# El Lenguaje Common Lisp

Manuel Vilares Ferro

Ciencias de la Computación e Inteligencia Artificial
Universidad de Vigo
Campus As Lagoas, s/n. 32004 Orense

# Índice general

# Capítulo 1

# Introducción

## 1.1. Programación funcional

Los lenguajes de programación funcional hacen uso de las propiedades matemáticas de las funciones. De hecho, el nombre de *funcional* se deriva del papel predominante que juegan dichas estructuras y su aplicación.

### 1.1.1. Funciones: respuestas y efectos de borde

- Una *función* es una regla de correspondencia de miembros de un conjunto, denominado *dominio*, con miembros de otro conjunto, denominado *rango*; de modo que a cada elemento del dominio se le asocia uno y sólo uno del rango.

$$\begin{array}{rcccc} f & : & \text{Dominio} & \longrightarrow & \text{Rango} \\ & & x & \longmapsto & f(x) \end{array}$$

- En la definición de una función se especifica su dominio, su rango y su regla de correspondencia.
- Las funciones matemáticas se definen expresando su regla de correspondencia en términos de la aplicación de otras funciones.

  La esencia de la programación funcional es combinar funciones para producir a su vez funciones más potentes.
- En la adaptación del concepto de función a los lenguajes de programación, se conoce como *respuesta* a la aplicación de la definición de la función a un elemento del dominio, esto es, a *f(x)*.

  En contraposición, se conoce como *efecto de borde* a cualquier cálculo efectuado durante la evaluación de la función[1], que pueda afectar al entorno de programación[2] no local a la función.

[1] es decir, el cálculo de la respuesta.

[2] esto es, al conjunto de variables asignadas en un momento dado de la sesión de trabajo.

Un lenguaje exclusivamente funcional no incluye estructuras posibilitando efectos de borde. En relación a ello, podemos establecer que una diferencia fundamental entre un lenguaje imperativo y uno funcional radica en que, mientras en el primero es imprescindible el uso de la asignación para almacenar los resultados intermedios de las operaciones, en el segundo se devuelven como el resultado de la aplicación de funciones. Esto es, no es necesario su almacenamiento explícito en variables intermedias.

- Otro rasgo importante de los lenguajes funcionales es su carácter *recursivo natural*, esto es, la definición de conceptos en función de ellos mismos.
- Un lenguaje de programación funcional tiene básicamente cuatro componentes:
  - Un conjunto de funciones primitivas, que constituyen el núcleo del sistema.
  - Un conjunto de formas funcionales, que permiten ampliar el lenguaje a partir de las primitivas.
  - La operación *aplicación*, que posibilita la evaluación de las funciones.
  - Un conjunto de objetos o datos.

### 1.1.2. Limitaciones

Se derivan de la imposibilidad física de simular ciertos procesos desde un punto de vista exclusivamente funcional. En efecto, problemas diversos exigen la inclusión del concepto de *efecto de borde*, como pueden ser el funcionamiento de los canales de entrada-salida de un *device*:

- Interfaces gráficas.
- Protocolos de comunicaciones.

## 1.2. Programación funcional en Lisp

Lisp no es un lenguaje funcional puro, incluyendo la posibilidad de considerar *efectos de borde*. En relación a las componentes del lenguaje:

- Las *funciones primitivas* son aquellas que aporta incialmente el lenguaje y no implementables en el lenguaje mismo: `car`, `cdr`, `cons`.
- Las *formas funcionales* son el mecanismo por el cual al aplicar varias funciones, combinándolas de forma adecuada, se puede definir cualquier otra: `defun` y `defmacro`.

- La *operación de aplicación* es una función primitiva esencial en el funcionamiento del motor del lenguaje, ya que entre otras cosas es la encargada de gestionar y realizar el proceso de interpretación: `funcall`, `apply`.
- Por último, los *objetos* que conforman el lenguaje son esencialmente símbolos que serán definidos más tarde.

### 1.2.1. Programación orientada a listas

- Las *listas* son estructuras de datos dinámicas, esto es, su talla puede variar en el curso de la sesión de trabajo.
- Son además estructuras de tipo secuencial. Esto es, para acceder a un elemento contenido en una lista es necesario recorrer previamente todos los que le preceden.
- En Lisp, las listas son las estructuras fundamentales de manejo de datos.

### 1.2.2. Los programas como estructuras de datos

- Una de las características más importantes de Lisp es la de no hacer distinción entre código ejecutable y datos.
- De hecho, las expresiones Lisp son listas, y por tanto manipulables como los datos.
- Si bien los lenguajes de programación imperativa hacen una distinción radical entre programa y datos, la característica expuesta de Lisp no tiene nada de revolucionario puesto que a nivel del ordenador tal diferencia no existe.
- Dicha característica hace de Lisp un lenguaje bien adaptado a lo que se ha dado en llamar *inteligencia artificial*, con posibilidades amplias de abordar problemas relacionados con el aprendizaje automático.

# Capítulo 2

# Primeros pasos en Lisp

Existe toda una variedad de intérpretes Lisp en el mercado: LeLisp, Common Lisp, Allegro, Emacs Lisp, ... Todos ellos mantienen un estilo de programación común que les define y que, con raras excepciones puntuales, gira en torno a un estandar sintáctico. Ello permite afirmar que el contenido del presente texto es aplicable a todos ellos, aunque por razones prácticas nos centremos en el caso concreto Common Lisp.

## 2.1. Entorno operativo e instalación del intérprete

Aunque la inmensa mayoría de intérpretes Lisp están disponibles en diferentes sistemas operativos, nuestra plataforma por defecto será Linux. Ello nos permitirá, en concreto, acceder a una de las implementaciones Common Lisp más populares, la conocida como gnu clisp, nuestra elección.

En este contexto, la instalación de gnu clisp es trivial a partir de la herramienta estandar de administración del sistema para la gestión de paquetes, esto es, de Synaptic. Basta con teclear clisp en el recuadro de búsqueda, marcarlo para su instación en los resultados obtenidos y proceder a la misma.

