. Se encuentra que lo más frecuente, es duplicar “el”; aquí, se llama a la función para detectar las palabras duplicadas, `the-the`.

Como primer paso, se podrían usar las siguientes expresiones regulares para buscar duplicados:

```
\\(\\w+[ \t\n]+\\)\\1
```

Este regexp asocia uno o más caracteres que constituyen palabras seguidas por uno o más espacios, tabuladores, o nuevas líneas. Sin embargo, eso no detecta palabras duplicadas en diferentes líneas, desde la finalización de la primera palabra, el fin de la línea, es diferente desde el fin de la segunda palabra, un espacio. (Para más información acerca de expresiones regulares, mira Capítulo 12 “Búsquedas de Expresiones Regulares”, página 135, tan bien como Sección “Sintaxis de Expresiones Regulares” in *El Manual de GNU Emacs*, y Sección “Expresiones Regulares” in *El Manual de Referencia GNU Emacs Lisp*.)

Se podrían intentar buscar caracteres duplicados pero no si el patrón detecta dobles tales como las dos ocurrencias de ‘th’ en ‘with the’.

Otro posible regexp busca caracteres constituyentes de palabras seguidos por caracteres de no palabras constituyentes, reduplicados. Aquí, ‘`\\w+`’ asocia a una o más caracteres de palabras constituyente y ‘`\\W*`’ asocia cero o más caracteres que no constituyen palabras.

```
\\(\\(\\w+\\)\\W*\\)\\1
```

De nuevo, no útil.

Aquí está el patrón que uso. No es perfecto, pero suficientemente bueno. ‘`\\b`’ asocia la cadena vacía provista al principio o fin de una palabra; ‘`[^@ \n\t]+`’ asocia una o más ocurrencias de qué caracteres que *no* son un @-signo, espacio, nueva línea, o tabulador.

```
\\b\\([^@ \n\t]+\\)[ \n\t]+\\1\\b
```

Uno puede escribir expresiones más complicadas, pero esta expresión es suficientemente buena así.

Aquí está la función `the-the`, como se incluye en mi fichero `.emacs`, a lo largo de un atajo global manejable:

```
(defun the-the ()
  "Busca hacia adelante para una palabra duplicada."
  (interactive)
  (message "Buscando palabras duplicadas ...")
  (push-mark)
  ;; Este regexp no es perfecto
  ;; pero es limpiamente bueno a pesar de todo:
  (if (re-search-forward
       "\\b\\([^@ \n\t]+\\)[ \n\t]+\\1\\b" nil 'move)
      (message "Palabra encontrada duplicada.")
    (message "Fin de búffer")))

;; Asocia ‘the-the’ a  C-c \
(global-set-key "\C-c\\" 'the-the)
```

Aquí está el test de texto:

```
uno dos tres cuatro cinco
cinco seis siete
```

Se pueden sustituir las otras expresiones regulares mostradas debajo en la definición de función y prueba cada una de ellas en esta lista.

Apéndice B Manejando el anillo de la muerte

El anillo de la muerte es una lista que es transformada dentro de un anillo que trabaja con la función `current-kill`. Los comandos `yank` y `yank-pop` usan la función `current-kill`.

Este apéndice describe la función `current-kill` tan bien como ambos comandos `yank` y `yank-pop`, pero primero, considera los trabajo del kill ring.

El anillo de la muerte *kill ring* tiene el tamaño máximo de sesenta elementos; hacer una explicación con este número máximo quedaría demasiado larga. En vez de eso, pensemos qué ocurre si se asígna a cuatro. Por favor, evalúe lo siguiente:

```
(setq old-kill-ring-max kill-ring-max)
(setq kill-ring-max 4)
```

Entonces, por favor, copie cada línea del siguiente ejemplo indentado dentro del anillo de la muerte *kill ring*. Se puede cortar cada línea con *C-k* o marcarla y copiarla con *M-w*.

(En un buffer de solo lectura, tal como el buffer `*info*`, el comando kill, *C-k* (`kill-line`), no eliminará el texto, solamente lo mueve al anillo de la muerte *kill ring*. Sin embargo, el ordenador puede avisar con un beep. Alternativamente, para silenciar, se puede copiar la región de cada línea con el comando *M-w* (`kill-ring-save`). Se debe marcar cada línea de este comando para tener éxito, pero no importa si al final se posiciona en el punto o la marca.)

Por favor, invoque las llamadas en orden, de modo que los cinco elementos rellenen el anillo de la muerte *kill ring*.

```
primero algo de texto
segunda pieza de texto
tercera línea
cuarta línea de texto
quinto bit de texto
```

Entonces encuentra el valor de `kill-ring` evaluando

```
kill-ring
```

Eso es:

```
("quinto bit de texto" "cuarta línea de texto"
"tercera línea" "segunda pieza de texto")
```

El primer elemento, '`primero algo de texto`', fué borrado.

Para devolver el viejo valor para el tamaño del kill ring, evalúe:

```
(setq kill-ring-max old-kill-ring-max)
```

B.1 La función `current-kill`

La función `current-kill` cambia el elemento en el anillo de la muerte *kill ring* para el que `kill-ring-yank-pointer` apunta. (También, la función `kill-new` asigna `kill-ring-yank-pointer` para apuntar al último elemento del anillo de la muerte *kill ring*. La función `kill-new` es usada directamente o indirectamente por `kill-append`, `copy-region-as-kill`, `kill-ring-save`, `kill-line`, y `kill-region`.)

La función `current-kill` es usada por `yank` y por `yank-pop`. Aquí está el código para `current-kill`:

```
(defun current-kill (n &optional do-not-move)
"Rota el punto de pegue por N lugares, y entonces devuelve lo cortado.
Si N es cero, ‘interprogram-paste-function’ se asigna, y si se llama
devuelve una cadena, entonces esta cadena se añade al frente del
anillo de la muerte kill ring y devuelve el último corte.
Si el argumento opcional DO-NOT-MOVE es no nulo, entonces no muevas el
punto de pegue; solo devuelve el Nth corte hacia adelante.
   (let ((interprogram-paste (and (= n 0)
                                  interprogram-paste-function
                                  (funcall interprogram-paste-function)))))
    (if interprogram-paste
        (progn
          ;; Deshabilita el programa de la función de corte cuando se
          ;; añade el nuevo texto al anillo de la muerte kill ring,
          ;; así Emacs no intenta poseer la selección
          ;; con idéntico texto.
          (let ((interprogram-cut-function nil))
            (kill-new interprogram-paste))
          interprogram-paste)
      (or kill-ring (error "Kill ring is empty"))
      (let ((ARGth-kill-element
             (nthcdr (mod (- n (length kill-ring-yank-pointer))
                          (length kill-ring))
                     kill-ring)))
        (or do-not-move
            (setq kill-ring-yank-pointer ARGth-kill-element))
        (car ARGth-kill-element)))))
```

Recuerde también que la función `kill-new` asigna `kill-ring-yank-pointer` al último elemento del anillo de la muerte *kill ring*, que significa que todas las funciones lo llaman y asigna el valor de manera indirecta: `kill-append`, `copy-region-as-kill`, `kill-ring-save`, `kill-line` y `kill-region`.

Aquí está la línea en `kill-new`, que se explica en la “La función `kill-new`”, página 94.

```
(setq kill-ring-yank-pointer kill-ring)
```

La función `current-kill` parece compleja, pero usual, eso puede ser comprendido tomándolo aparte pieza por pieza. Primero míralo en la forma esquelética:

```
(defun current-kill (n &optional do-not-move)
  "Rota el punto a pegar por N lugares, y entonces devuelve el texto cortado."
  (let varlist
    body...)
```

Esta función tiene dos argumentos, uno es opcional. Hay una cadena de documentación. *No* es una función interactiva.

El cuerpo de la definición de función es una expresión `let`, que por sí misma tiene tanto un cuerpo como una *varlist*.

La expresión `let` declara una variable que será solo usable con las asociaciones de esta función. Esta variable se llama `interprogram-paste` y se copia a otro programa. No se copia con esta instancia de GNU Emacs. La mayoría de los sistemas

de ventanas proveen una facilidad para pegar el interprograma. Tristemente, esta facilidad normalmente provee solo el último elemento. La mayoría de los sistemas de ventanas no han adoptado un anillo de muchas posibilidades, incluso aunque Emacs haya provisto esto durante décadas.

La expresión `if` tiene dos partes, una si existe `interprogram-paste` y otra si no.

