6 LISP

Este capítulo introduce el lenguaje Lisp, desarrollado por McCarthy [55]. La primera parte del capítulo aborda la instalación del lenguaje. La segunda parte introduce los conceptos básicos de Lisp, siguiendo la presentación de Graham [35], donde se pone especial atención en algunas peculiaridades del lenguaje que pueden dificultar un primer acercamiento. La tercera parte del capítulo abunda en las listas como representación primaria de Lisp y su manejo iterativo, recursivo y basado en funciones de orden superior (mapeos). Finalmente se aborda una técnica particular de Lisp que permite escribir programas que generan programas: Las macros. Algunos aspectos no incluidos en este capítulo (paquetes, librerías, objetos) se revisarán en los dos capítulos siguientes donde se aborda la aplicación de Lisp a problemas de la IA.

6.1 INSTALACIÓN

Existen numerosas implementaciones de Lisp que podemos usar en cualquier sistema operativo. El cuadro 6.1 muestra algunas de las alternativas disponibles, su página web y el tipo de licencia asociada. Nosotros utilizaremos principalmente la versión profesional de `LispWorks` 6.0.1 y para el trabajo en casa, `sbcl`; aunque en sentido estricto, cualquiera de estas versiones puede usarse en el curso.

Lisp	URL	Licencia
LispWorks	`http://www.lispworks.com`	comercial
Steel Bank Common Lisp	`http://www.sbcl.org`	libre
Allegro Common Lisp	`https://franz.com`	comercial
Clozure Common Lisp	`https://ccl.clozure.com`	libre
Clojure	`https://clojure.org`	libre
Racket	`http://racket-lang.org`	libre

Cuadro 6.1: Implementaciones de Lisp

Hay varias formas de instalar estas distribuciones de Lisp. Las implementaciones comerciales proveen un instalador que solicitará la licencia del producto al momento de ejecutar Lisp por primera vez. Su instalación está plenamente documentada en su distribución. Ambas proveen un ambiente de desarrollo gráfico para los principales sistemas operativos.

`Clozure CL`, aunque provee implementaciones para todos los sistemas operativos, fue originalmente diseñado para MacOS, por lo que no es de extrañar que se pueda descargar gratuitamente desde la Apple Store. También puede descargarse desde su página web y proceder a instalarlo à la UNIX. Tanto `sbcl` como `Clojure` pueden instalarse de esta última forma; aunque si estamos en MacOS, la forma más fácil de instalarlos es haciendo use de `Homebrew` [1]:

- brew install sbcl
- brew install clozure-cl
- brew install clojure

1 `https://brew.sh/index_es.html`

Mención aparte merece `Clojure`, por tratarse de un dialecto de Lisp (por tanto no sigue el estándar ANSI adoptado en este curso) que compila el código para ser ejecutado en una máquina virtual de Java (JVM), de manera que tienen una gran integración con este lenguaje orientado a objetos. De igual manera, `Scheme` puede resultar interesante como un dialecto de Lisp alternativo. En particular, la distribución de este dialecto conocida como `Racket` es otra alternativa promovida como un lenguaje para escribir lenguajes gracias a las macros. Este dialecto también pueden instalarse vía `Homebrew`, pero como un *cask*:

- brew cask install racket

Sea cual sea el método de instalación elegido, el resultado debe ser un ejecutable de Lisp que se puede invocar desde consola. En lo que sigue, ejemplificaré la instalación detallada con `LispWorks`, aunque mencionaré en su momento los cambios necesarios para usar `sbcl` en su lugar.

6.1.1 Emacs y slime

Aunque las versiones comerciales incluyen ambientes de desarrollo gráficos, lo normal es que éste no sea el caso. La ejecución de Lisp desde consola pone al lenguaje en modo de espera por una expresión a evaluar en lo que se conoce como ciclo REPL (*Read-Eval-Print Loop*). Lo normal es utilizar el editor de textos `emacs` [2] para trabajar con Lisp. La página web de emacs incluye instrucciones para su instalación en los principales sistemas operativos. El editor también se puede instalar en MacOS vía `Homebrew`, aunque todos los sistemas *UNIX-like* suelen incluir un `emacs` en su distribución.

La interacción de `emacs` con `lisp` se lleva a cabo gracias a `SLIME` [3] (*The Superior Lisp Interaction Mode for Emacs*). Con un poco de suerte, este paquete ya está instalando en la distribución de `emacs` que hemos elegido. Esto se puede verificar tratando de invocar `SLIME` de la siguiente manera: `M-x slime` (Recuerden que en `emacs` los comandos suelen ejecutarse tecleando `alt+x`, que suele denotarse como antes, o `ctrl+x` que se denota como `C-x`; seguidos de un argumento y `enter`). Si todo va bien, su `emacs` debe lucir como el de la figura 6.1. Emacs se ha dividido en dos espacios de trabajo (*buffers*). En el superior puedo editar mi código de Lisp, que será evaluado en el inferior. El prompt `CL-USER>` indica que Lisp está a la espera de una expresión para evaluarla.

Para configurar `SLIME` hay que agregar las siguientes líneas al archivo de configuración `.emacs`, que se encuentra en el directorio raíz del usuario. Si no existe ahí, hay que crearlo con las siguientes líneas:

```
1 (setq inferior-lisp-program "/usr/local/bin/lw")
2 (setq slime-contribs '(slime-fancy))
3 (setq auto-mode-alist (append '(("\\.lisp$" . lisp-mode)
4                                 ("\\.lsp$" . lisp-mode))
5                               auto-mode-alist))
```

En este caso, he ligado una imagen (`lw`) de `LispWorks` sin ambiente de desarrollo gráfico, que he generado con la versión profesional de esta versión del lenguaje. En la versión personal gratuita no puede hacerse esto, pero pueden usar `sbcl` en su lugar.

Si `SLIME` no se encuentra instalado, se producirá un error diciendo que el comando `slime` no es conocido. Para instalarlo, conviene revisar la configuración de `melpa`, el repositorio de paquetes para `emacs`. El archivo de configuración `.emcas` debería contener las siguientes líneas (en caso contrario agréguelas):

2 https://www.gnu.org/software/emacs/
3 https://common-lisp.net/project/slime/

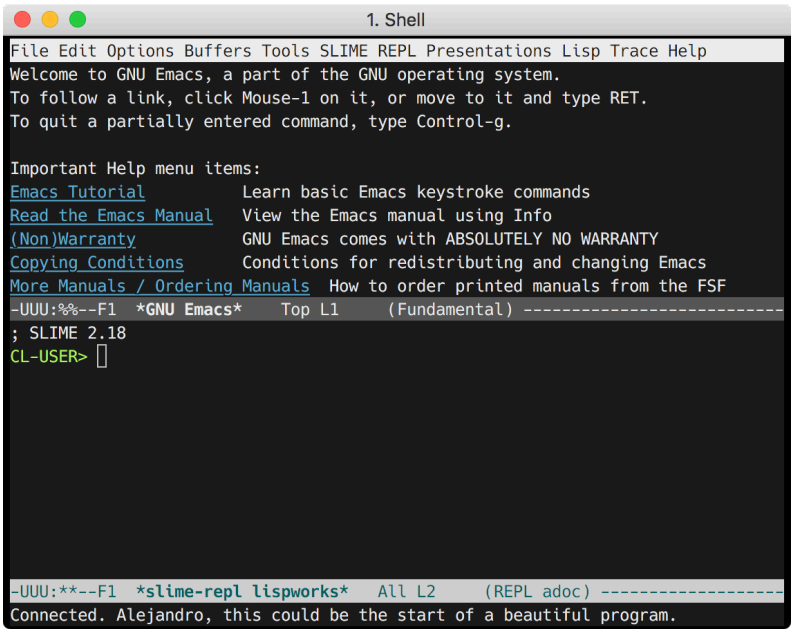

Figura 6.1: Emacs ejecutando SLIME.

```
(require 'package)

(add-to-list 'package-archives
             '("melpa" . "https://melpa.org/packages/"))
(when (< emacs-major-version 24)
  ;; For important compatibility libraries like cl-lib
  (add-to-list 'package-archives '("gnu" . "http://elpa.gnu.org/packages/")))

(package-initialize)
```

Reinicie `emacs` e invoque los siguientes comandos: Para actualizar el índice de paquetes disponibles`M-x package-refresh-contents`; para instar un paquete, que será `slime`, `M-x package-install`. Al terminar el proceso podrá invocar `slime` como se indicó anteriormente.

6.1.2 ASDF y QuickLisp

ASDF es una especie de `make` o `ant` para Lisp. Una herramienta para definir en que orden deben ser compiladas las fuentes de un sistema y sus dependencias. ASDF es la base de varios administradores de librerías, incluyendo QuickLisp que usaremos más adelante.

He descargado la ASDF 3.1.7 y compilado la librería usando el comando `make`, por lo que debería tener un archivo `asdf.lisp` en la carpeta `build` donde están las fuentes de ASDF. Desde LispWorks puedo compilar ese archivo de la siguiente manera (asumiendo que he ejecutado Lisp en esa carpeta, de otra forma hay que indicar el camino completo al archivo):

```
(compile-file "asdf.lisp")
```

Lo anterior resulta en un archivo `asdf.xfasl`, que es el archivo compilado para LispWorks.