## 2.2. Arrancando el intérprete

Obviamente, en este punto, la sintaxis sí depende del intérprete particular que utilicemos y del entorno operativo considerado. En nuestro caso, centrándonos en gnu clisp, el arranque se realiza desde un `xterm` mediante el comando `clisp`:

```
$ clisp
  i i i i i i i       ooooo    o        ooooooo   ooooo   ooooo
```

```
  I I I I I I I      8     8   8           8     8     o  8    8
  I  \ `+' /  I      8         8           8     8        8    8
   \  `-+-'  /       8         8           8      ooooo   8oooo
    `-__|__-'        8         8           8           8  8
        |            8     o   8           8     o     8  8
  ------+------       ooooo    8ooooooo  ooo8ooo   ooooo   8

Welcome to GNU CLISP 2.48 (2009-07-28) <http://clisp.cons.org/>

Copyright (c) Bruno Haible, Michael Stoll 1992, 1993
Copyright (c) Bruno Haible, Marcus Daniels 1994-1997
Copyright (c) Bruno Haible, Pierpaolo Bernardi, Sam Steingold 1998
Copyright (c) Bruno Haible, Sam Steingold 1999-2000
Copyright (c) Sam Steingold, Bruno Haible 2001-2009

Type :h and hit Enter for context help.

[1]>
```

donde el símbolo > es el *prompt* del intérprete. A partir de este momento, podemos ya evaluar expresiones Lisp

```
[1]> (defvar a 5)
5
[2]> a
5
```

lo que también incluye la posibilidad de definir funciones

```
[3]> (defun suma (a b) (+ a b))
SUMA
```

y de evaluarlas

```
[4]> (suma 2 3)
5
```

tal y como corresponde a un lenguaje interpretado. Al respecto, las expresiones Lisp pueden ser igualmente cargadas en el intérprete desde cualquier fichero, mediante la función predefinida `load`, indicando como argumento el camino hacia el mismo. La extensión habitual para ficheros fuente en los intérpretes COMMON LISP es `.cl`, aunque no es necesario explicitarla en el momento de la carga. Supongamos, por ejemplo, que he editado el siguiente fichero `mult.cl` en el directorio `~/lisp/CLisp/examples`

```
(defun mult (a b)
       (* a b))
```

y que yo he lanzado mi intérprete GNU CLISP desde el directorio ∼/lisp/CLisp/doc/Apuntes, esto es, un directorio diferente al de edición. Entonces, se producirá un error al intentar cargarlo sin indicar el camino de acceso

```
[5]> (load mult)

*** - SYSTEM::READ-EVAL-PRINT: variable MULT has no value
Es posible continuar en los siguientes puntos:
USE-VALUE      :R1      Input a value to be used instead of MULT.
STORE-VALUE    :R2      Input a new value for MULT.
ABORT          :R3      Abort main loop
Break 1 [6]>
```

al no coincidir el directorio de carga del fichero con el de lanzamiento del intérprete. De esta forma, el modo correcto de carga sería

```
[1]> (load "~/lisp/CLisp/examples/mult")
;; Loading file /home/vilares/lisp/CLisp/examples/mult.cl ...
;; Loaded file /home/vilares/lisp/CLisp/examples/mult.cl
T
```

donde `T` es un valor predefinido que coincide con el *true* lógico. En este caso, indica que el proceso de carga ha finalizado correctamente y que ya es posible evaluar cualquier expresión contenida en el fichero

```
[2]> (mult 2 3)
6
```

Amén de la vertiente interpretada, todos los entornos Lisp permiten la "compilación"[1] de sus programas, aunque en este punto hay diferencias sensibles. Así, por ejemplo, ALLEGRO compila automáticamente las fuentes una vez se cargan mediante `load` desde el fichero. En nuestro caso, sin embargo, el resultado por defecto de la carga de un fichero es su interpretación. La compilación es aquí opcional y debe requerirse de forma explícita mediante la función predefinida `compile-file`, considerando el nombre del fichero que contiene las expresiones a compilar, como argumento:

[1] el concepto de compilación aquí no se corresponde necesariamente con el de los lenguajes imperativos puesto que algunas de las funcionalidades de un lenguaje interpretado se perderían en una estrategia clásica de compilación llevada hasta sus últimas consecuencias. En particular, podemos referirnos aquí al tratamiento de facilidades ligadas al tratamiento dinámico de la información, como pueden ser los programas que se modifican a si mismos, algo incompatible con los procesos de compilación enteramente estáticos.

```
[1]> (compile-file "~/lisp/CLisp/examples/mult")
;; Compiling file /home/vilares/lisp/CLisp/examples/mult.cl ...
;; Wrote file /home/vilares/lisp/CLisp/examples/mult.fas
0 errores, 0 advertencias
#P"/home/vilares/lisp/CLisp/examples/mult.fas" ;
NIL ;
NIL
```

lo que genera en el directorio del fichero origen otro con el mismo nombre, pero extensión `.fas`, y que contiene el código compilado de las expresiones del original. En este caso, la función de compilación ha devuelto `nil`, un valor predefinido asimilable a un símbolo, a una lista vacía o a un valor booleano equivalente al clásico *false*. Ahora podemos cargar ese fichero mediante `load`, aunque para evitar confusiones deberíamos hacer explícita la extensión `.fas`:

```
[2]> (load "~/lisp/CLisp/examples/mult.fas")
;; Loading file /home/vilares/lisp/CLisp/examples/mult.fas ...
;; Loaded file /home/vilares/lisp/CLisp/examples/mult.fas
T
```

y como antes en la versión interpretada, ya tenemos acceso a las expresiones Lisp contenidas en el fichero:

```
[2]> (mult 2 3)
6
```

aunque ahora ya en forma compilada y no interpetada. Por otra parte, basta editar el fichero `mult.fas` generado en el proceso, para darse cuenta de que el concepto de compilación Lisp dista mucho de coincidir con el clásico en lenguajes imperativos.

