Técnicas Básicas de Lisp

David J. Cooper, Jr

14 de febrero de 2011

Prólogo

El poder de las computadoras y de las aplicaciones de software está presente en todos los aspectos de nuestra vida. Desde almacenes, a reservaciones aéreas, hasta consultorios dentales, nuestra dependencia hacia la tecnología es totalmente abarcadora. Y eso no es todo. Todos los días, nuestra expectativa sobre la tecnología y el software se incrementan:

- electrodomésticos inteligentes que se pueden controlar vía internet.
- mejores motores de búsqueda que generan la información que necesitamos actualmente.
- laptops que se activan por voz.
- autos que saben exactamente hacia donde vas.

La lista es interminable. Desafortunadamente, no hay un abastecimiento interminable de programadores y desarrolladores para satisfacer nuestro apetito insaciable por nuevos inventos y artefactos. Todos los días, cientos de revistas y artículos on-line tratan sobre el tiempo y los recursos humanos necesarios para soportar las futuras expectativas tecnológicas. Mas allá de eso, los días del financiamiento ilimitado han terminado. Los inversores quieren ver los resultados, rápidamente.

Common Lisp (CL), es uno de los pocos lenguajes y opciones de desarrollo que puede soportar estos desafíos. Poderoso, flexible, variable al vuelo, incremental, CL está jugando un papel importante en áreas que demandan soluciones a problemas complejos. Ingenieros en el campo de la bioinformática, planificación, extracción de datos, administración de documentos, B2B y E-comercio, están migrando a CL para completar sus aplicaciones a tiempo y dentro del presupuesto. Sin embargo, CL no solamente es apropiado para los problemas mas complejos. Aplicaciones de modesta complejidad, pero con demanda de ciclos rápidos de desarrollo y adaptaciones, son también candidatos ideales para CL.

Otros lenguajes han intentado imitar a CL, con éxito limitado. Perl, Python, Java, C++, C# - incorporan algunas de las características que dan a Lisp ese poder, pero esas implementaciones tienden a ser frágiles.

El propósito de este libro es mostrar las características que hacen a CL mucho mejor que estos imitadores, y darte una guía rápida para usar CL como entorno de desarrollo. Si eres un programador en otros lenguajes distintos de Lisp, esta guía te dará todas las herramientas que necesitas para comenzar a escribir aplicaciones en Lisp. Si haz usado Lisp en el pasado, esta guía ayudará a refrescar tu memoria y encenderá nuevas luces de Lisp para ti.

Pero ten cuidado, ¡Lisp puede ser adictivo! Esto es porque la mayor parte de las compañías de la lista de 500 empresas con mejores beneficios lo usarán en sus aplicaciones críticas todo el tiempo. Después de leer este libro, intentar nuestro software, y experimentar de 3 a 10 veces el incremento en productividad, creemos que usted se sentirá de la misma forma.

CONTENIDOS

Capítulo 1

Introducción

1.1 Pasado, presente y futuro de Common Lisp

1.1.1 Lisp ayer

John McCarthy descubrió los principios básicos de Lisp en 1958, cuando estuvo procesando listas matemáticas complejas en MIT. El Common Lisp (CL) es un lenguaje de computación de alto nivel, cuya sintaxis sigue una estructura de lista simple. El término "Lisp" tiene su origen en "LISt Processing". Cuando el desarrollo, testeo y ejecución de un programa CL, en el núcleo es una versión actual del procesamiento de listas original que procesa (compila, evalúa, etc.) los elementos de su programa. Estos elementos, en el nivel de código fuente, son representados como listas. Una lista, en este contexto, es solo una secuencia de ítems, que luce como una lista de compras del shopping. Originalmente la lista era la única estructura de datos soportado por Lisp, pero actualmente Common Lisp soporta un vasto rango de estructuras de datos flexibles y eficientes.

Un entorno de desarrollo y de ejecución típico se comporta como un completo sistema operativo, con procesos múltiples de ejecución y la habilidad de cargar nuevo código y redefinir objetos (funciones, etc.) *dinámicamente*, por ejemplo sin detener y reiniciar la "máquina".

Esta característica de "sistema operativo" también hace a CL significativamente mas flexible que otros lenguajes de programación populares. Mientras que otros lenguajes, así como los scripts de shell o scripts de Perl CGI, necesitan conectar los datos entre si, en CL todas las manipulaciones de datos pueden ejecutarse interactivamente sin un proceso único en un espacio de memoria determinado.

1.1.2 Lisp Hoy

La forma mas popular de Lisp usada actualmente es "ANSI Common Lisp" la cual fue inicialmente diseñada por Guy Steele en Common Lisp, el lenguaje, 2° edición ("CLtL2") - (ver la bibliografía en el Apéndice B). Esta versión de Lisp se volvió el estándar aceptado en la industria. En 1995, el Instituto Nacional de Estándares Americano reconoció una versión levemente actualizada como ANSI Common Lisp, el *primer* lenguaje orientado a objetos en recibir certificación, y ANSI CL permanece como el único lenguaje que cumple con todos los criterios definidos por el Grupo de Administración de Objetos (OMG) para ser un completo lenguaje orientado a objetos.

La estandarización del lenguaje oficial es importante, debido a que ello protege a los desarrolladores de cargas heredadas cuando se implementan las nuevas versiones de un lenguaje. Por ejemplo, esta carencia de estandarización ha sido un continuo problema para los desarrolladores de Perl. Desde que cada implementación de Perl define el comportamiento del lenguaje, no es extraño tener aplicaciones que requieran de versiones de Perl desactualizadas, haciendo casi imposible usarlo en un entorno comercial crítico.

1.1.3 Lisp Mañana

Muchos desarrolladores tenían la ilusión de que el proceso de desarrollo de software del futuro sería mas automatizado a través de las herramientas de Ingeniería de Software Asistida por Computadora (CASE). Tales herramientas permiten a programadores y no programadores diagramar sus aplicaciones visualmente y automáticamente generar código. Aunque es útil hasta cierto punto, las herramientas CASE tradicionales no cubren detalles de dominio específico y soportan todos los posibles tipos de adaptaciones – de esta manera los desarrolladores inevitablemente necesitan manejar el resto de código de sus aplicaciones, rompiendo el eslabón entre el modelo CASE y el código. Los sistemas CASE fallan al tratar de ser una verdadera "herramienta de solución de problemas".

La naturaleza recursiva de CL, y su habilidad natural para construir aplicaciones en capas y construirlas sobre si mismas – en esencia, *el código que escribe código que escribe código...*, hace a CL una mejor solución de ingeniería de software. Los programas CL pueden generar otros programas CL (usualmente en tiempo de compilación), permitiendo al desarrollador definir *todas* las partes de la aplicación en un lenguaje de alto nivel unificado. Usando macros y funciones, soportados naturalmente por la sintaxis de CL, el sistema puede convertir aplicaciones escritas en el lenguaje de alto nivel automáticamente dentro del código "final". De este modo, CL es tanto un lenguaje de programación como una poderosa herramienta CASE, y no hay necesidad de romper la conexión.

Otras características de CL también ahorran tiempo significante de desarrollo y recursos humanos:

- Administración automática de memoria /Recolector de basura: inherente en la arquitectura básica de CL.
- Tipos dinámicos: En CL, los valores tienen tipos, pero las variables no. Esto significa que no tienes que declarar variables, o marcadores de posición, delante de un tipo particular; puedes simplemente crearlos o modificarlos al vuelo.[1]
- Redefinición dinámica: puedes agregar nuevas operaciones y funcionalidades para la ejecución de procesos de CL sin la necesidad de tiempos de inactividad. En efecto, un programa puede ser ejecutado en un hilo, mientras partes del código son redefinidas en otro hilo. Además, tan pronto como las redefiniciones se terminan, la aplicación usará el nuevo código y está efectivamente modificado mientras se ejecuta. Esta característica es especialmente útil para aplicaciones de servidores que no pueden permitirse tiempos de inactividad – puedes parchearlos y repararlos mientras se están ejecutando.
- Portabilidad: La calidad de "máquina abstracta" de CL hace a los programas especialmente portables.
- Características Estándar: CL contiene muchas características internas y estructuras de datos que no son estándar en otros lenguajes. Características tales como tablas de símbolos y sistemas de paquetes, tablas hash, estructuras de lista y un sistema de objetos comprensivo que tiene varias características "automáticas" que requeriría un significativo tiempo de desarrollo para ser recreado en otros lenguajes.

1.2 Convergencia de Hardware y Software

Las críticas mas comunes sobre Lisp usualmente hechas por programadores no familiarizados con

1 Aunque no *tengas* que declarar tipos de variable, *puedes* declararlas. Haciéndolo ayudaras al compilador de CL a optimizar tu programa.

el lenguaje es que las aplicaciones Lisp son demasiado grandes y demasiado lentas. Aunque esto pudo haber sido así hace 20 años atrás, es totalmente falso hoy en día. La computadoras económicas actuales tienen mas memoria que algunas de las máquinas mas poderosas disponibles hace solo cinco años atrás, y se encuentran fácilmente las maquinas necesarias para un aplicación típica de CL. El Hardware no es un factor crítico. ¡Es el tiempo de los programadores y los desarrolladores!

Además, los equipos de programación de CL tienden a ser pequeños, debido a las características inherentes a CL que otorgan mas ventajas a los desarrolladores. Sin un esfuerzo excesivo, un solo desarrollador de CL puede crear y desplegar un programa sofisticado, poderoso en un período de tiempo mucho menor que si hubiera utilizado otro lenguaje.

1.3 El Modelo CL de Computación

La programación en CL se distingue de la programación en otros lenguajes debido a su sintaxis única y su modo de desarrollo. CL permite a los desarrolladores hacer cambios y pruebas inmediatamente, y de una manera incremental. En otros lenguajes, un desarrollador debe seguir el ciclo “compilar – linkar – ejecutar – testear”. Estos pasos extra agregan minutos u horas para cada ciclo de desarrollo, y rompe el proceso de pensamiento del programador. Multiplique estos hechos por días, semanas, y meses – y el potencial ahorrado es fenomenal.

Capítulo 2

Operar un entorno de Desarrollo CL

Este capítulo cubre algunas de las técnicas usadas cuando se trabaja con un entorno de desarrollo CL, específicamente, el entorno Allegro® CL de Franz Inc. Si usted no esta familiarizado con cualquier lenguaje Lisp, puede consultar el capítulo 3.

Idealmente, mientras lee este capítulo, podría tener acceso a una sesión de Common Lisp e intentar escribir alguno de los ejemplos.

Cuando trabajamos con Common Lisp, podemos entender el entorno como si se tratase de un sistema operativo, ubicado por encima de cualquier sistema operativo sobre el cual se trabaja. En efecto, hay gente que construyó completas estaciones de trabajo y sistemas operativos en Lisp. El entorno Symbolics Common Lisp aun se ejecuta como una máquina emulada por encima de la CPU Alpha Compaq/DEC de 64 bits, con su propio sistema operativo basado en Lisp.

Un listado completo y actualizado de sistemas CL disponibles puede encontrarse en el sitio web de la Asociación de Usuarios de Lisp (ALU) en:

http://www.alu.org/table/systems.htm

Estos sistemas siguen los mismos principios básicos, y la mayoría de lo que tu aprendas y escribas sobre un sistema puede ser aplicado y portado sobre los otros. Los ejemplos en esta guía fueron preparados usando Allegro CL para Linux. En lo posible, anotaremos la sintaxis y extensiones específicas de Allegro CL.

2.1 Instalar un Entorno CL

Instalar un entorno CL es generalmente un proceso sencillo. Para Allegro CL en Linux, básicamente se ejecuta un simple script de shell cuyos pasos te indican sobre informaciones de licencias, luego procede a descomprimir la distribución, y configurar el ejecutable CL y archivos de soporte.

La instalación sobre Windows es un tanto menos sobre escribir con teclado y mas sobre uso del ratón.

2.2 Ejecutar CL en una Shell de Window

La forma mas simple y primitiva de ejecutar CL es directamente desde el shell del Unix (Linux) dentro de una ventana de terminal. La mayoría de los programadores no trabajan de esta manera sobre estas bases cotidianas debido a que otros modos de uso proveen mas potencia y conveniencia con menores pulsaciones de tecla.

Este modo de trabajo es algunas veces usado para loguearse en un sistema CL en ejecución sobre un host remoto (ej. un proceso CL actuando como un servidor web) cuando usa una conexión dial-up lenta.

2.2.1 Comenzar CL desde una ventana de terminal

Para iniciar una sesión CL desde una ventana de terminal, simplemente escriba

```
alisp
```

Desde la línea de comandos del Unix (o DOS). Asumiendo que la ruta de ejecución del shell esta configurada adecuadamente y que CL esta correctamente instalado, deberás ver alguna información de introducción, luego se mostrará un prompt de comandos que luciría muy similar al siguiente:

```
CL-USER(1):
```

CL esta completamente dentro de un bucle *leer-evaluar-imprimir*, y está esperando a que escribas algo con lo que pueda luego, *leer-evaluar* para obtener un *valor-de-retorno*, y finalmente *imprimir* este valor-de-retorno resultante.

2.2.2 Finalizar CL desde una ventana de terminal

Ahora que has aprendido como iniciar CL, deberías probablemente saber como cerrarlo. En Allegro CL, una forma fácil de finalizar el proceso CL es escribir

```
(excl:exit)
```

en el prompt de comandos. Esto debería devolverte hacia el shell. En muy raras ocasiones, un martillo muy fuerte es requerido para detener el proceso CL, en cuyo caso tu puedes escribir

```
(excl:exit 0 :no-unwind t)
```

Inténtalo: Ahora intenta iniciar y para CL varias veces, para saber la sensación que da. En general, podrías notar que CL comienza mucho mas rápidamente en invocaciones subsecuentes. Esto es debido a que el archivo ejecutable entero ya ha sido leído en la memoria de la computadora y no tiene que leerse del disco cada vez.

Un acceso directo para el comando `(excl:exit)` es el comando de consola `:exit`. Los comandos de consola son palabras precedidas por dos puntos [:] que puedes escribir en el Prompt de comandos como una forma de hacer que CL haga algo.

Inténtalo: Inicia el entorno CL. Luego, usando tu editor de texto favorito, crea un archivo `/tmp/hello.lisp` y coloca el siguiente texto dentro de el (no te preocupes en entender el texto por ahora):

```
(in-package :user)

(defun hello ()
      (write-string “Hola, mundo!”))
```

Guarda el archivo al disco. Ahora, en el prompt CL en el shell donde tu iniciaste el entorno CL, compila y

carga el archivo como sigue *(el texto después del promt "USER" representa lo que tienes que escribir; todo lo demás es texto mostrado por CL):*

```
CL-USER(3): (compile-file "/tmp/hola.lisp")
;;; Compiling file /tmp/hola.lisp
;;; Writing fasl file /tmp/hola.fasl Fasl write complete
#p"/tmp/hola.fasl"
NIL
NIL

CL-USER(4): (load "/tmp/hola.fasl")
; Fast loadind /tmp/hello.fasl
TT
```

Por defecto, el comando `complile-file` observará archivos con el sufijo .lisp, y el comando load cargará el archivo compilado a código de máquina (binario) resultante, que por defecto tendrá la extensión .fasl. La extensión ".fasl" esta puesta por "FASt Loading" file (archivo de carga rápida).
Observe que, en general, simplemente cargando un archivo Lisp no compilado (usando load) funcionalmente tendrá el mismo efecto que cargar un archivo binario compilado (fasl). Pero el archivo fasl cargará mas rápido, y algunas funciones, objetos, etc. definidos en el actuarán mas rápido, debido a que han sido optimizados y traducidos en lenguaje que es mas cercano a las máquinas actuales. Distinto a los archivos de Java "class", el archivo fasl de CL usualmente representa código nativo específico de la máquina. Este medio que compila programas CL generalmente ejecutará mucho más rápido que los programas compilados de Java, pero los programas CL deben ser compilados separadamente para cada tipo de máquina en la que desees ejecutarlos.
Finalmente, intenta ejecutar la función definida `hola` escribiendo lo siguiente en tu línea de comandos:

```
CL-USER(5): (hola)
Hola, Mundo!
"Hola, Mundo!"
```

Capítulo 3

El Lenguaje CL

Si eres nuevo en el lenguaje de CL, recomendamos que complementes este capítulo con otros recursos. Vea la bibliografía en el Apéndice B para algunas sugerencias. La bibliografía también lista dos tutoriales interactivos de CL en línea de la Universidad de Tulane y Texas A&M, que puedes visitar si lo deseas.

Entretanto, este capítulo proveerá una vista condensada del lenguaje. Por favor observe, sin embargo, que este libro es entendido como un sumario, y no ahondará en algunos de los mas sutiles y poderosas técnicas posibles con Common Lisp.

3.1 Vista rápida sobre CL y su sintaxis

Lo primero que deberías notar sobre CL (y la mayoría de los lenguajes de la familia Lisp) es que usa una notación generalizada de *prefijos*.
Uno de las acciones mas frecuentes en un programa CL, o en un bucle de consola leer-evaluar-imprimir, es la llamada a una *función*. Esto es a menudo hecho por escritura de una *expresión* que es el nombre de la función, seguida por sus argumentos. Aquí hay un ejemplo:

```
(+ 2 2)
```

Esta expresión consiste de la función nombrada por el símbolo "+", seguida por los argumentos 2 y otro 2. Como puedes adivinar, cuando esta expresión es evaluada retornará el valor 4.
Inténtalo: Intenta escribir esta expresión en tu prompt de comandos, y vea el valor-de-retorno mostrado en la consola.
¿Qué es lo que está pasando aquí? Cuando CL es consultado para *evaluar* una *expresión* (como en el bucle de la consola leer-evaluar-imprimir), evalúa la expresión de acuerdo a las siguientes reglas:
1. Si la expresión es un número (ej. luce como un número), simplemente se evalúa a si mismo (un número):

```
CL-USER(8): 99
99
```

2. Si la expresión luce como una *cadena* (ej. esta entre dobles comillas), también simplemente se evalúa a si misma:

```
CL-USER(9): "Se valiente, no puedes cruzar un abismo en dos pequeños pasos."
"Se valiente, no puedes cruzar un abismo en dos pequeños pasos."
```

3. Si la expresión luce como un *símbolo* literal, simplemente evaluará a ese *símbolo* (mas sobre esto en la

Sección 3.1.4):

```
CL-USER(12): 'mi-simbolo
MI-SIMBOLO
```

4. Si la expresión luce como una *lista* (ej. esta entre paréntesis), CL asume que el *primer* elemento en esta lista es un *símbolo* que nombra una *función* o una *macro*, y el resto de los elementos en la lista representa a los *argumentos* para la función o macro. (Hablaremos sobre *funciones* primero, *macros* después). Una *función* puede tomar cero o mas argumentos, y puede *devolver* cero o mas *valores de retorno.* A menudo una función solo devuelve un valor de retorno.

```
CL-USER(14): (expt 2 5)
32
```

Inténtalo: Intenta escribir las siguientes expresiones funcionales en tu prompt de comandos, y convéncete que los valores de retorno impresos tienen sentido:

```
(+ 2 2)
(+ 2)
2
(+)
(+ (+ 2 2) (+ 3 3))
(+ (+ 2 2))
(sys:user-name)
(user-homedir-pathname)
(get-universal-time) ;;devuelve el numero de segundos desde el 1 enero 1900
(seach “Dr.” “Yo soy el Dr. Strangelove”
(subseq “Yo soy el Dr. Strangelove”
      (search “Dr.” “Yo soy el Dr. Strangelove”))
```

3.1.1 Evaluación de argumentos para una función

Observe que los argumentos para una función pueden *ellos mismos* ser cualquier tipo de las expresiones de arriba. Ellas están evaluadas en orden, de izquierda a derecha, y finalmente son pasadas a la función para ser evaluadas. Este tipo de anidación puede ser tan profunda como se quiera. No te preocupes por tener un océano de paréntesis – la mayoría de los editores de texto serios pueden manejar paréntesis profundamente anidados con facilidad, y además automáticamente hace el dentado de tus expresiones así como tú lo haces, como el programador, nunca mas tiene que preocuparse con la concordancia de los paréntesis.