Como pienso usar ASDF con algún administrador de librerías, debería crear la infraestructura para ello. En mi carpeta `home` voy a crear tres directorios:

```
> mkdir .asdf-install-dir
> mkdir .asdf-install-dir/site
> mkdir .asdf-install-dir/systems
```

Y ahora solo necesitamos integrar todo. Para ello debemos modificar o crear el archivo de configuración inicial de LispWorks. Si éste ya fue creado, debe haber un archivo `.lispworks` en mi directorio raíz. Debemos agregar las siguientes instrucciones en él:

```
#-:asdf (load "~/Dropbox/minted/lisp/asdf-3.1.7/build/asdf")
(pushnew "~/.asdf-install-dir/systems/" asdf: *central-registry*
:test #'equal)
```

Instalemos ahora el administrador de librerías `QuickLisp`, para facilitar las cosas. Solo hay que descargar el archivo `quicklisp.lisp` [4] y cargarlo en LispWorks:

```
USER> (load "~/Dropbox/minted/lisp/quicklisp/quicklisp.lisp")

  ==== quicklisp quickstart 2015-01-28 loaded ====

    To continue with installation, evaluate: (quicklisp-quickstart:install)

    For installation options, evaluate: (quicklisp-quickstart:help)

#P"/Users/aguerra/Dropbox/minted/lisp/quicklisp/quicklisp.lisp"
```

Procedemos ahora a instalar `QuickLisp`:

```
USER> (quicklisp-quickstart:install)

  ==== quicklisp installed ====

    To load a system, use: (ql:quickload "system-name")

    To find systems, use: (ql:system-apropos "term")

    To load Quicklisp every time you start Lisp, use: (ql:add-to-init-file)

    For more information, see http://www.quicklisp.org/beta/

NIL
```

Y a indicar a `LispWorks` que lo cargue cada vez que se inicie una sesión:

```
USER> (ql:add-to-init-file)
I will append the following lines to #P"~/.lispworks":

  ;;; The following lines added by ql:add-to-init-file:
  #-quicklisp
  (let ((quicklisp-init (merge-pathnames "quicklisp/setup.lisp"
                                         (user-homedir-pathname))))
    (when (probe-file quicklisp-init)
      (load quicklisp-init)))

Press Enter to continue.

#P"/Users/aguerra/.lispworks"
```

Ejemplo 6.1. *Supongamos que queremos partir una cadena de texto con la coma como separador. Podemos programar una función para ello o usar la librería* `split-sequence`*:*

```
USER> (ql:quickload "split-sequence") To load "split-sequence":
Install 1 Quicklisp release: split-sequence ; Fetching #<QL-HTTP:URL
"http://beta.quicklisp.org/archive/
```

4 http://www.quicklisp.org

```
split-sequence/2011-08-29/split-sequence-1.0.tgz"> ; 2.52KB
================================================== 2,580 bytes in
0.00 seconds (2519.53KB/sec) ; Loading "split-sequence" [package
split-sequence] ("split-sequence")

USER> (split-sequence:split-sequence #\, "hola, que, tal")
("hola" " que" " tal") 14
```

Con esto, hemos completado la instalación de nuestro Lisp.

6.2 CONCEPTOS BÁSICOS

El objetivo de esta sección es que puedan programar en Lisp tan pronto como sea posible. Al final de la misma, conocerán lo suficiente de Lisp como para comenzar a escribir sus propios programas. Este material debe revisarse como un tutorial básico sobre el lenguaje Lisp. Esta presentación se complementa con la revisión del capítulo tres del libro de Seibel [82].

6.2.1 Expresiones

Es particularmente cierto que la mejor forma de aprender Lisp es usándolo, porque se trata de un lenguaje interactivo. Cualquier sistema Lisp, incluye una interfaz interactiva llamada **top-level**. Uno escribe expresiones Lisp en el top-level, y el sistema despliega sus valores. El sistema normalmente despliega un indicador llamado **prompt** (>) señalando que está esperando que una expresión sea escrita. Por ejemplo, si escribimos el entero 1 después del prompt y tecleamos **enter**, tenemos:

```
> 1
1
>
```

el sistema despliega el valor de la expresión, seguida de un nuevo prompt, indicando que está listo para **evaluar** una nueva expresión. En este caso, el sistema desplegó lo mismo que tecleamos porque los números, como otras constantes, evalúan a si mismos. Las cosas son más interesantes cuando una expresión necesita algo de trabajo para ser evaluado, por ejemplo, sumar dos números:

```
> (+ 2 3)
5
>
```

En la expresión `(+ 2 3)` el símbolo `+` es llamado el **operador** y los números 3 y 4 son sus **argumentos** (o parámetros actuales, siguiendo la notación introducida en el capítulo anterior). Como el operador viene al principio de la expresión, esta notación se conoce como **prefija** y aunque parezca extraña, veremos que es muy práctica. Por ejemplo, si queremos sumar tres números en notación **infija**, necesitaríamos usar dos veces el operador +: 2+3+4. En Lisp, las siguientes sumas son válidas:

```
> (+)
0
> (+ 2)
2
> (+ 2 3)
5
> (+ 2 3 5)
10
```

Como los operadores pueden tomar un número variable de argumentos, es necesario utilizar los paréntesis para indicar donde inicia y donde termina una expresión. Las expresiones pueden anidarse, esto es, el argumento de una expresión puede ser otra expresión compleja. Ej.

```
> (/ (- 7 1)(- 4 2))
3
```

En español esto corresponde a siete menos uno, dividido por cuatro menos dos.

Estética minimalista, esto es todo lo que hay que decir sobre la notación en Lisp. Toda expresión Lisp es un *átomo*, como 1, o bien es una *lista* que consiste de cero o más expresiones delimitadas por paréntesis. Como veremos, código y datos usan la misma notación en Lisp.

6.2.2 Evaluación

Veamos más en detalle como las expresiones son evaluadas para desplegar su valor en el top-level. En Lisp, + es una función y `(+ 2 3)` es una **llamada** a la función. Cuando Lisp evalúa una llamada a alguna función, lo hace en dos pasos:

1. Los argumentos de la llamada son evaluados de izquierda a derecha. En este caso, los valores de los argumentos son 2 y 3, respectivamente.
2. Los valores de los argumentos son pasados a la función nombrada por el operador. En este caso la función + que regresa 5.

Si alguno de los argumentos es a su vez una llamada de función, será evaluado con las mismas reglas. Ej. Al evaluar la expresión (/ (- 7 1) (- 4 2)) pasa lo siguiente.

1. Lisp evalúa el primer argumento de izquierda a derecha (- 7 1). 7 es evaluado como 7 y 1 como 1. Estos valores son pasados a la función - que regresa 6.
2. El siguiente argumento (- 4 2) es evaluado. 4 es evaluado como 4 y 2 como 2. Estos valores son pasados a la función - que regresa 2.
3. Los valores 6 y 2 son pasados a la función / que regresa 3.

No todos los operadores en Lisp son funciones, pero la mayoría lo son. Todas las llamadas a función son evaluadas de esta forma, que se conoce como **regla de evaluación** de Lisp. Los operadores que no siguen la regla de evaluación se conocen como **operadores especiales**. Uno de estos operadores especiales es `quote` (’). La regla de evaluación de `quote` es –No evalúes nada, despliega lo que el usuario tecleo, *verbatim*. Por ejemplo:

```
> (quote (+ 2 3))
(+ 2 3)
> '(+ 2 3)
(+ 2 3)
```

Lisp provee el operador `quote` como una forma de evitar que una expresión sea evaluada. En la siguiente sección veremos porque esta protección puede ser útil.

6.2.3 Datos

Lisp ofrece los tipos de datos que podemos encontrar en otros lenguajes de programación, y otros que no. Ya hemos usado **enteros** en los ejemplos precedentes. Las**cadenas** de caracteres se delimita por comillas, por ejemplo, `“Hola mundo”`. Enteros y cadenas evalúan a ellos mismos, como las constantes.

Dos tipos de datos propios de Lisp son los **símbolos** y las **listas**. Los símbolos son palabras. Normalmente se evaluan como si estuvieran escritos en mayúsculas, independientemente de como fueron tecleados:

```
> 'uno
UNO
```

Los símbolos por lo general no evaluan a si mismos, así que si es necesario referirse a ellos, se debe usar `quote`, como en ejemplo anterior, de lo contrario, se producirá un error ya que el símbolo `uno` no está acotado (no tiene ligado ningún valor en este momento).

Las listas se representan como cero o más elementos entre paréntesis. Los elementos pueden ser de cualquier tipo, incluidas las listas. Se debe usar `quote` con las listas, ya que de otra forma Lisp las tomaría como una llamada a función. Veamos algunos ejemplos:

```
> '(Mis 2 "ciudades")
(MIS 2 "CIUDADES")
> '(La lista (a b c) tiene 3 elementos)
(LA LISTA (A B C) TIENE 3 ELEMENTOS)
```

Observen que `quote` protege a toda la expresión, incluidas las sub-expresiones en ella. La lista `(a b c)`, tampoco fue evaluada. También es posible construir listas usando la función `list`:

```
> (list 'mis (+ 4 2) "colegas")
(MIS 6 COLEGAS)
```

Estética minimalista y **pragmática**, observen que los programas Lisp se representan como listas. Si el argumento estético no bastará para defender la notación de Lisp, esto debe bastar –Un programa Lisp puede generar código Lisp! Por eso es necesario `quote`. Si una lista es precedida por el operador `quote`, la evaluación regresa la misma lista, en otro caso, la lista es evaluada como si fuese código. Por ejemplo:

```
> (list '(+ 2 3) (+ 2 3))
((+ 2 3) 5)
```

En Lisp hay dos formas de representar la lista vacía, como un par de paréntesis o con el símbolo `nil`. Ej.

```
> ()
NIL
> NIL
NIL
```

6.2.4 Operaciones básicas con listas

La función `cons` construye listas. Si su segundo argumento es una lista, regresa una nueva lista con el primer argumento agregado en el frente. Ej.

```
> (cons 'a '(b c d))
(A B C D)
> (cons 'a (cons 'b nil))
(A B)
```

Las funciones primitivas para accesar los elementos de una lista son `car` y `cdr`. El `car` de una lista es su primer elemento (el más a la izquierda) y el `cdr` es el resto de la lista (menos el primer elemento). Ej.

```
> (car '(a b c))
A
> (cdr '(a b c))
(B C)
```

Se pueden usar combinaciones de `car` y `cdr` para accesar cualquier elemento de la lista. Ej.

```
> (car (cdr (cdr '(a b c d))))
C
> (caddr '(a b c d))
C
> (third '(a b c d))
C
```

6.2.5 Valores de verdad

En Lisp, el símbolo `t` es la representación por default para verdadero. La representación por default de falso es `nil`. Ambos evalúan a si mismos. Por ejemplo, la función `listp` regresa verdadero si su argumento es una lista:

```
> (listp '(a b c))
T
> (listp 34)
NIL
```

Una función cuyo valor de regreso se interpreta como un valor de verdad (verdadero o falso) se conoce como **predicado**. En lisp es común que el símbolo de un predicado termine en `p`.