Permítenos considerar el 'si no' o la parte else de la función `current-kill`. (La parte then usa la función `kill-new`, que ya hemos descrito. Véase "La función `kill-new`.", página 94)

```
(or kill-ring (error "El Kill ring está vacío"))
(let ((ARGth-kill-element
       (nthcdr (mod (- n (length kill-ring-yank-pointer))
                    (length kill-ring))
               kill-ring)))
  (or do-not-move
      (setq kill-ring-yank-pointer ARGth-kill-element))
  (car ARGth-kill-element))
```

El código primero chequea si el kill ring *anillo de la muerte* tiene contenido; de otro modo señala un error.

Note que la expresión `or` es muy similar para testear el tamaño con un `if`:

```
(if (zerop (length kill-ring))          ; parte-si
    (error "Anillo de la muerte vacío"))       ; parte-entonces
  ;; No hay parte-resto
```

Si no hay nada en el kill ring *anillo de la muerte*, su tamaño debe ser cero y un mensaje de error se envía al usuario: '`El kill ring está vacío`'. La función `current-kill` usa una expresión `or` que es simple. Pero una expresión `if` recuerda lo que lleva.

Esta expresión `if` usa la función `zerop` que devuelve cierto si el valor que se chequea es cero. Cuando `zerop` chequea cierto, la parte then del `if` se evalúa. La parte then es una lista empezando con la función `error`, que es una función que es similar a la función `message` (véase Sección 1.8.5 "La Función `message`", página 14) que imprime un mensaje de una línea en el área echo. Sin embargo, además de imprimir un mensaje, `error` también evalúa la función que está embebida. Esto significa que el resto de la función no será evaluada si el tamaño del anillo de la muerte *kill ring* es cero.

Entonces la función `current-kill` selecciona el elemento a devolver. La selección depende del número de lugares que `current-kill` rota y donde `kill-ring-yank-pointer` apunta.

Lo siguiente, si el argumento `do-not-move` opcional es verdadero o el actual valor de `kill-ring-yank-pointer` se establece al punto de la lista. Finalmente, otra expresión devuelve el primer elemento de la lista incluso si el argumento `do-not-move` es verdadero.

En mi opinión, es ligeramente erróneo, al menos para humanos, usar el término 'error' como el nombre de la función `error`. Un término mejor sería 'cancelar'. Estrictamente hablando, de acuerdo, no se puede apuntar, mucho menos rotar un puntero a una lista que no tiene tamaño, así desde el punto de vista del ordenador,

la palabra 'error' es correcta. Pero un humano espera intentar algo, si solo si se encuentra el anillo de la muerte *kill ring* esté lleno o vacío. Esto es un acto de exploración.

Desde el punto de vista humano, el acto de exploración y descubrimiento no es necesariamente un error, y por esta razón no sería etiquetado como tal, incluso las vocales de un ordenador. Como tal, el código en Emacs implica que un humano que está actuando virtuosamente, explorando su entorno, está teniendo un error. Esto está mal. Incluso aunque el ordenador tome los mismos pasos como cuando hay 'error', un término tal como 'cancelar' tendría una clara connotación.

Entre otras acciones, la else-part de la expresión `if` asigna el valor de `kill-ring-yank-pointer` a `ARGth-kill-element` cuando el kill ring *anillo de la muerte* tiene alguna cosa dentro y el valor de `do-not-move` es `nil`.

El código se parece a esto:

```
(nthcdr (mod (- n (length kill-ring-yank-pointer))
             (length kill-ring))
        kill-ring)))
```

Esto necesita algún examen. A menos que no se suponga mover el puntero, la función `current-kill` cambia donde `kill-ring-yank-pointer` apunta. Esto es lo que el `(setq kill-ring-yank-pointer ARGth-kill-element)` expresión hace. También, claramente, `ARGth-kill-element` está siendo asignado a ser igual para algún CDR del anillo de la muerte *kill ring*, usando la función `nthcdr` que está descrita en una sección temprana. (Véase Sección 8.3 "copy-region-as-kill", página 90.) ¿Cómo se hace?

Como se ha visto antes (véase Sección 7.3 "nthcdr", página 77), la función `nthcdr` funciona repetidamente tomando el CDR de una lista — eso toma el CDR del CDR del CDR . . .

Las siguientes dos expresiones producen el mismo resultado:

```
(setq kill-ring-yank-pointer (cdr kill-ring))

(setq kill-ring-yank-pointer (nthcdr 1 kill-ring))
```

Sin embargo, la expresión `nthcdr` es más complicada. Usa la función `mod` para determinar que CDR para seleccionar.

(Se recordará buscar funciones propias primero, en vez de esto, tendremos que ir dentro del `mod`.)

La función `mod` devuelve el valor de su primer argumento modulo el segundo; que es decir, eso devuelve el resto después de dividir el primer argumento por el segundo. El valor devulto tiene el mismo signo que el segundo argumento.

De este modo,

```
(mod 12 4)
  ⇒ 0  ;; porque no hay resto
(mod 13 4)
  ⇒ 1
```

En este caso, el primer argumento es con frecuencia pequeño que el segundo. Que está bien.

```
(mod 0 4)
  ⇒ 0
(mod 1 4)
  ⇒ 1
```

Se puede adivinar que la función `-` hace. Eso es como `+` pero sustrae en vez de añadir; la función `-` sustrae su segundo argumento desde el primero. También, ya se sabe que la función `length` hace (véase Sección 7.2.1 "length", página 76). Eso devuelve el tamaño de una lista.

Y `n` es el nombre del argumento requerido a la función `current-kill`.

Así cuando el primer argumento a `nthcdr` es cero, la expresión `nthcdr` devuelve la lista entera, como se puede ver evaluando lo siguiente:

```
;; kill-ring-yank-pointer and kill-ring tener un tamaño de cuatro
;; and (mod (- 0 4) 4) ⇒ 0
(nthcdr (mod (- 0 4) 4)
        '("cuarta línea de texto"
          "tercera línea"
          "segunda pieza de texto"
          "primero algo de texto"))
```

Cuando el primer argumento a la función `current-kill` es uno, la expresión `nthcdr` devuelve la lista sin su primer elemento.

```
(nthcdr (mod (- 1 4) 4)
        '("cuarta línea de texto"
          "tercera línea"
          "segunda pieza de texto"
          "primero algo de texto"))
```

Incidentalmente, tanto `kill-ring` y `kill-ring-yank-pointer` son *variables globales*. Esto significa que cualquier expresión en Emacs Lisp puede acceder a ellas. Ellas no son como las variables locales asignadas por `let` o como los símbolos en una lista de argumentos. Las variables locales pueden solo ser accedidas con el `let` que los define o la función que los especifica en una lista de argumentos (y con expresiones llamadas por ellos).

B.2 pegar

Después de aprender acerca de `current-kill`, el código para la función `yank` es casi fácil.

La función `yank` no usa variable `kill-ring-yank-pointer` directamente. Eso llama a `insert-for-yank` que llama a `current-kill` que asigna la variable `kill-ring-yank-pointer`.

El código se parece a esto:

```
(defun yank (&optional arg)
  "Reinserta (\"pega\") el último logro del texto cortado.
Más precisamente, reinserta el texto cortado más recientemente.
Pon el punto al final, y asigna la marca al principio.
Solo con \\[universal-argument] como argumento, lo mismo pero pon el
punto al principio (y marca al final). Con el argumento N, reinserta
el N más recientemente cortado.
Cuando este comando inserta texto cortado dentro del búffer, eso
honra a ‘yank-excluded-properties’ y ‘yank-handler’ como se describe
la cadena de documentación para ‘insert-for-yank-1’, que se ve.
Ver también el comando \\[yank-pop]."
  (interactive "*P")
  (setq yank-window-start (window-start))
  ;; Si no tenemos todo el camino a través, crea last-command que
  ;; indique esto para el siguiente comando.
  (setq this-command t)
  (push-mark (point))
  (insert-for-yank (current-kill (cond
                                  ((listp arg) 0)
                                  ((eq arg '-) -2)
                                  (t (1- arg)))))
  (if (consp arg)
      ;; Esto es como like exchange-point-and-mark,
      ;;     pero no activa la marca.
      ;; Es limpio evitar la activación, incluso aunque el comando
      ;; loop would deactivaría la marca porque se
      ;; insertara el texto.
      (goto-char (prog1 (mark t)
                   (set-marker (mark-marker) (point) (current-buffer)))))
  ;; Si tenemos todo el camino, haz que this-command lo indique.
  (if (eq this-command t)
      (setq this-command 'yank))
  nil)
```

La expresión clave es `insert-for-yank`, que inserta la cadena devuelta por `current-kill`, pero elimina algo de propiedades de texto desde eso.