3.1.1 Simplicidad de la sintaxis de Lisp

Una de las cosas mas agradables de Lisp es la simplicidad y consistencia de su sintaxis. Tenemos cubierto

por lejos todo lo perteneciente a la evaluación de argumentos para funciones, así que solamente necesitas conocer sobre la sintaxis. Las reglas para las macros son muy similares, con la diferencia que todos los argumentos de una macro no son necesariamente evaluados cuando la macro es llamada – ellos pueden ser transformados en otra cosa primero.

Esta sintaxis consistente simple es un grato alivio respecto de otros lenguajes jóvenes como C, C++, Java; Perl y Python. Estos lenguajes demandan un uso "más natural" sintaxis "infija", pero en realidad su sintaxis es una confusa mezcla de prefijos, infijos y sufijos. Ellos usan infijos solamente para las operaciones aritméticas simples como

```
2 + 2
```

y

```
3 * 13
```

y usa sufijos en ciertos casos como

```
i++
```

```
and
```

```
char*
```

Pero en la mayoría de los casos usan una sintaxis de prefijo así como Lisp, como en:

```
split(@array, ":");
```

De esta manera, si has estado programando en estos otros lenguajes, has estado usando notación de prefijos todo el tiempo, pero no te has dado cuenta. Si ves de esta manera, la notación de prefijo de Lisp parecerá mucho menos extraña para ti.

3.1.3 Desactivar la Evaluación

Algunas veces tu deseas especificar una expresión en el bucle de consola leer-evaluar-imprimir, o dentro de un programa, pero no quieres que CL evalúe esta expresión. Quieres solo la expresión literal. CL provee el operador especial `quote` para este propósito. Por ejemplo,

```
(quote a)
```

devolverá el símbolo literal A. Observe que, por defecto, el lector CL convierte todos los nombres de símbolos a mayúscula cuando el símbolo es leído en el sistema. Observe también que los símbolos, si son evaluados "como son" (ej. sin `quote`), se comportarán por defecto como variables, y CL intentará devolver un valor asociado. Veremos mas sobre esto luego.

Aquí hay otro ejemplo del uso de `quote` para devolver una expresión literal:

```
(quote (+ 1 2))
```

De manera distinta a evaluar una expresión de lista "como es", este devolverá la lista literal (+ 1 2). Esta lista puede ahora ser accedida como una normal estructura de datos de lista[1].

Common Lisp define la abreviación ' como una forma rápida para "envolver" una expresión en una llamada a `quote`. De esta manera los ejemplos anteriores podrían ser escritos de forma equivalente a:

```
'a
```

```
'(+ 1 2)
```

1 Debería notarse, sin embargo, que cuando una lista literal con quote es creada como esta, es ahora técnicamente una constante – no podrías modificarla con operadores destructivos.

3.1.4 Tipos de Datos fundamentales de CL

Common Lisp en forma nativa soporta muchos tipos de datos comunes en otros lenguajes, como números, cadenas y arreglos. También son nativos de CL un conjunto de tipos que no puedes encontrar en otros lenguajes, como listas, símbolos y tablas hash. En esta guía rápida trataremos sobre números, cadenas, símbolos y listas. Luego en el libro daremos mas detalles de algunos tipos de datos específicos de CL.

En lo que se refiere a tipos de datos, CL permite un paradigma llamado escritura dinámica. Esto se trata básicamente de que los *valores* tienen tipos, pero las *variables* no necesariamente tienen tipo, y típicamente las variables no están "declaradas de antemano" para ser de un tipo particular.

Números

Los *números* CL forman una jerarquía de tipos, que incluye *enteros, fracciones, punto flotante* y números *complejos.* Para muchos propósitos solamente necesitas pensar a un valor como un "número" sin obtener nada mas específico que eso. La mayoría de las operaciones aritméticas, tales como +, –, *, /, etc, forzará automáticamente a cualquier tipo necesario sobre sus argumentos y devolverá un número del tipo apropiado.

CL soporta un amplio rango de números decimales de punto flotante, como fracciones, de manera que 1/3 significa un tercio, no 0.333333333 redondeado a alguna precisión arbitraria.

Como hemos visto, los números en CL son un tipo de datos nativo que se evalúan simplemente a si mismos cuando se colocan en la consola o si están incluidos en una expresión.

Cadenas

Las cadenas son actualmente un tipo especializado de arreglo, concretamente un arreglo de una sola dimensión (vector) hecho de caracteres. Estos caracteres pueden ser letras, números, o signos de puntuación, y en algunos casos pueden incluir caracteres del conjunto internacional de caracteres (ej. Unicode) como el Hanzi Chino o el Kanji Japonés. El delimitador de cadena en CL es el carácter de doble comilla (").

Como hemos visto, las cadenas en CL son un tipo de datos nativo que simplemente se evalúan a si mismos cuando son incluidos en una expresión.

Símbolos

Los símbolos son una estructura de datos tan importante en CL que las personas algunas veces se refieren a CL como el "Lenguaje de computación simbólico". Los símbolos son un tipo de objetos de CL que proveen a tu programa con un mecanismo interno para almacenar y devolver valores y funciones, usados en su propia forma correcta. Un símbolo es a menudo conocido por su nombre (actualmente una cadena), pero de hecho hay mucho mas sobre un símbolo que su nombre. En adición al nombre, los símbolos también contienen un espacio de *función*, un espacio de *valor*, y un espacio de *lista de propiedades* abierto en la cual tu puedes almacenar un número arbitrario de propiedades con nombre.

Para una función nombrada como +, el espacio de función del símbolo + contiene la función actual misma. El espacio de valor de un símbolo puede contener cualquier valor, permitiendo al símbolo actuar como una variable global, o *parámetro.* Y el espacio de lista de propiedades, o espacio *plist*, puede contener una cantidad arbitraria de información.

Esta separación de la estructura de datos símbolo en espacios de función, valor y plist es una distinción obvia entre Common Lisp y la mayoría de los otros dialectos Lisp. La mayoría de los otros dialectos permiten solo una (1) "cosa" para ser almacenada en la estructura de datos símbolo, otra mas que su nombre (ej. una función o un valor, pero no ambos al mismo tiempo). Debido a que Common Lisp no impone esta restricción, no es necesario ingeniárselas con los nombres, por ejemplo para tus variables, para evitar conflictos con "palabras reservadas" existentes en el sistema. Por ejemplo, "`list`" es el nombre de una función interna en CL. Pero tu puedes deliberadamente usar "`list`" como una variable. No hay necesidad de ingeniárselas con abreviaciones arbitrarias como "`lst`".

Como los símbolos son evaluados según donde están ubicados en una expresión. Como hemos visto, si un símbolo aparece *primero* en una expresión de lista, como con el + en `(+ 2 2)`, el símbolo es evaluado para su espacio de función. Si el primer elemento de una expresión es un símbolo que realmente contiene una función en su espacio de función, entonces cualquier símbolo que luego aparezca en *cualquier lugar* (en el *resto*) de la expresión es tomado como una variable, y es evaluado para su valor global o local, dependiendo de su alcance. Mas sobre variables y alcance después.

Como vimos en la seccion 3.1.3, si buscas un símbolo literal, una forma de lograr esto es agregar "`quote`" al nombre del símbolo:

```
'a
```

Otra manera es que el símbolo aparezca en una expresión de lista con quote:

```
'(a b c)
'(a (b c) d)
```

Observe que la comilla (') se aplica a todo lo que esta en la expresión de lista, incluyendo cualquier sub-expresión.

Listas

Lisp toma su nombre de su duro soporte para la estructura de datos lista. El concepto de lista es importante para CL por mas de una razón – las listas son importantes porque *¡todos los programas de CL son listas!* Teniendo la lista como una estructura de datos nativa, así como la forma de todos los programas, cosa que es especialmente sencillo para programas CL para computar y generar otros programas CL. De la misma manera, los programas CL pueden leer y manipular otros programas CL de una forma natural. *Esto no puede decirse de la mayoría de otros lenguajes, y es una de las distintas características primarias de CL.*

Textualmente, una lista es definida con cero o mas elementos rodeados por paréntesis. Los elementos pueden ser objetos de cualquier tipo de datos de CL válido, como números, cadenas, símbolos, listas, u otra clase de objetos. Como hemos visto, debes poner quote a una lista literal para evaluarla o CL asumirá que estás llamando a una función.

Ahora observa la siguiente lista:

```
(defun hello() (write-string "hola, mundo!"))
```

Esta lista también es un programa válido (definición de función, en este caso). No te preocupes sobre analizar la función ahora mismo, pero tomate algunos minutos para asegurarte que esto cumple con los requerimientos de una lista. ¿Cuáles son los tipos de elementos en esta lista?

Además de usar la comilla (') para producir una lista literal, otra manera de hacer una lista es llamar a la función `list` . La función `list` toma cualquier número de argumentos, y devuelve una lista hecha del

resultado de evaluar cada argumento (como con todas las funciones, los argumentos para la función `list` son evaluados, de izquierda a derecha, antes de ser pasados a la función). Por ejemplo,

```
(list 'a 'b (+ 2 2)
```

devolverá la lista `(A B 4)`. Los dos símbolos con comilla se evalúan como símbolos, y la función llamada `(+ 2 2)` se evalúa como el número `4`.

3.1.5 Funciones

Las funciones forman los bloques básicos de construcción de CL. Aquí daremos una breve guía sobre como definir una función; luego iremos con mas detalle sobre que es una función.

Una manera común de definir funciones con nombre en CL es con la macro `defun`, cuyo nombre está por DEFinición de una FUNción. `Defun` toma como argumentos un símbolo, una l*ista de argumentos* y un *cuerpo*:

```
(defun mi-primera-funcion-lisp ()
      (list 'hola 'mundo))
```

Debido a que `defun` es una *macro*, mas que una función, no sigue las reglas de que todos sus argumentos son evaluados como expresiones – específicamente, los símbolos que nombran a la función no tienen que estar con comilla, tampoco tienen listas de argumentos. Estos son tomados como un símbolo literal y una lista literal, respectivamente.

Una vez que la función ha sido definida con `defun`, puedes llamarla como llamarías cualquier otra función, envuelta entre paréntesis junto a sus argumentos:

```
CL-USER(56): (mi-primera-funcion-lisp)
(HOLA MUNDO)

CL-USER(57): (defun cuadrado(x)
                  (* x x))
CUADRADO

CL-USER(58): (cuadrado 4)
16
```

Las declaraciones de los tipos de argumentos para una función no son requeridos.

3.1.6 Variables Locales y Globales

Variables Globales

Las variables globales en CL son usualmente conocidas también como variables especiales. Ellas pueden ser establecidas por llamadas a `defparameter` o `defvar` desde el bucle leer-evaluar-imprimir, o desde un archivo que tu compilas y cargas en CL:

```
(defparameter *temp-actual* 31)
(defvar *humedad-actual* 70)
```

Los asteriscos alrededor de estos nombres de símbolos son parte ellos mismos de los nombres de los símbolos, y no tienen significado para CL. Esto es simplemente una convención de nombres para variables globales (parámetros), para hacerlas mas fácil de ubicar al ojo humano.

`Defvar` y `defparameter` difieren de una manera importante: Si `defvar` es evaluada con un símbolo que

es una variable global, no la sobreescribe con un valor nuevo:

```
CL-USER(4): *temp-actual*
31
CL-USER(5): *humedad-actual*
70
CL-USER(6): (defparameter *temp-actual* 100)
*TEMP-ACTUAL*
CL-USER(7): (defvar *humedad-actual* 50)
*HUMEDAD-ACTUAL*
CL-USER(8): *temp-actual*
100
CL-USER(9): *humedad-actual*
70
```

`*humedad-actual*` no cambia debido a que nosotros hemos usado `defvar`, y esta ya tiene un valor previo.

El valor para cualquier parámetro puede siempre ser cambiado desde la consola con `setq`:

```
CL-USER(11): (setq *humedad-actual* 30)
30
CL-USER(12): *humedad-actual*
30
```

Aunque setq puede trabajar con nuevas variables, debería solo ser usada con variables ya establecidas.

Durante el desarrollo de la aplicación, a menudo tiene sentido usar defvar antes que defparameter para variables cuyos valores pueden cambiar durante la ejecución y testeo del programa. Esta forma, no resetearás sin querer los valores de un parámetro si compilas y cargas el código fuente donde ésta es definida.

Variables Locales

Se pueden introducir nuevas variables locales con la macro `let`:

```
CL-USER(13): (let ((a 20)
                   (b 30))
               (+ a b))
```

Como se vio en el ejemplo de arriba, `let` toma una sección de asignación y un cuerpo. `let` es una macro, en vez de una función[1], así que no sigue la regla pura y dura de que todos sus argumentos son evaluados. Específicamente, la sección de asignación

```
((a 20)
 (b 30))
```

no se evalúa (de hecho, la macro tiene el efecto de transformar esto en algo distinto antes de que el evaluador CL ni siquiera lo vea). Debido a que esta sección de asignación no se evalúa por la macro `let`, no ha de ser puesta entre comillas, como una lista que sea un argumento de función lo sería. Como se puede ver, la sección de asignación es una lista de listas, donde cada lista interna es un par cuyo `first` es un

1 Véase la Sección 3.8.5 acerca de cómo reconocer las macros y las funciones

símbolo y cuyo `second` es un valor (que será evaluado). Estos símbolos (*a* y *b*) son las variables locales y son asignadas a los valores respectivos.

El *cuerpo* consiste en cualquier número de expresiones que van tras la sección de asignación y antes del paréntesis de cierre de la sentencia `let`. Las expresiones en este cuerpo se evalúan normalmente y, por supuesto, cualquier expresión puede referirse al valor de cualquiera de las variables locales simplemente refiriéndose directamente a su símbolo. Si el cuerpo consiste en más de una expresión, el valor de retorno final del cuerpo es el valor de retorno de su última expresión.

Las nuevas variables locales CL se dice que tienen ámbito *léxico*, lo que significa que sólo son accesibles en el código donde están textualmente contenidas dentro del cuerpo de `let`. El término '*léxico*' se deriva del hecho de que el comportamiento de estas variables puede ser determinado simplemente leyendo el texto del código fuente, y no es afectado por lo que ocurre durante la ejecución del programa.

El ámbito *dinámico* ocurre cuando básicamente se mezcla el concepto de parámetros globales y de variables `let` locales. Esto es, si se usa el nombre de un parámetro previamente establecido dentro de la sección de asignación de una `let`, como esto:

```
(let ((*todays-humidity* 50))
  (do-something))
```

el parámetro tendrá el valor especificado no sólo textualmente dentro del cuerpo de la `let`, sino también en cualquiera de las funciones que puedan ser llamadas desde dentro del cuerpo de la `let`. El valor global del parámetro en cualquier otro contexto no quedará afectado. Debido a que este comportamiento de ámbito es determinado por la ejecución "dinámica" en tiempo de ejecución del programa, nos referimos a él como el ámbito dinámico.

El ámbito dinámico se usa con frecuencia para cambiar el valor de un parámetro global particular sólo dentro de un árbol particular de llamadas a función. Usando el ámbito dinámico, se puede conseguir esto sin afectar a otros sitios en el programa que puedan referirse al parámetro. Además, no hay que recordar hacer que el código "restablezca" el parámetro de regreso a su valor global predeterminado, ya que automáticamente "rebotará" de regreso a su valor global normal.

La capacidad para el ámbito dinámico es especialmente útil en un CL *multi-hilo*, e.d., un proceso CL que puede tener muchos hilos (virtuales) simultáneos de ejecución. Un parámetro puede tomar un valor de ámbito dinámico en un hilo, sin afectar el valor del parámetro en ninguno de los otros hilos que se estén ejecutando concurrentemente.

3.2 La lista como estructura de datos

En esta sección presentaremos algunos de los operadores nativos de CL fundamentales para la manipulación de listas como estructuras de datos. Estos incluyen operadores para hacer cosas como:

1. encontrar la longitud de una lista
2. acceder a miembros particulares de una lista
3. agregar juntas múltiples listas para crear una nueva
4. extraer elementos de una lista para crear una nueva

3.2.1 Acceso a los elementos de las listas

Common Lisp define las funciones de acceso desde la `first` hasta la `tenth` como un medio para acceder los primeros diez elementos de una lista:

```
CL-USER(5): (first '(a b c))
 A

CL-USER(6): (second '(a b c))
 B

CL-USER(7): (third '(a b c))
 C
```

Para acceder a elementos en una posición arbitraria de la lista, se puede usar la función `nth`, que toma un entero y una lista como sus dos argumentos:

```
CL-USER(8): (nth 0 '(a b c))
 A

CL-USER(9): (nth 1 '(a b c))
 B

CL-USER(10): (nth 2 '(a b c))
 C

CL-USER(11): (nth 12 '(a b c d e f g h i j k l m n o p))
 M
```

Observe que `nth` comienza su indexación en cero (0), así que (`nth` 0 ...) es equivalente a (`first` ...) y (`nth` 1 ...) es equivalente a (`second` ...), etc.

3.2.2 El "resto" de la historia

Una operación muy común en CL es realizar alguna operación en el `first` de una lista, entonces realiza la misma operación en cada `first` del `rest` de la lista, repitiendo el procedimiento hasta el final de la lista, e.d. se alcance la Lista Vacía. La función `rest` es muy útil en tales casos:

```
CL-USER(59): (rest '(a b c))
(B C)
```

Como se puede ver en este ejemplo, rest devuelve una lista consistente en todos sus argumentos menos el primero.

3.2.3 La Lista Vacía

El símbolo NIL se define en Common Lisp como equivalente a la Lista Vacía, (). NIL también tiene la interesante propiedad de que su valor es él mismo, lo que significa que siempre se evalúa en sí mismo, esté o no entre comillas. Así NIL, 'NIL y () todos se evalúan en la Lista Vacía, cuya representación impresa predeterminada es NIL:

```
CL-USER(14): nil
NIL
```

```
CL-USER(15): 'nil
NIL

CL-USER(16): ()
NIL
```

La función null se puede usar para comprobar si algo es o no la Lista Vacía:

```
CL-USER(17): (null '(a b c))
NIL

CL-USER(18): (null nil)
T

CL-USER(19): (null ())
T
```

Como se habrá deducido de los ejemplos de arriba, el símbolo T es la representación predeterminada de CL para "verdadero". Como en el caso de NIL, T se evalúa en sí mismo.

3.2.4 ¿Eres una lista?

Para comprobar si un objeto particular es o no una lista, CL proporciona la función `listp`. Como muchas otras funciones que finalizan con la letra "p", esta función toma un solo argumento y comprueba si cumple ciertos criterios (en este caso, si califica como lista). Estas funciones predicado que finalizan con la letra "p" siempre devolverán T o NIL:

```
CL-USER(20): (listp '(pontiac cadillac chevrolet))
T

CL-USER(21): (listp 99)
NIL

CL-USER(22): (listp nil)
T
```

Observe que `(listp nil)` devuelve T, ya que NIL también es una lista (aunque sea la lista vacía).