Alternativamente, también es posible compilar una función de forma aíslada y previamente cargada en el sistema en su forma interpretada, mediante la función `compile` y teniendo como argumento el nombre de la función. Supongamos, por ejemplo, la siguiente función, incluida en el fichero `tedioso.cl` del directorio ∼`/lisp/CLisp/examples`:

```
(defun tedioso (n)
       ; un bucle desde i=0 hasta i=n-1
       (dotimes (i n)   ; loop desde i=0 hasta i= n-1
                (if (> (* i i) n) (return i))))
```

donde ";" precede a una línea de comentario. Carguémoslo ahora en el sistema como interpretado

```
[2]> (load "~/lisp/CLisp/examples/tedioso")
;; Loading file /home/vilares/lisp/CLisp/examples/tedioso.cl ...
;; Loaded file /home/vilares/lisp/CLisp/examples/tedioso.cl
T
```

apliquemos un test simple para apreciar su rendimiento, por medio del tiempo de ejecución

```
[3]> (times (tedioso 5000000))
                                          Permanent            Temporary
Class                                  instances   bytes    instances   bytes
-----                                  --------- ---------  --------- ---------
-----                                  --------- ---------  --------- ---------
Total                                          0         0          0         0
Real time: 0.002372 sec.
Run time: 0.0 sec.
Space: 0 Bytes
2237
```

compilemos ahora la función `tedioso` directamente

```
[4]> (compile 'tedioso)
TEDIOSO ;
NIL ;
NIL
```

donde `tedioso` va precedida de una *quote* para ser evaluada como símbolo y no como variable. Volvamos ahora a aplicar el mismo test de rendimiento que antes

```
[5]> (times (tedioso 5000000))
                                          Permanent            Temporary
Class                                  instances   bytes    instances   bytes
-----                                  --------- ---------  --------- ---------
-----                                  --------- ---------  --------- ---------
Total                                          0         0          0         0
Real time: 3.13E-4 sec.
Run time: 0.0 sec.
Space: 0 Bytes
2237
```

resulta evidente el impacto de la compilación en la eficacia computacional del proceso de ejecución.

Finalmente, indicar una función predefinida esencial del sistema, la que permite la salida del intérprete, la función `quit`.

```
[6]> (quit)
Adios.
$
```

y que nos devuelve al *prompt*, $, de Linux.

# Capítulo 3

# Conceptos básicos del lenguaje

## 3.1. Objetos en Lisp

Un objeto Lisp es cualquier estructura susceptible de ser manipulada por el lenguaje. En atención a su naturaleza distinguiremos entre *objetos atómicos* y *objetos compuestos*.

### 3.1.1. Objetos atómicos

Un *objeto atómico*, o *átomo*, es un objeto irreducible del lenguaje, pudiendo ser de tipo numérico o simbólico:

- Los *símbolos* juegan el papel de identificadores, pudiéndo asociársele un valor.

  Un símbolo puede asociar una variable, una función o una lista de propiedades[1], sin exclusión mútua[2]. El valor de un símbolo como tal es él mismo:

  ```
  [1]> (type-of 'a)
  SYMBOL
  [2]> 'a
  A
  ```

  Por defecto los átomos simbólicos tienen un valor como variable igual al booleano *false*[3].

[1] a definir más adelante.

[2] esto es, un símbolo puede referirse al mismo tiempo a una variable, una función y una lista de propiedades.

[3] también representado por `nil` o la lista vacía ().

- Los *átomos numéricos* se dividen a su vez en enteros y reales. El valor de un átomo numérico es él mismo:

```
[1]> (setq a 6)
6
[2]> a
6
[3]> (type-of a)
(INTEGER 0 281474976710655)
[4]> (type-of 'a)
SYMBOL
```

- Los *tableros*. En la práctica, la definición de un átomo como objeto irreducible equivale a afirmar que un átomo es cualquier objeto Lisp diferente de una lista. Ello incluye a los vectores. El concepto es el mismo conocido y aplicado en los lenguajes imperativos clásicos:

  - Un tablero es esencialmente una dirección en la memoria, que referencia su primer elemento, y una talla indicando el número de elementos del tablero.
  - La diferencia en relación a las listas estriba en que en aquellas, el acceso a los componentes era secuencial, mientras que en los tableros es directo.

  Formalmente, en Lisp un tablero se representa mediante una sequencia entre paréntesis, y precedida por el símbolo `#`, de elementos separados por espacios en blanco. Como ejemplo consideremos la definición del tablero de elementos `1`, `b`, `''hola''` y `2`:

```
[13]> (setq a '#(1 a "hola" 2))
#(1 A "hola" 2)
[14]> (type-of a)
(SIMPLE-VECTOR 4)
```

  los elementos del tablero pueden accederse a partir de su posición en la secuencia, iniciándose la numeración de éstas posiciones en 0 (y no en 1). La función para accederlas es `aref`, que toma como argumentos al propio tablero y a la posición que deseamos acceder:

```
[15]> (aref a 0)
1
```

de igual modo, podemos asignar el valor de una celda dada mediante `setf` y tomando como argumento la celda y el valor asignado:

```
[16]> (setf (aref a 0) 3)
3
[17]> (aref a 0)
3
```

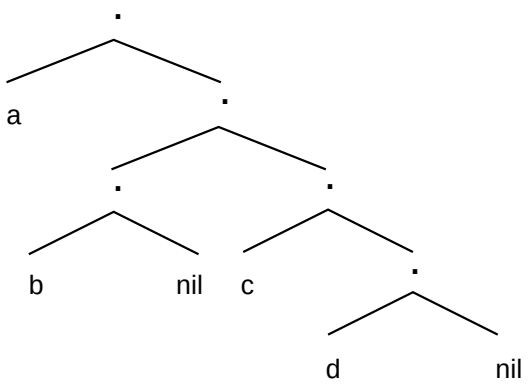
Cuando los tableros son unidimensionales, hablamos habitualmente de *vectores*.

### 3.1.2. Listas

La implementación física de la estructura de datos *lista* se centra en el concepto de *célula* o *doblete*. Elemento mínimo de una lista, la célula está compuesta de dos partes; el `car` y el `cdr`:

- El `car` de una célula se corresponde al contenido de la dirección memoria de dicha célula. Esto es, el `car` de una lista es su primer elemento.
- El `cdr` de una célula es el contenido de la dirección siguiente a la representada por el `car`. Esto es, el `cdr` de una lista es la lista una vez eliminado su primer elemento.
- En el caso de Lisp, la estructura de célula se representa mediante el operador · en la forma indicada en la figura 3.1.