Sin embargo, antes de tener esta expresión, la función asigna el valor de `yank-window-start` a la posición devuelta por la expresión `(window-start)`, la posición que muestra lo que actualmente empieza. La función `yank` también asigna `this-command` y empuja la marca.

Después de pegar el elemento apropiado, si el argumento opcional es un CONS en vez de un número o nada, se pone el punto al principio del texto pegado y se marca al final.

(La función `prog1` es como `progn` pero devuelve el valor de su primer argumento en vez del valor de su último argumento. Su primer argumento fuerza devolver la marca del búffer como un entero. Se puede ver la documentación para estas funciones emplazando el punto a través de ellas en este búffer y entonces escribiendo *`C-h f`* (`describe-function`) seguido por un *`RET`*; por defecto es la función.)

La última parte de la función cuenta que hacer cuando eso sucede.

B.3 yank-pop

Después de comprender `yank` y `current-kill`, se conoce como enfocar la función `yank-pop`. Dejando fuera la documentación para guardar el espacio, se parece a esto:

```
(defun yank-pop (&optional arg)
  "..."
  (interactive "*p")
  (if (not (eq last-command 'yank))
      (error "El comando previo no fué un corte"))
  (setq this-command 'yank)
  (unless arg (setq arg 1))
  (let ((inhibit-read-only t)
        (before (< (point) (mark t))))
    (if before
        (funcall (or yank-undo-function 'delete-region) (point) (mark t))
      (funcall (or yank-undo-function 'delete-region) (mark t) (point)))
    (setq yank-undo-function nil)
    (set-marker (mark-marker) (point) (current-buffer))
    (insert-for-yank (current-kill arg))
    ;; Asigna la ventana a volver donde estaba el comando yank,
    ;; si es posible
    (set-window-start (selected-window) yank-window-start t)
    (if before
        ;; Esto es como exchange-point-and-mark,
        ;; pero no activa la marca.
        ;; Es limpio evitar la activación, incluso aunque el comando
        ;; desactivase la marca porque se insertara el texto.
        (goto-char (prog1 (mark t)
                     (set-marker (mark-marker)
                                 (point)
                                 (current-buffer))))))
  nil)
```

La función es interactive con una pequeña '`p`' así el argumento prefijo es procesado y pasado a la función. El comando puede solo ser usado después del yank previo; de otro modo un mensaje de error se envía. Este chequeo usa la variable `last-command` que se asigna por `yank` y discutida de algún modo. (Véase Sección 8.3 "copy-region-as-kill", página 90.)

La cláusula `let` asigna la variable `before` a cierto o falso dependiendo de si el punto está antes o después de la marca y entonce la región entre punto y marca se borra. Esta es la región que fué insertada por el yank previo y eso es este texto que será reemplazado.

`funcall` llama a su primer argumento como una función, pasando los argumentos que permanecen. El primer argumento es el que la expresión `or` devuelve. Los dos argumentos que permanecen son las posiciones de punto y marca asignadas por el comando `yank` precedente.

Hay más, pero esta es la parte más dura.

B.4 El fichero `ring.el`

De manera interesante, GNU Emacs posee un fichero llamado `ring.el` que provee muchas de las funcionalidades que ahora se discuten. Pero las funciones tales como `kill-ring-yank-pointer` no usan esta librería, posiblemente porque fueron escritas pronto.

Apéndice C Un grafo con ejes etiquetados

Los ejes impresos ayudan a comprender un grafo. Para crear escalas. En un capítulo anterior (véase Capítulo 15 "Leyendo un grafo", página 182), se escribió el código para imprimir el cuerpo de un grafo. Aquí se escribe el código para imprimir y etiquetar ejes horizontales y verticales, a lo largo del cuerpo en sí.

Puesto que las inserciones rellenan un búffer a la derecha y debajo del punto, el nuevo grafo imprime la función que primero imprimiría el eje vertical Y, después el cuerpo del grafo, y finalmente el eje horizontal X. Esta secuencia nos da los contenidos de la función:

1. Configura código.
2. Imprime el eje Y.
3. Imprime el cuerpo del grafo.
4. Imprime el eje X.

Aquí hay un ejemplo de como se ve un grafo finalizado:

```
10 -
                *
                *  *
                *  **
                *  ***
 5 -      *    *******
        * *** *******
        ************
      ***************
 1 - ****************
     |   |    |    |
     1   5   10   15
```

En este grafo, en ambos ejes vertical y horizontal se etiquetan con números. Sin embargo, en algunos grafos, el eje horizontal es tiempo y estaría mejor etiquetarlo con meses, así:

```
5 -      *
       * ** *
       *******
      ********** **
1 - **************
    |    ^       |
  Enero Junio Enero
```

Dentro, con un pequeño pensamiento, se puede fácilmente venir con una variedad de esquemas de etiquetado verticales y horizontales. Nuestra tarea podría llegar a ser complicada. Pero las complicaciones generan confusión. En vez de permitir esto, es mejor elegir un simple esquema de etiquetado para nuestro primer esfuerzo, y modificarlo o reemplazarlo después.

Estas consideraciones sugieren el siguiente outline para la función `print-graph`:

```
(defun print-graph (numbers-list)
  "documentation..."
  (let ((height  ...
         ...))
    (print-Y-axis height ... )
    (graph-body-print numbers-list)
    (print-X-axis ... )))
```

Nosotros podemos trabajar en cada parte de la definición de función `print-graph`.

C.1 La varlist `print-graph`

Para escribir la función `print-graph`, la primera tarea es escribir la varlist en la expresión `let`. (Nosotros dejaremos por ahora cualquier pensamiento acerca de hacer la función interactive o acerca de los contenidos de su cadena de documentación.)

La varlist asignaría varios valores. Claramente, la etiqueta superior del eje vertical debe ser al menos la altura del grafo, que significa que debe obtener esta información aquí. Note que la función `print-graph-body` también requiere esta información. No hay razón para calcular la altura del grafo en dos lugares diferentes, así cambiaría `print-graph-body` desde el camino que definimos pronto para tomar ventaja del cálculo.

De manera similar, tanto la función para imprimir la etiqueta del eje X y la función `print-graph-body` se necesita aprender el valor del ancho de cada símbolo. Se puede desarrollar el cálculo aquí y cambiar la definición para `print-graph-body` desde el camino que se definió en el capítulo previo.

El tamaño de la etiqueta para el eje horizontal debe ser al menos tan largo como el grafo. Sin embargo, esta información es usada solo en la función que imprime el eje horizontal, así no necesita calcularse aquí.

Estos pensamientos nos llevan directamente a la siguiente forma para la varlist en el `let` para `print-graph`:

```
(let ((height (apply 'max numbers-list)) ; Primera versión.
      (symbol-width (length graph-blank)))
```

Como se verá, esta expresión no es bastante correcta.

C.2 La función `print-Y-axis`

El trabajo de la función `print-Y-axis` es imprimir una etiqueta para el eje vertical que se parece a esto:

```
10 -


 5 -


 1 -
```

La función se pasaría a lo alto del grafo, y así construyen e insertan los números y marcas apropiados.

Es suficientemente fácil ver en la figura que la etiqueta del eje Y se miraría así; pero decir en palabras, y entonces escribir una definición de función para hacer el trabajo es otra materia. No es bastante verdad decir que se quiere un número y un tic cada cinco líneas: solo hay tres líneas entre el '`1`' y el '`5`' (líneas 2, 3 y 4), pero cuatro líneas entre el '`5`' y el '`10`' (líneas 6, 7, 8 y 9). Es mejor decir que se quiere un número y un tic en la quinta línea desde abajo a cada línea que es un múltiplo de cinco.

La siguiente cuestión es a que altura se etiquetaría. Supón que la máxima altura de la columna mayor del grafo es siete. La etiqueta superior en el eje Y sería '`5 -`', ¿y el grafo se pegaría debajo de la etiqueta?, ¿o la etiqueta superior sería '`7 -`', y marcar la vertical del grafo? ¿o sería la etiqueta superior `10 -`, que es múltiplo de cinco, y es superior al valor más alto del grafo?

La última forma es preferida. La mayoría de los grafos son rectángulos cuyos lados son un número integral de pasos a lo largo — 5, 10, 15, y así para un paso a distancia de cinco. Pero tan pronto se decide usar un paso alto para el eje vertical, se descubre que la expresión simple en la varlist para la altura de la computación es errónea. La expresión es `(apply 'max numbers-list)`. Esto devuelve la altura precisa, no la altura máxima más de lo que es necesario para redondear el múltiplo de cinco. Una expresión más compleja es requerida.

Como es normal en casos como este, un problema complejo llega a ser simple si eso está dividido en varios problemas pequeños.