3.2.5 El condicional `if`

Antes de continuar con otras funciones básicas para listas, cubriremos la macro `if`, que permite condicionales simples. Toma tres argumentos, una forma de comprobación, una forma entonces, y una forma si no. Cuando se evalúa una forma `if`, primero evalúa su forma de comprobación. Si la forma devuelve no-`NIL`, evaluará la forma entonces, y si no, evaluará la forma si no. Es posible el anidamiento de múltiples expresiones `if`, pero no aconsejado; más tarde cubriremos otras construcciones que son más apropiadas para tales casos. Estos son algunos ejemplos simples que usan el operador `if`:

```
CL-USER(2): (if (> 3 4)
                "no"
              "yes")
"yes"
```

```
CL-USER(3): (if (listp '("Chicago" "Detroit" "Toronto"))
                "it is a list"
              "it ain't a list")
"it is a list"

CL-USER(4): (if (listp 99)
                "it is a list"
              "it ain't a list")
"it ain't a list"
```

3.2.6 La longitud (`length`) de una lista

Normalmente se puede usar la función length para obtener el número de elementos de una lista, como un entero:

```
CL-USER(5): (length '(gm ford chrysler volkswagen))
4

CL-USER(6): (length nil)
0
```

La función length, como casi todo en Common Lisp, puede ella misma estar implementada en CL. Esta es una versión simplificada de la función '`our-length`' que ilustra cómo puede implementarse length:

```
(defun our-length (list)
  (if (null list)
      0
    (+ (our-length (rest list)) 1)))
```

Observe que la función usa el símbolo `list` para nombrar a su argumento, lo que es perfectamente válido como decíamos en la sección 3.1.4.

En español, esta función dice básicamente: "Si la lista está vacía, su longitud es cero. De lo contrario su longitud es uno más que la longitud de su `rest`". Como en el caso de muchas funciones que operan sobre listas, esta definición recursiva es una manera natural de expresar la operación `length`.

3.2.7 Miembro de una lista

La función `member` ayudará a determinar si un elemento particular es miembro de una lista particular. Como muchas funciones similares, `member` usa `eql` para comprobar la igualdad, que es una de las funciones de igualdad más básicas de CL (véase la Sección 3.8.4 para consideraciones acerca de la igualdad en CL). `eql` básicamente significa que los dos objetos deben ser el mismo símbolo, entero u objeto real (e.d. la misma dirección en la memoria). `member` toma dos argumentos: un elemento y una lista. Si el elemento no está en la lista, devuelve `NIL`. De lo contrario, devuelve el resto de la lista, empezando por el elemento encontrado:

```
CL-USER(7): (member 'madrid '(bogotá buenos-aires quito))
NIL

CL-USER(8): (member 'buenos-aires '(bogotá buenos-aires quito))
(BUENOS-AIRES QUITO)
```

Como con length, podríamos definir member usando una definición de función que es muy cercana a la descripción inglesa de lo que se supone que hace la función[1]:

```
(defun nuestro-miembro (elem list)
  (if (null list)
      nil
    (if (eql elem (first list))
       list
      (nuestro-miembro elem (rest list)))))
```

En inglés, se podría leer esta función como que dice "Si la lista está vacía, devuélvase `NIL`. De lo contrario, si el elemento deseado es el mismo que el primero de la lista, devuélvase la lista entera. De lo contrario, hágase lo mismo en el resto de la lista". Observe que, para el propósito de cualquier tipo de operadores lógicos, devolver cualquier valor no-`NIL` es "tan bueno como" devolver T. Así los posibles valores de retorno de la función `member` son tan buenos devolviendo `T` o `NIL` hasta donde les concierne a los operadores lógicos.

3.2.8 Obtención de parte de una lista

`subseq` es una función común que se usa para devolver una porción de una lista (o realmente cualquier tipo de secuencia). `subseq` toma al menos dos argumentos, una lista y un entero que indica la posición desde donde empezar. También toma un tercer argumento optativo, un entero que indica la posición donde detenerse. Observe que la posición indicada por este tercer argumento no se incluye en el la sub-lista devuelta:

```
CL-USER(9): (subseq '(a b c d) 1 3)
(B C)

CL-USER(10): (subseq '(a b c d) 1 2)
(B)

CL-USER(11): (subseq '(a b c d) 1)
(B C D)
```

Observe también que el tercer argumento optativo, de no darse, se entiende la longitud de la lista.

3.2.9 Añadido de listas (append)

La función `append` toma cualquier número de listas, y devuelve una nueva lista que resulta de añadirlas juntas. Como muchas funciones CL, `append` no tiene efectos colaterales, esto es, simplemente devuelve una nueva lista como valor de retorno, pero no modifica sus argumentos de ninguna forma:

```
CL-USER(6): (setq mi-pase '(introducción bienvenida listas funciones))
 (INTRODUCCION BIENVENIDA LISTAS FUNCIONES)

CL-USER(7): (append mi-pase '(números))
 (INTRODUCCION BIENVENIDA LISTAS FUNCIONES NUMEROS)

CL-USER(8): mi-pase
 (INTRODUCCION BIENVENIDA LISTAS FUNCIONES)
```

1 El uso de la sentencia `if` "anidada" de este ejemplo, aunque funcionalmente correcta, es una violación del buen estilo CL. Más tarde aprenderemos a evitar sentencias `if` anidadas.

```
CL-USER(9): (setq mi-pase (append mi-pase '(números)))
 (INTRODUCCION BIENVENIDA LISTAS FUNCIONES NUMEROS)

CL-USER(10): mi-pase
 (INTRODUCCION BIENVENIDA LISTAS FUNCIONES NUMEROS)
```

Observe que la simple llamada a `append` no afecta a la variable *mi-pase*. Si deseamos modificar el valor de esta variable, sin embargo, una manera de hacerlo es usando `setq` en combinación con la llamada a `append`. Observe también el uso de `setq` directamente en el nivel máximo con un nuevo nombre de variable. Para probar y "hackear" en la línea de comandos, esto es aceptable, pero en un programa finalizado todas esas variables globales deberían en realidad ser declaradas formalmente con `defparameter` o `defvar`.

3.2.10 Añadido de elementos a las listas

Para añadir un solo elemento al frente de una lista, se puede usar la función cons:

```
CL-USER(13): (cons 'a '(b c d))
 (A B C D)
```

`cons` es de hecho la función primitiva de más bajo nivel sobre la que están construidas muchas de las otras funciones constructoras de listas. Ya se habrá adivinado que '`cons`' significa 'CONStruir', como en "construir una lista". Cuando se lea u oiga a la gente hablar acerca de un programa CL que hace mucho "consing", se están refiriendo al comportamiento del programa de construir un montón de estructuras de listas, muchas de las cuales serán transitorias y hará falta "liberarlas" con el sub-sistema automático de gestión de memoria, o "recolector de basura". Como con `append`, `cons` es no-destructiva, lo que significa que no tiene efectos colaterales (modificaciones) sobre sus argumentos.

3.2.11 Remoción de elementos de las listas

La función remove toma dos argumentos, cualquier elemento y una lista, y devuelve una nueva lista con todas las presencias del elemento:

```
CL-USER(15): (setq data '(1 2 3 1000 4))
 (1 2 3 1000 4)

CL-USER(16): (remove 1000 data)
 (1 2 3 4)

CL-USER(17): data
 (1 2 3 1000 4)

CL-USER(18): (setq data (remove 1000 data))
 (1 2 3 4)

CL-USER(19): data
 (1 2 3 4)
```

Como `append` y `cons`, `remove` es no-destructiva y así no modifica sus argumentos. Como antes, una manera de conseguir modificaciones en variables es con `setq`.

3.2.12 Ordenamiento de listas

La función `sort` ordenará cualquier secuencia (incluyendo, por supuesto, una lista), basándose en la comparación de sus elementos mediante el empleo de cualquier función de comparación aplicable, o función predicado. Debido a razones de eficiencia, los diseñadores de CL tomaron la decisión de permitir que sort modificara ("reciclar" la memoria en) su secuencia argumental, así que siempre se deberá capturar ("catch") el resultado del ordenamiento haciendo algo explícitamente con el valor de retorno:

```
CL-USER(20): (setq data '(1 3 5 7 2 4 6))
 (1 3 5 7 2 4 6)

CL-USER(21): (setq data (sort data #'<))
 (1 2 3 4 5 6 7)

CL-USER(22): (setq data (sort data #'>))
 (7 6 5 4 3 2 1)
```

[–> NdT: Si `sort` es destructiva y de hecho reordena el contenido de data, dejándolo así (como acabo de comprobrar haciendo tan solo un `sort` y después mirando el valor de data, ¿para qué hace falta rodear esa operación con `setq` data??]

Adviértase que la función de comparación debe ser una función que pueda tomar dos argumentos: cualesquier dos elementos representativos de la secuencia dada, y compararlos para que den un resultado T o NIL. La notación "#'" es un atajo para recuperar el objeto de función real asociado con un símbolo o expresión (en este caso, el símbolo < o el >). Cubriremos más este tema en la Sección 3.4. En los ejemplos de arriba, simplemente reponemos el valor de la variable *data* en el resultado de `sort`, que es una cosa común. Si uno quisiera retener la secuencia original sin ordenar, una versión "segura" de `sort` podría definirse como sigue:

```
(defun safe-sort (list predicate)
  (let ((new-list (copy-list list)))
    (sort new-list predicate)))
```

3.2.13 Tratamiento de las listas como conjuntos

Las funciones `union`, `intersection` y `set-difference` toman dos listas y computan la operación de conjuntos correspondiente. Debido a que los conjuntos matemáticos no poseen la noción de ordenamiento, el orden de los resultados devueltos por estas funciones es puramente arbitrario, así que nunca se debería depender de él:

```
CL-USER(23): (union '(1 2 3) '(2 3 4))
 (1 2 3 4)

CL-USER(25): (intersection '(1 2 3) '(2 3 4))
 (3 2)

CL-USER(26): (set-difference '(1 2 3 4) '(2 3))
 (4 1)
```

3.2.14 Mapeo de una función en una lista

Si se tiene una lista, y se desea otra lista de la misma longitud, existe una buena oportunidad de que se pueda usar una de las funciones de mapeo. mapcar es la más común de tales funciones. mapcar toma una función y una o más listas, y mapea la función a través de cada elemento de la lista, produciendo una nueva lista resultante. El término 'car' proviene de la manera original en que Lisp referenciaba al primer elemento de una lista ("contents of address register"). Por tanto 'mapcar' toma su función y la aplica al primero de cada resto sucesivo de la lista:

```
CL-USER(29): (defun twice (num)
               (* num 2))
TWICE

CL-USER(30): (mapcar #'twice '(1 2 3 4))
(2 4 6 8)
```

Las funciones `Lambda` (sin nombre) se usan con mucha frecuencia con `mapcar` y funciones de mapeo similares. Más sobre esto en la Sección 3.4.3.

3.2.15 Listas de propiedades

Las listas de propiedades ('plistas') proporcionan una vía simple pero poderosa de manejar pares de palabras-clave/valores. Una plista es simplemente una lista, con un número par de elementos, donde cada pareja de elementos representa un valor con nombre. Este es un ejemplo de plista:

```
(:michigan "Lansing" :illinois "Springfield"
 :pennsylvania "Harrisburg")
```

En esta plista, las *claves* son símbolos de palabras clave, y los *valores* son cadenas. Las claves en una plista son con mucha frecuencia símbolos de palabras clave. Los símbolos de palabras clave son símbolos cuyos nombres van precedidos por un signo de dos puntos (':'), y que son usadas generalmente justo para comparar el símbolo en sí (e.d. típicamente no se usan por su valor del símbolo o función del símbolo). Para acceder a los miembros de una plista, CL proporciona la función getf, que toma una plista y una clave:

```
CL-USER(34): (getf '(:michigan "Lansing"
                     :illinois "Springfield"
                     :pennsylvania "Harrisburg")
                   :illinois)
"Springfield"
```

3.3 Control de ejecución

El control de la ejecución en CL se convierte en el control de la evaluación. Usando algunas macros estándares y comunes, se controlan qué formas se evalúan, entre ciertas elecciones.

```
(if (eql status :all-systems-go)
    (progn (broadcast-countdown) (flash-colored-lights)
           (play-eerie-music)(launch-rocket))
  (progn (broadcast-apology) (shut-down-rocket)))
```

Figura 3.1: `progn` empleada con `if`

3.3.1 La forma `if`

Quizás el operador más básico para imponer control es el operador `if`, que ya hemos introducido brevemente en la Sección 3.2.5. El operador `if` toma exactamente tres argumentos, cada uno de los cuales es una expresión que puede ser evaluada. El primero es una forma de comprobación, el segundo una forma "entonces", y el tercero una forma "si no" (que puede dejarse en un NIL por omisión). Cuando CL evalúa la forma `if`, primero evaluará la forma de comprobación. Dependiendo de si el valor de retorno de la forma de comprobación es no-`NIL` o `NIL`, se evaluará la forma entonces o la forma si no.

Si se quieren agrupar múltiples expresiones juntas como parte de la forma entonces o de la forma si no, se las debe envolver dentro de un bloque que las encierre. Se usa comúnmente `progn` para este propósito. `progn` aceptará cualquier número de formas en calidad de argumentos, y evaluará cada una de ella por turno, devolviendo finalmente el valor de retorno de la última de ellas (véase la Figura 3.1).

3.3.2 La forma when

En algunos aspectos `when` es similar a `if`, pero se debería usar `when` en casos donde se sepa que la cláusula `else` es simplemente `NIL`. Esto se revela como una situación común. Otra ventaja del uso de `when` en estos casos es que no hay necesidad de que `progn` agrupe múltiples formas: `when` toma simplemente una forma de comprobación y entonces cualquier número de formas "entonces":

```
CL-USER(36): (when (eql database-connection :active)
             (read-record-from-database)
             (chew-on-data-from-database)
             (calculate-final-result))
999

CL-USER(37): (when (> 3 4)
               (princ "I don't think so..."))
 NIL
```

(Observe que las funciones de bases de datos de arriba son sólo ejemplos y no están predefinidas en CL).

3.3.3 Operadores lógicos

Los operadores lógicos `and` y `or` evalúan una o más expresiones dadas como argumentos, dependiendo de los valores de retorno de las expresiones. Como dijimos en la sección 3.2.3, T es la representación predeterminada de "verdadero", NIL (equivalente a la Lista Vacía) es la representación predeterminada en CL de "falso" y cualquier valor no-NIL es "tan bueno como" T, al menos en lo que concierne al valor de "verdad":

```
CL-USER(38): (and (listp '(chicago detroit boston new-york))
                  (listp 3.1415))
 NIL

CL-USER(39): (or (listp '(chicago detroit boston new-york))
                 (listp 3.1415))
 T
```

De hecho lo que ocurre aquí es que el operador `and` evalúa sus argumentos (expresiones) uno cada vez, de

izquierda a derecha, devolviendo NIL tan pronto como encuentre uno que devuelva NIL; de lo contrario devolverá el valor de su última expresión. `or` evalúa sus argumentos hasta que encuentre uno que devuelva no-NIL, y devolverá ese valor no-NIL si encuentra uno. De lo contrario terminará devolviendo NIL. `not` toma una expresión única como argumento y la niega; esto es, si la expresión devuelve NIL, `not` devolverá T, y si el argumento devuelve no-NIL, not devolverá NIL. Desde un punto de vista lógico, `not` se comporta idénticamente a la función `null`, pero semánticamente, `null` porta el significado de "comprobar una Lista Vacía", mientras que `not` porta el significado de la negación lógica:

```
CL-USER(45): (not NIL)
 T

CL-USER(46): (not (listp 3.1415))
 T
```

3.3.4 La forma cond

Debido a que el uso de sentencias `if` "anidadas" no es apropiado, CL proporciona la macro `cond` para acomodar esto. Se usa `cond` en situaciones en las que en otros lenguajes se podría usar un "`If...then...else if...else if...else if...`" repetido. `cond` toma como argumentos cualquier número de formas de comprobación de expresiones. Cada una de estas formas consiste en una lista cuyo first es una expresión a ser evaluada, y cuyo `rest` es cualquier número de expresiones que serán evaluadas si la primera forma se evalúa en no-NIL. Tan pronto como se encuentre una forma no-NIL, sus expresiones correspondientes son evaluadas y la `cond` entera finaliza (e.d. no hay necesidad de una sentencia "`break`" explícita o nada de esa naturaleza):

```
CL-USER(49): (let ((n 4))
               (cond ((> n 10) "It's a lot")
                     ((= n 10) "It's kind of a lot")
                     ((< n 10) "It's not a lot")))
 "It's not a lot"
```

Debido a que T como expresión simplemente se evalúa en sí misma, una manera conveniente de especificar una cláusula "predeterminada" o "todoterreno" en una `cond` es usar T como la expresión condicional final:

```
CL-USER(50): (let ((n 4))
              (cond ((> n 10) "It's a lot")
                    ((= n 10) "It's kind of a lot")
                    (t "It's not a lot")))
 "It's not a lot"
```

3.3.5 La forma case

Si se quiere tomar una decisión basada en la comparación de una variable frente a un conjunto conocido de valores constantes, case es con frecuencia más conciso que `cond`:

```
CL-USER(51): (let ((color :red))
               (case color
                 (:blue "Blue is okay")
```

```
                  (:red "Red is actually her favorite color")
                  (:green "Are you nuts?")))
 "Red is actually her favorite color"
```

Observe que case emplea `eql` para hacer su comparación, así que sólo funcionará con constantes que sean símbolos (incluyendo símbolos de palabras clave), enteros, etc. No lo hará con cadenas. CL también proporciona las macros `ecase` y `ccase`, que funcionan justo como case, pero proporcionan alguna manipulación de errores automática para los casos en los que el valor dado no concuerde con ninguna de las claves. El símbolo `otherwise` tiene un significado especial en la macro case: indica un caso "predeterminado," o "todoterreno":

```
CL-USER(52): (let ((color :orange))
               (case color
                 (:blue "Blue is okay")
                 (:red "Red is actually her favorite color")
                 (:green "Are you nuts?")
                 (otherwise "We have never heard of that!")))
 "We have never heard of that!"
```

3.3.6 Iteraciones

Además de su habilidad innata para la recursividad y el mapeado, CL proporciona varios operadores para hacer reiteraciones de tipo tradicional ("bucles"). Estas incluyen macros tales como do, `dolist`, `dotimes` y la macro reiterativa "para todo", `loop`. `dolist y dotimes` son dos de las más usadas, así que las cubriremos aquí:

```
CL-USER(53): (let ((result 0))
               (dolist (elem '(4 5 6 7) result)
                 (setq result (+ result elem))))
22
```

Como se ve arriba, `dolist` toma una lista de inicialización y un cuerpo. La lista de inicialización consiste en una variable-iteradora, una forma-lista, y una forma-de-valor-de-retorno optativa (por omisión NIL). El cuerpo será evaluado una vez con la variable-iteradora puesta en cada elemento de la lista que es devuelto por la forma-lista. Cuando la lista se ha agotado, la forma-de-valor-de-retorno será evaluada y dicho valor será devuelto para la `dolist` entera. En otras palabras, el número de reiteraciones de una dolist lo determina la longitud de la lista devuelta por su forma-lista.

`dotimes` es similar en muchos aspectos a `dolist`, pero en vez de una forma-lista, toma una expresión que debería evaluarse en un entero. El cuerpo se evalúa una vez con la variable-iteradora puesta en cada entero empezando desde cero (0) hasta uno menos del valor de la forma-entero:

```
CL-USER(54): (let ((result 1))
               (dotimes (n 10 result)
                 (setq result (+ result n))))
 46
```

En situaciones donde se quiere que el programa salga pronto de una reiteración, usualmente se puede conseguir llamando explícitamente a `return`.

3.4 Las funciones como objetos

3.4.1 Funciones con nombre

Cuando se trabaja con CL, es importante comprender que una función es realmente un tipo de objeto, separada de cualquier símbolo o nombre de símbolo con el que pueda estar asociada. Se puede acceder al objeto-función en la ranura de función de un símbolo llamando a la función `symbol-function` sobre un símbolo, así:

```
(symbol-function '+)
#<Function +>
```

o así:

```
(symbol-function 'list)
#<Function LIST>
```

Como se puede ver, la *representación impresa* de un objeto-función con nombre contiene el nombre de símbolo de la función encerrado entre paréntesis angulares, precedido por un signo de almohadilla.