Como `nil` juega dos roles en Lisp, las funciones `null` (lista vacía) y `not` (negación) hacen exactamente lo mismo:

```
> (null nil)
T
> (not nil)
T
```

El condicional (estructura de control) más simple en Lisp es `if`. Normalmente toma tres argumentos: una expresión **test**, una expresión **then** y una expresión **else**. La expresión **test** es evaluada, si su valor es verdadero, la expresión **then** es evaluada; si su valor es falso, la expresión **else** es evaluada. Ej.

```
> (if (listp '(a b c d))
    (+ 1 2)
    (+ 3 4))
3
> (if (listp 34)
    (+ 1 2)
    (+ 3 4))
7
```

Como `quote`, `if` es un operador especial. No puede implementarse como una función, porque los argumentos de una función siempre se evaluan, y la idea al usar `if` es que sólo uno de sus argumentos sea evaluado.

Si bien el default para representar verdadero es `t`, todo excepto `nil` cuenta como verdadero en un contexto lógico. Ej.

```
> (if 27 1 2)
1
> (if nil 1 2)
2
```

Los operadores lógicos `and` y `or` parecen condicionales. Ambos toman cualquier número de argumentos, pero solo evaluan los necesarios para decidir que valor regresar. Si todos los argumentos son verdaderos (diferentes de `nil`), entonces `and` regresa el valor del último argumento. Ej.

```
> (and t (+ 1 2))
3
```

Pero si uno de los argumentos de `and` resulta falso, ninguno de los argumentos posteriores es evaluado y el operador regresa `nil`. De manera similar, `or` se detiene en cuanto encuentra un elemento verdadero.

```
> (or nil nil (+ 1 2) nil)
3
```

Observen que los operadores lógicos son operadores especiales, en este caso definidos como **macros**.

6.2.6 Funciones

Es posible definir nuevas funciones con `defun` que toma normalmente tres argumentos: un nombre, una lista de parámetros y una o más expresiones que conforman el cuerpo de la función. Ej. Así definiríamos `third`:

```
> (defun tercero (lst)
    (caddr lst))
TERCERO
```

El primer argumento de `defun` indica que el nombre de nuestra función definida será `tercero`. El segundo argumento `(lst)` indica que la función tiene un sólo argumento, `lst`. Un símbolo usado de esta forma se conoce como **variable**. Cuando la variable representa el argumento de una función, se conoce como `parámetro`. El resto de la definición indica lo que se debe hacer para calcular el valor de la función, en este caso, para cualquier `lst`, se calculará el primer elemento, del resto, del resto del parámetro `(caddr lst)`. Ej.

```
> (tercero '(a b c d e))
C
```

Ahora que hemos introducido el concepto de variable, es más sencillo entender lo que es un símbolo. Los símbolos son nombres de variables, que existen con derechos propios en el lenguaje Lisp. Por ello símbolos y listas deben protegerse con `quote` para ser accesados. Una lista debe protegerse porque de otra forma es procesada como si fuese código; un símbolo debe protegerse porque de otra forma es procesado como si fuese una variable.

Podríamos decir que la definición de una función corresponde a la versión generalizada de una expresión Lisp. Ej. La siguiente expresión verifica si la suma de 1 y 4 es mayor que 3:

```
> (> (+ 1 4) 3)
T
```

Substituyendo los números particulares por variables, podemos definir una función que verifica si la suma de sus dos primeros argumentos es mayor que el tercero:

```
> (defun suma-mayor-que (x y z)
    (> (+ x y) z))
SUMA-MAYOR-QUE
> (suma-mayor-que 1 4 3)
T
```

Lisp no distingue entre programa, procedimiento y función; todos cuentan como funciones y de hecho, casi todo el lenguaje está compuesto de funciones. Si se desea considerar una función en partícular como **main**, es posible hacerlo, pero cualquier función puede ser llamada desde el top-level. Entre otras cosas, esto significa que posible probar nuestros programas, pieza por pieza, conforme los vamos escribiendo, lo que se conoce como **programación incremental** (*bottom-up*).

6.2.7 Recursividad

Las funciones que hemos definido hasta ahora, llaman a otras funciones para hacer una parte de sus cálculos. Ej. `suma-mayor-que` llama a las funciones + y >. Una función puede llamar a cualquier otra función, incluida ella misma.

Una función que se llama a si misma se conoce como **recursiva**. Ej. En Lisp la función `member` verifica cuando algo es miembro de una lista. He aquí una versión recursiva simplificada de esta función:

```
> (defun miembro (obj lst)
    (if (null lst)
        nil
        (if (eql (car lst) obj)
            lst
            (miembro obj (cdr lst)))))
MIEMBRO
```

El predicado `eql` verifica si sus dos argumentos son idénticos, el resto lo hemos visto previamente. La llamada a `miembro` es como sigue:

```
> (miembro 'b '(a b c))
(B C)
> (miembro 'z '(a b c))
NIL
```

La descripción en español de lo que hace la función `miembro` es como sigue:

1. Primero, verificar si la lista `lst` está vacía, en cuyo caso es evidente que `obj` no es un miembro de `lst`.
2. De otra forma, si `obj` es el primer elemento de `lst` entonces es miembro de la lista.
3. De otra forma, `obj` es miembro de `lst` únicamente si es miembro del resto de `lst`.

Traducir una función recursiva a una descripción como la anterior, siempre ayuda a entenderla. Al principio, es común toparse con dificultades para entender la recursividad.

Una metáfora adecuada es ver a las funciones como procesos que vamos resolviendo. Solemos usar procesos recursivos en nuestras actividades diarias. Por ejemplo, supongan a un historiador interesado en los cambios de población a través de la historia de Europa. El proceso que el historiador utilizaría para examinar un documento es el siguiente:

1. Obtener una copia del documento que le interesa.
2. Buscar en él la información relativa a cambios de población en Europa.
3. Si el documento menciona otros documentos que puede ser útiles, examinarlos.

Piensen en `miembro` como las reglas que definen cuando algo es miembro de una lista y no como una máquina que computa si algo es miembro de una lista. De esta forma, la paradoja desaparece.

6.2.8 Leyendo y escribiendo Lisp

Estética. Si bien los paréntesis delimitan las expresiones en Lisp, un programador en realidad usa los margenes en el código para hacerlo más legible. Casi todo editor puede configurarse para verificar paréntesis bien balanceados. Ej. `:set sm` en el editor `vi`; o `M-x lisp-mode` en `Emacs`. Cualquier hacker en Lisp tendría problemas para leer algo como:

```
> (defun miembro (obj lst) (if (null lst) nil (if
(eql (car lst) obj) lst (miembro obj (cdr lst)))))
MIEMBRO
```

6.2.9 Entradas y salidas

Hasta el momento hemos procesado las E/S implícitamente, utilizando el top-level. Para programas interactivos esto no es suficiente[5], así que veremos algunas operaciones básicas de E/S.

La función de salida más general en Lisp es `format`. Esta función toma dos o más argumentos: el primero indica donde debe imprimirse la salida, el segundo es una cadena que se usa como molde (**template**), y el resto son generalmente objetos cuya representación impresa será insertada en la cadena molde. Ej.

```
> (format t "~A mas ~A igual a ~A. ~%" 2 3 (+ 2 3))
2 MAS 3 IGUAL A 5.
NIL
```

Observen que dos líneas fueron desplegadas en el ejemplo anterior. La primera es producida por `format` y la segunda es el valor devuelto por la llamada a `format`, desplegada por el top-level como se hace con toda función. Normalmente no llamamos a `format` en el top-level, sino dentro de alguna función, por lo que el valor que regresa queda generalmente oculto.

El primer argumento de `format`, `t`, indica que la salida será desplegada en el dispositivo estándar de salida, generalmente el top-level. El segundo argumento es una cadena que sirve como molde de lo que será impreso. Dentro de esta cadena, cada `~A` reserva espacio para insertar un objeto y el `~%` indica un salto de línea. Los espacios reservados de esta forma, son ocupados por el resto de los argumentos en el orden en que son evaluados.

La función estándar de entrada es `read`. Sin argumentos, normalmente la lectura se hace a partir del top-level. Ej. Una función que despliega un mensaje y lee la respuesta el usuario:

```
> (defun pregunta (string)
    (format t "~A" string)
    (read))
PREGUNTA
> (pregunta "Su edad: ")
Su edad: 34
34
```

Puesto que `read` espera indefinidamente a que el usuario escriba algo, no es recomendable usar `read` sin desplegar antes un mensaje que solicite la información al usuario. De otra forma el sistema dará la impresión de haberse plantado.

Se debe mencionar que `read` hace mucho más que leer caracteres, es un auténtico **parser** de Lisp que evalua su entrada y regresa los objetos que se hallan generado.

La función `pregunta`, aunque corta, muestra algo que no habíamos visto antes: su cuerpo incluye más de una expresión Lisp. El cuerpo de una función puede incluir

5 De hecho, casi todo el software actual incluye alguna interfaz gráfica con un sistema de ventaneo, por ejemplo, en lisp: CLIM, Common Graphics, etc.

cualquier número de expresiones. Cuando la función es llamada, las expresiones en su cuerpo son evaluadas en orden y la función regresará el valor de la última expresión evaluada.

Hasta el momento, lo que hemos mostrado se conoce como **Lisp puro**, esto es, Lisp sin efectos colaterales. Un efecto colateral es un cambio en el sistema Lisp producto de la evaluación de una expresión. Cuando evaluamos `(+ 2 3)`, no hay efectos colaterales, el sistema simplemente regresa el valor 5. Pero al usar `format`, además de obtener el valor `nil`, el sistema imprime algo, esto es un tipo de efecto colateral.

Cuando se escribe código sin efectos colaterales, no hay razón alguna para definir funciones cuyo cuerpo incluye más de una expresión. La última expresión evaluada en el cuerpo producirá el valor de la función, pero el valor de las expresiones evaluadas antes se perderá.

6.2.10 Variables

Uno de los operadores más comunes en Lisp es `let`, que permite la creación de nuevas variables locales. Ej.

```
> (let ((x 1)(y 2))
    (+ x y))
3
```

Una expresión `let` tiene dos partes: Primero viene una lista de expresiones definiendo las nuevas variables locales, cada una de ellas con la forma (**variable expresión**). Cada variable es inicializada con el valor que regrese la expresión asociada a ella. En el ejemplo anterior se han creado dos variables, `x` e `y`, con los valores `1` y `2` respectivamente. Esas variables son válidas dentro del cuerpo de `let`.