Primero, considera el caso cuando el valor superior del grafo es un múltiplo integral de cinco — cuando eso es 5, 10, 15, o algún múltiplo de cinco. Se puede usar este valor como la altura del eje Y.

Un camino simple y limpio para determinar si un número es múltiplo de cinco se divide por cinco y mira si la división devuelve resti. Si no hay resto, el número es un múltiplo de cinco. De este modo, siete dividido tiene un resto de dos, y siete no es un entero múltiplo de cinco. Dicho de otra manera, recordando la escuela, cinco entre siete es uno y me llevo dos. Sin embargo, diez entre dos, no tiene resto: diez es un múltiplo entero de cinco.

C.2.1 Viaje lateral: Calcula un resto

En Lisp, la función para calcular un resto es `%`. La función devuelve el resto de su primer argumento dividido por su segundo argumento. Como ocurre, `%` es una función en Emacs Lisp que no se puede implementar usando `apropos`: no se puede encontrar nada si se escribe *`M-x apropos RET resto RET`*. El único camino para aprender la existencia de `%` es leer acerca de eso en un libro tal como este o en las fuentes de Emacs Lisp.

Se puede probar la función `%` evaluando las siguientes dos expresiones:

```
(% 7 5)

(% 10 5)
```

La primera expresión devuelve 2 y la segunda expresión devuelve 0.

Para chequear si el valor devuelto es cero o algún otro número, se puede usar la función `zerop`. Esta función devuelve `t` si su argumento debe ser un número, es cero.

```
(zerop (% 7 5))
     ⇒ nil

(zerop (% 10 5))
     ⇒ t
```

De este modo, la siguiente expresión devolverá `t` si la altura del grafo es divisible por cinco:

```
(zerop (% height 5))
```

(El valor de `height`, de acuerdo, puede ser encontrado desde `(apply 'max numbers-list)`.)

Por otro lado, si el valor de `height` no es un múltiplo de cinco, nosotros queremos resetear el valor al siguiente múltiplo de cinco. Esta es la aritmética sencilla usando funciones con las que ya se está familiarizado. Primero, se divide el valor de `height` por cinco para determinar cuantas veces cinco va dentro del número. De este modo, cinco va dentro doce veces. Si se añade uno a este cociente y se multiplica por cinco, obtendremos el valor del siguiente múltiplo de cinco que es más largo que el mayor. Cinco va dentro de doce dos veces. Añade uno a dos, y multiplica por cinco; el resultado es quince, que es el siguiente múltiplo de cinco que es mayor que doce. La expresión Lisp para esto es:

```
(* (1+ (/ height 5)) 5)
```

Por ejemplo, si se evalúa lo siguiente, el resultado es 15:

```
(* (1+ (/ 12 5)) 5)
```

Todo a través de esta discusión, se ha estado usando 'cinco' como el valor para las etiquetas espaciadas en el eje Y; pero se puede querer usar algún otro valor. Generalmente, reemplazaría 'cinco' con una variable a la que poder asignar un valor. El mejor nombre que puedo pensar para esta variable es `Y-axis-label-spacing`.

Usando este término, y una expresión `if`, se produce lo siguiente:

```
(if (zerop (% height Y-axis-label-spacing))
    height
  ;; else
  (* (1+ (/ height Y-axis-label-spacing))
     Y-axis-label-spacing))
```

Esta expresión devuelve el valor de `height` en sí si la altura es incluso un múltiplo del valor del `Y-axis-label-spacing` o lo demás computa y devuelve un valor de `height` que es igual al siguiente múltiplo mayor del valor del `Y-axis-label-spacing`.

Se puede ahora incluir esta expresión en la expresión `let` de la función `print-graph` (después de la primera configuración del valor de `Y-axis-label-spacing`):

```
(defvar Y-axis-label-spacing 5
  "Número de líneas desde una etiqueta del eje Y al siguiente.")

...
(let* ((height (apply 'max numbers-list))
       (height-of-top-line
        (if (zerop (% height Y-axis-label-spacing))
            height
          ;; else
          (* (1+ (/ height Y-axis-label-spacing))
             Y-axis-label-spacing)))
       (symbol-width (length graph-blank))))
...
```

(Nota el uso de la función `let*`: el valor inicial de la altura es calculada una vez por la expresión `(apply 'max numbers-list)` y entonces el valor resultado de `height` es usado para computar su valor final. Véase "La expresión `let*`", página 142, para más acerca de `let*`.)

C.2.2 Construye un elemento del eje Y

Cuando se imprime el eje vertical, se quieren insertar cadenas tales como '`5 -`' y '`10 -` ' cada cinco líneas. Más allá, se quieren los números agitados para alinear, así pocos números deben ser acuñados con espacios de guía. Si alguna de las cadenas usan dos dígitos, las cadenas con un simple dígito deben incluir una guía en blanco antes del número.

Para figurarse el tamaño del número, se usa la función `length`. Pero la función `length` funciona solo con una cadena, no con un número. Así el número tiene que ser convertido desde un número a una cadena. Esto es hecho con la función `number-to-string`. Por ejemplo,

```
(length (number-to-string 35))
     ⇒ 2

(length (number-to-string 100))
     ⇒ 3
```

(`number-to-string` es también llamado `int-to-string`; se verá este nombre alternativo en varias fuentes.)

Además, en cada etiqueta, cada número es seguido por una cadena tal como ‘ - ’, que llamará al marcador `Y-axis-tic`. Esta variable está definida con `defvar`:

```
(defvar Y-axis-tic " - "
   "La Cadena que sigue el número en una etiqueta del eje Y.")
```

El tamaño de la etiqueta Y es la suma del tamaño del eje Y y el tamaño del número del alto del grafo.

```
(length (concat (number-to-string height) Y-axis-tic)))
```

Este valor será calculado por la función `print-graph` en su varlist como `full-Y-label-width` y se pasa dentro. (Note que no se pensaba en incluir esto en el varlist cuando se propuso.)

Para crear un eje vertical completo, una marca de tic es concatenada con un número; y los dos juntos pueden ser precedidos por uno o más espacios dependiendo de cómo de largo es el número. La etiqueta consiste de tres partes: los espacios que se lideran (opcional), el número, y la marca tic. La función se pasa al valor del número para la fila específica, y el valor del ancho de la línea de arriba, que es calculada (solo una vez) por `print-graph`.

```
(defun Y-axis-element (number full-Y-label-width)
  "Construye una etiqueta NUMERADA
Un elemento numerado se parece a esto ‘ 5 - ’,
y está tan acuñado como se necesita así todo se
alinea con el elemento para el número mayor."
  (let* ((leading-spaces
          (- full-Y-label-width
             (length
              (concat (number-to-string number)
                      Y-axis-tic)))))
    (concat
     (make-string leading-spaces ? )
     (number-to-string number)
     Y-axis-tic)))
```

La función `Y-axis-element` concatena junto los espacios que se lideran si cualquiera; el número, como una cadena; y la marca tic.

Para imaginarnos cuantos espacios guía la etiqueta necesita, la función sustrae el tamaño de la etiqueta — el tamaño del número más el tamaño de la marca tic — desde el ancho de la etiqueta deseada.

Los espacios en blanco se insertan usando la función `make-string`. Esta función tiene dos argumentos: lo primero cuenta como de larga será a cadena y el segundo es un símbolo para el caracter a insertar, en un formato espcial. El formato es una marca de pregunta seguida por un espacio en blanco, como este, ‘`?`’. Véase Sección “Tipo de Caracter” in *El Manual de Referencia Emacs Lisp*, para una descripción de la sintaxis para caracteres. (De acuerdo, se podría querer reemplazar el espacio en blanco por algún otro caracter . . . . Tu sabes qué hacer.)

La función `number-to-string` es usada en la expresión de concatenación, para convertir el número a una cadena que es concatenada con los espacios que se lideran y la marca de tic.

C.2.3 Crea un eje de la columna Y

Las funciones precedentes proporcionan todas las herramientas necesarias para construir una función que genera una lista de cadenas enumeradas y en blanco para inserta como la etiqueta para el eje vertical:

```
(defun Y-axis-column (height width-of-label)
  "Construye la lista de ejes Y etiquetadas y cadenas en blanco.
Para height la altura de la línea de debajo y width-of-label."
  (let (Y-axis)

    (while (> height 1)
      (if (zerop (% height Y-axis-label-spacing))
          ;; Insertar etiqueta.
          (setq Y-axis
                (cons
                 (Y-axis-element height width-of-label)
                 Y-axis))

        ;; Else, insertar blancos.
        (setq Y-axis
              (cons
               (make-string width-of-label ? )
               Y-axis)))
      (setq height (1- height)))
    ;; Insertar la línea base.
    (setq Y-axis
          (cons (Y-axis-element 1 width-of-label) Y-axis))
    (nreverse Y-axis)))
```

En esta función, nosotros empezamos con el valor de `height` y repetitivamente sustrae uno desde su valor. Después de cada sustración, se chequea para ver si el valor es una integral múltiple del `Y-axis-label-spacing`. Si eso es, se construye una etiqueta numerada usando la función `Y-axis-element`; si no, se construye una etiqueta blanca usando la función `make-string`. La línea base consiste del número uno seguido por una marca tic.