Figura 3.1: Representación interna de la lista *(a (b) c d)*




Formalmente, una lista Lisp es una sequencia entre paréntesis de elementos separados por espacios en blanco. Como ejemplo consideremos la definición de la lista de elementos `1`, `b`, `‘‘hola’’` y `2`:

```
[6]> (setq a '(1 b "hola" 2))
(1 B "hola" 2)
[7]> (type-of a)
CONS
```

donde podemos observar la mezcolanza de tipos, así como el hecho de que `cons` es el tipo de un objeto lista. Otra cuestión importante es que, de nuevo, hemos usado una *quote* para omitir la acción del evaluador Lisp. De hecho, si no hubiésemos usado una *quote*, el resultado sería el error ilustrado en el siguiente ejemplo:

```
[8]> (1 b "hola" 2)

*** - EVAL: 1 is not a function name; try using a symbol instead
Es posible continuar en los siguientes puntos:
USE-VALUE      :R1      Input a value to be used instead.
ABORT          :R2      Abort main loop
Break 1 [9]>
```

al pretender evaluar `1` como función con tres argumentos. Al respecto, la expresión `Break 1` simplemente indica que hemos reentrado en el bucle del intérprete una vez ha sido detectado un error. Si deseamos volver al anterior nivel del bucle, basta con teclear `abort`:

```
Break 1 [9]> abort
[10]>
```

Las primitivas `car` y `cdr` permiten acceder a los elementos homónimos de la lista en argumento:

```
[11]> (car a)
1
[12]> (cdr a)
(B "hola" 2)
```

Comparando ahora las características de las listas frente a los vectores, podemos apuntar:

- Las listas son adecuadas cuando el acceso directo no es esencial o cuando el factor memoria es determinante por su limitación.
- Los vectores son adecuados cuando la velocidad es primordial y la memoria suficientemente amplia.

## 3.2. Acciones básicas del intérprete

Cuando se teclea sobre el intérprete Lisp una expresión, sobre ella actúa automáticamente el mecanismo evaluador del sistema. Éste devuelve como resultado un objeto Lisp que constituye el *valor* de dicha expresión.

### 3.2.1. Evaluación de objetos atómicos

- La evaluación de un átomo numérico devuelve el propio átomo.
- La evaluación de un átomo simbólico depende del contexto sintáctico en el que se encuentre en el programa.
- Sobre los vectores, la evaluación se limita a una evaluación recursiva sobre cada una de las células del mismo.

de este modo, tenemos que:

```
[1]> 'a
A
[2]> #(1 2)
#(1 2)
[3]> (+ 1 2)
3
[4]> 5
5
```

### 3.2.2. Evaluación de listas

En este caso, el evaluador del intérprete Lisp actúa como sigue:

- Verifica que el primer elemento de la lista es una función, en caso contrario da error.
- Evalúa los argumentos uno a uno de izquierda a derecha según el tipo de función considerado[4], aplicando recursivamente este mismo procedimiento.
- Aplica la función sobre el resultado de la evaluación de cada argumento y devuelve el valor obtenido.

[4] algunas funciones no evalúan sus elementos.

### 3.2.3. Inhibición de la evaluación

A este nivel, surge la necesidad en ciertas ocasiones de inhibir la evaluación. Consideremos dos casos prácticos muy sencillos:

- Algunas funciones toman como argumento una lista, por ejemplo las primitivas `car` y `cdr`. Supongamos la llamada:

```
[18]> (car (cdr (1 a 3)))

*** - EVAL: 1 is not a function name; try using a symbol instead
Es posible continuar en los siguientes puntos:
USE-VALUE      :R1      Input a value to be used instead.
ABORT          :R2      Abort main loop
Break 1 [15]>
```

  su respuesta no es el elemento `a` como cabría esperar, sino un mensaje de error indicando que no existe ninguna función de nombre 1. Esto es, el intérprete ha evalúado `(1 a 3)` como si de una llamada a una función se tratase. La solución al problema planteado estaría en inhibir dicha evaluación. Una posible solución ya ha sido antes comentada, y pasa por anteponer una *quote* a la sequencia, de forma a inhibir su evaluación como función. Así, por ejemplo:

```
[19]> (car (cdr '(1 a 3)))
A
```

- Algunas funciones toman como argumento símbolos que previamente evalúan, como en el caso de `list` que genera una lista a partir de la evaluación de sus argumentos. Así, la llamada

```
[20]> (list 1 a 3)
(1 (1 B "hola" 2) 3)
```

  genera una lista cuyo segundo elemento es el resultado de evaluar la variable `a` que previamente habíamos inicializado

```
[21]> a
(1 B "hola" 2)
```

Si realmente deseamos que el segundo elemento de la sequencia sea el símbolo `a`, la solución pasa por inhibir su evaluación con una *quote*, bien explícitamente

```
[22]> (list 1 'a 3)
(1 A 3)
```

bien utilizando la función `quote`

```
[23]> (list 1 (quote a) 3)
(1 A 3)
```

La combinación de la *backquote* y del carácter coma puede usarse para realizar la evaluación *selectiva* de elementos en una lista:

```
[24]> (setq a 1)
1
[25]> (setq b 2)
2
[26]> (defvar lista `(a ,a b ,b))
(A 1 B 2)
```

### 3.2.4. Evaluación de funciones

Ante todo diferenciaremos entre lo que es la llamada a una función, y su definición:

- La *llamada a una función* en Lisp es, como comentado con anterioridad, una lista donde:
  - El `car` es el nombre de la función.
  - El `cdr` es el conjunto de los argumentos de la llamada a la función.

  de este modo, tenemos que:

  ```
  [30]> (list 1 'a 3 "adios")
  (1 A 3 "adios")
  ```

- La *definición de la función* es una entrada en una tabla de símbolos indexada por el símbolo que da nombre a la misma. Según sea la forma de dicha entrada se evaluará de forma diferente.

En relación al tipo de evaluación aplicada, podemos distinguir en GNU CLISP dos tipos diferenciados de funciones[5]:

- Las funciones que evalúan sus argumentos, que denominamos de tipo `defun`.
- Las funciones que no evalúan sus argumentos, que denominamos de tipo `defmacro`.

Las definiciones de las funciones están implementadas en Lisp mismo, usando una expresión anónima `lambda` de la forma:

(**lambda** lista-de-variables expresión$_1$ ... expresión$_n$)

y ligándose al nombre de la función mediante las formas funcionales `defun` y `defmacro`, según el tipo de evaluación considerada.

Analizaremos ahora cada uno de estos tipos de función por separado, indicando su naturaleza y el interés de su utilización. Con este objetivo y para facilitar la descripción que sigue, introduciremos previamente la función `defmacro` de nombre `ppmx`[6]. Ésta no es estandar en las implementaciones COMMON LISP y el usuario deberá primera verificar su disponibilidad. En el caso concreto de GNU CLISP, el usuario deberá implementarla explícitamente. El código es el siguiente:

```
(defmacro ppmx (forma)
  `(let* ((exp1 (macroexpand-1 ',forma))
          (exp (macroexpand exp1))
          (*print-circle* nil))
      (cond ((equal exp exp1)
             (format t "~&Macro expansion:")
             (pprint exp))
            (t (format t "~&Primer paso de expansion:")
               (pprint exp1)
               (format t "~%~%Expansion final:")
               (pprint exp)))
      (format t "~%~%")
      (values)))
```

y nos permitirá desvelar los pasos en la evaluación de una función, determinados tanto por su definición como por su tipo particular.