Después de la lista de variables y valores, viene el cuerpo de `let` constituido por una serie de expresiones que son evaluadas en orden. En el ejemplo, sólo hay una llamada a +. Presento ahora como ejemplo una función `preguntar` más selectiva:

```
> (defun preguntar-num ()
     (format t "Por favor, escriba un numero: ")
     (let ((val (read)))
        (if (numberp val)
            val
            (preguntar-num))))
```

Esta función crea la variable local `var` para guardar el valor que regresa `read`. Como este valor puede ahora ser manipulado por Lisp, la función revisa que se ha escrito para decidir que hacer. Si el usuario ha escrito algo que no es un número, la función vuelve a llamarse a si misma:

```
> (preguntar-num)
Por favor, escriba un numero: a
Por favor, escriba un numero: (un numero)
Por favor, escriba un numero: 3
3
```

Las variables de este estilo se conocen como `locales` porque sólo son válidas en cierto contexto. Existe otro tipo de variables llamadas `globales`, que son visibles donde sea[6].

Se puede crear una variable global usando `defparameter`:

```
> (defparameter *glob* 1970)
*GLOB*
```

6 La distinción propia es entre variables léxicas y especiales, pero por el momento no es necesario entrar en detalles

Esta variable es visible donde sea, salvo en contextos que definan una variable local con el mismo nombre. Para evitar errores accidentales con los nombre de las variables, se usa la convención de nombrar a las variables globales con símbolos que inicien y terminen en asterisco.

Se pueden definir también constantes globales usando `defconstant`:

```
> (defconstant limit (+ *glob* 1))
```

No hay necesidad de dar a las constantes nombres distintivos porque si se intenta usar el nombre de una constante para una variable se produce un error. Para verificar si un símbolo es el nombre de una variable global o constante, se puede usar `boundp`:

```
> (boundp '*glob*)
T
```

6.2.11 Asignaciones

En Lisp el operador de asignación más común es `setf`. Se puede usar para asignar valores a cualquier tipo de variable. Ej.

```
> (setf *glob* 2000)
2000
> (let ((n 10))
    (setf n 2)
  n)
2
```

Cuando el primer argumento de `setf` es un símbolo que no es el nombre de una variable local, se asume que se trata de una variable global.

```
> (setf x (list 'a 'b 'c))
(A B C)
> (car x)
A
```

Esto es, es posible crear implícitamente variables globales con sólo asignarles valores. Como sea, es preferible usar explícitamente `defparameter`.

Se puede hacer más que simplemente asignar valores a variables. El primer argumento de `setf` puede ser tanto una expresión, como un nombre de variable. En el primer caso, el valor representado por el segundo argumento es insertado en el lugar al que hace referencia la expresión. Ej.

```
> (setf (car x) 'n)
N
> x
(N B C)
```

Se puede dar cualquier número de argumentos pares a `setf`. Una expresión de la forma:

```
> (setf a 'b
        c 'd
        e 'f)
F
```

es equivalente a:

```
> (set a 'b)
B
> (set b 'c)
C
> (set e 'f)
F
```

6.2.12 Programación funcional

La **programación funcional** significa, entre otras cosas, escribir programas que trabajan regresando valores, en lugar de modificar cosas. Es el paradigma de programación dominante en Lisp. La mayor parte de las funciones predefinidas en el lenguaje, se espera sean llamadas por el valor que producen y no por sus efectos colaterales.

La función `remove`, por ejemplo, toma un objeto y una lista y regresa una nueva lista que contiene todos los elementos de la lista original, menos el objeto indicado:

```
> (setf lst '(k a r a t e))
(K A R A T E)
> (remove 'a lst)
(K R T E)
```

¿Por qué no decir simplemente que `remove` remueve un objeto dado de una lista? Porque esto no es lo que la función hace. La lista original no es modificada:

```
> lst
(K A R A T E)
```

Si se desea que la lista original sea afectada, se puede evaluar la siguiente expresión:

```
> (setf lst (remove 'a lst))
(K R T E)
> lst
(K R T E)
```

La programación funcional significa, esencialmente, evitar `setf` y otras expresiones con el mismo tipo de efecto colateral. Esto puede parecer contra intuitivo y hasta no deseable. Si bien programar totalmente sin efectos colaterales es inconveniente, a medida que practiquen Lisp, se sorprenderán de lo poco que en realidad se necesita este tipo de efecto.

Una de las ventajas de la programación funcional es que permite la **verificación interactiva**. En código puramente funcional, se puede verificar cada función a medida que se va escribiendo. Si la función regresa los valores que esperamos, se puede confiar en que es correcta. La confianza agregada al proceder de este modo, hace una gran diferencia: un nuevo estilo de programación.

6.2.13 Iteración

Cuando deseamos programar algo repetitivo, algunas veces la iteración resulta más natural que la recursividad. El caso típico de iteración consiste en generar algún tipo de tabla. Ej.

```
> (defun cuadrados (inicio fin)
    (do ((i inicio (+ i 1)))
        ((> i fin) 'final)
      (format t "~A ~A ~%" i (* i i))))
CUADRADOS
> (cuadradros 2 5)
2 4
3 9
4 16
5 25
FINAL
```

La macro `do` es el operador fundamental de iteración en Lisp. Como `let`, `do` puede crear variables y su primer argumento es una lista de especificación de variables.

Cada elemento de esta lista toma la forma (**variable valor-inicial actualización**). En cada iteración el valor de las variables definidas de esta forma, cambia como lo especifica la actualización.

En el ejemplo anterior, `do` crea únicamente la variable local `i`. En la primer iteración `i` tomará el valor de `inicio` y en las sucesivas iteraciones su valor se incrementará en `1`.

El segundo argumento de `do` debe ser una lista que incluya una o más expresiones. La primera expresión se usa como prueba para determinar cuando debe parar la iteración. En el ejemplo, esta prueba es `(> i fin)`. El resto de la lista será evaluado en orden cuando la iteración termine. La última expresión evaluada será el valor de `do`, por lo que `cuadrados`, regresa siempre el valor `'final`.

El resto de los argumentos de `do`, constituyen el cuerpo del ciclo y serán evaluados en orden en cada iteración, donde: las variables son actualizadas, se evalua la prueba de fin de iteración y si esta falla, se evalua el cuerpo de `do`. Para comparar, se presenta aquí una versión recursiva de `cuadrados-rec`:

```
> (defun cuadrados-rec (inicio fin)
    (if (> inicio fin)
        'final
        (progn
          (format t "~A ~A ~%" inicio (* inicio inicio))
          (cuadrados-rec (+ inicio 1) fin))))
CUADRADOS
```

La única novedad en esta función es `progn` que toma cualquier número de expresiones como argumentos, las evalua en orden y regresa el valor de la última expresión evaluada.

Lisp provee operadores de iteración más sencillos para casos especiales, por ejemplo, `dolist` para iterar sobre los elementos de una lista. Ej. Una función que calcula la longitud de una lista:

```
> (defun longitud (lst)
    (let ((len 0))
       (dolist (obj lst)
          (setf len (+ len 1)))
       len))
LONGITUD
> (longitud '(a b c))
3
```

El primer argumento de `dolist` toma la forma (**variable expresión**), el resto de los argumentos son expresiones que constituyen el cuerpo de `dolist`. Este cuerpo será evaluado con la `variable` instanciada con elementos sucesivos de la lista que regresa `expresión`. La función del ejemplo, dice – por cada `obj` en `lst`, incrementar en uno `len`.

La versión recursiva obvia de esta función longitud es:

```
> (defun longitud-rec (lst)
    (if (null lst)
        0
        (+ (longitud-rec (cdr lst)) 1)))
LONGITUD
```

Esta versión será menos eficiente que la iterativa porque no es recursiva a la cola (**tail-recursive**), es decir, al terminar la recursividad, la función debe seguir haciendo algo. La definición recursiva a la cola de longitud es:

```
> (defun longitud-tr (lst)
   (labels ((longaux (lst acc)
               (if (null lst)
```

```
                    acc
                    (longaux (cdr lst) (+ 1 acc)))))
      (longaux lst 0)))
LONGITUD-TR
```

Lo nuevo aquí es `labels` que permite definir funciones locales como `longaux`.

6.2.14 Funciones como objetos

En Lisp las funciones son objetos regulares como los símbolos, las cadenas y las listas. Si le damos a `function` el nombre de una función, nos regresará el objeto asociado a ese nombre. Como `quote`, `function` es un operador especial, así que no necesitamos proteger su argumento. Ej.

```
> (function +)
#<Compiled-Function + 17BA4E>
```

Esta extraño valor corresponde a la forma en que una función sería desplegada en una implementación Lisp. Hasta ahora, hemos trabajado con objetos que lucen igual cuando los escribimos y cuando Lisp los evalua. Esto no suecede con las funciones, cuya representación interna corresponde más a un segmento de código máquina, que a la forma como la definimos.

Al igual que usamos `'` para abreviar `quote`, podemos usar `#'`, para abreviar `function`.

```
> #'+
#<Compiled-Function + 17BA4E>
```

Como sucede con otros objetos, en Lisp podemos pasar funciones como argumentos. Una función que toma una función como argumento es `apply`. Ej.

```
> (apply #'+ '(1 2 3))
6
> (+ 1 2 3)
6
```

Se le puede dar cualquier número de argumentos, si se respeta que el último de ellos sea una lista. Ej.

```
> (apply #'+ 1 2 '(3 4 5))
15
```

La función `funcall` hace lo mismo, pero no necesita que sus argumentos estén empaquetados en forma de lista. Ej.

```
> (funcall #'+ 1 2 3)
6
```

La macro `defun` crea una función y le da un nombre, pero las funciones no tienen porque tener nombre y no necesitamos `defun` para crearlas. Como los otros tipos de objeto en Lisp, podemos definir a las funciones literalmente.