C.2.4 La versión no demasiado final de `print-Y-axis`

La lista construida por la función `Y-axis-column` está pasada a la función `print-Y-axis`, que inserta la lista como una columna.

```
(defun print-Y-axis (height full-Y-label-width)
  "Inserta el eje Y usando HEIGHT y FULL-Y-LABEL-WIDTH.
La altura debe ser la máxima altura del grafo.
El ancho completo es el ancho del mayor elemento de la
etiqueta"
;; El valor del alto y full-Y-label-width
;; son pasadas por 'print-graph'.
  (let ((start (point)))
    (insert-rectangle
     (Y-axis-column height full-Y-label-width))
    ;; Posiciona el punto listo para inserta el grafo.
    (goto-char start)
    ;; Mueve el punto hacia adelante por valor de full-Y-label-width
    (forward-char full-Y-label-width)))
```

El `print-Y-axis` usa la función `insert-rectangle` para inserta el eje Y creado por la función `Y-axis-column`. Además, eso emplaza el punto en la posición correcta para imprimir el cuerpo del grafo.

Se puede chequear `print-Y-axis`:

1. Instalar
   ```
   Y-axis-label-spacing
   Y-axis-tic
   Y-axis-element
   Y-axis-column
   print-Y-axis
   ```
2. Copia la siguiente expresión:
   ```
   (print-Y-axis 12 5)
   ```
3. Cambia al búffer `*scratch*` y emplaza el cursor donde se quiere el eje etiquetado para empezar.
4. Escribe *`M-:`* (`eval-expression`).
5. Pega la expresión `graph-body-print` dentro del minibuffer con *`C-y`* (`yank`).
6. Presiona `RET` para evaluar la expresión

Emacs imprimirá etiquetas verticalmente, el primero siendo '`10 - `'. (La función `print-graph` pasará el valor de `height-of-top-line`, que en este caso finalizará en 15, por esto lo que se obtiene podría aparecer como un error.)

C.3 La función `print-X-axis`

Las etiquetas del eje X son como las etiquetas del eje Y, excepto que los ticks son un línea debajo de los números. Las etiquetas se parece como esto:

```
|   |     |     |
1   5    10    15
```

El primer tic está bajo la primera columna del grafo y está precedido por varios espacios en blanco. Estos espacios proporcionan la habitación en filas de debajo para las etiquetas del eje Y. El segundo, tercer, cuarto, y subsiguientes ticks son todos espaciados igualmente, de acuerdo al valor de `X-axis-label-spacing`.

La segunda fila del eje X consiste de números, precedidos por varios espacios en blanco y también separado de acuerdo al valor de la variable `X-axis-label-spacing`.

El valor de la variable `X-axis-label-spacing` sería medido en unidades de `symbol-width`, puesto que se puede querer cambiar el ancho de los símbolos que estás usando para imprimir el cuerpo del grafo sin cambiar los caminos del grafo que está etiquetado.

La función `print-X-axis` está construida más o menos del mismo modo como que la función `print-Y-axis` excepto que tiene dos líneas: la línea de marcas tic y los números. Nosotros escribiremos una función separado a imprimir cada línea y entonces combinarlo con la función `print-X-axis`.

Esto es un proceso de tres pasos:

1. Escribe una función para imprimir el eje X marca tic, `print-X-axis-tic-line`.

2. Escribe una función imprime los números X, `print-X-axis-numbered-line`.

3. Escribe una función para imprimir ambas líneas, la función `print-X-axis`, usando `print-X-axis-tic-line` y `print-X-axis-numbered-line`.

C.3.1 Eje X marca tic

La primera función imprimiría las marcas de tic del eje X. Se deben especificar las marcas en sí y su espacio:

```
(defvar X-axis-label-spacing
  (if (boundp 'graph-blank)
      (* 5 (length graph-blank)) 5)
  "Números de unidades desde un eje X al siguiente.")
```

(Note que el valor de `graph-blank` est'a asignado por otro `defvar`. El predicado `boundp` chequea si ya ha sido asignado; `boundp` devuelve `nil` si no lo tiene. Si `graph-blank` fuera disociado y no usara esta construcción condicional, en un GNU Emacs reciente, se introduciría el depurador y mirará un mensaje de error diciendo '`Debugger entered--Lisp error: (void-variable graph-blank)`'

Aquí está el `defvar` para `X-axis-tic-symbol`:

```
(defvar X-axis-tic-symbol "|"
  "Cadena para insertar para apuntar a una columna en el eje X.")
```

El objetivo es crear una línea que se parece a esto:

```
        |   |    |    |
```

El primer tic es indentado así que está bajo la primera columna, que es indentado para proveer espacio para las etiquetas del eje Y.

Un elemento tic consiste en espacios en blanco que se extienden desde un tic al siguiente más un símbolo tic. El número de espacios en blanco se determinan por el ancho del símbolo tic y el `X-axis-label-spacing`.

El código se parece a esto:

```
;;; X-axis-tic-element
...
(concat
 (make-string
  ;; Crea una cadena de blancos.
  (-  (* symbol-width X-axis-label-spacing)
      (length X-axis-tic-symbol))
  ? )
 ;; Concatena blancos con símbolos.
 X-axis-tic-symbol)
...
```

Lo siguiente, determina cuantos espacios en blanco son necesarios para indentar la primera marca tic a la primera del grafo. Esto usa el valor de `full-Y-label-width` pasaba por la función `print-graph`.

El código para crear `X-axis-leading-spaces` se parece a esto:

```
;; X-axis-leading-spaces
...
(make-string full-Y-label-width ? )
...
```

También necesita determinar el tamaño del eje horizontal, que es el tamaño de la lista de números, y el número de ticks en el eje horizontal:

```
;; X-length
...
(length numbers-list)

;; tic-width
...
(* symbol-width X-axis-label-spacing)

;; number-of-X-ticks
(if (zerop (% (X-length tic-width)))
    (/ (X-length tic-width))
  (1+ (/ (X-length tic-width))))
```

Todo esto lidera directamente a la función para imprimir el eje X:

```
(defun print-X-axis-tic-line
  (number-of-X-tics X-axis-leading-spaces X-axis-tic-element)
  "Imprime ticks para el eje X."
    (insert X-axis-leading-spaces)
    (insert X-axis-tic-symbol)  ; En la primera columna.
    ;; Inserta el segundo tic en el lugar adecuado.
    (insert (concat
             (make-string
              (-  (* symbol-width X-axis-label-spacing)
                  ;; Inserta el espacio en blanco al segundo símbolo tic.
                  (* 2 (length X-axis-tic-symbol)))
              ? )
             X-axis-tic-symbol))
    ;; Inserta los ticks que permanecen.
    (while (> number-of-X-tics 1)
      (insert X-axis-tic-element)
      (setq number-of-X-tics (1- number-of-X-tics))))
```

La línea de números es igualmente simple:

Primero, creamos un elemento numerado con espacios en blanco antes de cada número:

```
(defun X-axis-element (number)
  "Construye un elemento del eje X numerado."
  (let ((leading-spaces
         (-  (* symbol-width X-axis-label-spacing)
             (length (number-to-string number)))))
    (concat (make-string leading-spaces ? )
            (number-to-string number))))
```

Lo siguiente, se crea la función para imprimir la línea numerada, empezando con el número “1” para la primera columna:

```
(defun print-X-axis-numbered-line
  (number-of-X-tics X-axis-leading-spaces)
  "Imprime la líneas de números del eje X"
  (let ((number X-axis-label-spacing))
    (insert X-axis-leading-spaces)
    (insert "1")
    (insert (concat
             (make-string
              ;; Inserta espacios en blanco al siguiente número.
              (-  (* symbol-width X-axis-label-spacing) 2)
              ? )
             (number-to-string number)))
    ;; Insertar números.
    (setq number (+ number X-axis-label-spacing))
    (while (> number-of-X-tics 1)
      (insert (X-axis-element number))
      (setq number (+ number X-axis-label-spacing))
      (setq number-of-X-tics (1- number-of-X-tics)))))
```

Finalmente, se necesita escribir lo que `print-X-axis` que usa `print-X-axis-tic-line` y `print-X-axis-numbered-line`.