[5] en LELISP se consideran hasta cuatro formas diferentes de evaluación de una función.
[6] por Pretty Print Macro eXpansion.

**Funciones de tipo `defun`** El proceso exacto de evaluación es el que sigue:

- Los resultados de evaluar los parámetros de la llamada, son guardados en una pila mientras los nombres de los argumentos son asociados a éstos.
- Las expresiones en el cuerpo de la función son evalúadas, coincidiendo el resultado devuelto por la función con el de la última de dichas evaluaciones.
- Las asignaciones realizadas en el primer paso se deshacen. Los antiguos valores asociados a los nombres de los parámetros, que estaban guardados en una pila, son restaurados. En consecuencia las llamadas son, en las funciones de tipo `defun`, por valor.

Es el tipo más común de función utilizada. La asociación entre símbolo y definición de función se realiza mediante la forma funcional `defun`.

Ilustraremos con un ejemplo el proceso de evaluación, utilizando la `defmacro`, que habremos primero de implementar y cargar en el intérprete, `ppmx`:

```
[1]> (defvar saludo "hola")
SALUDO
[2]> (defun saludo (texto) (print texto))
SALUDO
[3]> (load "~/lisp/CLisp/examples/ppmx")
;; Loading file /home/vilares/lisp/CLisp/examples/ppmx.cl ...
;; Loaded file /home/vilares/lisp/CLisp/examples/ppmx.cl
T
[4]> (ppmx (saludo saludo))
Macro expansion:
(SALUDO SALUDO)
```

lo que viene a ilustrar que la evaluación de una función `defun` se realiza en un solo paso, sin fases intermedias de expansión, algo que las distinguirá de las `defmacro`, como veremos.

**Funciones de tipo `defmacro`** El proceso de evaluación es ahora el siguiente:

- El `cdr` de la forma, sin evaluar, es usado como argumento.
- Una vez realizada una primera evaluación del cuerpo, la forma entera es físicamente reemplazada por el valor resultante de la primera evaluación.

Como consecuencia, debemos evitar la definición de macros recursivas. El uso de las funciones de tipo `defmacro` suele justificarse por dos razones:

- Implementación de pequeños fragmentos de código, en los que por lo exíguo de su talla desea evitarse el costo adicional que representa la utilización de funciones de tipo `defun`. Será, típicamente, el caso de las funciones aplicativas como, por ejemplo, `apply` o `mapcar`.
- Clarificación del código.

En esta ocasión, la asociación entre símbolo y definición de función se realiza mediante la forma funcional `defmacro`. Para ilustrar el proceso de evaluación, retomamos el anterior ejemplo usado con `defun`, pero en forma de `defmacro`:

```
[5]> (setq saludo "hola")
SALUDO
[6]> (defmacro saludo (texto) `(print ,texto))
SALUDO
[7]> (load "~/lisp/CLisp/examples/ppmx")
;; Loading file /home/vilares/lisp/CLisp/examples/ppmx.cl ...
;; Loaded file /home/vilares/lisp/CLisp/examples/ppmx.cl
T
[8]> (ppmx (saludo saludo))
Macro expansion:
(PRINT SALUDO)
```

lo que viene a determinar el valor de la expresión que sustituirá a la llamada `(saludo saludo)` en el texto del programa una vez esta evaluada.

En este punto, es importante hacer notar, que en ocasiones la macro expansión puede requerir varios pasos, dependiendo del tipo de expresiones utilizadas en la definición de la macro. Consideremos el siguiente ejemplo:

```
[9]> (load "~/lisp/CLisp/examples/ppmx")
;; Loading file /home/vilares/lisp/CLisp/examples/ppmx.cl ...
;; Loaded file /home/vilares/lisp/CLisp/examples/ppmx.cl
T
[10]> (load "~/lisp/CLisp/examples/incr")
;; Loading file /home/vilares/lisp/CLisp/examples/incr.cl ...
;; Loaded file /home/vilares/lisp/CLisp/examples/incr.cl
T
[11]> (ppmx (incr a))
First step of expansion:
(SETF A (+ A 1))
```

```
Final expansion:
(SETQ A (+ A 1))
```

lo que implica, al contrario que en el ejemplo anterior, dos pasos diferenciados en la expansión de la `defmacro` original `incr`. Ello se debe a que la función `setf`, que forma parte de la definición de `incr`, es a su vez una `defmacro` que se expande a la forma `setq`. El segundo paso, el que expande `setf` a `setq`, sólo puede efectuarse una vez se ejecute `setf`, lo que puede ocurrir sólo después del paso representado por la primera fase de expansión.

# Capítulo 4

# Predicados básicos

Un *predicado* es un tipo especial de función en el sentido de que los valores que devuelve poseen siempre un significado que puede entenderse como *false* o *true*. En Lisp, el booleano *false* es equivalente al valor `()` ó `nil`, y el booleano *true* es equivalente a cualquier otro distinto de `()`. Ciertos predicados que deben devolver el valor boolenao *true*, usan la constante simbólica `t`. Por definición, esta constante es diferente de `()`, puesto que el valor del símbolo `t` es el símbolo `t` el mismo.

## 4.1. Predicados booleanos

En Lisp, todo valor diferente de `()` se considera con un valor booleano `t`. En este sentido, nos limitaremos aquí a los más básicos. Aquellos que justamente verifican si el valor es `t` ó `()` y a los que cambian ese valor.

**(NULL forma)**

Testea si el valor la `forma` es igual a `()`. En caso afirmativo, devuelve `t`, sino devuelve `()`.

```
[12> (null ())
T
[13]> (null 2)
NIL
[14]> (null t)
NIL
[15]> (null nil)
T
```

**(NOT forma)**

Inversa del valor booleano de la `forma`, devolviendo dicho valor inverso.

Es equivalente al anterior `nul`, puesto que en Lisp, cualquier valor diferente de `()` ó `nil` es siempre considerado cierto.

```
[16]> (not ())
T
[17]> (not 2)
NIL
[18]> (not t)
NIL
[19]> (not nil)
T
```

## 4.2. Predicados de tipo

Son predicados que devuelven `t` cuando su argumento pertenece al tipo de dato por el que se interroga y `nil` en otro caso. Existe un predicado de esta clase asociado a cada tipo de dato Lisp.