Para referirnos literalmente a un entero usamos una secuencia de dígitos; para referirnos literalmente a una función usamos una expresión **lambda** cuyo primer elemento es el símbolo `lambda`, seguido de una lista de parámetros y el cuerpo de la función. Ej.

```
> (lambda (x y)
    (x + y))
```

Una expresión lambda puede considerarse como el nombre de una función. Ej. Puede ser el primer elemento de una llamada de función:

```
> ((lambda (x) (+ x 100)) 1)
101
```

o usarse con `funcall`:

```
> (funcall #'(lambda (x) (+ x 100)) 1)
101
```

Entre otras cosas, esta notación nos permite usar funciones sin necesidad de nombrarlas. También podemos usar este truco para computar mapeos sobre listas:

```
> (mapcar #'(lambda(x) (* 2 x)) '(1 2 3 4))
(2 4 6 8)
> (defun dobles (lst)
(mapcar #'(lambda(x)(* 2 x)) lst))
DOBLES
> (dobles '(1 2 3 4))
(2 4 6 8)
```

6.2.15 Tipos

Lisp utiliza un inusual enfoque flexible sobre tipos. En muchos lenguajes, las variables tienen un tipo asociado y no es posible usar una variable sin especificar su tipo. En Lisp, los valores tienen un tipo, no las variables. Imaginen que cada objeto en Lisp tiene asociada una etiqueta que especifica su tipo. Este enfoque se conoce como **tipificación manifiesta**. No es necesario declarar el tipo de una variable porque cualquier variable puede recibir cualquier objeto de cualquier tipo. De cualquier forma, es posible definir el tipo de una variable para optimizar el código antes de la compilación.

Lisp incluye una jerarquía predefinida de subtipos y supertipos. Un objeto siempre tiene más de un tipo. Ej. el número 27 es de tipo `fixnum`, `integer`, `rational`, `real`, `number`, `atom` y `t`, en orden de generalidad incremental. El tipo `t` es el supertipo de todo tipo.

La función `typep` toma como argumentos un objeto y un especificador de tipo y regresa `t` si el objeto es de ese tipo. Ej.

```
> (typep 27 'integer)
T
```

6.2.16 Consideraciones

Aunque este documento presenta un bosquejo rápido de Lisp, es posible apreciar ya el retrato de un lenguaje de programación inusual. Un lenguaje con una sola sintaxis para expresar programas y datos. Esta sintaxis se basa en listas, que son a su vez objetos en Lisp. Las funciones, que son objetos del lenguaje también, se expresan como listas. Y Lisp mismo es un programa Lisp, programado casi por completo con funciones Lisp que en nada difieren a las que podemos definir.

No debe preocuparles que la relación entre todas estas ideas no sea del todo clara. Lisp introduce tal cantidad de conceptos nuevos que toma tiempo acostumbrarse a ellos y como usarlos. Solo una cosa debe estar clara: Las ideas detrás Lisp son extremadamente elegantes.

Si C es el lenguaje para escribir UNIX[7], entonces podríamos describir a Lisp como el lenguaje para describir Lisp, pero eso es una historia totalmente diferente. Un lenguaje que puede ser escrito en si mismo, es algo distinto de un lenguaje para escribir una clase partícular de aplicaciones. Ofrece una nueva forma de programar: así como es posible escribir un programa en el lenguaje, ¡el lenguaje puede mejorarse para acomodarse al programa! Esto es un buen inicio para entender la esencia de Lisp.

7 Richard Gabriel

6.3 LISTAS

Las listas fueron originalmente la principal estructura de datos en Lisp. De hecho, el nombre del lenguaje es un acrónimo de "LISt Processing". Las implementaciones modernas de Lisp incluyen otras estructuras de datos. El desarrollo de programas en Lisp refleja esta historia. Las versiones iniciales del programa suelen hacer un uso intensivo de listas, que posteriormente se convierten a otros tipos de datos, más rápidos o especializados. Este capítulo muestra qué es lo que uno puede hacer con las listas y las usa para ejemplificar algunos conceptos generales de Lisp.

6.3.1 Conses

En el capítulo anterior se introdujeron las funciones primitivas *cons*/2, *car*/1 y *cdr*/1 para el manejo de listas. Lo que *cons* hace es combinar dos objetos, en uno formado de dos partes llamado **cons**. Conceptualmente, un *cons* es un par de apuntadores: el primero es el *car* y el segundo el *cdr*.

Los *conses* proveen una representación conveniente para cualquier tipo de pares. Los dos apuntadores del *cons* pueden dirigirse a cualquier objeto, incluyendo otros *conses*. Es esta última propiedad, la que explotamos para definir **listas** en términos de *cons*. Una lista puede definirse como el par formado por su primer elemento y el resto de la lista. La mitad del *cons* apunta a ese primer elemento; la otra mitad al resto de la lista.

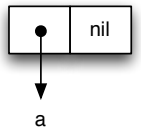

Figura 6.2: Una lista de un elemento, como *cons a nil*.

Cuando aplicamos *cons* a un elemento y una lista vacía, el resultado es un solo *cons* mostrado en la figura 6.2. Como cada *cons* representa un par de apuntadores, *car* regresa el objeto apuntado como primer componente del *cons* y *cdr* el segundo:

```
> (cons 'a nil)
(A)
> (car '(a))
A
> (cdr '(a))
NIL
```

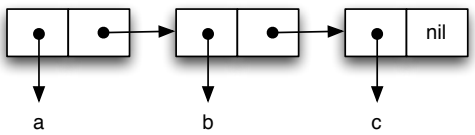

Figura 6.3: Una lista de varios elementos.

Cuando construimos una lista con múltiples componentes, el resultado es una cadena de *conses*. La figura 6.3 ilustra la siguiente construcción:

```
> (list 'a 'b 'c)
(A B C)
> (cdr (list 'a 'b 'c))
(B C)
```

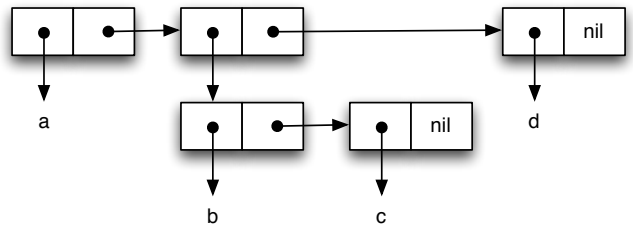

Figura 6.4: Una lista de varios elementos, incluida otra lista.

Las listas pueden tener elementos de cualquier tipo, incluidas otras listas, como lo ilustran las siguientes definiciones y la figura 6.4:

```
> (list 'a (list 'b 'c) 'd)
(A (B C) D)
```

Las listas que no incluyen otras listas, se conocen como **listas planas**. En caso contrario, decimos que se trata de una **lista anidada**.

La función *consp*/1 regresa *true* si su argumento es un *cons*, así que podemos definir *listp*/1 que regresa *true* si su argumento es una lista como sigue:

```
  > (defun mi-listp (x)
            (or (null x) (consp x)))
  MI-LISTP
  > (defun mi-atomp (x)
            (not (consp x)))
  MI-ATOMP
  > (mi-listp '(1 2 3))
  T
  > (mi-atomp '(1 2 3))
  NIL
  > (mi-listp nil)
  T
  > (mi-atomp nil)
  T
```

la definición de *mi − atomp* se basa en que todo lo que no es un *cons* es un **átomo**. Observen que *nil* es lista y átomo a la vez.

6.3.2 Cons e igualdad

Cada vez que invocamos a *cons*, Lisp reserva memoria para dos apuntadores, así que si llamamos a *cons* con el mismo argumento dos veces, Lisp regresa dos valores que aparentemente son el mismo, pero en realidad se trata de diferentes objetos:

```
> (eql (cons 1 nil) (cons 1 nil))
NIL
```

Sería conveniente contar con una función *mi − eql* que regrese *true* cuando dos listas tienen los mismos elementos, aunque se trate de objetos distintos:

```
> (defun mi-eql (lst1 lst2)
      (or (eql lst1 lst2)
          (and (consp lst1)
               (consp lst2)
               (mi-eql (car lst1) (car lst2))
               (mi-eql (cdr lst1) (cdr lst2)))))
MI-EQL
> (mi-eql (cons 1 nil) (cons 1 nil))
T
```

en realidad, lisp cuenta con una función predefinida *equal*/2 que cumple con nuestros objetivos. Como la definición de nuestro *eql* lo sugiere, si dos objetos son *eql*, también son *equal*.

Uno de los secretos para comprender Lisp es darse cuenta de que las variables tienen valores, en el mismo sentido en que las listas tienen elementos. Así como los *conses* tienen apuntadores a sus elementos, las variables tienen apuntadores a sus valores.

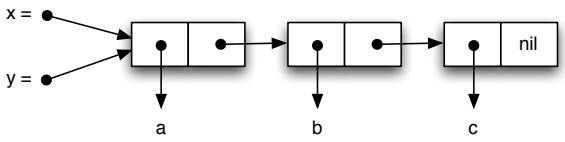

Figura 6.5: Una lista de varios elementos, incluida otra lista.

La diferencia entre Lisp y otros lenguajes de programación donde manipulamos los apuntadores explícitamente, es que en el primero caso, el lenguaje administra los apuntadores por uno. Ya vimos ejemplos de esto, relacionados con la creación de listas. Algo similar pasa con las variables. Por ejemplo, si asignamos a dos variables la misma lista:

```
1 > (setf x '(a b c))
2 (A B C)
3 > (setf y x)
4 (A B C)
5 > (eql x y)
6 T
```

¿Qué sucede al hacer la segunda asignación (línea 3)? En realidad, la localidad de memoria asociada a x no contiene la lista (a, b, c) sino un apuntador a esta lista. El *setf* en cuestión copia ese mismo apuntador a la localidad de memoria asociada a y. Es decir, Lisp copia el apuntador relevante, no la lista completa. La figura 6.5 ilustra este caso. Por eso *eql* en la llamada de la línea 5, regresa *true*.

6.3.3 Construyendo listas

Ya hemos visto ejemplos de construcción de listas con *list* y *cons*. Ahora imaginen que deseamos una función $copia - lista$ que una lista como su primer argumento y regresa una copia de ella. La lista resultante tiene los mismos elementos que la lista original, pero está contenida en *conses* diferentes:

```
1  > (defun copia-lista (lst)
2        (if (atom lst)
3            lst
4            (cons (car lst) (copia-lista (cdr lst)))))
5
6  COPIA-LISTA
7  > (setf x '(a b c)
8          y (copia-lista x))
9  (A B C)
10 > (eql x y)
11 NIL
```

evidentemente, x y su copia nunca serán *eql* pero si *equal*, al menos que x sea *nil*.