La función debe determinar los valores locales de las variables usadas por `print-X-axis-tic-line` y `print-X-axis-numbered-line`, y entonces eso debe llamarlas. También, debe imprimir el retorno de carro que separe las dos líneas.

La función consiste de una varlist que especifica cinco variables locales, y llama cada una de las dos líneas imprimiendo funciones:

```
(defun print-X-axis (numbers-list)
  "Imprime el eje X etique al tamaño de NUMBERS-LIST."
  (let* ((leading-spaces
          (make-string full-Y-label-width ? ))
         ;; symbol-width se provee por graph-body-print
         (tic-width (* symbol-width X-axis-label-spacing))
         (X-length (length numbers-list))
         (X-tic
          (concat
           (make-string
            ;; Crea una cadena de espacios en blanco.
            (-  (* symbol-width X-axis-label-spacing)
                (length X-axis-tic-symbol))
            ? )
           ;; Concatena espacio en blanco con símbolos
       tic.
           X-axis-tic-symbol))
         (tic-number
          (if (zerop (% X-length tic-width))
              (/ X-length tic-width)
            (1+ (/ X-length tic-width)))))
    (print-X-axis-tic-line tic-number leading-spaces X-tic)
    (insert "\n")
    (print-X-axis-numbered-line tic-number leading-spaces)))
```

Se puede testear `print-X-axis`:

1. Instale `X-axis-tic-symbol`, `X-axis-label-spacing`, `print-X-axis-tic-line`, tanto como `X-axis-element`, `print-X-axis-numbered-line`, y `print-X-axis`.
2. Copia la siguiente expresión:

```
(progn
 (let ((full-Y-label-width 5)
       (symbol-width 1))
   (print-X-axis
    '(1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16))))
```

3. Cambia al búffer `*scratch*` y emplaza el cursor donde se quiere el eje etiquetado para empezar.
4. Escribe *M-:* (`eval-expression`).
5. Pegue la expresión de test dentro del minibuffer con *C-y* (`yank`).
6. Presiona `RET` para evaluar la expresión

Emacs imprimirá el eje horizontal así

```
|   |    |    |    |
1   5   10   15   20
```

C.4 Imprimiendo el grafo completo

Ahora estamos listos para imprimir el grafo completo.

La función para imprimir el grafo con las etiquetas apropiadas sigue el esquema que creamos antes (véase Apéndice C "Un Grafo con Ejes Etiquetados", página 227), pero con adiciones.

Aquí está el esquema:

```
(defun print-graph (numbers-list)
  "documentation..."
  (let ((height  ...
         ...))
    (print-Y-axis height ... )
    (graph-body-print numbers-list)
    (print-X-axis ... )))
```

La versión final es diferente desde que se planea en dos caminos: primero, contiene los valores adicionales calculadas una vez que en la varlist; segundo, eso trae una opción para específicar las etiquetas se incrementa la fila. Esta última funcionalidad cambia a ser esencial; de otro modo, un grafo puede tener más filas que ajustarse en una muestra o en una hoja de papel.

Esta nueva funcionalidad requiere un cambio a la función `Y-axis-column`, para añadir `vertical-step` para eso. Esta función es parece a esto:

```
;;; Versión Final.
(defun Y-axis-column
  (height width-of-label &optional vertical-step)
  "Construye una lista de etiquetas para el eje Y.
HEIGHT es la máxima altura del grafo.
WIDTH-OF-LABEL es el máximo ancho de la etiqueta.
VERTICAL-STEP, una opción, es un entero positivo
que especifica cuanto una etiqueta de eje Y incrementa
cada línea. Por ejemplo, un paso de 5
significa que cada línea es cinco unidades
del grafo."
  (let (Y-axis
        (number-per-line (or vertical-step 1)))
    (while (> height 1)
      (if (zerop (% height Y-axis-label-spacing))
          ;; Inserta etiqueta.
          (setq Y-axis
                (cons
                 (Y-axis-element
                  (* height number-per-line)
                  width-of-label)
                 Y-axis))
        ;; Else, inserta espacios en blanco.
        (setq Y-axis
              (cons
               (make-string width-of-label ? )
               Y-axis)))
      (setq height (1- height)))
    ;; Inserta línea base.
    (setq Y-axis (cons (Y-axis-element
                        (or vertical-step 1)
                        width-of-label)
                       Y-axis))
    (nreverse Y-axis)))
```

Los valores para la máxima altura del grafo y el ancho de un símbolo se computan por `print-graph` es su expresión `let`; así `graph-body-print` debe ser cambiado para aceptarlos.

```
;;; Versión Final.
(defun graph-body-print (numbers-list height symbol-width)
  "Imprime una gráfica de barras del NUMBERS-LIST.
El numbers-list consiste en los valores del eje Y.
HEIGHT es la máxisma altura del grafo.
SYMBOL-WIDTH es el número de cada columna."
```

```
  (let (from-position)
    (while numbers-list
      (setq from-position (point))
      (insert-rectangle
       (column-of-graph height (car numbers-list)))
      (goto-char from-position)
      (forward-char symbol-width)
      ;; Dibuja el grafo columna por columna.
      (sit-for 0)
      (setq numbers-list (cdr numbers-list)))
    ;; Posiciona el punto para las etiquetas del eje X.
    (forward-line height)
    (insert "\n")))
```

Finalmente, el código para la función `print-graph`:

```
;;; Versión Final.
(defun print-graph
  (numbers-list &optional vertical-step)
  "El gráfico de barras etiquetadas del NUMBERS-LIST.
El numbers-list consiste en los valores de eje Y.

Opcionalmente, VERTICAL-STEP, un entero positivo, especifica cuanto el
eje Y incrementa cada línea. Por ejemplo, un paso de 5
significa que cada fila es de cinco unidades.
  (let* ((symbol-width (length graph-blank))
         ;; height en ambos es el número más largo
         ;; y el número con la mayoría de los digitos.
         (height (apply 'max numbers-list))
         (height-of-top-line
          (if (zerop (% height Y-axis-label-spacing))
              height
            ;; else
            (* (1+ (/ height Y-axis-label-spacing))
               Y-axis-label-spacing)))
         (vertical-step (or vertical-step 1))
         (full-Y-label-width
          (length
           (concat
            (number-to-string
             (* height-of-top-line vertical-step))
            Y-axis-tic))))

    (print-Y-axis
     height-of-top-line full-Y-label-width vertical-step)
    (graph-body-print
     numbers-list height-of-top-line symbol-width)
    (print-X-axis numbers-list)))
```

C.4.1 Testeando `print-graph`

Se puede chequear la función `print-graph` con una lista ordenada de números:

1. Instala las versiones finales de `Y-axis-column`, `graph-body-print`, y `print-graph` (además del resto del código.)
2. Copia la siguiente expresión:

```
(print-graph '(3 2 5 6 7 5 3 4 6 4 3 2 1))
```

3. Cambia al búffer `*scratch*` y emplaza el cursor donde se quiere el eje etiquetado para empezar.
4. Escribe *M-:* (`eval-expression`).
5. Pegue la expresión de test dentro del minibuffer con *C-y* (`yank`).
6. Presiona `RET` para evaluar la expresión

Emacs imprimirá un grafo que se parece a:

```
10 -


         *
        **   *
 5 -    ****  *
       **** ***
     * *********
     ************
 1 - *************

     |   |    |    |
     1   5   10   15
```

Por otro lado, si se pasa a `print-graph` un `vertical-step` valor de 2, evaluando esta expresión:

```
(print-graph '(3 2 5 6 7 5 3 4 6 4 3 2 1) 2)
```

El grafo se parece a esto:

```
20 -


         *
        **   *
10 -    ****  *
       **** ***
     * *********
     ************
 2 - *************

     |   |    |    |
     1   5   10   15
```

(Una pregunta: ¿es el '2' debajo del eje vertical un error o una funcionalidad? Si se piensa que es un error, y sería un '1', (o incluso un '0'), se pueden modificar las fuentes.)

C.4.2 Creando gráficas de números de palabras y símbolos

Ahora para el gráfico para el que todo este código fué escrito: un gráfico que muestra cuantas definiciones de función contienen unas pocas 10 palabras y símbolos, cuantas contienen entre 10 y 19 palabras y símbolos, cuantos contienen entre 20 y 29 palabras y símbolos, y así.

Esto es un proceso de múltiples pasos. Primero asegúrate que has cargado todo el requisito del código.