Se puede llamar explícitamente a un objeto-función con la función `funcall`:

```
CL-USER(10): (funcall (symbol-function '+) 1 2)
 3

CL-USER(11): (setq my-function (symbol-function '+))
 #<Function +>

CL-USER(12): (funcall my-function 1 2)
 3
```

Una manera rápida de escribir symbol-function es escribir '#' antes del nombre de una función:

```
CL-USER(10): (funcall #'+ 1 2)
 3

CL-USER(11): (setq my-function #'+)
 #<Function +>

CL-USER(12): (funcall my-function 1 2)
 3
```

3.4.2 Argumentos funcionales

Los *argumentos funcionales* son los argumentos de las funciones tal que ellos mismos son objetos-funciones. Ya lo hemos visto con `mapcar, sort y funcall`:

```
CL-USER(13): (defun add-3 (num)
               (+ num 3))
 ADD-3

CL-USER(14): (symbol-function 'add-3)
 #<Interpreted Function ADD-3>

CL-USER(15): (mapcar #'add-3 '(2 3 4))
 (5 6 7)
```

```
CL-USER(17): (funcall #'add-3 2)
 5
```

La idea de pasar funciones a otras funciones es a veces referida como funciones de orden superior. CL proporciona soporte natural para este concepto.

3.4.3 Funciones anónimas

Alguna vez se usará una función tan raramente que difícilmente tiene valor tener que crear una función con nombre mediante `defun`. Para estos casos, CL proporciona el concepto de '`lambda`', o función "anónima" (sin nombre). Para usar una función 'lambda', se pone la definición de función entera en la posición donde normalmente se pondría tan sólo el símbolo que bautiza a la función, identificada con el símbolo especial '`lambda`':

[NdT: No es una explicación conceptualmente correcta. "`Lambda`" es un concepto mucho más amplio y profundo. En el Cálculo Lambda y por tanto en Lisp en realidad toda función con nombre es verdaderamente una función lambda "enmascarada", así que la definición del autor es rápida, fácil de entender y práctica, pero es falsa, y genera una aproximación superficial a los conceptos profundos del lenguaje, que en su momento pasarán factura inevitablemente. Aunque no es al único autor que le veo explicarlo así...]

```
CL-USER(19): (mapcar #'(lambda(num) (+ num 3))
                     '(2 3 4))
 (5 6 7)

CL-USER(20): (sort '((4 "Buffy") (2 "Keiko") (1 "Judy") (3 "Aruna"))
                  #'(lambda(x y)
                  (< (first x) (first y))))
((1 "Judy") (2 "Keiko") (3 "Aruna") (4 "Buffy"))
```

3.4.4 Argumentos optativos

Los argumentos optativos de una función deben ser especificados después de cualquier argumento requerido, y se identifican con el símbolo `&optional` en la lista de argumentos. Cada argumento optativo puede ser o bien un símbolo o una lista que contenga un símbolo y una expresión que devuelva un valor predeterminado. En el caso de un símbolo, el valor predeterminado se dice que es NIL:

```
CL-USER(23): (defun greeting-list (&optional (username "Jake"))
               (list 'hello 'there username))
GREETING-LIST

CL-USER(24): (greeting-list)
(HELLO THERE "Jake")

CL-USER(25): (greeting-list "Joe")
(HELLO THERE "Joe")
```

3.4.5 Argumentos de palabras clave

Los argumentos de palabras clave de una función son básicamente argumentos optativos con nombre. Se

definen casi igual que los argumentos optativos:

```
(defun greeting-list (&key (username "Jake")
                           (greeting "how are you?"))
  (list 'hello 'there username greeting))
```

Pero cuando se llama a una función con argumentos de palabras clave, se los conecta a una plista que contiene los símbolos de las palabras clave como nombres de los argumentos junto con sus valores:

```
CL-USER(27): (greeting-list :greeting "how have you been?")
(HELLO THERE "Jake" "how have you been?")

CL-USER(28): (greeting-list :greeting "how have you been?" :username
"Joe")
(HELLO THERE "Joe" "how have you been?")
```

Observe que con argumentos de palabras clave, se usa un símbolo normal (e.d. *sin* el signo precedente de dos puntos) en la *definición* de la función, pero un símbolo de palabra clave como la "etiqueta" cuando se llama a la función. Existe una razón lógica detrás de esto, que el lector entenderá pronto.

3.5 Entrada, Salida, Flujos y Cadenas

La entrada y la salida en CL se hace mediante objetos llamados flujos. Un flujo es una fuente o un destino para piezas de datos que generalmente están conectados a alguna fuente o destino físicos, tales como una ventana de la consola de la terminal, un fichero en el disco duro de la computadora, o un navegador web. Cada hilo de ejecución en CL define parámetros globales que representan algunos flujos estándares:

- **standard-input** (entrada estándar) es la fuente de datos predeterminada.
- **standard-output** (salida estándar) es el destino predeterminado de los datos.
- **terminal-io** (e/s de la terminal) está asociado a la ventana de la consola (indicador de comandos), y por omisión es idéntico tanto a la `*standard-input*` como a la `*standard-output*`.

3.5.1 El mecanismo `read`

`read` es el mecanismo estándar para leer elementos de datos en CL. `read` toma un argumento único optativo para su flujo, y lee un único elemento de datos desde ese flujo. Lo predeterminado para el flujo es *`*standard-input*`*. Si se escribe simplemente (`read`) en el indicador de comandos, CL esperará hasta que el usuario introduzca algún elemento Lisp válido, y devolverá ese elemento. `read` simplemente lee, no evalúa, así que no hay necesidad de comillar los datos que se lean. Se puede establecer una variable en el resultado de una llamada a `read`:

```
(setq myvar (read))
```

La función `read-line` es similar a `read`, pero leerá caracteres hasta que encuentre un salto de línea o un final de fichero, y devolverá esos caracteres en la forma de cadena. `read-line` es apropiada para leer datos desde ficheros que estén en un formato "basado en líneas" más que formateados como expresiones Lisp.

3.5.2 Las funciones `Print` y `Prin1`

`print` y `prin1` toman cada una exactamente un objeto CL y le da salida a su representación impresa en un flujo, por omisión la salida estándar (**standard-output**). `print` pone un espacio y un salto de línea después del elemento (separando efectivamente los elementos individuales con espacio en blanco), y `Prin1` le da salida al elemento sin espacios extra:

```
CL-USER(29): (print 'hola)

HOLA
HOLA
```

En el ejemplo de arriba, se ve el símbolo *hola* aparecer dos veces: la primera de ellas es la salida que realmente se está imprimiendo en la consola, y la segunda es el valor de retorno normal de la llamada a `print` que se está imprimiendo en la consola por el bucle de lectura-evaluación-impresión. `print` y `prin1` ambas imprimen su salida legiblemente, lo que significa que la función `read` de CL será capaz de leer los datos de vuelta. Por ejemplo, las comillas dobles de una cadena se incluirán en la salida:

```
CL-USER(30): (print "hello")

"hello"
"hello"
```

3.5.3 La función `princ`

`princ` se distingue de `print` y de `prin1` en que imprime su salida en un formato más legible para los humanos, lo que significa por ejemplo que una cadena quedará despojada de las comillas dobles en su salida:

```
CL-USER(31): (princ "hello")
hello
"hello"
```

En el ejemplo de arriba, como antes, veremos la salida por partida doble: una para la impresión real en la consola, y otra de nuevo como el valor de retorno de la llamada a `princ` impreso por el bucle de lectura-evaluación-impresión.

3.5.4 La función Format

`format` es una poderosa generalización de `prin1`, `princ` y otras funciones básicas de impresión, y puede usarse para casi toda salida. `format` toma un flujo, una cadena-de-control, y optativamente cualquier número de argumentos adicionales para rellenar los contenedores de la cadena-de-control:

```
CL-USER(33): (format t "Hello There ~a" 'bob)
Hello There BOB
NIL
```

En el ejemplo de arriba, se usa t como un atajo para especificar **standard-output** como el flujo, y ~a es un contenedor de cadena-de-control que procesa su argumento como lo hace `princ`. Si se especifica ya

sea un flujo real o T como el segundo argumento de `format`, imprimirá por efecto lateral y devolverá NIL como su valor de retorno. Sin embargo, si se especifica NIL como segundo argumento, `format` no dirige su salida a ningún flujo mediante efectos colaterales, sino que en su lugar devuelve una cadena que contiene lo que se le hubiera dado salida a un flujo si se hubiera proporcionado uno:

```
CL-USER(34): (format nil "Hello There ~a" 'bob)
"Hello There BOB"
```

Además de ~a, existe todo un surtido de otras directivas de `format` para diversas elecciones de salidas, tales como darle salida a los elementos como si lo hiciera `prin1`, darle salida a listas con un cierto carácter después de todos menos el último elemento, darle salida a números en muchos formatos distintos incluyendo números romanos y palabras inglesas, etc.

3.5.5 Nombres de rutas (Pathnames)

Para nombrar directorios y ficheros de un sistema de ficheros de computadora de una manera independiente al sistema operativo, CL proporciona el sistema *pathname*. Los nombres de ruta de CL no contienen cadenas de nombre de ruta específicas del sistema tales como las barras de los nombres de fichero de Unix/GNU-Linux o las barras invertidas de MSDOS/Windows. Un nombre de ruta CL es básicamente una estructura que soporta una función constructora, `make-pathname` y varias funciones de acceso, tales como `pathname-name`, `pathname-directory` y `pathname-type`. La estructura de `pathname` contiene seis ranuras:

- Directory
- Name
- Type
- Device
- Version
- Host

En los sistemas de ficheros estándares de Unix sólo las ranuras directorio, nombre y tipo son relevantes. En los sistemas de ficheros MSDOS/Windows, la ranura de dispositivo también lo es. En la plataforma Symbolics, que tiene un sistema de ficheros más avanzado, las ranuras de versión y anfitrión también tienen relevancia. La función constructora `make-pathname` toma argumentos de palabra clave correspondientes a las seis posibles ranuras de `pathname`. El siguiente es un ejemplo de creación de un nombre de ruta que en Unix se presentaría como `/tmp/try.txt`:

```
CL-USER(15): (defparameter *my-path*
                 (make-pathname :directory (list :absolute "tmp")
                                :name "try"
                                :type "txt"))
 *MY-PATH*

CL-USER(16): *my-path*
#p"/tmp/readme.txt"
```

Como se ve en el ejemplo de arriba, la ranura `:directory` es técnicamente una lista, cuyo `first` es un

símbolo de palabra clave (ya sea `:absolute` o `:relative`), y cuyo `rest` son los componentes de directorios individuales representados como cadenas.

3.5.6 Entrada y salida de ficheros

Hacer entrada y salida de ficheros en CL es esencialmente un asunto de combinar los conceptos de flujo y nombres de rutas. Primero se debe abrir un fichero, que conecta un flujo al fichero. CL proporciona la función `open`, que abre directamente un fichero le asocia un flujo. Por una serie de razones, sin embargo, generalmente se usará una macro llamada `with-open-file` que no solo abre el fichero, sino que lo cierra automáticamente cuando se haya acabado con él, y realiza limpieza y manejo de errores adicionales. `with-open-file` toma una *lista de especificaciones* y un *cuerpo*. La lista de especificaciones consiste en una *variable de flujo* (un símbolo) y un nombre de ruta, seguidos por varios argumentos de palabras clave optativos que especifican, por ejemplo, si el fichero es para la entrada, la salida, o para ambas. Ahora un ejemplo de apertura de un fichero y de lectura de un elemento en él (adviértase que el fichero denotado por *`*mypath*`* debe existir, o se señalará un error. Se puede usar la función `probe-file` para comprobar la existencia de un fichero.):

```
(with-open-file (input-stream *my-path*)
  (read input-stream))
```

No se requieren argumentos de palabras clave extra en la lista de especificaciones de este ejemplo, ya que el modo "`read-only`" es el predeterminado para abrir un fichero. Para escribir en un fichero, sin embargo, se deben proporcionar argumentos de palabras clave adicionales:

```
(with-open-file (output-stream *my-path*
                 :direction :output
                 :if-exists :supersede
                 :if-does-not-exist :create)
  (format output-stream "Hello There"))
```

3.6 Tablas hash, arreglos, Estructuras y Clases

CL proporciona varias otras estructuras de datos predefinidas para cubrir un amplio abanico de necesidades. Las arreglos y las tablas hash soportan el almacenamiento de valores de una manera similar a las listas y las plistas, pero con una velocidad de rastreo muy superior para los conjuntos de datos grandes. Las estructuras proporcionan otra construcción al estilo de plistas, pero más eficiente y estructurada.

Las clases extienden la idea de las estructuras para proporcionar un paradigma completo de programación orientada a objetos que soporta la herencia múltiple, y *métodos* que pueden despacharse basándose en cualquier número de argumentos especializados ("multi-métodos"). Esta colección de características de CL se denomina el Sistema de Objetos de Common Lisp, o CLOS.

3.6.1 Tablas hash

Una tabla hash es similar a lo que en otros lenguajes es conocido como arreglo asociativo. Es comparable a una arreglo de una dimensión que admite claves que son cualquier objeto CL arbitrario, que será comparado usando `eql`, v.g. símbolos o enteros. Las tablas hash soportan un rastreo de los datos

virtualmente de tiempo constante, lo que significa que el tiempo que se toma en buscar un elemento en la tabla hash permanecerá estable, incluso si el número de entradas en la tabla se incrementa.

Para trabajar con tablas hash, CL proporciona la función constructora `make-hash-table` y la función de acceso `gethash`. Para establecer entradas en una tabla hash se usa el operador `setf` en conjunción con la función de acceso `gethash`. `setf` es una generalización de `setq` que puede usarse en sitios resultantes de una llamada a función, además de exclusivamente con símbolos como con `setq`. Este es un ejemplo de creación de una tabla hash, que pone un valor en ella y que recupera un valor de ella:

```
CL-USER(35): (setq os-ratings (make-hash-table))
#<EQL hash-table with 0 entries  #x209cf822>

CL-USER(36): (setf (gethash :windows os-ratings) :no-comment)
:NO-COMMENT

CL-USER(37): (gethash :windows os-ratings)
:NO-COMMENT
T
```

Observe que la función `gethash` devuelve dos valores. Este es el primer ejemplo que vemos de una función que devuelve más de un valor. El primer valor de retorno es el valor correspondiente a la entrada encontrada en la tabla hash, si existiera. El segundo valor de retorno es un valor booleano (e.d. T o NIL), que indica si el valor se se encontró de hecho en la tabla hash. Si el valor no se encuentra, el primer valor de retorno será NIL y el segundo será también NIL. Este segundo valor es necesario para distinguir los casos en los que la entrada sí se encuentra en la tabla hash, pero sucede que su valor es precisamente NIL. Existen varias vías para acceder a múltiples valores de retorno de una función, v.g. empleando multiple-value-bind y multiple-value-list.

3.6.2 Los Arreglos

Los arreglos son estructuras de datos que pueden albergar "rejillas" simples (o multi-dimensionales) llenas de valores, que son indexadas por uno o más enteros. Como las tablas hash, los arreglos soportan un acceso muy rápido a los datos basado en índices de enteros. Se crean arreglos empleando la función constructora `make-array` y los elementos del arreglo se refieren usando la función de acceso `aref`. Como en las tablas hash, se pueden establecer elementos individuales en el arreglo usando `setf`:

```
CL-USER(38): (setq my-array (make-array (list 3 3)))
#2A((NIL NIL NIL) (NIL NIL NIL) (NIL NIL NIL))

CL-USER(39): (setf (aref my-array 0 0) "Dave")
"Dave"

CL-USER(40): (setf (aref my-array 0 1) "John")
"John"

CL-USER(41): (aref my-array 0 0)
"Dave"

CL-USER(42): (aref my-array 0 1)
"John"
```

La función `make-array` también admite una miríada de argumentos de palabras clave optativos para

inicializar el arreglo en el momento de su creación.

3.6.3 Estructuras

Las *estructuras* admiten la creación de estructuras de datos propias con ranuras nombradas, similares a, por ejemplo, la estructura de datos `pathname` que ya hemos visto. Las estructuras se definen con la macro `defstruct`, que admite muchas opciones para establecer las funciones constructoras de la instancia, las ranuras y las funciones de acceso a la estructura. Para un tratamiento completo de las estructuras, remitimos al lector a una de las referencias CL o a la documentación en línea de `defstruct`.

3.6.4 Clases y Métodos

Las clases y los métodos forman el corazón del Sistema de Objetos de Common Lisp (CLOS), y le agregan capacidades completas de orientación a objetos a Common Lisp.

Un tratamiento completo del CLOS está más allá del alcance de este libro, pero proporcionaremos una breve revisión. Las clases son básicamente "blueprints" o "prototipos" de objetos. Los objetos, en este contexto, son un tipo de estructura para agrupar juntos tanto a los datos como a la funcionalidad de cómputo. Los métodos son muy similares a las funciones, pero pueden estar especializados para su aplicación sobre combinaciones particulares de tipos de objetos.

CLOS es un sistema orientado a objetos de *funciones genéricas*, lo que significa que los métodos existen independientemente de las clases (e.d. los métodos no "pertenecen" a las clases). Los métodos pueden tomar diferentes combinaciones de tipos de instancias de clases como argumentos, sin embargo, y pueden estar especializados basándose en las combinaciones particulares de los tipos de sus argumentos. Los tipos de datos simples de CL, tales como las cadenas y los números, también se consideran como clases.

Ahora ponemos un ejemplo de definición de una clase y de un método, creando una instancia de la clase, y llamando al método empleando la instancia como argumento:

```
CL-USER(43): (defclass box () ((length :initform 10)
                               (width :initform 10)
                               (height :initform 10)))
 #<STANDARD-CLASS BOX>

CL-USER(44): (defmethod volume ((self box))
               (* (slot-value self 'length)
                  (slot-value self 'width)
                  (slot-value self 'height)))
 #<STANDARD-METHOD VOLUME (BOX)>

CL-USER(45): (setq mybox (make-instance 'box))
 #<BOX  #x209d135a>

CL-USER(46): (volume mybox)
 1000
```

Como se ve en el ejemplo de arriba, la definición de la clase toma al menos tres argumentos: un símbolo que identifique el nombre de la clase, una lista intercalada que nombra a las superclases (ninguna en este ejemplo), y una lista de ranuras para la clase, con valores predeterminados especificados por el argumento

de palabra clave `:initform`.

La definición del método es muy parecida a una definición de función, pero sus argumentos (argumento único en este caso) están especializados en un tipo de clase particular. Cuando llamamos al método, pasamos una instancia de la clase como argumento, empleando una lista de argumentos normales. Esto es distinto a Java o C++, donde un método se considera parte "de " una instancia de clase particular, y sólo los argumentos adicionales se pasan en una lista de argumentos.

CL es más consistente a este respecto.

Debido a que los tipos de datos normales de CL también se consideran clases, podemos definir métodos que se especialicen en ellos:

```
CL-USER(47): (defmethod add ((value1 string) (value2 string))
               (concatenate 'string value1 value2))
 #<STANDARD-METHOD ADD (STRING STRING)>

CL-USER(48): (defmethod add ((value1 number) (value2 number))
               (+ value1 value2))
 #<STANDARD-METHOD ADD (NUMBER NUMBER)>

CL-USER(49): (add "hello " "there")
 "hello there"

CL-USER(50): (add 3 4)
 7
```

3.7 Paquetes

Lo normal es que los símbolos existan en CL dentro de un *paquete* particular. Se puede pensar en los paquetes como en "códigos de área" para los símbolos. Son nombres de espacios usados dentro de CL para ayudar a evitar colisiones entre nombres.