### (ARRAYP tablero)

Predicado que devuelve `t` si el argumento es un tablero, sino devuelve `()`.

**Ejemplo 4.1** *Ilustraremos el funcionamiento de* `arrayp` *con algunas llamadas tecleadas directamente sobre el intérprete:*

```
[154]> (setq tablero (make-array '(4 3) :initial-contents '((a b c)
                                                            (1 2 3)
                                                            (E F G)
                                                            (4 5 6))))
#2A((A B C) (1 2 3) (E F G) (4 5 6))
[155]> (arrayp tablero)
T
[156]> (arrayp (vector 1 2 3 4))
T
```

### (CONSP forma)

Testea si el valor de `forma` es de tipo `cons`.

```
[20]> (consp ())
NIL
[21]> (consp '(1 2))
T
[22]> (consp 1)
```

```
NIL
[23]> (consp 'a)
NIL
[24]> (consp #(1 2))
NIL
```

Observar que `()` no es de tipo `cons`, esto es, la lista vacía no es una lista[1].

### (ATOM forma)

Testea si el valor de `forma` es de un tipo diferente a `cons`, esto es, tiene un comportamiento contrario a `consp`.

```
[25]> (atom ())
T
[26]> (atom '(1 2))
NIL
[27]> (atom 1)
T
[28]> (atom 'a)
T
[29]> (atom #(1 2))
T
```

### (CONSTANTP forma)

Testea si el valor de `forma` es una constante, esto es, si el objeto evaluado es idéntico al objeto sin evaluar.

```
[30]> (constantp ())
T
[31]> (constantp nil)
T
[32]> (constantp 'Hola)
NIL
[33]> (constantp 1)
T
[34]> (constantp "Hola")
T
[35]> (constantp #(1 2))
T
```

### (FUNCTIONP forma)

[1] si lo fuera, tendría un `car` y un `cdr`, y no es el caso.

Testea si el valor de `forma` es una función, esto es, un objeto susceptible de ser aplicable a argumentos.

```
[30]> (functionp #'append)
T
[31]> (functionp #'apply)
T
[32]> (functionp #'numberp)
T
[33]> (functionp #'+)
T
```

### (COMPILED-FUNCTION-P forma)

Testea si el valor de `forma` es una función compilada.

```
[30]> (compiled-function-p #'map)
T
[31]> (defun hola () (print "Hola mundo"))
HOLA
[32]> (compiled-function-p #'hola)
NIL
```

### (PACKAGEP forma)

Testea si el valor de `forma` es un package.

```
[30]> (package (make-package 'saludo))
T
```

### (SYMBOLP forma)

Testea si el valor de `forma` es un símbolo.

```
[37]> (symbolp ())
T
[38]> (symbolp 'Hola)
T
[39]> (symbolp "Hola")
NIL
[40]> (symbolp #(1 2))
NIL
[41]> (symbolp '(1 2))
NIL
[42]> (symbolp 1)
NIL
```

### (FLOATP forma)

Testea si el valor de `forma` es un número real flotante.

```
[37]> (floatp 1)
NIL
[38]> (floatp 1.1)
T
```

### (NUMBERP forma)

Testea si el valor de `forma` es un número.

```
[43]> (numberp "Hola")
NIL
[44]> (numberp 'Hola)
NIL
[45]> (numberp #(1 2))
NIL
[46]> (numberp 1)
T
[47]> (setq a 1)
A
[48]> (numberp a)
T
```

### (RATIONALP forma)

Testea si el valor de `forma` es un número racional.

```
[43]> (rationalp 3)
T
[44]> (rationalp 3/2)
T
[45]> (rationalp 3.1)
NIL
```

### (REALP forma)

Testea si el valor de `forma` es un número real.

```
[43]> (realp 3)
T
[44]> (realp 3.1)
T
```

```
[45]> (realp 3/2)
T
[46]> (realp #c(5/3 7.2))
NIL
```

### (COMPLEXP forma)

Testea si el valor de `forma` es un número complejo.

```
[43]> (complexp 3)
NIL
[44]> (complexp 3.1)
NIL
[45]> (complexp 3/2)
NIL
[46]> (complexp #c(5/3 7.2))
T
```

### (CHARACTERP forma)

Testea si el valor de `forma` es un carácter.

```
[43]> (characterp #\a)
T
[44]> (characterp 'a)
NIL
[45]> (characterp 2)
NIL
```

### (SIMPLE-VECTOR-P forma)

Testea si el valor de `forma` es un vector simple.

```
[53]> (simple-vector-p (vector 1 2 3 4))
T
[54]> (simple-vector-p (make-array '(2 3)))
NIL
```

### (VECTORP forma)

Testea si el valor de `forma` es un vector.

```
[49]> (vectorp #(1 2))
T
[50]> (vectorp '(1 2))
```

```
NIL
[51]> (vectorp "Hola")
T
[52]> (vectorp 'Hola)
NIL
```

**(STRINGP forma)**

Testea si el valor de `forma` es una cadena de caracteres.

```
[53]> (stringp #(1 2))
NIL
[54]> (stringp '(1 2))
NIL
[55]> (stringp 'Hola)
NIL
[56]> (stringp "Hola")
T
```

**(LISTP forma)**

Testea si el valor de `forma` es una lista, vacía o no.

```
[57]> (listp ())
T
[58]> (listp '(1 2))
T
[59]> (listp #(1 2))
NIL
[60]> (listp 'Hola)
NIL
[61]> (listp "Hola")
NIL
```

## 4.3. Predicados de igualdad

Lisp proporciona dos predicados distintos para establecer la igualdad entre dos objetos. Ambos son usados con profusión en multitud de otras funciones Lisp predefinidas, dando lugar a dos versiones distintas de una misma funcionalidad fundamental.

**(EQUAL forma forma)**

Es la clase más general de comparación disponible en Lisp, y debe ser usada

cuando se desconoce el tipo exacto de sus argumentos. El predicado testea la igualdad estructural del resultado de la evaluación de sus argumentos, devolviendo `t` si el test es cierto y `()` en otro caso.