Eso es una buena idea para eliminar el valor de `top-of-ranges` en caso de que has asignado a algún valor diferente. Se puede evaluar lo siguiente:

```
(setq top-of-ranges
 '(10  20  30  40  50
   60  70  80  90 100
  110 120 130 140 150
  160 170 180 190 200
  210 220 230 240 250
  260 270 280 290 300)
```

Lo siguiente crea una lista del número de palabras y símbolos en cada rango.

Evalúa lo siguiente:

```
(setq list-for-graph
      (defuns-per-range
        (sort
         (recursive-lengths-list-many-files
          (directory-files "/usr/local/emacs/lisp"
                           t ".+el$"))
         '<)
        top-of-ranges))
```

En mi vieja máquina, esto lleva como una hora. Se parece a 303 ficheros Lisp en mi copia de Emacs version 19.23. Después de toda esta computación, el `list-for-graph` tenía este valor:

```
(537 1027 955 785 594 483 349 292 224 199 166 120 116 99
90 80 67 48 52 45 41 33 28 26 25 20 12 28 11 13 220)
```

Esto significa que mi copia de Emacs tiene 537 definiciones de funciones con poco menos de 10 palabras o símbolos en sí, 1027 definiciones de función con 10 a 19 palabras o símbolos dentro, 955 definiciones de función con 20 a 29 palabras o símbolos dentro, y así.

Claramente, solo buscando esta lista se puede ver que la mayoría de definiciones de función contienen de diez a treinta palabras y símbolos.

Ahora para imprimir. Nosotros *no* queremos imprimir un grafo que es de 1030 líneas de alto .... En vez de eso, imprimiría un grafo que es mejor que venticinco líneas de alto. Un grafo cuya altura puede ser mostrada en casi cualquier monitor, y fácilmente impreso en una hoja de papel.

Esto significa que cada valor en `list-for-graph` debe ser reducido a un quinceavo de su valor presente.

Aquí hay una corta función para hacer esto, usando dos funciones que no se han visto todavía, `mapcar` y `lambda`.

```
(defun one-fiftieth (full-range)
  "Devuelve la lista, con el cincuenteavo de cada elemento."
 (mapcar '(lambda (arg) (/ arg 50)) full-range))
```

C.4.3 Una expresión `lambda`: Anonimicidad útil

`lambda` es el símbolo para una función anónima, una función sin un nombre. Cada vez que se use una función anónima, se necesita incluir su cuerpo completo.
De este modo,

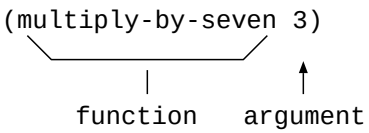
```
(lambda (arg) (/ arg 50))
```

es una definición de función que dice 'devuelve el valor resultante de dividir cualquier cosa que es pasada como `arg` por 50'.

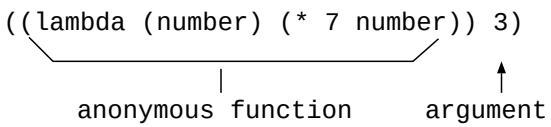
Pronto, por ejemplo, se tenía una función `multiply-by-seven`; se multiplica su argumento por 7. Esta función es similar, excepto que divide su argumento por 50; y, no tiene nombre. El equivalente anónimo de `multiply-by-seven` es:

```
(lambda (number) (* 7 number))
```

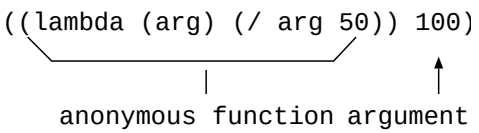
(Véase Sección 3.1 "La forma especial `defun`", página 26.)
Si queremos multiplicar 3 por 7, podemos escribir:



Esta expresión devuelve 21.
De manera similar, se puede escribir:



Si queremos dividir 100 por 50, se puede escribir:



Esta expresión devuelve 2. El 100 es pasado para la función, que divide este número por 50.

Véase Sección "Expresiones Lambda" in *El Manual de Referencia GNU Emacs Lisp*, para más acerca de `lambda`. Lisp y expresiones Lambda se derivan del Cálculo Lambda.

C.4.4 La función `mapcar`

`mapcar` es una función que llama a su primer argumento con cada elemento de su segundo argumento. El segundo argumento debe ser una secuencia.

La parte '`map`' del nombre viene de la frase matemática, 'mapeando a través de un dominio', significa hace apply a una función a cada uno de los elementos en un dominio. La frase matemática está basada en la metáfora de un superviviente paseando, un paso en un momento, a través de un área él está mapeando. Y '`car`', de acuerdo, viene desde la noción Lisp del primero de una lista.

Por ejemplo,

```
(mapcar '1+ '(2 4 6))
     ⇒ (3 5 7)
```

La función `1+` añade uno a su argumento, es ejecutada en *each* de la lista, y una nueva lista es devuelta.

En contraste con esto `apply`, se aplica su primer argumento a todo lo que permanece. (Véase Capítulo 15 "Leyendo un grafo", página 182, para una explicación de `apply`.)

En la definición de `one-fiftieth`, el primer argumento es la función anónima:

```
(lambda (arg) (/ arg 50))
```

y el segundo argumento es `full-range`, que será asociado para `list-for-graph`.

La expresión completa se parece a esto:

```
(mapcar (lambda (arg) (/ arg 50)) full-range))
```

Véase Sección "Mapeando Funciones" in *El Manual de Referencia de GNU Emacs Lisp*, para más acerca de `mapcar`.

Usando la función `one-fiftieth`, se puede generar una lista en el que cada elemento es un cincuenteavo del tamaño del correspondiente elemento en `list-for-graph`.

```
(setq fiftieth-list-for-graph
      (one-fiftieth list-for-graph))
```

La lista resultante se parece a esto:

```
(10 20 19 15 11 9 6 5 4 3 3 2 2
1 1 1 1 0 1 0 0 0 0 0 0 0 0 0 0 4)
```

Así, ¡ya estamos casi listos para imprimir! (También se notifica la pérdida de información: muchos de los rangos superiores son 0, esto significa que menos de 50 funciones tenían muchas palabras o símbolos — pero no necesariamente significando que niguna tenía muchas palabras o símbolos.)

C.4.5 Otro error . . . más insidioso

¡Se dijo 'casi listo para imprimir'! De acuerdo, hay un error en la función `print-graph` .... Esta tiene una opción `vertical-step`, pero no una opción `horizontal-step`. La escala `top-of-range` va desde 10 a 300 por decenas. Pero la función `print-graph` imprimirá solo uno por uno.

Esto es un ejemplo clásico de lo que algunos consideramos el tipo más insidioso de error, el error de omisión. Este no es el tipo de error que se puede encontrar estudiando el código, para eso no es el código; es una funcionalidad omitida. Tus mejores acciones son probar tu programa pronto y con frecuencia; e intentar poner en orden, tanto como se pueda, escribir código que sea fácil de comprender y fácil de cambiar. Intenta ser consciente, siempre y cuando se pueda, esto es siempre que tengas que escribir, *será* reescrito, si no pronto, eventualmente. Un máximo duro de seguir.

Esta es la función `print-X-axis-numbered-line` que necesita el trabajo; y entonces el `print-X-axis` y la función `print-graph` necesita ser adaptada. No se necesita mucho para ser hecho; hay uno simpático: los números podrían alinearse con marcas de tic. Esto toma un pequeño pensamiento.

Aquí está el `print-X-axis-numbered-line` corregido:

```
(defun print-X-axis-numbered-line
  (number-of-X-tics X-axis-leading-spaces
   &optional horizontal-step)
  "Imprime la líneas de números X-axis"
  (let ((number X-axis-label-spacing)
        (horizontal-step (or horizontal-step 1)))
    (insert X-axis-leading-spaces)
    ;; Elimina espacios extra de guía.
    (delete-char
     (- (1-
         (length (number-to-string horizontal-step)))))
    (insert (concat
             (make-string
              ;; Inserta espacio en blanco.
              (-  (* symbol-width
                     X-axis-label-spacing)
                  (1-
                   (length
                    (number-to-string horizontal-step)))
                  2)
              ? )
             (number-to-string
              (* number horizontal-step))))
    ;; Insertar los números que permanecen.
    (setq number (+ number X-axis-label-spacing))
    (while (> number-of-X-tics 1)
      (insert (X-axis-element
               (* number horizontal-step)))
      (setq number (+ number X-axis-label-spacing))
      (setq number-of-X-tics (1- number-of-X-tics)))))
```

Si se está leyendo esto en Info, se pueden ver las nuevas versiones `print-X-axis` y `print-graph` y los evaluarlas. Si se está leyendo esto en un libro impreso, se pueden ver las líneas cambiadas aquí (el texto completo es mucho para imprimir).