Los nombres de los paquetes se representan generalmente mediante símbolos palabras clave (e.d. símbolos cuyos nombres van precedidos por un signo de dos puntos). ANSI CL tiene un sistema de paquetes "plano", lo que significa que los paquetes no "contienen" a otros paquetes al estilo jerárquico, pero se puede conseguir esencialmente el mismo efecto "superponiendo" paquetes. De hecho, los paquetes jerárquicos han sido añadidos a Allegro CL como una extensión al ANSI estándar, y pueden ser añadidos al ANSI estándar en algún momento. Sin embargo, el uso real de los paquetes jerárquicos es evitar conflictos de nombres con los mismos nombres de paquetes. En todo caso, esto se puede hacer simplemente separando los componentes de un nombre de paquete con un delimitador, por ejemplo, un punto: :com.genworks.books.

En una sesión CL dada, CL siempre tiene una noción del *paquete en uso* que está almacenado en la variable (dinámica) *`*package*`*. Es importante siempre ser consciente de cual es el paquete en uso, porque si se está en un paquete diferente del que se piensa, se pueden presentar resultados muy sorprendentes. Por ejemplo, las funciones que están definidas en ':paquete-1' no estarán necesariamente visibles en ':paquete-2', así que si se intenta ejecutar las funciones del ':paquete-2', CL pensará que están sin definir.

Normalmente el indicador de comandos de CL imprime el nombre del paquete en uso. Se puede mover a un paquete distinto, v.g. foo, con el comando de nivel superior `:package :foo`.

Se pueden desarrollar pequeños programas en el paquete `:user` pero, en general, una aplicación larga debería ser desarrollada en su propio paquete.

3.7.1 Importación y exportación de símbolos

Los paquetes pueden exportar símbolos para hacerlos accesibles a otros paquetes, e importar símbolos para que parezcan que son locales al paquete.

Si se exporta un símbolo desde un paquete pero no se importa al paquete en uso, aún es válido referirse a él. Para referirse a símbolos en otros paquetes, se cualifica al símbolo con el nombre del paquete y un solo signo de dos puntos: `package-2:foo`.

Si un símbolo no se exporta desde el paquete exterior, aún podemos referirnos a él, pero esto representaría una violación de estilo, como se significa con el hecho de que habría de usarse un doble signo de dos puntos: `package-2::bar`.

3.7.2 El paquete Keyword

Hemos visto muchos casos de símbolos cuyos nombres están precedidos por un signo de dos puntos (':'). Estos símbolos de hecho residen dentro del paquete `:keyword`, y los dos puntos son un atajo para escribirlos e imprimirlos. Los símbolos de palabras clave se usan generalmente para valores enumerados y para nombrar argumentos de funciones (palabras clave). Son "inmunes" al paquete (e.d. no hay posibilidad de confusión acerca de en qué paquete están), lo que los hace especialmente convenientes para estos propósitos.

3.8 Tropiezos comunes

Esta sección trata de errores y tropiezos comunes en los que incurren con frecuencia los nuevos usuarios de CL.

3.8.1 Comillas

Un despiste común es no entender *cuándo hay que comillar* las expresiones. Comillar un solo símbolo simplemente devuelve el símbolo, mientras que sin comilla, el símbolo se considera una variable:

```
CL-USER(6): (setq x 1)
1

CL-USER(7): x
1

CL-USER(8): 'x
X
```

Comillar una lista devuelve la lista, mientras que sin comilla, la lista se considera una llamada a función (o a macro):

```
CL-USER(10): '(+ 1 2)
(+ 1 2)
```

```
CL-USER(11): (first '(+ 1 2))
+

CL-USER(12): (+ 1 2)
3

CL-USER(13): (first (+ 1 2))
Error: Attempt to take the car of 3 which is not listp.
```

3.8.2 Listas de argumentos de funciones

Cuando se define una función con `defun`, los argumentos van dentro de su propia lista:

```
CL-USER(19): (defun square (num)
               (* num num))
SQUARE
```

Pero cuando se llama a la función, la lista de argumentos viene junta directamente tras el nombre de la función:

```
CL-USER(20): (square 4)
16
```

Una equivocación común es poner la lista de argumentos dentro de su propia lista cuando se llama a la función, así:

```
CL-USER(21) (square (4))
Error: Funcall of 4 which is a non-function.
```

Si se está acostumbrado a programar en Perl o Java, probablemente se hará esto de vez en cuando, hasta que uno se acostumbre a la sintaxis CL.

3.8.3 Símbolos vs. Cadenas

Otro tipo de confusión es la diferencia entre símbolos y cadenas. El símbolo y la cadena 'AAA' se tratan de maneras completamente distintas en CL. Los símbolos residen en paquetes; las cadenas no. Más aún, todas las referencias a un símbolo dado en un paquete dado refieren a la misma dirección real en la memoria: un símbolo dado en un paquete dado sólo se adjudica una vez. En contraste, CL puede adjudicar nueva memoria cada vez que el usuario defina una cadena, así que podría ocurrir que hubiera múltiples copias de la misma cadena de caracteres en la memoria. Considérese este ejemplo:

```
CL-USER(17): (setq a1 'aaa)
AAA

CL-USER(18): (setq a2 'aaa)
AAA

CL-USER(19): (eql a1 a2)
T
```

`eql` compara punteros o enteros existentes: a1 y a2 apuntan ambos al mismo símbolo en la misma parte de la memoria.

```
CL-USER(20): (setq b1 "bbb")
"bbb"

CL-USER(21): (setq b2 "bbb")
"bbb"

CL-USER(22): (eql b1 b2)
NIL

CL-USER(23): (string-equal b1 b2)
T
```

De nuevo, `eql` compara punteros, pero aquí b1 y b2 apuntan a diferentes direcciones de memoria, aunque ocurra que esas direcciones contengan la misma cadena. De ahí que la comparación con `string-equal`, que compara cadenas carácter a carácter, en vez de los punteros a esas cadenas, devuelva T.

Otra diferencia entre símbolos y cadenas es que CL proporciona una cantidad de operaciones para manipular cadenas que no funcionan para los símbolos:

```
CL-USER(25): (setq c "Jane dates only Lisp programmers")
 "Jane dates only Lisp programmers"

CL-USER(26): (char c 3)
#\e

CL-USER(27): (char 'xyz 1)
Error: `XYZ' is not of the expected type `STRING'

CL-USER(29): (subseq c 0 4)
"jane"

CL-USER(30): (subseq c (position #\space c))
" dates only lisp programmers"

CL-USER(31): (subseq c 11 (length c))
"only lisp programmers"
```

```
(defun nearly-equal? (num1 num2 &optional (tolerance *zero-epsilon*))
  (< (abs (- num1 num2)) tolerance))
```

Figura 3.2: Función para la comparación de comas flotantes

3.8.4 Igualdad

CL posee varios predicados diferentes para comprobar la igualdad. Esto es así debido a los muchos tipos de objetos distintos que hay en CL, que dictan diferentes nociones de lo que es igualdad.

Una regla simple es:

- Se usa `eql` para comparar símbolos, punteros (como los punteros a listas) y enteros
- Se usa = para comparar la mayoría de los valores numéricos
- Se usa `string-equal` para comparación de cadenas sin sensibilidad a mayúsculas/minúsculas
- Se usa `string=` para comparación de cadenas con sensibilidad a mayúsculas
- Se usa `equal` para comparar otros tipos de objetos (tales como listas y caracteres solos)

- Para comparar números con coma flotante que pueden diferir en un margen muy pequeño, pero aún así deberían considerarse iguales, la técnica más segura es tomar el valor absoluto de su diferencia, y compararlo con un valor 'epsilon' con tolerancia cero. En la Figura 3.2 es una sola función la que hace esto, que asume que hay definido un parámetro global, **zero-epsilon**, como valor numérico muy cercano a cero: Ejemplos:

```
CL-USER(33): (setq x :foo)
:FOO

CL-USER(34): (eql x :foo)
T

CL-USER(35): (= 100 (* 20 5))
T

CL-USER(36): (string-equal "xyz" "XYZ")
T

CL-USER(37): (setq abc '(a b c))
(A B C)

CL-USER(38): (equal abc (cons 'a '(b c)))
T

53

CL-USER(39): (equal #\a (char "abc" 0))
T

CL-USER(40): (eql abc (cons 'a '(b c)))
NIL
```

La última expresión devuelve NIL porque, aunque las dos listas tienen el mismo contenido, residen en diferentes lugares de la memoria, y por tanto los punteros a esas listas son distintos.

Observe que varias de las funciones que usan por defecto `eql` para la comparación tomarán un argumento de palabra clave optativo que puede sobre-escribir esta predefinición:

```
CL-USER(54): (member "blue" '("red" "blue" "green" "yellow")
                    :test #'string-equal)
("blue" "green" "yellow")
```

Recuérdese en la Sección 3.2.7 que los miembros devuelven el rest de la lista empezando en el elemento encontrado.

3.8.5 Distinción entre Macros y Funciones

Como las macros[1] y las funciones ambas se presentan como el primer elemento de una expresión, a primera vista puede parecer difícil distinguirlas. En la práctica, esto raramente se convierte en un problema. Para los principiantes en el lenguaje, la mejor aproximación es hacer la asunción predeterminada de que algo es una función a menos que se reconozca como macro. La mayoría de las macros serán rápidamente identificables porque les será de aplicación una de las posibilidades siguientes:

1 Los operadores especiales son otra categoría de operadores en CL, que se implementan internamente de manera diferente a las macros, pero que para nuestros propósitos podemos considerar como macros.

- Es una parte familiar del núcleo de CL, tal como `if, cond, let, setq, setf`, etc.
- Su nombre comienza con '`def`', tal como en `defun`, `defparameter`, etc.
- Su nombre empieza con '`with-`', como en `with-open-file, with-output-to-string`, etc.
- Su nombre comienza con '`do`', por ejemplo `dolist` y `dotimes`.

Más allá de estas reglas nemotécnicas generales, si uno se encuentra con dudas acerca de si algo es una función o una macro, se puede:

1. Consultar la documentación de referencia en línea o en un libro de manual de referencia de CL (véase la Bibliografía en el Apéndice B)
2. Emplear la función symbol-function para preguntarle a la sesión de CL, así:

```
CL-USER(24): (symbol-function 'list)
 #<Function LIST>

CL-USER(25): (symbol-function 'with-open-file)
 #<macro WITH-OPEN-FILE  #x2000b07a>
```

3.8.6 Operaciones que construyen (cons)

Debido a que los programas CL no tienen que liberar ni adjudicar memoria explícitamente, pueden escribirse mucho más rápidamente. En la medida en que evoluciona una aplicación, algunas operaciones que construyen ("cons") pueden ser reemplazadas por operaciones que "reciclan" la memoria, reduciendo la cantidad de trabajo que tenga que hacer el recolector de basura (subsistema de administración automática de memoria de CL).

Para escribir programas que se conserven a la hora de recoger la basura, una técnica es simplemente evitar el uso innecesario de operaciones que construyan (cons). Esta habilidad se irá obteniendo con la experiencia.

Otra técnica es usar operaciones destructivas, cuyo empleo es preferible sólo tras haber ganado más experiencia trabajando con CL. Las operaciones destructivas están en su mayor parte más allá del alcance de este libro.

Capítulo 4

Diversas Interfaces

Uno de los puntos fuertes de los entornos modernos CL con soporte comercial es su flexibilidad en trabajar con otros entornos. En este capítulo presentamos una muestra de algunas interfaces y paquetes que son útiles para desarrollar aplicaciones CL para el mundo real y conectarlas con el entorno exterior. Algunos de los paquetes cubiertos en este capítulo son extensiones comerciales a Common Lisp según las ofrece Franz Inc's Allegro CL, y algunas otras están disponibles como ofertas de Software Libre.

4.1 Interfaz con el Sistema Operativo

Uno de los medios más básicos para interactuar con el mundo exterior es simplemente invocar comandos a nivel de sistema operativo, en un estilo similar al comando "system" de Perl. En Allegro CL, una manera de conseguirlo es con la función excl.osi:command-output[1]. Este comando toma una cadena que representa un comando (posiblemente con argumentos) a ser ejecutado a través de una shell del sistema operativo, tal como la shell "C" en Unix:

```
(excl.osi:command-output "ls")
```

Cuando se invoca de esta manera, esta función devuelve la salida estándar del comando de la shell como una cadena. Alternativamente, se puede especificar que la salida vaya a otra ubicación, tal como un fichero o una variable definida dentro de la sesión CL.

Cuando se use `excl.osi:command-output`, también se puede especificar si CL debería a que el comando de la shell se complete, o simplemente debería seguir con sus asuntos y dejar que el comando de la shell se complete asíncronamente.

4.2 Interfaz a funciones ajenas

La interfaz de Allegro CL para *funciones ajenas* permite cargar bibliotecas compiladas y objetos código de C, C++, Fortran y otros lenguajes, y llamar a funciones definidas en este código como si estuvieran definidas nativamente en CL. Esta técnica requiere un poquito más de trabajo de configuración que usar simplemente comandos de la shell con `excl.osi:command-output`, pero proporciona mejor rendimiento y operaciones más transparentes una vez esté configurada.

4.3 Interfaz con Corba

Corba, o la Common Object Request Broker Architecture, es un mecanismo industrial estándar para conectar aplicaciones escritas en distintos lenguajes orientados a objetos.

Para usar Corba, se define una interfaz estándar para la aplicación en un lenguaje llamado Interface

1 Se debe tener un ACL 6.2 o posterior completamente parcheado para acceder a la funcionalidad excl.osi (si la máquina puede acceder a internet, se puede actualizar la instalación ACL al nivel de parcheo del momento con la función `sys:update-allegro`). Las versiones previas de ACL tienen una función `excl:run-shell-command` que opera de manera semejante a `excl.osi:command-output.`

Definition Language (Corba IDL), y cada aplicación debe compilar la interfaz estándar e implementar ya sea un sirviente o un cliente para la interfaz. De esta manera, los sirvientes y los clientes pueden comunicarse a través de un protocolo común, independientemente del lenguaje en el que estén escritos.

Para usar Corba con CL se requiere un compilador Corba IDL implementado en CL, así como un ORB de tiempo de ejecución, u Object Request Broker, para manejar las comunicaciones cliente/servidor que se den. Hay varios compiladores Corba IDL y ORBs disponibles para CL. Un ejemplo de un sistema así es Orblink, que está disponible como paquete optativo para Allegro CL. Orblink consiste en un compilador Corba IDL completo, y un ORB basado en CL, que se ejecuta como un hilo CL en vez de como un proceso de sistema operativo separado.

El empleo de Orblink para integrar una aplicación CL a un entorno cliente/servidor Corba es un proceso elemental. Para empezar, simplemente se obtienen los ficheros Corba IDL para la interfaz deseada, y se les compila en CL con el comando corba:idl. Esto automáticamente definirá un conjunto de clases en CL correspondientes a todas las clases, métodos, etc. que constituyen la interfaz.

Para usar CL como cliente, se puede entonces simplemente crear instancias de las clases deseadas, y usarlas para llamar a los métodos deseados. El ORB de Orblink se preocupará de comunicar estas llamadas a métodos a través de la red, donde serán invocadas en el proceso en curso donde el sirviente correspondiente esté implementado (que podría estar en una máquina completamente diferente escrita en un lenguaje completamente diferente).

Para usar CL como sirviente, se deben implementar las interfaces que hagan al caso. Esto consiste en definir clases en CL que *hereden* desde el sirviente "stub" automáticamente las clases definidas por la compilación del Corba IDL, después definir métodos que operen sobre esas clases, y que rindan según lo advertido en la interfaz. Tipicamente, una interfaz Corba sólo expondrá una pequeña pieza de la aplicación entera, así que implementar las clases del sirviente habitualmente representa un pequeño núcleo en relación a la aplicacion en sí.

4.4 Conexiones de Socket personalizadas

Casi todas las implementaciones de CL proporcionan un mecanismo para programar directamente con sockets de red, en orden de implementar "escuchadores" y aplicaciones clientes de red. Como ejemplo práctico, la Figura 4.1 muestra un servidor Telnet, escrito por John Foderaro de Franz Inc., en 22 líneas de código en Allegro CL. Este servidor Telnet permitirá a un proceso CL en ejecución aceptar conexiones Telnet en el puerto 4000 de la máquina donde esté ejecutándose, y

```
(defun start-telnet (&optional (port 4000))
  (let ((passive (socket:make-socket :connect :passive
                                     :local-host "127.1"
                                     :local-port port
                                     :reuse-address t)))
    (mp:process-run-function
     "telnet-listener"
     #'(lambda (pass)
         (let ((count 0))
           (loop
             (let ((con (socket:accept-connection pass)))
               (mp:process-run-function
                (format nil "tel~d" (incf count))
```

```
                    #'(lambda (con)
                        (unwind-protect
                            (tpl::start-interactive-top-level
                             con
                             #'tpl::top-level-read-eval-print-loop
                             nil)
                          (ignore-errors (close con :abort t))))
                     con)))))
        passive)))
```

Figura 4.1: Servidor telnet en 22 líneas

presentará el cliente con un indicador de comandos CL normal (por seguridad, sólo aceptará por defecto conexiones desde la máquina local).

4.5 Interfaz con Windows (COM, DLL, DDE)

La mayoría de las implementaciones comerciales de CL para plataformas Microsoft Windows permitirán a las aplicaciones CL usar varios medios específicos de Microsoft para integrarse con la plataforma Microsoft. Por ejemplo, una aplicación Allegro CL puede configurarse para ejecutarse como servidor COM, o compilarse en una DLL para que la usen otras aplicaciones.

4.6 Generación de código en otros lenguajes

CL es excelente leyendo y generando código CL. También es una herramienta poderosa para analizar y generar código en otros lenguajes. Un ejemplo de esto se puede encontrar en el producto htmlgen de Franz Inc, que proporciona una conveniente macro en CL para generar HTML para una página web.

Otro ejemplo es el producto AutoSim de MSC, que usa un programa CL para modelar un automóvil, y después genera programas C optimizados para resolver sistemas especializados de ecuaciones lineales relativas a la dinámica del vehículo.

¡CL también desempeñó un papel crucial en la prevención del potencial desastre del año 2000 analizando y reparando automáticamente millones de líneas de COBOL!

4.7 Multiprocesamiento

El multiprocesamiento permite que la aplicación se lleve en tareas separadas virtualmente simultáneas. Virtualmente, porque en una máquina con un único procesador sólo puede ocurrir físicamente una operación cada vez. Sin embargo, existen situaciones en las que se quiere que el programa parezca que está ejecutando simultaneamente hilos, o procesos, separados. El servicio Web es un buen ejemplo. Como se describió en la Sección 4.9, se puede estar ejecutando la aplicación CL como servidor web. Esto significa que múltiples usuarios, en diferentes máquinas en distintas partes de la compañía o incluso a lo largo del mundo, pueden estar enviando solicitudes a la aplicación en la forma de comandos HTTP de navegador web (cliente). Si la solicitud de un usuario se toma un tiempo especialmente largo para procesarse, probablemente no se quiere que todos los demás usuarios tengan que esperar por una respuesta mientras esa otra solicitud se esté procesando.

El multiprocesamiento encara tales problemas permitiendo a la aplicación CL continuar atendiendo otras solicitudes, mientras "simultáneamente" completa ese otro cómputo largo.

El multiprocesamiento de Allegro se cubre en el documento multiprocessing.htm que va incluido en todas las ediciones de Allegro CL.