Existe otro constructor de listas que toma como argumento varias listas para construir una sola:

```
1 CL-USER> (append '(1 2) '(3) '(4))
2 (1 2 3 4)
```

6.3.4 Compresión de datos *run-length*

Consideremos un ejemplo para utilizar los conceptos introducidos hasta ahora. RLE (*run-length encoding*) es un algoritmo de compresión de datos muy sencilla. Funciona como los meseros: si los comensales pidieron una tortilla española, otra, otra más y una ensalada verde; el mesero pedirá tres tortillas españolas y una ensalada verde. El código de esta estrategia es como sigue:

```
(defun rle (lst)
  (if (consp lst)
      (compress (car lst) 1 (cdr lst))
      lst))

(defun compress (elt n lst)
  (if (null lst)
      (list (n-elts elt n))
      (let ((sig (car lst)))
        (if (eql sig elt)
            (compress elt (+ n 1) (cdr lst))
            (cons (n-elts elt n)
                  (compress sig 1 (cdr lst)))))))

(defun n-elts (elt n)
  (if (> n 1)
      (list n elt)
      elt))
```

Una llamada a *rle* sería como sigue:

```
> (rle '(1 1 1 0 1 0 0 0 0 1))
((3 1) 0 1 (4 0) 1)
```

Ejercicio sugerido. Programen una función inversa a *rle*, esto es dado una lista que es un código *rle*, esta función regresa la cadena original. Observen que este método de comprensión no tiene perdida de información. Prueben su solución con la ayuda de un generador de listas de n elementos aleatorios.

6.3.5 Funciones básicas

A continuación definiremos una biblioteca mínima de operaciones sobre listas, que por cuestiones de eficiencia serán declaradas:

```
(proclaim '(inline last1 single append1 conc1 mklist))
(proclaim '(optimize speed))

(defun last1 (lst)
  (car (last lst)))

(defun single (lst)
  (and (consp lst) (not (cdr lst))))

(defun append1 (lst obj)
  (append lst (list obj)))

(defun conc1 (lst obj)
  (nconc lst (list obj)))

(defun mklist (obj)
  (if (listp obj) obj (list obj)))
```

La función `last1` regresa el último elemento de una lista. La función predefinida `last` regresa el último *cons* de una lista, no su último elemento. Generalmente obtenemos tal elemento usando `(car(last ...))`. ¿Vale la pena definir una nueva función para una función predefinida? La respuesta es afirmativa cuando la nueva función remplaza efectivamente a la función predefinida.

Observen que `last1` no lleva a cabo ningún chequeo de error. En general, ninguna de las funciones del curso harán chequeo de errores. En parte esto se debe a que de esta manera los ejemplos serán más claros; y en parte se debe a que no es razonable hacer chequeo de errores en utilidades tan pequeñas. Si intentamos:

```
> (last1 "prueba")
value "prueba" is not of the expected type LIST.
[Condition of type TYPE-ERROR]
```

el error es capturado y reportado por la función predefinida `last`. Cuando las utilidades son tan pequeñas, forman una capa de abstracción tan delgada, que comienzan por ser transparentes. Uno puede ver en `last1` para interpretar los errores que ocurren en sus funciones subyacentes.

La función `single` prueba si algo es una lista de un elemento. Los programas Lisp necesitan hacer esta prueba bastantes veces. Al principio, uno está tentado a utilizar la traducción natural del español a Lisp:

```
(= (length lst) 1)
```

pero escrita de esta forma, la función sería muy ineficiente.

Las funciones `append1` y `conc1` agregan un elemento al final de una lista, `conc1` de manera destructiva. Estas funciones son pequeñas, pero se usan tantas veces que vale la pena incluirlas en la librería. De hecho, `append1` ha sido predefinida en muchos dialectos Lisp.

La función `mklst` nos asegura que su argumento sea una lista. Muchas funciones Lisp están escritas para regresar una lista o un elemento. Supongamos que `lookup` es una de estas funciones. Si queremos colectar el resultado de aplicar esta función a todos los miembros de una lista, podemos escribir:

```
(mapcar #'(lambda (d) (mklist (lookup d)))
  data)
```

Veamos ahora otros ejemplos de utilidades más grandes que operan sobre listas:

```
(defun longer (x y)
  (labels ((compare (x y)
             (and (consp x)
                  (or (null y)
                      (compare (cdr x) (cdr y))))))
    (if (and (listp x) (listp y))
        (compare x y)
        (> (length x) (length y)))))

(defun filter (fn lst)
  (let ((acc nil))
    (dolist (x lst)
      (let ((val (funcall fn x)))
        (if val (push val acc))))
    (nreverse acc)))

(defun group (source n)
  (if (zerop n) (error "zero length"))
  (labels ((rec (source acc)
             (let ((rest (nthcdr n source)))
               (if (consp rest)
```

```
                  (rec rest (cons (subseq source 0 n) acc))
                  (nreverse (cons source acc))))))
    (if source (rec source nil) nil)))
```

Al comparar la longitud de dos lista, lo más inmediato es usar `(>(length x) (length y))`, pero esto es ineficiente. En partícular si una de las listas es mucho más corta que la otra. Lo mejor, es usar `longer` y recorrerlas en paralelo con la función local `compare`, en caso de que x e y sean listas. Si este no es el caso, por ejemplo, si los argumentos de `longer` son cadenas de texto, solo entonces usaremos `length`.

La función `filter` aplica su primer argumento `fn` a cada elemento de la lista `lst` guardando aquellos resultados diferentes a `nil`.

```
> (filter #'null '(nil t nil t 5 6))
(T T)
> (filter #'(lambda (x)
              (when (> x 0) x))
                 '(-1 2 -3 4 -5 6))
(2 4 6)
> (filter #'(lambda (x)
              (if (numberp x) (1+ x)))
        '(a 1 2 b 3 c d 4))
(2 3 4 5)
```

Observen el uso del acumulador `acc` en la definición de `filter`. La combinación de `push` y `nreverse` es la forma estándar de producir una lista acumulador en Lisp.

La función `group` agrupa una lista `lst` en sublistas de tamaño `n`:

```
> (group '(a b c d e f g) 2)
((A B) (C D) (E F) (G))
```

Esta función si lleva a cabo un chequeo de error, porque si $n = 0$ `group` entra en un ciclo infinito.

Otras funciones (doblemente recursivas) sobre listas son:

```
(defun flatten (x)
  (labels ((rec (x acc)
             (cond ((null x) acc)
                   ((atom x) (cons x acc))
                   (t (rec (car x) (rec (cdr x) acc))))))
    (rec x nil)))

(defun prune (test tree)
  (labels ((rec (tree acc)
             (cond ((null tree) (nreverse acc))
                   ((consp (car tree))
                    (rec (cdr tree)
                         (cons (rec (car tree) nil) acc)))
                   (t (rec (cdr tree)
                           (if (funcall test (car tree))
                               acc
                               (cons (car tree) acc)))))))
    (rec tree nil)))
```

estas funciones recurren sobre listas anidadas para hacer su trabajo. La primera de ellas, `flatten`, es una aplanadora de listas. Si su argumento es una lista anidada, regresa los elementos de la lista original, pero eliminando el anidamiento:

```
> (flatten '(a (b c) ((d e) f)))
(A B C D E F)
```

La segunda función, `prune`, elimina de una lista anidada a aquellos elementos atómicos que satisfacen el predicado `test`, de forma que:

```
> (prune #'evenp '(1 2 (3 (4 5) 6) 7 8 (9)))
(1 (3 (5)) 7 (9))
```

6.3.6 Mapeos

Otra clase de funciones ampliamente usadas en Lisp son los mapeos, que aplican una función a la secuencia de sus argumentos. La más conocida de estas funciones es `mapcar`. Definiremos otras funciones de mapeo a continuación:

```
(defun map0-n (fn n)
  (mapa-b fn 0 n))

(defun map1-n (fn n)
  (mapa-b fn 1 n))

(defun mapa-b (fn a b &optional (step 1))
  (do ((i a (+ i step))
       (result nil))
      ((> i b) (nreverse result))
    (push (funcall fn i) result)))

(defun map-> (fn start test-fn succ-fn)
  (do ((i start (funcall succ-fn i))
       (result nil))
      ((funcall test-fn i) (nreverse result))
    (push (funcall fn i) result)))

(defun mappend (fn &rest lsts)
  (apply #'append (apply #'mapcar fn lsts)))

(defun mapcars (fn &rest lsts)
  (let ((result nil))
    (dolist (lst lsts)
      (dolist (obj lst)
        (push (funcall fn obj) result)))
    (nreverse result)))

(defun rmapcar (fn &rest args)
  (if (some #'atom args)
      (apply fn args)
      (apply #'mapcar
             #'(lambda (&rest args)
                 (apply #'rmapcar fn args))
             args)))
```

Las primeras tres funciones mapean funciones sobre rangos de números sin tener que hacer `cons` para guardar la lista resultante. Las primeras dos `map0-n` y `map1-n` funcionan con rangos positivos de enteros:

```
> (map0-n #'1+ 5)
(1 2 3 4 5 6)

> (map1-n #'1+ 5)
(2 3 4 5 6)
```

Ambas fueron escritas usando `mapa-b` que funciona para cualquier rango de números:

```
> (mapa-b #'1+ 1 4 0.5)
(2 2.5 3.0 3.5 4.0 4.5 5.0)
```

A continuación se implementa un mapeo más general con `map->`, el cual trabaja para cualquier tipo de secuencias de objetos de cualquier tipo. La secuencia comienza con el objeto dado como segundo argumento; el final de la secuencia está dado por el tercer argumento como una función; y los sucesores del primer elemento se generan de acuerdo a la función que se da como cuarto argumento. Con esta función es posible navegar en estructuras de datos arbitrarias, así como operar sobre secuencias de números. Por ejemplo, `mapa-b` puede definirse en términos de `map->` como:

```
(defun my-mapa-b (fn a b &optional (step 1))
  (map-> fn
         a
         #'(lambda(x) (> x b))
         #'(lambda(X) (+ x step))))
```

La función `mapcars` es útil cuando queremos aplicar `mapcar` a varias listas. Las siguientes dos expresiones `(mapcar #'sqrt (append list1 list2))` y `(mapcars #'sqrt list1 list2`, son equivalentes. Sólo que la segunda versión no hace `conses` innecesarios.