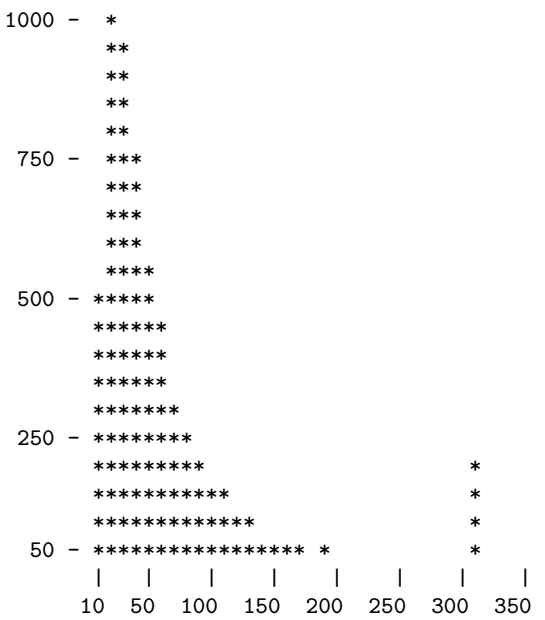
```
(defun print-X-axis (numbers-list horizontal-step)
  ...
    (print-X-axis-numbered-line
     tic-number leading-spaces horizontal-step))
(defun print-graph
  (numbers-list
   &optional vertical-step horizontal-step)
  ...
    (print-X-axis numbers-list horizontal-step))
```

C.4.6 El gráfico impreso

Cuando esté hecho e instalado, se puede llamar al comando `print-graph` como esto:

```
(print-graph fiftieth-list-for-graph 50 10)
```

Aquí está el gráfico:



El grupo largo de funciones contienen de 10 a 19 palabras y símbolos.

Apéndice D Software Libre y Manuales Libres

por Richard M. Stallman

La mayor deficiencia en sistemas operativos libres no está en el software — es la falta de buenos manuales libres que se puedan incluir en estos sistemas. Muchos de nuestros programas más importantes no vienen con manuales completos. La documentación es una parte esencial de cualquier paquete de software; cuando un paquete de software libre no viene con un manual libre, es una brecha mayor. Nosotros tenemos muchas brechas hoy.

Érase una vez, hace muchos años, se piensa que aprendería Perl. Se tiene una copia de un manual libre, pero se encontró difícil de leer. Cuando pregunto a los usuarios de Perl acerca de alternativas, me contaron que serían mejor los manuales introductorios — pero estos no eran libres.

¿Por qué era esto? Los autores de los buenos manuales los habían escrito para O'Reilly Associates, que los publicaron con términos restrictivos — no copiando, no modificando, los ficheros fuentes están disponibles — que los excluyen desde la comunidad de software libre.

No era la primera vez que esto ocurría, y (para nuestra comunidad es una gran pérdida) eso está lejos desde el último. Las editoriales de manuales privativos han logrado que muchos autores restrinjan sus manuales desde entonces. Muchas veces se ha oido que un usuario de GNU hábil me cuente que un manual que está escribiendo, con el que él espera ayudar al proyecto GNU — y entonces tenía mis esperanzas frustradas, como se ha procedió a explicar que él tenía que haber firmado un contrato con una editorial que restringiría eso, así que no puede usarlo.

Debido a que escribir buen inglés es una habilidad rara entre programadores, se pueden perder manuales por esto.

La documentación, como el software, es una cuestión de libertad, no de precio. El problema con estos manuales no eran que O'Reilly Associates impusiera un precio por las copias impresas — que en sí estaban bien. La Free Software Foundation *Fundación por el Software Libre* vende copias impresas de manuales libres de GNU, también. Pero los manuales de GNU están disponibles en forma de código fuente, mientras que estos manuales están disponibles solo en papel. Los manuales de GNU vienen con permiso para copiar y modificar; los manuales de Perl no. Estas restricciones son un problema.

El criterio para un manual libre es parecido al del software libre: es una cuestión de dar a todos los usuarios ciertas libertades. La redistribución (incluyendo redistribución comercial) debe ser permitida, así el manual puede acompañar cada copia del programa, en líne o en papel. El permiso para modificar es crucial también.

Como regla general, no se cree que sea esencial para la gente tener permisos para modificar todas las partes de artículos y libro. Las cuestiones para escritos no son necesariamente las mismas como estas para el software. Por ejemplo, no se sabe si se está obligado a dar permisos para modificar artículos como este, que describen nuestras acciones y nuestras vistas.

Pero hay una razón particular de por qué la libertad de modificar es crucial para la documentación de software libre. Cuando las personas ejercita su derecho a modificar el software, y añadir o cambiar sus funcionalidades, si son consciente ellos cambiarán el manual también — así se puede proveer documentación usable y cuidada con el programa modificado. Un manual que prohibe a los programadores ser consciente y finalizar el trabajo, o más precisamente requiere escribir un nuevo manual desde cero si ellos cambian el programa, no se ajusta a las necesidades de nuestra comunidad.

Mientras una serie de prohibiciones en la modificación es inaceptable, algunos tipos de límites en el método de modificar no tiene tanto problema. Por ejemplo, los requisitos para preservar la noticia de autores del copyright, los términos de distribución, o la lista de autores, estén ok. Eso es también no da problemas para requerir versiones modificadas para incluir notificar que fueron modificadas, incluso tienen secciones enteras que puede no ser eliminadas o cambiadas, tan largo como estas secciones tratan con asuntos no técnicos. (Algunos manuales de GNU los tienen).

Estos tipos de restricciones no son un problema porque, como materia práctica, no para al programador consciente desde la adaptación del manual para ajustar el programa modificado. En otras palabras, no se bloquea la comunidad del software libre haciendo el uso completo del manual.

Sin embargo, debe ser posible modificar todo el contenido técnico del manual, y entonces se distribuye el resultado en todos los medios usuales, a través de todos los canales usuales; de otro modo, las restricciones bloquean la comunidad, el manual no es libre, y así no se necesita otro manual.

Desafortunadamente, con frecuencia es duro encontrar a alguien a escribir otro manual cuando un manual privativo. El obstáculo es que muchos usuario piensan que un manual privativo es suficientemente bueno — así ellos no ven la necesidad de escribir un manual libre. Ellos no ven que el sistema operativo tiene un gazapo que necesita se rellenado.

¿Por qué los usuarios piensan que los manuales privativos son suficientemente buenos? Algunos no han considerado la cuestión. Espero que este artículo hará alguna cosa para cambiar esto.

Otros usuarios considera manuales privativos aceptables para la misma razón así muchas personas software privativo aceptable: ellos judgan en términos puramente prácticos, no usando la liberta como un criterio. Estas personas son tituladas a sus opiniones, pero desde que estas opciones crezcan desde valores que no incluyen libertad, ellas no están guiadas por esto quienes valoran la libertad.

Por favor, populariza esta cuestión. Se continúa a perder manuales para publicación privativa. Si se populariza que los manuales privativos no son suficientes, quizás la siguiente persona que quiere ayudar a GNU escribiendo documentación realizará, antes de que sea demasiado tarde, lo que él debe que todo sea libre.

Se puede también animar editoriales comerciales a vender manuales libres o con copyleft en vez de uno privativo. Un camino que se puede ayudar esto chequea los términos de la distribución de un manual antes de que se compre, y preferimos manuales copyleft a los no copyleft.

Note: La Fundación para el Software Libre mantiene una página en su sitio Web que liste libros libres disponibles desde otras editoriales:
http://www.gnu.org/doc/other-free-books.html

Apéndice E GNU Free Documentation License

Version 1.3, 3 November 2008

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

   You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

   You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

   If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

   If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

   If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

   It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance

and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a

unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

   You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

   However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

   Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

   Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

    The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

    Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

    "Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

    "CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with

a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright (C) *year your name*.
> Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled ‘‘GNU Free Documentation License’’.

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with. . . Texts." line with this:

> with the Invariant Sections being *list their titles*, with the Front-Cover Texts being *list*, and with the Back-Cover Texts being *list*.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Índice

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

V

W

X

Y

Z

Acerca del Autor

Robert J. Chassell ha trabajado con GNU Emacs desde 1985. Él escribe, edita y enseña Emacs y Emacs Lisp, y habla alrededor del mundo acerca de la libertad del software. Chassell estuvo fué un Director fundador y Tesorero de la Fundación por el Software Libre, Inc. Él se graduó la Universidad de Cambridge, en Inglaterra. Él tiene un interés contínuo en historia económica y social y vuela su propio aeroplano