4.7.1 Inicio de un proceso en segundo plano

Empezar otro hilo de ejecución en Allegro CL puede ser tan simple como llamar a la función `mp:process-run-function`. Esta función cubre un hilo separado de ejecución, mientras la ejecución del código desde donde se la llamó continúa. Este es un ejemplo simple que se puede ejecutar desde la línea de comandos:

```
(defvar *count* 0)
(mp:process-run-function
  (list :name "contar hasta un millón en segundo plano"
        :priority -10)
   #'(lambda() (dotimes (n 1000000) (setq *count* n))))
```

Esto inicia un hilo llamado "contar hasta un millón en segundo plano" con una prioridad relativa de -10. Observe que después de que se haya llamado a esta función, el indicador de máximo nivel regresa inmediatamente. El bucle de Lectura-Evaluación-Impresión de máximo nivel continúa alegremente, mientras que el nuevo proceso se ejecuta en segundo plano, contando hasta un millón (de hecho, 999999). Mientras el proceso en segundo plano está en ejecución, se puede extraer el valor de la cuenta en cualquier momento:

```
CL-USER(16): *count*
424120

CL-USER(17): *count*
593783

CL-USER(18): *count*
679401

CL-USER(19): *count*
765012
```

Observe que en un procesador veloz se puede necesitar incrementar el valor de este ejemplo para poder extraer muestras antes de que el código se complete.

4.7.2 Control de concurrencia

A veces se quiere prevenir que dos hilos puedan modificar a la vez un valor en la memoria. Para este propósito, se puede usar el bloqueo de procesos. Un bloqueo de procesos es un objeto que puede ser poseído como máximo por un proceso en ejecución cada vez. Usando bloqueos de procesos, uno puede asegurarse de que los múltiples hilos de ejecución interactuarán como se espera.

En nuestro ejemplo contador previo, digamos que se quiere prevenir que otro hilo modifique la variable contabilizadora hasta que el conteo esté completo. Primero se establecería un bloqueo de procesos específico para este propósito:

```
CL-USER(20): (setq count-lock (mp:make-process-lock))
#<MULTIPROCESSING:PROCESS-LOCK  #x5a02fba>
```

Ahora, se ejecuta la función en segundo plano dentro del contexto de un bloqueo que la protege:

```
CL-USER(21): (mp:process-run-function
               (list :name "counting to a million in background"
                     :priority -10)
               #'(lambda()
                   (mp:with-process-lock (count-lock)
                     (dotimes (n 1000000) (setq count n)))))
#<MULTIPROCESSING:PROCESS counting to a million in background  #x5a0332a>
```

Ahora se puede adoptar una política de que cualquier pieza de código que desee modificar el valor del contador primero debe hacerse cargo del `count-lock`. Esto asegurará que nada interferirá en el proceso de conteo. Si se hace lo siguiente mientras el proceso de conteo está en ejecución:

```
CL-USER(22): (mp:with-process-lock (count-lock) (setq count 10))
```

se advertirá que la ejecución quedará bloqueada hasta que el proceso de conteo esté finalizado. Esto es porque la `count-lock` ya está poseída, y la ejecución aguarda a que este bloqueo se libere antes de proceder.

Habitualmente es una buena idea crear los bloqueos de procesos tan ajustados como sea posible, para evitar hacer que los procesos esperen innecesariamente.

Desde el indicador de máximo nivel, se puede monitorizar qué procesos están corriendo en la imagen CL en uso con el comando de máximo nivel :proc command:

```
CL-USER(28): :proc
```

P	Bix	Dis	Sec	dSec	Priority	State	Process Name, Whostate, Arrest
*	7	1	2	2.0	-10	runnable	counting to a million in background
*	1	8	28	0.1	0	runnable	Initial Lisp Listener
*	3	0	0	0.0	0	waiting	Connect to Emacs daemon, waiting for input
*	4	0	0	0.0	0	inactive	Run Bar Process
*	6	0	3	0.0	0	waiting	Editor Server, waiting for input

Véase el fichero multiprocessing.htm para detalles sobre cada columna de este informe, pero esencialmente esto está mostrando el estado de cada proceso existente en esos momentos en el sistema. Este ejemplo muestra que Allegro CL emplea hilos separados para mantener la conexión con el editor de texto Emacs.

Como veremos en la Sección 4.9, el servidor web AllegroServe crea y gestiona automáticamente hilos separados para enfrentarse con conexiones web simultáneas.

4.8 Interfaces a Bases de datos

CL en general, y Allegro CL específicamente, proporciona una miríada de mecanismos para conectarse a bases de datos. La mayoría de estos mecanismos sólo difieren ligeramente en su operación, así que generalmente se puede cambiar entre esos diferentes mecanismos sin mayor trabajo en la aplicación. Los dos que cubriremos aquí se incluyen tanto en la edición Enterprise como en la Platinum de Allegro CL.

4.8.1 Conexión mediante ODBC

ODBC, u Open Database Connectivity, es un medio estándar de conectarse con una base de datos

relacional basada en SQL (Structured Query Language, el lenguaje estándar industrial para Definiciones y Manipulación de Datos para las bases de datos relacionales). Virtualmente todos los sistemas de bases de datos populares pueden ser conectados usando ODBC. En Windows estas facilidades son gratuitas; en Unix pueden serlo o no. En cualquier caso, ODBC es una buena solución a considerar si se quiere conectar nuestra aplicación a varios sistemas de bases de datos relacionales diferentes sin escribir y mantener piezas separadas de código específicas para cada uno de ellos.

ODBC emplea el concepto de *fuentes de datos*, cada una con un nombre. Existen vías dependientes del sistema operativo para configurar fuentes de datos en la máquina para apuntar a las bases de datos físicas reales, que pueden estar hospedadas en la máquina local o en remotas. Asumiendo que tenemos la fuente de datos llamada "login" con una tabla llamada "USER", conectar y consultar a la base de datos puede ser tan simple como lo siguiente:

```
CL-USER(40): (setq handle (dbi:connect :data-source-name "login"))
#<DBI::ODBC-DB "DSN=login;DB=login;...[truncated]"  #x4b2c3b2>

CL-USER(41): (dbi:sql "select * from USER" :db handle)
(("1" "dcooper8" "guessme" "Dave" NIL "Cooper" NIL NIL NIL NIL ...)
 ("7" "awolven" NIL NIL NIL NIL NIL NIL NIL NIL ...))
("GEN_ID" "LOGIN" "PASSWORD" "FIRST_NAME" "MIDDLE_NAME" "LAST_NAME"
 "COMPANY" "ADDRESS_1" "ADDRESS_2" "CITY" ...)
```

Como con la mayoría de las interfaces a bases de datos de CL, la interfaz ODBC devuelve todas las consultas SQL con múltiples valores: el primer (y principal) valor de retorno es una lista de listas, correspondiente a las filas en la consulta devuelta. El segundo valor de retorno es una lista de nombres de campos, correspondiente a los valores en cada una de las primeras listas.

Debido a que ODBC por definición debe mantener la portabilidad entre sistemas de bases de datos, es en general incapaz de aprovechar todas las ventajas de las características específicas de cada base de datos y optimizaciones en el rendimiento.

4.8.2 Base de datos MySQL

MySQL es una popular base de datos relacional que está disponible tanto con soporte comercial como en forma libre. Aunque se puede conectar a una base de datos MySQL usando ODBC, Allegro CL también provee una conexión directa a MySQL, que realiza una conexión más rápida y más fácil de configurar que la ODBC. En el nivel de aplicación, su uso es suficientemente simple tal que una aplicación que emplee la interfaz directa a MySQL podría ser convertida para su uso con ODBC sin mayor retrabajo, si eso se hiciera necesario. Por tanto, si se sabe que la aplicación Allegro CL va a usar MySQL en el futuro inmediato, recomendamos usar la Interfaz Directa MySQL en vez de la ODBC.

Este es el mismo ejemplo de arriba, empleando la interfaz directa MySQL en vez de la ODBC:

```
CL-USER(50): (setq handle (dbi.mysql:connect :user "dcooper8"
                                             :password "guessme"
                                             :database "login"))
#<DBI.MYSQL:MYSQL connected to localhost/3306  #x5bd114a>

CL-USER(51): (dbi.mysql:sql "select * from USER" :db handle)
((1 "dcooper8" "bhaga<<" "Dave" #:|null| "Cooper" #:|null| ...)
 (7 "awolven" #:|null| #:|null| #:|null| #:|null| #:|null| ...))
("GEN_ID" "LOGIN" "PASSWORD" "FIRST_NAME" "MIDDLE_NAME"
```

```
"LAST_NAME" "COMPANY" "ADDRESS_1" "ADDRESS_2" "CITY" ...)
```

Una cuantas diferencias a advertir con ODBC:

- Las funciones están en el paquete dbi.mysql en vez de en el paquete dbi.
- Ya que no hay necesidad de configurar fuentes de datos separadas para MySQL Direct, conectamos con la instancia de MySQL en la máquina local directamente especificando un usuario, una contraseña y un nombre de base de datos.
- Los valores NULL en el retorno de la consulta se representan con el símbolo especial #:|null| en vez de con NIL como en ODBC.

4.9 La World Wide Web

4.9.1 Los servidores y la generación de HTML

La naturaleza dinámica y multi-hilo de AllegroCL hace que sea apto de manera natural para alimentar poderosas aplicaciones para servidores web. Franz proporciona un popular servidor web de código abierto llamado AllegroServe que está diseñado justo para este propósito.

Con una instalación estándar de Allegro CL 6.2 se puede cargar AllegroServe y acceder a sus símbolos exportados como sigue:

```
(require :aserve)
(use-package :net.aserve)
(use-package :net.html.generator)
(use-package :net.aserve.client)
```

Con AllegroServe, se pueden mapear direcciones de sitios web a funciones reales que se ejecutarán en respuesta a una solicitud. El siguiente es un ejemplo simple:

```
(defun hello (req ent)
  (with-http-response (req ent)
    (with-http-body (req ent)
      (html "Hello World"))))

(net.aserve:publish :path "/hello.html"
:function #'hello)
```

Ahora, siempre que un visitante web vaya a la URI /hello.html de nuestro servidor, recibirá en respuesta una página con el sencillo texto "Hello World". Serían, por supuesto, más interesantes ejes más dinámicos, y presentamos algunos en el capítulo siguiente.

Observe el uso del operador html en el ejemplo previo. Es una macro del paquete HTMLgen que va con Allegroserve. HTMLgen permite la generación conveniente de HTML bien estructurado dinámico a partir de un formato fuente de estilo lisp. Véase la documentación de AllegroServe y de HTMLgen en http://opensource.franz.com/aserve/index.html para los detalles sobre ambos.

4.9.1 Los servidores y la generación de HTML

La naturaleza dinámica y multi-hilo de AllegroCL hace que sea apto de manera natural para alimentar poderosas aplicaciones para servidores web. Franz proporciona un popular servidor web de código abierto

llamado AllegroServe que está diseñado justo para este propósito.

Con una instalación estándar de Allegro CL 6.2 se puede cargar AllegroServe y acceder a sus símbolos exportados como sigue:

```
(require :aserve)
(use-package :net.aserve)
(use-package :net.html.generator)
(use-package :net.aserve.client)
```

Con AllegroServe, se pueden mapear direcciones de sitios web a funciones reales que se ejecutarán en respuesta a una solicitud. El siguiente es un ejemplo simple:

```
(defun hello (req ent)
  (with-http-response (req ent)
    (with-http-body (req ent)
      (html "Hello World"))))

(net.aserve:publish :path "/hello.html"
:function #'hello)
```

Ahora, siempre que un visitante web vaya a la URI /hello.html de nuestro servidor, recibirá en respuesta una página con el sencillo texto "Hello World". Serían, por supuesto, más interesantes ejes más dinámicos, y presentamos algunos en el capítulo siguiente.

Observe el uso del operador html en el ejemplo previo. Es una macro del paquete HTMLgen que va con Allegroserve. HTMLgen permite la generación conveniente de HTML bien estructurado dinámico a partir de un formato fuente de estilo lisp. Véase la documentación de AllegroServe y de HTMLgen en http://opensource.franz.com/aserve/index.html para los detalles sobre ambos.

4.10 Expresiones Regulares

Muchas aplicaciones tienen que trabajar con cadenas de caracteres. Hacer esto de una manera eficiente puede ser la clave de la eficiencia y rendimiento de toda la aplicación. Allegro CL proporciona un módulo de Expresiones Regulares para este propósito. El módulo Regular Expression permite buscar, reemplazar y manipular patrones en las cadenas de caracteres de una manera eficiente y optimizada.

Como ejemplo simple, reemplacemos un carácter por otro en una cadena. Asumimos que tenemos algunos nombres de campos como variables en CL, que pueden contener legalmente caracteres de guiones (‘-’). Digamos que queremos usar estos nombres como la base para los nombres de campos en una base de datos SQL, donde los guiones no se permiten como parte de un nombre. Reemplazaremos todos los guiones con subrayados:

```
(defun replace-dashes (word)
  (excl:replace-regexp word "-" "_"))

CL-USER(59): (replace-dashes "top-o-the-morning")
"top_o_the_morning"
```

En el documento regexp.htm, que se incluye en todas las ediciones de Allegro CL, existe una referencia completa de las Expresiones Regulares.

4.11 Correo electrónico

El correo electrónico fue la primera "aplicación matadora" de Internet, y se puede argumentar que continúa en ese papel hasta hoy. A despecho de las amenazas de los virus y del correo no deseado, mucha gente en el mundo moderno aún depende del correo para sus comunicaciones diarias. CL proporciona un excelente entorno para el procesado automático de correo. No es coincidencia que en el siguiente capítulo presentemos un ejemplo de uso de CL para realizar clasificación de texto en el correo, un medio efectivo para filtrar correo no querido ni solicitado.

4.11.1 Envío de correo

Franz proporciona un conveniente paquete de código abierto para enviar mensajes a un servidor SMTP (Simple Mail Transfer Protocol).

Con una instalación estándar de Allegro CL 6.2 se puede cargar este paquete y acceder sus símbolos exportados como sigue:

```
(require :smtp)
(use-package :net.post-office)
```

Este paquete proporciona la función send-letter. Esta función toma el nombre de un servidor SMTP junto con el destinatario, el texto de un correo, y otros argumentos optativos, y envía el correo al anfitrión SMTP especificado. Este es un ejemplo:

```
(send-letter "smtp.myisp.com" "dcooper@genworks.com" "joe@bigcompany.com"
 "See you at the game tonight." :subject "Tonight's Game"
```

Esto es todo lo que le lleva a la aplicación enviar un correo.

4.11.2 Recepción de correo

Los dos estándares principales de Internet para recibir correo son POP e IMAP. Aunque IMAP es más nuevo y con más características que POP, muchos ISP aún sólo le dan soporte a POP, así que cubriremos la interfaz POP aquí. Franz proporciona un único fichero, imap.cl que contiene interfaces de cliente tanto para POP como para IMAP.

Con una instalación estándar Allegro CL 6.se puede cargar este paquete y acceder a sus símbolos exportados como sigue:

```
(require :imap)
(use-package :net.post-office)
```

Este es un ejemplo de uso de la interfaz de cliente de POP. Primero se debe crear un manejador de conexiones para el servidor POP:

```
CL-USER(4): (setq mailbox (net.post-office:make-pop-connection
"pop.myisp.com"
                                                    :user "joe"
                                                    :password
"guessme"))
#<NET.POST-OFFICE::POP-MAILBOX  #x5dcc33a>
```

Ahora podemos obtener una cuenta de los mensajes en el buzón de entrada:

```
GDL-USER(5): (net.post-office:mailbox-message-count mailbox)
1
```

y capturar el mensaje:

```
GDL-USER(9): (net.post-office:fetch-letter mailbox 1)
"Content-type: text/plain; charset=US-ASCII
MIME-Version: 1.0 ...
..."
```

Finalmente, podemos eliminar este mensaje y cerrar la conexión:

```
GDL-USER(10): (net.post-office:delete-letter mailbox 1)
NIL

GDL-USER(12): (net.post-office:close-connection mailbox)
T
```

Véase la documentación en http://opensource.franz.com/postoffice/index.html para una cobertura completa de las facilidades POP e IMAP.

En el siguiente capítulo combinaremos buena parte de lo que hemos aprendido en este capítulo para poner juntos un programa basado en web para navegar y clasificar correo. Debido a que hacemos uso de bibliotecas predefinidas y de interfaces como las que hemos descrito en este capítulo, podemos crear un aplicación multi-usuario en funcionamiento con menos de 500 líneas de código.

Capítulo 5

Cliente de correo Squeakymail

Este capítulo presentará una aplicación de muestra en CL, llamada "Squeakymail", para filtrar y navegar por el correo electrónico. Squeakymail demuestra el uso de muchas de las interfaces de Allegro CLASE y las características especiales cubierts en el capítulo anterior, en una aplicación por debajo de las 500 líneas.

Por favor, adviértase que Squeakymail es una aplicación de código abierto y que aún se encuentra en evolución, así que el lector debería obtener el código último si desea trabajar con él. Contáctese con Genworks en info@genworks.com para tener información acerca de cómo obtener el código base del Squeakymail más actual.

5.1 Vista general de Squeakymail

Squeakymail es una aplicación multi-usuario, para servidores, que realiza dos tareas principales:

1. Coge el correo de cada usuario de una o más cuents POP remotas (Post Office Protocol), lo analiza, le da un valor en su probabilidad de que sea "spam" (correo no deseado), y lo almacena en una cola local para su clasificación final por el usuario. Esta captura de mensajes es perpetua, como un hilo en segundo plano ejecutándose en la imagen CL.
2. Proporciona una interfaz web simple para que cada usuario pueda ver las cabeceras y cuerpos del correo entrante, y confirma o sobre-escribe la prevaloración de cada mensaje. Una vez clasificado, los fragmentos individuales (palabras) de cada mensaje se añaden a las tablas de probabilidad de spam ya memorizadas, tal que la exactitud de la valoración pueda mejorar con el tiempo.

Existen algunas extensiones obvias que se le podrían hacer a Squeakymail, por ejemplo proporcionarle una opción de auto-clasificación de mensajes cuya probabilidad de ser spam exceda cierto umbral, evitando así que el usuario ni siquiera tenga que ver esos mensajes.

5.2 Captura y Valoración (Puntuación)

5.2.1 Captura

Una parte de nuestra aplicación se ejecutará en un hilo en segundo plano, consultando periódicamente las fuentes de correo POP3 remotas del usuario para saber si hay mensajes nuevos. Específicamente, el hilo de captura de correo realizará lo siguiente para cada cuenta POP3 remota de cada usuario local:

```
    (defun fetch-mails ()
      (mp:process-run-function "fetching mail in the background"
        #'(lambda()
            (do ()()
              (let ((start (get-universal-time)))
                (with-db-rows ((login) :db *mysql-logins* :table "USER")
                  (fetch-mail login))
                (sleep (max 0 (- *mail-fetching-interval*
```

```
                                    (- (get-universal-time) start)))))))))
```

Figura 5.1: Inicio de un hilo en segundo plano

1. Coge un bloqueo de procesos para el usuario local en curso, para prevenir que otros hilos modifiquen simultáneamente los datos del usuario (en ficheros o en la memoria).
2. Crea una conexión POP3 al servidor POP3 especificado.
3. Abre el fichero de Correo Entrante local del usuario para añadir correo.
4. Captura el Contador de Mensajes que haya (e.d., número de procesos) del servidor POP
5. Itera a través de cada mensaje numerado, haciendo lo siguiente:
    a. Captura el contenido completo del mensaje y lo pone en la memoria.
    b. Analiza el mensaje y lo descompone en sus elementos individuales de cabecera y de cuerpo.
    c. Computa los fragmentos individuales de cada cabecera y del cuerpo.
    d. Computa una "puntuación" de probabilidad de spam probability basándose en dichos fragmentos.
    e. Escribe una entrada en el fichero local de Correo Entrante del usuario, que contendrá las cabeceras, el cuerpo, la lista de fragmentos, y la puntuación de probabilidad de spam
    f. Finalmente marca el mensaje para su eliminación remota en el servidor POP3.