**Ejemplo 4.2** *Ilustraremos el funcionamiento de* `equal` *mediante unas sencillas evaluaciones en modo interactivo, directamente sobre el intérprete:*

```
[16]> (equal () nil)
T
[17]> (equal nil 'nil)
T
[18]> (equal 'a (car '(a)))
T
[19]> (equal 5 (+ 1 4))
T
[20]> (equal '(a b) '(a b))
T
[21]> (equal 5 5)
T
[22]> (setq a 5)
A
[23]> (equal 5 a)
T
[24]> (equal #(1 2) #(1 2))
NIL
```

*Observar el inusual comportamiento de* `equal` *en el último ejemplo, sobre dos vectores con la misma estructura. En otros intérpretes Lisp, como en el caso de* LeLisp*, la respuesta en ese caso sería* `t`*.*

### (EQUALP forma forma)

Es una versión más felxible de `equal`. En particular, compara las cadenas de caracteres sin atender a si están o no en mayúsculas, y los números sin tener en cuenta el tipo específico. Los vectores y listas son comparados recursivamente. En el resto de los casos la evaluación es idéntica a la de `equal`.

**Ejemplo 4.3** *Ilustraremos el funcionamiento de* `equalp` *mediante unas sencillas evaluaciones en modo interactivo, directamente sobre el intérprete:*

```
[25]> (equal 3 3.0)
NIL
[26]> (equalp 3 3.0)
T
```

```
[27]> (equal "Hola" "hola")
NIL
[28]> (equalp "Hola" "hola")
T
[29]> (equal #("Hola" 3) #("hola" 3.0))
NIL
[30]> (equalp #("Hola" 3) #("hola" 3.0))
T
[31]> (equal '("Hola" 3) '("hola" 3.0))
NIL
[32]> (equalp '("Hola" 3) '("hola" 3.0))
T
```

**(EQ forma forma)**

Testea si las direcciones de memoria de los resultados de la evaluación de sus dos argumentos es igual, y devuelve `t` si este es el caso. De otra manera, devuelve (). En caso de que los argumentos sean símbolos, simplemente testea su igualdad.

**Ejemplo 4.4** *Ilustraremos el funcionamiento de* `eq` *mediante unas sencillas evaluaciones en modo interactivo, directamente sobre el intérprete:*

```
[5]> (eq () nil)
T
[6]> (eq nil 'nil)
T
[7]> (eq 'a (car '(a)))
T
[8]> (eq 5 (+ 1 4))
T
[9]> (eq '(a b) '(a b))
NIL
[10]> (eq 5 5)
T
[11]> (setq a 5)
A
[12]> (eq 5 a)
T
[13]> (eq #(1 2) #(1 2))
NIL
```

**(EQL forma forma)**

Función idéntica a `eq`, salvo que si los resultados de la evaluación de las `formas` son números del mismo tipo, entonces son comparados como si de `equal` se tratara; esto es, por igualdad numérica. Ello sólo marcará una diferencia con `eq` en versiones compiladas con soporte de punto flotante[2].

[2]en ese caso, los flotantes son implementados como celdas de tipo `cons`, lo que implica expresiones como `(eq 3 3.0)` no son necesariamente `t`. Sin embargo, `(eql 3 3.0)` si sería `t` en cualquier caso.

# Capítulo 5

# Funciones predefinidas

Se trata de funciones residentes en el sistema y disponibles en todo momento. Organizaremos su descripción por su analogía operativa.

## 5.1. Funciones de definición de funciones

Para definir una función y darle un nombre, necesitamos especificar:

- *Nombre de la función.* Puede ser cualquier átomo no numérico distinto de `t` ó `nil`.
- *Argumentos de la función.* Pueden representarse mediante cualquier átomo no numérico distinto de `t` ó `nil`. Se escriben en una lista.
- *Cuerpo de la función.* Expresión simbólica que establece cómo se calcula el valor que devolverá la función al dar valores concretos a los argumentos.

### 5.1.1. Funciones estáticas

Permiten cambiar permanentemente las funciones asociadas con los símbolos. Se trata de las primitivas ya introducidas `defun` y `defmacro`.

### 5.1.2. Funciones dinámicas

Permiten la definición dinámica de funciones.

#### (FLET ({(símbolo ({argumentos}*) {forma}*)}* {forma}*)

Permite la definición dinámica y local de una función a partir de cada una de las cláusulas `(símbolo ({argumentos})* {forma}*)`. De este modo asigna simultáneamente a cada `símbolo` una definición de función con los `argumentos` asociados y con el cuerpo definido por las `formas` también

asociadas en la cláusula correspondiente. Las definiciones son locales al cuerpo {`forma`}* de la función `flet`, devolviendo el valor de la última de estas `formas`.

**Ejemplo 5.1** *Como ejemplo, ilustraremos como asignar temporalmente la funcionalidad de la función* `cdr` *a la función* `car`*:*

```
[13]> (flet ((car (x) (cdr x))) (car '(a b c)))
(B C)
```

## 5.2. Funciones de manipulación del entorno

Cambian dinámicamente el *entorno de trabajo*, esto es, el conjunto de pares variable/valor asignados en un momento dado de la sesión de trabajo.

### (LET ({(variable forma)}*) {forma}*)

Permite la asignación local de variables. Asigna dinámica y simultáneamente a cada `variable` el resultado de la evaluación de su `forma` asociada. Dicha asignación es local al cuerpo {`forma`}* de la función `let`, devolviendo el valor de la última de estas `formas`.

**Ejemplo 5.2** *Supongamos la siguiente función en el fichero* ∼`/lisp/CLisp/examples/ejemplo-let.cl`*, ilustrando la función* `let`*:*

```
(defun ejemplo-let ()
       (let ((global-1 3)
             (global-2 (+ global-1 1)))
          (format t "Localmente, global-1 = ~S ~%" global-1)
          (format t "Localmente, global-2 = ~S ~%" global-2)
          ()))
```

*y apliquemos la siguiente secuencia de acciones sobre el intérprete:*

```
[1]> (defvar global-1 1)
GLOBAL-1
[2]> (defvar global-2 5)
GLOBAL-2
[3]> (load "~/lisp/CLisp/examples/ejemplo-let")
;; Loading file /home/vilares/lisp/CLisp/examples/ejemplo-let.cl ...
;; Loaded file /home/vilares/lisp/CLisp/examples/ejemplo-let.cl
T
[4]> (ejemplo-let)
Localmente, global-1 = 3
```

```
Localmente, global-2 = 2
NIL
[5]> global-1
1
[6]> global-2
5
```

### (LET* ({(variable forma)}*) {forma}*)

La descripción de argumentos y la funcionalidad son las mismas detalladas para `let`, con la salvedad de que aquí las asignaciones de los pares `(variable forma)` se hacen de forma secuencial.