Finalmente `rmapcar` es un acrónimo para `mapcar` recursivo:

```
> (rmapcar #'princ '(1 2 (3 4 (5) 6) 7 (8 9)))
123456789
(1 2 (3 4 (5) 6) 7 (8 9))

> (rmapcar #'+ '(1 (2 (3) 4)) '(10 (20 (30) 40)))
(11 (22 (33) 44))
```

6.4 MACROS

La definición de una macro es esencialmente el de una función que genera código Lisp –un programa que genera programas. Este mecanismo ofrece grandes posibilidades, pero también da lugar a problemas inesperados. Este capítulo explica como funcionan las macros, ofrece técnicas para escribirlas y probarlas, y revisa el estilo correcto para su definición.

6.4.1 ¿Cómo funcionan las macros?

Debido a que las macros, al igual que las funciones, pueden invocarse para computar valores de salida, existe una confusión sobre que operadores predefinidos son funciones y cuales macros. La definición de una macro se asemeja a la definición de una función y, de manera informal, incluso los programadores identifican a operadores como `do`, como una función predefinida, cuando en realidad es una macro. Sin embargo, llevar la analogía entre macros y funciones demasiado lejos, puede ocasionar problemas. Las macros tienen un funcionamiento diferente al de las funciones y entender tales diferencias es clave para usar las macros correctamente. Una función produce un resultado, pero una macro produce una expresión que al ser evaluada produce un resultado.

Veamos un ejemplo. Supongan que queremos escribir la macro `nil!` que asigna a su argumento el valor `nil`. Queremos que `(nil! x)` tenga el mismo efecto que `(setq x nil)`. Lo que hacemos es definir `nil!` como una macro que trasforma casos de la primera forma en casos de la segunda forma:

```
> (defmacro nil! (var)
    (list 'setq var nil))
```

```
NIL!
> (nil! x)
NIL
> x
NIL
```

Parafraseada al español, la definición anterior le dice a Lisp "Siempre que encuentres una expresión de la forma `(nil! var)`, conviertela en una expresión de la forma `(setq var nil)` y entonces procede a evaluar la expresión resultante".

La expresión generada por la macro será evaluada en lugar de la llamada original a la macro. Una llamada a una macro es una lista cuyo primer elemento es el nombre de la macro. ¿Qué sucede cuando llamamos a la macro con `(nil! x)` en el toplevel? Lisp identifica a `nil!` como una macro y:

- Construye la expresión especificada por la definición de la marco, y entonces
- Evalúa la expresión en lugar de la llamada original a la macro.

El paso que construye la nueva expresión a ser evaluada se llama *macro-expansión*. Lisp busca en la definición de `nil!` la forma de transformar la llamada original a la macro en una expresión de remplazo. La definición de la macro se aplica a los argumentos de la llamada de manera habitual. En nuestro ejemplo, esto produce la lista `(setq x nil)`.

Tras la macro-expansión, el segundo paso es la *evaluación*. Lisp evalúa la macro-expansión resultante del primer paso, en nuestro ejemplo `(setq x nil)`, y la evalua como si el programador la hubiese tecleado. Aunque, la evaluación no siempre se produce inmediatamente después de la expansión, como sucede normalmente en el toplevel. Una llamada a una macro que ocurre en la definición de una función será expandida cuando la función sea compilada, pero la expansión – o la expresión que resulta de la expansión, no será evaluada hasta que la función sea llamada.

Muchas de las dificultades propias del uso de las macros pueden evitarse si se tiene clara la diferencia entre macro-expansión y evaluación. Al escribir macros, se debe identificar que computaciones serán ejecutadas en fase de macro-expansión y cuales durante la evaluación. La macro-expansión trabaja con expresiones, y la evaluación lo hace con sus valores.

Algunas veces, la macro-expansión puede ser más complicada que con `nil!`. La expansión de `nil!` era una llamada a una forma especial pre-definida, pero en algunas ocasiones la expansión de una macro puede producir otra macro, como una matrushka rusa. En esos casos, la expansión continua hasta que se produce una expresión que no es una llamada a una macro. El proceso puede tener cuantos pasos sean necesarios, a condición de que eventualmente termine.

Muchos lenguajes ofrecen alguna forma de macro, pero el mecanismo ofrecido por Lisp es en particular poderoso. Cuando un archivo de Lisp es compilado, un parser lee el código en él y envía la salida al compilador. He aquí el truco: la salida del parser consiste en una lista de objetos Lisp. Con las macros, podemos manipular el programa mientras está en esta forma intermedia entre el parser y el compilador. Si es necesario, esta manipulación puede ser muy extensa. Una macro generando su expansión tiene a su disposición todo el repertorio de Lisp. De hecho, una macro es realmente una función Lisp que regresa expresiones. La definición de `nil!` contiene únicamente una llamada a `list`, pero es posible que la definición de una macro utilice subprogramas completos para generar la expansión deseada.

La capacidad de cambiar lo que el compilador ve, es casi como tener la capacidad de reescribir el compilador. Podemos agregar cualquier constructor al lenguaje que pueda ser definido mediante transformaciones a constructores existentes.

6.4.2 Backquote

El *backquote* es una versión especial de nuestro conocido *quote* que puede ser usado para definir moldes de expresiones Lisp. Su uso más común está en la definición de macros. Cuando hacemos apóstrofo invertido sobre una expresión, se comporta igual que quote: `'(a b c)` es equivalente a `‘(a b c)`. Backquote se vuelve útil cuando se usa conjuntamente con coma "`,`" y coma-arroba "`,@`". Si el apóstrofo invertido es usado para crear un molde, la coma crea las ranuras en el molde. Una lista bajo apóstrofo invertido es equivalente a una llamada a `list` con sus argumentos bajo quote. Esto es:

```
> (list 'a 'b 'c)
(A B C)
> `(a b c)
(A B C)
```

Bajo el alcance de apóstrofo invertido, una coma le dice a Lisp "deten el efecto de quote". Cuando una coma aparece antes de un elemento de una lista, tiene el efecto de cancelar el quote, de forma que:

```
> (setf a 1 b 2 c 3)
3
> `(a (,b c))
(A (2 C))
```

El apóstrofo invertido se usa normalmente para construir listas. Cualquier lista generada de esta forma, puede generarse también usando `list` y quotes regulares. La ventaja del apóstrofo invertido es que hace que las expresiones sean más fáciles de leer, debido a que la expresión con apóstrofo invertido es similar a su macro-expansión. Vean las definiciones de `nil!`

```
(defmacro nil! (var)
  (list 'setq var nil))
(defmacro nil! (var)
  `(setq ,var nil))
```

Aunque en el ejemplo la diferencia es mínima, entre más grande sea la definición de una macro, más relevante es el uso de apóstrofo invertido para hacer su lectura más clara. Veamos un segundo ejemplo más complicado, un if numérico donde el primer argumento debe evaluar a un número y los otros tres argumentos son evaluados dependiendo si el número fue positivo, cero o negativo, respectivamente. Por ejemplo:

```
(mapcar #'(lambda(x)
  (nif x 'p 'c 'n))
  '(0 2.5 -8))
```

La versión con apóstrofo invertido es:

```
(defmacro nif (expr pos zero neg)
  `(case (truncate (signum ,expr))
     (1 ,pos)
     (0 ,zero)
     (-1 ,neg)))
```

La versión sin apóstrofo invertido es como sigue:

```
(defmacro nif (expr pos zero neg)
   (list 'case
      (list 'truncate (list 'signum expr))
      (list 1 pos)
      (list 0 zero)
      (list -1 neg)))
```

La coma-arroba es una variante del operador coma. Funciona como la coma normal, sólo que en lugar de insertar el valor de la expresión que antecede, una coma-arroba inserta tal valor removiendo sus paréntesis más externos:

```
> (setq b '(1 2 3))
(1 2 3)
> `(a ,b c)
(A (1 2 3) C)
> `(a ,@b c)
(A 1 2 3 C)
```

Observen que la coma causa que la lista `(1 2 3)` sea insertada en el lugar de la `b`, mientras que la coma-arroba, hace que los elementos de la lista `1 2 3` sean insertados en el lugar de `b`. Existen restricciones adicionales en el uso de coma-arroba:

- Para que sus argumentos sean insertados, la coma-arroba debe ocurrir dentro de una secuencia.
- El objeto a insertar debe ser una lista, o en caso contrario, ser insertado al final de la secuencia destino.

El operador coma-arroba se usa normalmente en macros que toman un número no determinado de argumentos y los pasan a funciones o macros que a su vez reciben un número indeterminado de argumentos. Esta situación es común al definir bloques implícitos. Lisp provee diversos operadores para agrupar código en bloques, incluyendo `block`, `tagbody`, y el más conocido `progn`. Estos operadores rara vez se usan directamente en un programa por lo que se dice que son implícitos –escondidos por las macros.

Un bloque implícito ocurre en cualquier macro predefinida que tenga un cuerpo de varias expresiones. Tanto `let` como `cond` proveen un `progn` explícito. Posiblemente la macro más simple que hace esto es `when`:

```
(when (test)
  (funcion1)
  (funcion2)
  obj)
```

si el `test` es verdadero, las expresiones en el cuerpo de la macro son ejecutadas secuencialmente y se regresa el valor de la última expresión `obj`. Como un ejemplo del uso de coma-arroba, definiremos nuestro propio `mi-when`:

```
(defmacro mi-when (test &body body)
  '(if ,test
    (progn ,@body)))
```

el parámetro `&body` toma un número arbitrario de argumentos y el operador coma-arroba los inserta en un sólo `progn`. La mayoría de las macros de iteración insertan sus argumentos de esta forma.

Aunque las macros normalmente construyen listas, también pueden regresar funciones. Finalmente, el apóstrofo invertido puede usarse en cualquier expresión Lisp,no sólo en las macros:

```
(defun hola (nombre)
   `(hola ,nombre))
```

6.4.3 Definiendo macros simples

Comencemos por escribir una macro que sea una variante de la función predefinida `member`. Por default, ésta función utiliza `eql` para probar igualdad. Si se quiere probar membresía usando `eq`, esto debe indicarse explícitamente:

```
(member obj lst :test #'eq)
```

Si hacemos esto muchas veces, nos gustaría tener una variante de `member` que siempre use `eq`. Aunque normalmente definiríamos esta versión como una función *inline* (ver clase anterior), su definición es un buen ejercicio sobre codificación de macros.