Las Figuras 5.1 y 5.2 contienen el código básico de esta parte del programa. El código de la Figura 5.1 inicia el hilo en segundo plano con una llamada a mp:process-run-function, dándole un nombre al proceso y una expresión lambda (objeto de función anónima) a ser ejecutada por el proceso. Esa función entra en una iteración infinita (con do). En cada iteración de este bucle infinito escanea la tabla de cuentas locales de usuarios empleando la macro with-db-rows del paquete dbi.mysql. Para cada línea se pone la variable léxica login en el nombre de login del usuario respectivo, y la función fetch-mail de la Figura 5.2 se invoca para ese usuario. Finalmente, el proceso en modo Sleep hasta que se produzca el siguiente intervalo planificado, en cuyo momento se volverá a despertar y realizará otra iteración de su bucle infinito.

El código de la Figura 5.2 captura y puntúa el correo de un usuario local individual. Primero establece un bloqueo de proceso para ese usuario en particular para el contexto del resto del cuerpo de la función. El objeto de bloqueo en sí proviene de una llamada a la función user-process-lock (Figura 5.3), que toma de una tabla hash un bloqueo específico para el usuario local, o crea un

```
(defun fetch-mail (user)
 (mp:with-process-lock ((user-process-lock user))
   (with-db-rows ((host login password) :table "POP_ACCOUNT"
                                        :db *mysql-local*
                                        :where (=text local_login user))
     (format t "Fetching mail from ~a for local user ~a~%" host user)
     (with-open-pop-box (mb host login password)
       (with-open-file
           (out (string-append (format nil "/var/mail/incoming/~a" user))
            :direction :output :if-exists :append :if-does-not-exist :create)
         (let ((message-count (mailbox-message-count mb)))
           (dotimes (n message-count)
             (let* ((message (fetch-letter mb (1+ n))))
               (multiple-value-bind (headers body)
                   (parse-mail-header message)
```

```
                    (let ((tokens (tokenize headers body)))
                      (let ((*package* *token-package*))
                        (multiple-value-bind (score scored-tokens)
                            (score-tokens tokens user)
                          (print (list :headers headers
                                       :body body
                                       :tokens tokens
                                       :score score
                                       :scored-tokens scored-tokens) out)))))))
              (delete-letter mb (1+ n))))))))
```

Figura 5.2: Captura y valoración del correo remoto de un usuario local

```
(defun user-process-lock (user)
  (or (gethash user *mailbox-locks-hash*)
      (setf (gethash user *mailbox-locks-hash*)
        (mp:make-process-lock :name (format nil "fetching mail for ~a" user)))))
```

Figura 5.3: Recuperación o creación de un bloqueo de proceso

```
(defmacro with-open-pop-box
    ((handle server user password &optional (timeout 10)) &body body)
  `(let ((,handle (make-pop-connection ,server :user ,user :password ,password
                                       :timeout ,timeout)))
     (unwind-protect (progn ,@body)
       (when ,handle (close-connection ,handle)))))
```

Figura 5.4: Una macro para abrir y asegurar el cierre limpio de las conexiones POP3

bloqueo nuevo si aún no existiera ninguno. La ejecución del resto del cuerpo de la función dentro de este proceso asegurará que ningún otro hilo de ejecución interfiera con los datos de este usuario durante la operación de captura. Por ejemplo, otra parte de nuestra aplicación proporcionará una interfaz web a los usuarios para que éstos naveguen y clasifiquen su correo interactivamente. Esta clasificación también modifica el fichero de Bandeja de Entrada, habitualmente mediante la eliminación de uno o más sus mensajes. Al ejecutar nuestra captura de correo dentro de un bloqueo específico del usuario, eliminamos la posibilidad de que estas distintas operaciones de actualización ocurran simultáneamente. Porque de hacerlo del último modo con probabilidad corrompería el fichero de Correo Entrante, con la pérdida de mensajes como resultado, o aún peor.

Este bloqueo de procesos implica, sin embargo, que los usuarios interactivos pueden experimentar una ligera demora cuando intenten visitar su página web de Clasificación de Correo, si se les ocurre hacerlo en el momento en que el ciclo de captura de sus correos esté en marcha. Esto se debe a que el proceso interactivo también requiere ese bloqueo específico del usuario, y se pondrá en modo de "espera" hasta que el proceso de captura se complete y libere el bloqueo.

La función fetch-mail posteriormente imprime un mensaje de estado en la consola indicando el usuario local y el anfitrión remoto de la operación en curso.

Después entra en el código escrito por la macro with-open-pop-box (Figura 5.4), que abre una conexión a la cuenta POP3 especificada, le asigna el nombre mb a esta conexión y finalmente garantiza que la conexión se cierre cuando el código que hay dentro de with-open-pop-box se complete o resulte en cualquier condición de error. Entonces la función fetch-mail entra en otro contexto, with-open-file, que nos da un flujo llamado out para la escritura del fichero de Bandeja de Entrada del usuario.

Entonces le consultamos a la conexión POP mb el número de mensajes, y asocia este resultado a la variable *message-count*. Esto nos dice cuántas veces tenemos que capturar mensajes, lo que hacemos dentro del código de una dolist, empleando la función fetch-letter del paquete net.post-office. Observe que añadimos uno (1) a la variable *n* de dotimes cuando se captura la carta, ya que los números de mensajes POP3 comienzan con 1, en vez de con 0 como en el caso de las variables iteradoras de dolist.

Entonces analizamos el mensaje y lo descomponemos en cabeceras y cuerpo, empleando multiple-value-bind, ya que la función parse-mail-header devuelve las cabeceras y el cuerpo como dos valores de retorno. Lo siguiente es fragmentar el mensaje (descrito en la Sección 5.2.1), computar una puntuación basada en los fragmentos (Sección 5.2.2) y escribir toda la información pertinente en el fichero de Correo Entrante del usuario en la conveniente forma de una plista.

Finalmente, marcamos el mensaje para su eliminación en el servidor POP remoto, con la función delete-letter, también del paquete :net.post-office.

```
(defun tokenize (headers body)
  (setq body (excl:replace-regexp
              (excl:replace-regexp body "<!-.[^>]*->" "") "<![^>]*>" ""))
  (append (tokenize-headers headers)
          (mapcan #'(lambda(item) (if (stringp item)
                                      (tokenize-string item)
                                    (list item)))
                  (flatten (parse-html body)))))


(defun tokenize-headers (headers)
  (mapcan #'(lambda(cons)
              (let ((header (first cons))
                    (string (rest cons)))
                (let ((strings (excl:split-regexp "\b+" string)))
                  (mapcar #'(lambda(string)
                              (intern (string-append header "*" string)
                                      *token-package*)) strings)))) headers))


(defun tokenize-string (string)
  (let ((words (excl:split-regexp "\b+" string)) result)
    (dolist (word words (nreverse result))
      (when (and (<= (length word) *max-token-length*)
                 (tokenizable? word))
        (push (intern word *token-package*) result)))))
```

Figura 5.5: Fragmentación utilizando expresiones regulares

5.2.2 Segmentación

La segmentación, el proceso de repartir el texto en palabras o unidades individuales, está en el corazón de nuestra estrategia de filtrado de spam. Nuestra versión actual de la segmentación hace un uso extenso de la API de Expresiones regulares de Allegro CL, un conjunto de funciones para manipular texto. Las tres funciones listadas en la Figure 5.5 resumen nuestro actual sistema de segmentación. La primera función, tokenize, usa primero excl:replace-regexp para remover todos los comentarios HTML y XML del cuerpo (la inserción de comentarios HTML aleatorios es un truco común usado por los espammers para confundir a los filtros). La función devuelve entonces la lista de las cabeceras segmentadas añadida a la lista de las palabras segmentadas del cuerpo del mensaje. Para producir la lista de las palabras segmentadas, primero

usamos el analizador HTML de Franz para retirar las etiquetas y cadenas HTML individuales (esto no tendrá efecto si el mensaje no comienza con HTML). Debido a que el analizador devuelve una lista anidada de etiquetas, empleamos flatten para convertirla en una lista plana. Después pasamos cada cadena a la función tokenize-string.

tokenize-headers acepta una lista de cabeceras, cada una de las cuales es una lista de pares con puntos cuyo first es el nombre de la cabecera, y cuyo rest es una cadena con el valor de la cabecera. Separamos el valor de la cabecera en palabras individuales empleando excl:split-regexp para hacer el corte en el espacio en blanco. Por cada palabra así obtenida, le anteponemos el nombre de la cabecera. Esto distingue los segmentos de la cabecera de los segmentos del cuerpo, ya que un segmento dado en una línea de 'Asunto', por ejemplo, puede tener una probabilidad muy diferente de spam con respecto al mismo segmento en el cuerpo del mensaje. Así, un mensaje:

```
Subject: Yo te quiero
```

generará los fragmentos 'Subject*Yo', 'Subject*te', y 'Subject*quiero'.

La función tokenize-string toma una cadena y esencialmente la divide en los espacios en blanco para producir palabras individuales. Rechazamos las palabras más largas que **max-token-length**, y comprobamos que la palabra es "fragmentable", lo que quiere decir que no confundirá al lector CL cuando lea el segmento como símbolo. Actualmente la única secuencia de caracteres que confundirá al lector CL es una secuencia de más de un punto, v.g. '...', así que también quedará rechazada. Todas las cadenas quedan convertidas en símbolos e internalizadas en un determinado paquete CL que hemos creado para este propósito. La conversión de las cadenas en símbolos acelera mucho el rastreo posterior de los segmentos en las tablas hash.

5.2.3 Puntuación (Valoración)

Una vez que tenemos el mensaje segmentado, los cómputos para valorarlo son bastante elementales. Llamamos a la función score-tokens con la lista de los trocitos extraidos del mensaje. Esta función emplea la técnica de Paul Graham[1] "Plan for Spam" para valorar los mensajes. Asume la existencia de una tabla hash que mapea los fragmentos en sus probabilidades individuales de spam. En la Section 5.3 veremos cómo se crean y mantienen estas tablas hash, para cada usuario.

La función de valoración asocia primero ht a la tabla de probabilidades específica del usuario empleando la función restore-probabilities, que obtiene la tabla de probabilidades de la memoria, la restaura de un fichero guardado, o crea una nueva si aún no existiera. Después rastrea cada fragmento en la tabla de

```
(defun score-tokens (tokens user)
  (let ((ht (restore-probabilities user)))
    (let* ((scored-tokens
            (sort (mapcar #'(lambda(token)
                              (list token (or (gethash token ht)
                                              *unknown-probability*)))
                          tokens)
                  #'> :key #'(lambda(token) (abs (- (second token) 0.5)))))
           (probs (mapcar #'second
                          (subseq scored-tokens 0
                                  (min *number-of-relevant-tokens*
                                       (length scored-tokens))))))
```

```
          (values
            (let ((prod (apply #'* probs)))
              (/ prod (+ prod (apply #'* (mapcar #'(lambda (x) (- 1 x)) probs)))))
            scored-tokens))))
```

Figura 5.6: Valoración de un mensaje basada en sus fragmentos

probabilidades, creando una lista que empareja cada fragmento con su probabilidad de spam correspondiente. Después ordena este resultado basándose en la distancia de la probabilidad con respecto al 0,5 neutral.

Los valores de probabilidad reales, *probs*, se computan por consiguiente tomando los valores superiores **number-of-relevant-tokens** (lo predeterminado, 15) de esta lista. La probabilidad combinada se computa empleando la fórmula de Bayes como se describe en el artículo de Paul Graham "Plan for Spam".

Una mejora posible a la valoración, mencionada por Paul Graham, sería no permitir fragmentos duplicados (o limitar su número) al computar los valores superiores.

5.3 Navegación y Clasificación

El "cerebro izquierdo" de nuestra aplicación le presentará una interfaz a los usuarios, permitiéndoles "probar" el clasificador del correo. La interfaz que cubrimos aquí está hasta cierto punto en los huesos; también hemos preparado una interfaz mucho más rica empleando el sistema GDL/GWL de Genworks. El código de esta se incluye en el Apéndice A.

A la que presentamos aquí le faltan algunas características clave. La más obvia es un mecanismo de seguridad para prevenir que alguien acceda al correo de otro usuario (GWL proporciona un mecanismo de seguridad prefabricado para la identificación con contraseña; tal sistema se deja como ejercicio a aquellos que desean implementar algo al estilo de Squeakymail en Common Lisp y AllegroServe).

Nuestra interfaz presentará una página web creada dinámicamente que mostrará las cabeceras del correo entrante del usuario, junto con la valoración de cada uno y el estado resultante de spam/no-spam. Empleando un botón de selección para cada correo, el usuario podrá corregir cualquier clasificación mal hecha.

```
(defun classify (req ent)
   (let ((query (request-query req)))
     (let ((keys (mapcar #'first query))
           (values (mapcar #'rest query)))
       (let ((user (rest (assoc "user" query :test #'string-equal)))
             (spam-indices (let ((ht (make-hash-table :values nil)))
                             (mapc #'(lambda(key value)
                                       (when (and (search "spamp-" key) (string-equal
                                         (setf (gethash (read-from-string (subseq key
              (nonspam-indices (let ((ht (make-hash-table :values nil)))
                                 (mapc #'(lambda(key value)
                                           (when (and (search "spamp-" key) (string-e
                                             (setf (gethash (read-from-string (subseq
         (let ((messages (classify-and-write-files user spam-indices nonspam-indices))
           (mp:process-run-function (list :name (format nil "Squeakymail: Recomputing
                                          :priority -5 :quantum 1) #'(lambda()(sleep 5
           (with-http-response (req ent) (with-http-body (req ent) (classify-form messa
```

```
(publish :path "/classify" :function #'classify)
```

Figura 5.7: Función principal para producir la página web de clasificación

La función principal classify se muestra en la Figura 5.7, y también queda publicada (mapeada en una URL) en la Figura 5.7. La función classify, como todas las funciones de respuesta http de Allegroserve, toma los argumentos req y ent. El argumento req proviene del navegador, y contiene valores de cualquier formulario html enviado. Estos valores están asociados a la variable *query*. Cuando la función classify se llama por primera vez, no contiene datos de formulario. En invocaciones subsiguientes, sin embargo, puede contener datos de formularios que indiquen qué mensajes son spam y cuáles no. Más adelante vermos cómo crear el formulario para enviar estos valores.

Una vez que tengamos las claves (nombres de campos) y valores del formulario de la información de consulta, computamos dos tablas hashs: spam-indices y nonspam-indices. Estas contienen los índices numéricos que representan la selección del usuario de botones de selección sobre cada mensaje. Lo siguiente es llamar a la función classify-and-write-files con los spam-indices y nonspam-indices. Explicaremos pronto las interioridades de esta función; basicamente mueve mensajes del fichero de correo entrante del usuario, y devuelve cualesquier mensajes restantes (sin clasificar) junto con los mensajes nuevos que hayan llegado mientras tanto.

Entonces creamos un fichero temporal como incoming-buffer-pathname para que contenga los mensajes sin clasificar, spams-pathname para que contenga el cuerpo del nuestro spam, nonspams-pathname para que contenga el cuerpo de nuestro no-spam, y popqueue para añadirlo al fichero existente de correo del usuario.

Ahora, empleando una expresión cond, distribuimos cada mensaje en el fichero apropiado.

La función classify entonces crea un proceso hijo en segundo plano de baja prioridad para recomputar las probabilidades de fragmentos de spam.

Finalmente, genera la página web con los mensajes y el formulario de clasificación, dentro de unas with-http-response y with-http-body, llamando a la función classify-form.

La Figura 5.8 contiene el código de la función classify-and-write-files. Esta función toma una lista de spam-indices y nonspam-indices, y mueve apropiadamente los mensajes. Todo esto se hace dentro del contexto de bloqueo específico del usuario, para prevenir que las solicitudes de dos usuarios reorganicen los ficheros simultáneamente, lo que podría conducir a ficheros corruptos.

La función classify-and-write-files lo primero que hace es poblar una lista de mensajes, asociada a la variable *messages* de let mediante su apertura y lectura desde el fichero de bandeja de entrada del usuario. Esto se hace con la función read-incoming-messages, listada en la Figura 5.9.

La página web real proviene de la función classify-form, listada en la Figura 5.10. Este formulario se crea con la macro html, que nos permite escribir una plantilla html "lispificada"

```
(defun classify-and-write-files (user spam-indices nonspam-indices)
  (let (remaining-messages)
    (mp:with-process-lock ((user-process-lock user))
      (let (*read-eval*)
        (let ((messages (read-incoming-messages user)))
```

```
            (when messages
              (let ((incoming-buffer-pathname (sys:make-temp-file-name))(count -1)
                    (spams-pathname
                     (progn (ensure-directories-exist (string-append "/var/mail/" user
                            (string-append "/var/mail/" user ".data/corpi/spams")))
                    (nonspams-pathname (string-append "/var/mail/" user ".data/corpi/no
               (with-open-file (spams spams-pathname
                                :direction :output :if-exists :append :if-does-not-exis
                 (with-open-file (nonspams nonspams-pathname
                                  :direction :output :if-exists :append :if-does-not-ex
                   (with-open-file (popqueue (string-append "/var/mail/" user)
                                    :direction :output :if-exists :append :if-does-not-
                     (with-open-file (incoming-buffer incoming-buffer-pathname
                                      :direction :output :if-exists :supersede :if-does
                       (dolist (message messages)
                         (let ((index (getf message :index)))
                           (cond ((gethash index spam-indices)
                                  (record-tokens (mapcar #'(lambda(token)(intern token
                                                             (getf message :tokens)) :spam
                                  (print message spams))
                                 ((gethash index nonspam-indices)
                                  (record-tokens (mapcar #'(lambda(token) (intern token
                                                             (getf message :tokens)) :nonsp
                                  (print message nonspams)
                                  (pop-format message popqueue))
                                 (t
                                  (setf (getf message :index) (incf count))
                                  (push message remaining-messages)
                                  (print message incoming-buffer)))))))))
               (sys:copy-file incoming-buffer-pathname (string-append "/var/mail/incomin
                              :overwrite t)
               (delete-file incoming-buffer-pathname))))))
        (nreverse remaining-messages)))
```

Figura 5.8: Función para distribuir los mensajes en los ficheros apropiados

con código Lisp incrustado que será evaluado. De esta manera, creamos la página html incluyendo un formulario cuya acción (URL de respuesta) invoca a la misma función classify que se describió arriba. Entonces agrupamos los mensajes, basándonos en su valoración pre-computada, en una lista de los spams y no-spams esperados. Finalmente presentamos las cabeceras de estos mensajes al usuario, cada una con su botón de selección pre-seleccionado apropiadamente si tenemos suficiente nivel de certeza de la probabilidad spam/no-spam del mensaje.

Cuando el usuario presiona en el botón <Enviar>, el formulario se envía y responderá la misma función classify, distribuyendo los mensajes en los ficheros apropiados. Este proceso se repite indefinidamente en tanto el usuario continúe presionando el botón <Enviar>.

```
(defun read-incoming-messages (user)
  (fetch-mail user)
  (let (result (*read-eval* nil) (count -1)
        (incoming-pathname (string-append "/var/mail/incoming/" user)))
    (when (probe-file incoming-pathname)
      (with-open-file (in incoming-pathname)
        (do ((message (read in nil nil) (read in nil nil)))
            ((null message) (nreverse result))
          (setf (getf message :index) (incf count))
          (push message result))))))
```

Figura 5.9: Lectura de mensaje entrantes

```
(defun classify-form (messages user)
  (html
   (:html
    (:head (:title "Classify Incoming Mail for " (:princ user)))
    (:body (:h2 (:center "Classify Incoming Mail for " (:princ user)))
           ((:form :action "/classify" :method :post)
            (:p ((:input :type :submit :value "Classify!")))
            (:center
             ((:input :type :hidden :name :user :value user))
             ((:table :bgcolor :black :cellpadding 1 :cellspacing 1)
              (let ((grouped-messages (list nil nil)))
                (mapc #'(lambda(message)
                          (if (> (getf message :score) 0.8)
                              (push message (second grouped-messages))
                            (push message (first grouped-messages)))) messages)
                (setq grouped-messages (append (nreverse (first grouped-messages))
                                               (nreverse (second grouped-messages))))
                (when grouped-messages
                  (html ((:tr :bgcolor :yellow) (:th "Bad") (:th "Good") (:th "From"
                (dolist (message grouped-messages)
                  (let ((headers (getf message :headers))
                        (score (getf message :score)))
                    (let ((from (rest (assoc "from" headers :test #'string-equal)))
                          (subject (rest (assoc "subject" headers :test #'string-equa
                          (date (rest (assoc "date" headers :test #'string-equal))))
                      (html ((:tr :bgcolor (if (> score 0.8) :pink "#aaffaa"))
                             ((:td :bgcolor :red)
                              ((:input :type :radio :name (format nil "spamp-~a" (ge
                                       :value :spam :if* (> score 0.8) :checked :chec
                             ((:td :bgcolor :green)
                              ((:input :type :radio :name (format nil "spamp-~a" (ge
                                       :value :non-spam :if* (< score 0.5) :checked
                             (:td (:small (:princ (string-append (subseq subject 0 (
                                                                 (if (> (length subje
                             (:td (:small (:princ (short-date date))))
                             (:td (:princ score)))))))))))
           (:p ((:input :type :submit :value "Classify!")))))))))
```

Figura 5.10: Generación de la página web

Apéndice A Squeakymail con GDL/GWL de Genworks

Genworks International produce y vende un sistema de Base de Conocimientos basado en Allegro CL llamado General-purpose Declarative Language. Adjunto a GDL está también GWL, Generative Web Language, un servidor de aplicaciones web integrado que se ejecuta en conjunción con GDL y AllegroServe.

GDL y GWL también pueden generar y manipular entidades geométricas en 3D, proporcionando poder adicional en un conjunto de herramientas para crear aplicaciones de negocios e ingeniería.

Para una introducción a GDL/GWL, véase el GDL Tutorial, disponible en formato PDF en: http://www.genworks.com/papers/gdl-tutorial.pdf

En este capítulo presentamos una interfaz alternativa de clasificación/exploración para Squeakymail, basada en GDL/GWL. Esta versión de la interfaz contiene algunas características significativas que no se encuentran en el Capítulo 5:

- Identificación segura de usuario, proporcionada por un módulo GWL separado.
- Habilidad de previsualizar los cuerpos de los mensajes en una página enlazada.
- Interfaces convenientes para que el usuario añada, actualice y elimine cuentas POP remotas y fitros explícitos de mensajes.
- Facilidad para "enchufar" un acceso de usuario a Squeakymail dentro de un marco de aplicación web más amplio, consistente en, por ejemplo, aplicaciones de calendario y gestión y planificación de tareas personales.

A.1 Página de inicio de máximo nivel de Squeakymail

El código de la Figura A.1 muestra la definición de objeto correspondiente a la página de inicio de la aplicación Squeakymail. Véase la fila de cliente [customer-row] que se pasa como una ranura de entrada. Esta fila de cliente es un registro de base de datos, proporcionado por un módulo GWL separado, que contiene información acerca del usuario identificado.

Los objetos hijos classify, pop-accounts y filters se presentan como hiper-vínculos a páginas específicas dentro de la página principal de Squeakymail. classify es una página que le permite al usuario explorar y clasificar el correo. pop-accounts le permite al usuario gestionar cuentas POP remotas. Finalmente, filters le permite al usuario administrar filtros explícitos, que le permite a los mensajes puentear el mecanismo estándar de filtrado Baysiano.

La vista definida en la Figure A.2 define una función de salida main-sheet, que genera el HTML para la página.

A.2 Página para la navegación y clasificación

La Figura A.3 construye una secuencia de los mensajes en curso, cada uno de ellos con la posibilidad de ser mostrados en su propia sub-página, y la vista definida en A.4 define una función de salida main-sheet que procesa el correo entrante, después llama a la función de salida classify-form para que emita el HTML del formulario de clasificación.

El objeto message definido en la Figura A.5 es simplemente un mensaje que toma su entrada de from, subject, to, body, score y scored-tokens. Cada mensaje puede potencialmente ser procesado en HTML empleando la función de salida main-sheet definida en la vista de la Figura A.6.

```
(define-object assembly (base-html-sheet)
  :input-slots
  (customer-row)

  :computed-slots
  ((local-login (the customer-row login)))

  :objects
  ((classify :type 'classify
             :strings-for-display "Check and Classify"
             :pass-down (local-login))

   (pop-accounts :type 'html-sql-table
                 :sql-server *local-connection*
                 :table-name 'pop-account
                 :fixed-fields (list :local-login (the customer-row login))
                 :strings-for-display "Pop Accounts"
                 :columns-and-names
                 `(("host" "Mail Server")
                   ("login" "Username")
                   ("password" "Password")))

   (filters :type 'html-sql-table
            :sql-server *local-connection*
            :table-name 'filters
            :strings-for-display "Filters"
            :fixed-fields (list :local-login (the customer-row login))
            :columns-and-names
            `(("source" "Source"
               :choices (:explicit ("Subject" "To" "From" "Body")))
              ("expression" "Expression" :size 45)
              ("destination" "Put In"
               :choices (:explicit ("Spam" "Nonspam")))))))
```

Figura A.1: Objeto de la página de inicio de Squeakymail

```
(define-view (html-format assembly)()

  :output-functions
  ((main-sheet
    ()
    (html (:html (:head
                  (:title "Squeakymail - Enjoy a squeaky-clean email experience"))
                 (:body (when *developing?* (the write-development-links))
                        (when (the return-object) (the write-back-link))
                        (:center (:h2 "Welcome to Squeakymail")
                                 (:i "Enjoy a Squeaky-Clean Email Experience"))
```

```
                          (:p (the classify (write-self-link)))
                          (:p (the filters (write-self-link)))
                          (:p (the pop-accounts (write-self-link)))

                          (the write-standard-footer)))))))
```

Figura A.2: Función de salida para la página de inicio de Squeakymail

```
    (define-object classify (base-html-sheet)

      :input-slots
      (local-login)

      :computed-slots
      ((message-list   (progn (fetch-mail (the local-login))
                              (read-incoming-messages (the local-login)))))

      :objects
      ((messages :type 'message
                 :sequence (:size (length (the message-list)))
                 :data (nth (the-child index) (the message-list))
                 :headers (getf (the-child :data) :headers)
                 :scored-tokens (getf (the-child :data) :scored-tokens)
                 :from (or (rest (assoc "from" (the-child headers)
                                        :test #'string-equal)) "")
                 :subject (or (rest (assoc "subject" (the-child headers)
                                           :test #'string-equal)) "")
                 :to (or (rest (assoc "to" (the-child headers)
                                      :test #'string-equal)) "")
                 :date (or (rest (assoc "date" (the-child headers)
                                        :test #'string-equal)) "")
                 :body (getf (the-child :data) :body)
                 :score (getf (the-child :data) :score))))
```

Figura A.3: Objeto de la página de clasificación de Squeakymail

```
    (define-view (html-format classify)()

      :output-functions
      ((main-sheet
        ()
        (let ((query-plist (the query-plist)))
          (classify-and-write-files query-plist (the local-login)))
        (the (restore-attribute-defaults! (list :message-list :query-plist)))

        (write-the (classify-form (list-elements (the messages)) (the local-login))))

        (classify-form
         (messages user)
         (html
          (:html
           (:head (:title (if messages (html "Classify Incoming Mail for ")
                              (html "No Mail for ")) (:princ user)))
           (:body (:p (when gwl:*developing?* (the write-development-links)))
                  (:h2 (:center (if messages (html "Classify Incoming Mail for ")
                                    (html "No Mail for ")) (:princ user)))
                  ((:form :action "/answer" :method :post)
                   (:p (the write-back-link))
                   (:p ((:input :type :submit :value (if messages "Classify!" "Check Aga
                   (:center
                    (the (:write-form-fields))
```

```
                  ((:table :bgcolor :black :cellpadding 1 :cellspacing 1)
                   (let ((grouped-messages (list nil nil nil)))
                     (mapc #'(lambda(message)
                               (let ((score (the-object message score)))
                                 (cond ((null score) (push message (first grouped-messa
                                       ((> score 0.8) (push message (third grouped-mess
                                       (t (push message (second grouped-messages))))))
                     (setq grouped-messages (append (nreverse (first grouped-messages)
                                                    (nreverse (second grouped-messages
                                                    (nreverse (third grouped-messages)
                     (when grouped-messages
                       (html ((:tr :bgcolor :yellow) (:th "Bad") (:th "Good") (:th "Fro
                                                     (:th "Subject") (:th "Date") (:th
                     (dolist (message grouped-messages)
                       (let ((from (the-object message from))
                             (subject (the-object message subject))
                             (date (the-object message date))
                             (score (the-object message score)))
                         (html ((:tr :bgcolor (cond ((null score) (gethash :sky-summer
                                                    ((> (the-object message score) 0.8
                                ((:td :bgcolor :red)
                                 ((:input :type :radio :name (format nil "spamp-~a" (th
                                          :value :spam :if* (and score (> (the-object m
                                ((:td :bgcolor :green)
                                 ((:input :type :radio :name (format nil "spamp-~a" (th
                                          :value :non-spam :if* (and score (< (the-obje
                                (:td (:small (:princ-safe (excl:replace-regexp from ""
                                (:td (:small (the-object message
                                                          (write-self-link
                                                           :display-string
                                                           (with-output-to-string(*html-
                                                             (html
                                                              (:princ-safe
                                                               (string-append (subseq su

                                                                              (if (> (le
                                                                                  "..."


                 (when messages (html (:p ((:input :type :submit :value "Classify!"))))
```

Figura A.4: Función de salida para la página de Clasificación

```
(define-object message (base-html-sheet)
  :input-slots
  (from subject to body score scored-tokens))
```

Figura A.5: Objeto de mensaje único

```
(define-view (html-format message)()
  :output-functions
  ((main-sheet
    ()
    (html
     (:html
      (:head
       (:title (:princ (the subject)))
       (:body
        (:p (the write-back-link))
        (:p (:table (:tr (:td "From") (:td (:princ-safe (the from))))
                    (:tr (:td "To") (:td (:princ-safe (the to))))
```

```
                    (:tr (:td "Subject") (:td (:princ-safe (the subject))))
                    (:tr (:td "Score") (:td (:princ-safe (the score))))
                    (:tr ((:td :colspan 2) (:pre (:princ (the body)))))))
          (when *show-tokens?*
            (html (:p ((:table :bgcolor :black)
                       (dolist (pair (subseq (the scored-tokens) 0
                                             (min (length (the scored-tokens)) 50)))
                         (html ((:tr :bgcolor (if (> (second pair) 0.5)
                                                  :pink "#aaffaa"))
                                (:td (:princ-safe (first pair)))
                                (:td (format *html-stream* "~2,7f"
                                             (second pair))))))))))))
```

Figura A.6: Función de salida para mostrar un mensaje

Apéndice B Bibliografía

Esta Bibliografía también podría considerarse como una lista de lectura recomendada.

- ANSI Common Lisp, by Paul Graham. Prentice-Hall, Inc, Englewood Cliffs, New Jersey. 1996.
- Common Lisp, the Language, 2nd Edition, by Steele, Guy L., Jr., with Scott E. Fahlman, Richard P. Gabriel, David A. Moon, Daniel L. Weinreb, Daniel G. Bobrow, Linda G. DeMichiel, Sonya E. Keene, Gregor Kiczales, Crispin Perdue, Kent M. Pitman, Richard C. Waters, and Jon L. White. Digital Press, Bedford, MA. 1990.
- ANSI CL Hyperspec, by Kent M. Pitman. Available at http://www.xanalys.com
- Learning Gnu Emacs, by Eric Raymond. O'Reilly and Associates. 1996.
- Sitio Web de la Asociación de Usuarios de Lisp, que contiene muchos otros recursos y referencias, en http://www.alu.org
- Grupo de Noticias de Usenet comp.lang.lisp
- Successful Lisp, de David Lamkin, disponible en http://psg.com/~dlamkins/left/sl/sl.html
- Franz Website (Distribuidor comercial de CL), en http://www.franz.com
- Sitio Web de Xanalys Lispworks (Distribuidor comercial de CL), en http://www.lispworks.com
- Symbolics Website (Distribuidor comercial de CL), en http://www.symbolics.com
- Sitio Web de Digitool (Distribuidor de Common Lisp para Macintosh), en http://www.digitool.com
- Corman Lisp Website (Commercial CL Vendor), at http://www.corman.net
- Genworks International Website (CL-based tools, services, and pointers to other resources) at http://www.genworks.com
- Knowledge Technologies International Website, at http://www.ktiworld.com
- Tulane Lisp Tutorial, available at http://www.cs.tulane.edu/www/Villamil/lisp/lisp1.html
- Texas A&M Lisp Tutorial, available at http://grimpeur.tamu.edu/colin/lp/

Apéndice C Personalización de Emacs

C.1 Modo Lisp

Cuando se edita un fichero Lisp en Emacs, Emacs debería automáticamente ponerse en mdo Lisp o Common Lisp. Se verá en la barra de estado en la zona inferior de la pantalla. El modo Lisp proporciona muchos comandos convenientes para trabajar con expresiones de listas, por lo demás llamadas Expresiones Simbólicas, o s-expresiones, o sexp abreviando.

Para una introducción completa al modo Lisp, se emite el comando M-x describe-mode en Emacs. La Tabla C.1 describe una muestra de los comandos disponibles en el modo Lisp[1].

C.2 Creación de combinaciones de teclas propias

Se puede desear automatizar algunos comandos que no poseen acordes de teclas asociados con ellos. La manera habitual de hacerlo es invocar algunos comandos global-set-key en el fichero .emacs (el fichero de inicialización de Emacs) en el directorio personal. Estos son algunas entradas de ejemplo que se podrían poner en este fichero, junto con los comentarios que describen lo que hacen:

Teclas	Acción	Función de Emacs
C-M-espacio	Marca la sexp que empieza en el punto	mark-sexp
M-w	Copia la expresión marcada	kill-ring-save
C-y	Pega la expresión copiada	yank
C-M-k	Corta la sexp que empieza en el punto	kill-sexp
C-M-f	Mueve el punto una sexp hacia adelante	forward-sexp
C-M-b	Mueve el punto una sexp hacia atrás	backward-sexp
M-/	Expande dinámicamente la palabra en curso (rota distintas elecciones)	dabbrev-expand

Tabla C.1: Comandos comunes de la interfaz de Emacs-Lisp

```
;; Make C-x & switch to the *common-lisp* buffer
(global-set-key "\C-x&" '(lambda() (interactive)
                           (switch-to-buffer "*common-lisp*")))

;; Make function key 5 (F5) start or visit an OS shell.
(global-set-key [f5] '(lambda() (interactive) (shell)))

;; Make sure M-p functions to scroll through previous commands.
(global-set-key "\M-p" 'fi:pop-input)

;; Make sure M-n functions to scroll through next commands.
(global-set-key "\M-n" 'fi:push-input)

;; Enable the hilighting of selected text to show up.
(transient-mark-mode 1)
```

C.3 Mapeo del teclado

Si se planea emplear un tiempo significativo trabajando con el teclado, considérese invertir algo de tiempo

1 Algunos de estos comandos de hecho también están disponibles en el modo Fundamental

en configurarlo tal que se pueda usar con efectividad. Como mínimo habría que asegurarse de que las teclas <Control> y <Meta> están configuradas tal que se puedan alcanzar sin tener que mover las manos de las "teclas caseras" del teclado. Esto significa que:

- La tecla <Control> debería estar a la izquierda de la tecla <A>, accesible con el meñique de la mano izquierda.
- Las teclas <Meta> deberían estar a cada lado de la barra espaciadora, accesibles tamborileando con los pulgares derecho o izquierdo bajo la palma de la mano.

Si se está trabajando con una terminal XWindow (v.g. una máquina Unix o GNU/Linux, o una PC ejecutando un servidor X tal como Hummingbird eXceed) se puede personalizar el teclado con el comando de Unix xmodmap. Para usar este comando, se prepara un fichero llamado .Xmodmap con las personalizaciones deseadas, después se invoca este fichero con el comando

```
xmodmap .Xmodmap
```

Esto es un ejemplo de fichero .Xmodmap para un teclado PC, que hace que la tecla de <bloqueo de mayúsculas> funcione como <Control>, y asegura que ambas teclas <Alt> funcionen como <Meta>:

```
remove Lock = Caps_Lock
keysym Caps_Lock = Control_L
add Control = Control_L
keysym Alt_L = Meta_L Alt_L
```

Para PC que ejecutan Microsoft Windows, el Panel de Control de Windows debería contener opciones para personalizar el teclado de una manera similar.

Apéndice D Postfacio

D.1 Acerca de este libro

Este libro fue tipografiado usando un lenguaje de marcas basado en Lisp. Para la salida impresa, the lenguaje de marcas basado en Lisp generó código LaTeX y fue tipografiado empleando LaTex de Leslie Lamport, que es una extensión de TeX de Donald Knuth.

[NdT: Esta traducción fue tipografiada con Texinfo, siguiendo todos los estándares de marcado de documentación propios del Proyecto GNU]

D.2 Reconocimientos

He recibido inspiración e instrucción de muchas personas, lo que me ha permitido escribir este libro. Me gustaría agradecer, sin orden en particular, a John McCarthy, Erik Naggum de Naggum Software y comp.lang.lisp, John Foderaro de Franz Inc., al catedrático John Laird que me enseñó CL en la facultad; John Mallery y Paul Graham, que me hicieron comprender que CL es la plataforma a usar para aplicaciones de servidor, James Russell, Andrew Wolven. Me gustaría agradecer especialmente a Lisa Fettner, Dr. Sheng-Chuan Wu y a otros en Franz Inc por su ojo avizor, Peter Karp from Stanford Research Institute por crear el bosquejo original, y a Hannu Koivisto por su ayuda en la salida en PDF. También agradecer a Michael Drumheller de la Boeing Co. por proporcionar sugerencias acerca de la primera edición. "Gracias" a mi esposa Kai, por mantener mis ilusiones mientras escribía esto (y en general). También quiero agradecer a Fritz Kunze de Franz Inc. por decirme "venga, escribe un libro".

Cualesquier errores que permanezcan es esta versión son, por supuesto, míos.

D.3 Acerca del autor

David J. Cooper ha sido ingeniero jefe de Genworks International desde Noviembre de 1997. Su actividad principal es trabajar con grandes compañías manufactureras, ayudándoles a solucionar problemas de ingeniería y manufactura, mayormente con herramientas basadas en CL. También ha impartido cursos de instrucción en General Motors, Visteon Automotive, Raytheon Aircraft, Ford Motor Company, Knowledge Technologies International y otros. Antes de su carrera en Genworks, Mr. Cooper trabajó para la Ford Motor Company, principalmente en la instalación de sistemas basados en CL en el campo de la ingeniería y manufactura mecánica. También tiene experiencia en el desarollo de software en CAD y bases de datos. Recibió su título de M.S. en Ciencias de la Computación en la Universidad de Michigan en 1993, y también tiene un B.S. en Ciencias de la Computación y un B.A. en Alemán, también en Michigan. Sus intereses actuales incluyen la venta y promoción de Genworks' GDL/GWL Knowledge Base. También está disponible para consultas y preguntas sobre productos a través de Genworks International en http://www.genworks.com david.cooper@genworks.com +1 (248) 910-0912.