El método para definir una macro es como sigue: se comienza con la llamada a la macro que queremos definir. Escríbanla en un papel y abajo escriban la expresión que quieren producir con la macro:

```
llamada: (mem-eq obj lst)
expansión: (member obj lst :test #'eq)
```

La llamada nos sirve para definir los parámetros de la macro. En este caso, como necesitamos dos argumentos, el inicio de la macro será:

```
(defmacro mem-eq (obj lst)
```

Ahora volvamos a las dos expresiones iniciales. Para cada argumento en la llamada a la macro, tracen un línea hacía donde son insertadas en la expansión. Para escribir el cuerpo de la macro presten atención a estas líneas paralelas. Comiencen el cuerpo con un apóstrofo invertido. Ahora lean la expansión expresión por expresión. Donde quiera que encuentre un paréntesis que no es parte de los argumentos de llamada de la macro, pongan un paréntesis en la definición de la macro. Así tenemos que para cada expresión:

1. Si no hay una línea conectado la expresión en la llamada, entonces escribir la expresión tal cual en el cuerpo de la macro.
2. Si hay una conexión, escriban la expresión precedida por una coma.

```
(defmacro mem-eq (obj lst)
  `(member ,obj ,lst :test #'eq))
```

Hasta ahora hemos escrito macros que tienen un número determinado de argumentos. Ahora supongan que queremos escribir la macro `while` que toma una expresión de prueba `test` y algún `cuerpo` que ejecutará repetidamente mientras el `test` sea verdadero. Esta macro requiere modificar el método anterior ligeramente. Comencemos por escribir la llamada a la macro:

```
(defmacro while (test &body body)
```

Ahora escriban la expansión deseada y como el caso anterior, sin embargo cuando tengan una secuencia de argumentos bajo `rest` o `body`, trátenlos como un grupo, dibujando una sola línea para toda la secuencia. Y claro, al insertar la expresión en la definición de la macro, hay que recurrir a coma-arroba:

```
(defmacro while (test &body body)
  `(do ()
       ((not ,test))
     ,@body))
```

6.4.4 Probando la expansión de las macros

Una vez que hemos escrito una macro ¿Cómo podemos probarla? Las macros simples, como `mem-eq` pueden probarse directamente, observando si su salida es la esperada. Para macros más complejas, es necesario poder observar si la expansión ha sido correcta. Para ello lisp provee las funciones predefinidas `macroexpand` y `macroexpand-1`. La primera muestra como la macro se expandiría antes de ser evaluada. Si la macro hace uso de otras macros, esta revisión de la expansión es de poca utilidad. La función `macroexpand-1` muestra la expansión de un sólo paso. Veamos un ejemplo basado en `while`:

```
CL-USER 2 > (pprint(macroexpand '(while (puedas) (rie))))
(BLOCK NIL
  (LET ()
    (DECLARE (IGNORABLE))
    (DECLARE)
    (TAGBODY
     #:G747  (WHEN (NOT (PUEDAS)) (RETURN-FROM NIL NIL))
             (RIE)
             (GO #:G747))))

CL-USER 3 > (pprint (macroexpand-1 '(while (puedas) (rie))))
(DO () ((NOT (PUEDAS)))  (RIE))
```

Observen que la expansión completa es más difícil de leer, mientras que la producida por `macroexpand-1` es más útil en este caso. La expansión depende de la implementación de Lisp que estén usando, en este caso LispWorks 5.1.2. Otros Lisp pueden implementar `do` en términos de `unless` en lugar de `unless`. Si vamos a hacer esto muchas veces, nos convine definir una macro:

```
(defmacro mac (expr)
  `(pprint (macroexpand-1 ',expr)))
```

de forma que podemos evaluar ahora:

```
CL-USER> (mac (while (puedas) rie))
(DO () ((NOT (PUEDAS)))  RIE)
; No value
```

La expansión obtenida puede reevaluarse en el `TOP-LEVEL` para experimentar con la macro:

```
> (setq aux (macroexpand-1 '(mem-eq 'a '(a b c))))
(MEMBER 'A '(A B C) :TEST #'EQ)
> (eval aux)
(A B C)
```

Estas herramientas no sólo son útiles para probar las macros, sino para aprender a escribir macros correctamente. Como he mencionado, numerosas facilidades de Lisp están implementadas como macros. Es posible entonces usar `macroexpand-1` para ver que forma intermedia generan esas expresiones!

```
> (mac (when t nil))
(IF T (PROGN NIL))
; No value
```

6.4.5 Ejemplos

Una pequeña librería de macros:

```
(defmacro for (var start stop &body body)
  (let ((gstop (gensym)))
    `(do ((,var ,start (1+ ,var))
          (,gstop ,stop))
         ((> ,var ,gstop))
       ,@body)))

(defmacro in (obj &rest choices)
  (let ((insym (gensym)))
    `(let ((,insym ,obj))
       (or ,@(mapcar #'(lambda(c) `(eql ,insym ,c))
                     choices)))))
```

```
13
14 (defmacro random-choice (&rest exprs)
15   `(case (random ,(length exprs))
16      ,@(let ((key -1))
17          (mapcar #'(lambda(expr)
18                      `(,(incf key) ,expr))
19                  exprs))))
20
21 (defmacro avg (&rest args)
22   `(/ (+ ,@args) ,(length args)))
23
24 (defmacro with-gensym (syms &body body)
25   `(let ,(mapcar #'(lambda(s)
26                      `(,s (gensym)))
27                  syms)
28      ,@body))
29
30 (defmacro aif (test then &optional else)
31   `(let ((it ,test))
32      (if it ,then ,else)))
```

y sus corridas:

```
1  > (for x 1 8 (princ x))
2  12345678
3  NIL
4  > (in 3 1 2 3)
5  T
6  > (random-choice 1 2 3)
7  1
8  > (random-choice 1 2 3)
9  3
10 > (random-choice 1 2 3)
11 3
12 > (random-choice 1 2 3)
13 1
14 > (random-choice 1 2 3)
15 1
16 > (random-choice 1 2 3)
17 2
18 > (avg 2 4 8)
19 14/3
```

6.5 LECTURAS Y EJERCICIOS SUGERIDOS

McCarthy [55] introdujo Lisp como un lenguaje de programación simbólica para la IA. El manual de Lisp 1.5 [56] fue publicado dos años más tarde. También publicó una historia de Lisp [57] que da cuenta de la génesis del lenguaje. En ese sentido, Costanza y col. [22] publicaron un artículo sobre los 50 años de Lisp, lo que lo hace uno de los lenguajes activos más antiguos.

Quizá el libro de introducción más próximo a nuestros intereses sea el de Norvig [65], por tratarse de un estudio de los paradigmas de la IA implementados en Lisp. El mismo nos ofrece un tutorial de lo que constituye un buen estilo de programación en este lenguaje [66]. Si se quiere una introducción más general, Graham [34] nos presenta detalladamente el estándar ANSI de Common Lisp y un compendio de técnicas avanzadas en este lenguaje [35]. Otras introducciones interesantes, incluyen las de Shapiro [84], Touretzkty [91], y Cooper [21]. Una introducción a Lisp más reciente y generalista es la de Seibel [82].

Gabriel [29] aborda el desempeño del lenguaje y la evaluación de sistemas implementados en Lisp. Bobrow y col. [6] nos presenta CLOS, la especificación del sistema orientado a objetos de Common Lisp. Kiczales, Rivières y Bobrow [43] introducen el concepto del protocolo meta-objetos. Queinnec [70] revisa los detalles de implementación de Lisp.

De especial interés para este curso, resulta el texto de Warren, Pereira y Pereira [95] comparando las implementaciones de Prolog y Lisp. Una revisión parecida es presentada por O'Keefe [67]. Cattaneo y Loia [11] nos presenta una implementación de un Prolog extendido en Common Lisp. Finalmente, Gat [30] hace una comparativa entre Lisp y Java.

Ejercicios

Ejercicio 6.1. *Implemente en Lisp las siguientes operaciones sobre conjuntos representados como listas:*

- *Subconjunto:*

```
> (subset '(1 3) '(1 2 3 4))
true.
> (subset '() (1 2)).
true.
```

- *Intersección:*

```
> (inter '(1 2 3) '(2 3 4)).
(2 3)
```

- *Unión:*

```
> (union '(1 2 3 4) '(2 3 4 5))
(1 2 3 4 5)
```

- *Diferencia:*

```
> (dif '(1 2 3 4) '(2 3 4 5))
(1)
> (dif '(1 2 3) '(1 4 5))
(2 3)
```

Ejercicio 6.2. *Escriba un programa que tome una lista de números como primer argumento y regrese una lista de los mismos números incrementados en 1. Por ejemplo:*

```
> (inc '(1 2 3 4))
(2 3 4 5)
```

Ejercicio 6.3. *Escriba un programa que regrese en su segundo argumento la lista de todas las permutaciones de la lista que es su primer argumento. Por ejemplo:*

```
> (perms '(1 2 3))
((1 2 3) (1 3 2) (2 3 1) (2 1 3) (3 1 2) (3 2 1))
```

Ejercicio 6.4. *Escriba un programa en que elimine todas las ocurrencias de un elemento (primer argumento) en una lista (segundo argumento); y regrese la lista filtrada como tercer argumento. Por ejemplo:*

```
> (eliminar 3 '(1 3 2 4 5 3 6 7))
(1 2 4 5 6 7)
```

Ejercicio 6.5. *Defina una representación basa en listas para árboles binarios. Defina las siguientes operaciones:*

- *Obtener la raíz de un árbol.*

- *Obtener la rama izquierda de un árbol.*
- *Obtener la rama derecha de un árbol.*
- *Crear un árbol a partir de un elemento, rama izquierda y rama derecha.*
- *Agregar un elemento a un árbol dado.*
- *Convertir una lista en árbol binario.*
- *Recorrer un árbol en orden infijo, prefijo y postfijo.*