**Ejemplo 5.3** *Supongamos la siguiente función en el fichero* ∼`/lisp/CLisp/examples/ejemplo-let-estrella.cl`*, ilustrando la función* `let*`*:*

```
(defun ejemplo-let-estrella ()
       (let* ((global-1 3)
              (global-2 (+ global-1 1)))
          (format t "Localmente, global-1 = ~S ~%" global-1)
          (format t "Localmente, global-2 = ~S ~%" global-2)
          ()))
```

*y apliquemos la siguiente secuencia de acciones sobre el intérprete:*

```
[1]> (defvar global-1 1)
GLOBAL-1
[2]> (defvar global-2 5)
GLOBAL-2
[3]> (load "~/lisp/CLisp/examples/ejemplo-let-estrella")
;; Loading file /home/vilares/lisp/CLisp/examples/ejemplo-let-estrella.cl ...
;; Loaded file /home/vilares/lisp/CLisp/examples/ejemplo-let-estrella.cl
T
[4]> (ejemplo-let-estrella)
Localmente, global-1 = 3
Localmente, global-2 = 4
NIL
[5]> global-1
1
[6]> global-2
5
```

## 5.3. Funciones de evaluación

Se sitúan a la base de funcionamiento del intérprete Lisp, que evalúa sistemáticamente las expresiones que se le sumnistran, salvo que se le indique expresamente lo contrario.

**(EVAL forma)**

Es la función de base del evaluador del intérprete Lisp, evaluando la `forma` y devolviendo el resultado de la misma:

```
[7]> (eval '(+ 1 2))
3
[8]> (eval (list '+ 8 '(+ 1 3) 3))
15
```

**Ejemplo 5.4** *Implementamos ahora una función* `(mi-eprogn lista)`*, que valúa secuencialmente todos los elementos de la lista* `lista`*, devolviendo el valor del último elemento de ésta:*

```
(defun mi-eprogn (l)
       (if (atom l)
           (if (null l)
               ()
              (print "Error en mi-progn"))
          (if (atom (cdr l))
              (eval (car l))
             (and (eval (car l))
                  (mi-eprogn (cdr l))))))
```

*que ahora pasamos a interpretar en la siguiente secuencia de acciones:*

```
[1]> (defvar lista '((print 1) (print 2) (print 3)))
LISTA
[2]> (load "~/lisp/CLisp/examples/ejemplo-mi-eprogn")
;; Loading file /home/vilares/lisp/CLisp/examples/ejemplo-mi-eprogn.cl ...
;; Loaded file /home/vilares/lisp/CLisp/examples/ejemplo-mi-eprogn.cl
T
[3]> (mi-eprogn lista)

1
2
3
3
```

### (PROG1 {forma}*)

Evalúa secuencialmente las `formas`, y devuelve el valor de la primera de ellas:

```
[1]> (prog1 (print 1) (print 2) (print 3))

1
2
3
1
```

Una posible implementación en Lisp para la función `prog1` sería la siguiente:

```
(defun mi-prog1 (primero &rest resto)
       (let ((resultado (eval primero)))
           (mi-eprogn resto)
           resultado))
```

donde `&rest` nos permite diferenciar entre el conjunto de argumentos obligatorios, en este caso `primero`, y el resto. Ello permite la definición de funciones con un número variable de argumentos.

### (PROG2 {forma}*)

Evalúa secuencialmente las `formas`, y devuelve el valor de la segunda de ellas:

```
[2]> (prog2 (print 1) (print 2) (print 3))

1
2
3
2
```

Una posible implementación en GNU CLISP para la función `prog2` sería la siguiente:

```
(defun mi-prog2 (primero segundo &rest resto)
       (eval primero)
       (let ((resultado (eval segundo)))
           (mi-eprogn resto)
           resultado))
```

### (PROGN {forma}*)

Evalúa secuencialmente las `formas`, y devuelve el valor de la última:

```
[3]> (progn (print 1) (print 2) (print 3))

1
2
3
3
```

Una posible implementación en GNU CLISP para la función `progn` sería la siguiente:

```
(defun mi-progn (primero &rest resto)
       (mi-eprogn l))
```

**(QUOTE forma)**

Devuelve la `forma`, sin evaluar. Podemos usar 'como abreviatura de `quote`, de forma que una expresión del estilo `(quote cons)` es equivalente a `'cons`. Esta función es usada para inhibir la evaluación de los argumentos en las funciones.

```
[1]> (defvar a 1)
A
[2]> a
1
[3]> (quote a)
A
[4]> 'a
A
```

**(FUNCTION símbolo)**

Devuelve la *interpretación funcional* de su argumento, sin evaluar éste. Esto es, devuelve el contenido de la entrada en la tabla de símbolos correspondiente a la función de nombre `símbolo`, en caso de existir. Podemos usar ' como abreviatura de `quote`, de forma que una expresión del estilo `(function cons)` es equivalente a `#'cons`.

```
[5]> (function cons)
#<SYSTEM-FUNCTION CONS>
[6]> #'cons
#<SYSTEM-FUNCTION CONS>
```

## 5.4. Funciones de control

Permiten el encadenamiento secuencial de evaluaciones, con la posibilidad de romperlas. Ello implica que los argumentos no son evaluados a la llamada de la función, sino explícitamente al interior de las mismas.

### 5.4.1. Funciones locales

Reciben el nombre de locales, por reducir su acción de control al entorno representado por la función que las incluye. Las fundamentales son las siguientes:

**(IF forma forma {forma}*)**

Si el valor de la forma inicial es distinto de `()`, `if` devuelve el valor de la segunda `forma`, en otro caso evalúa en secuencia las `formas` a partir de la tercera y devuelve el valor de la última.

**Ejemplo 5.5** *Implementaremos la función de Ackermann, que matemáticamente viene dada por:*

$$\begin{aligned} ackermann(0,N) &= N{+}1 \\ ackermann(M,0) &= ackermann(M\text{-}1,1) \\ ackermann(M,N) &= ackermann(M\text{-}1, ackermann(M,N\text{-}1)) \end{aligned}$$

*en el fichero* ∼`/lisp/CLisp/examples/ejemplo-ackermann.cl`*, usando el código siguiente:*

```
(defun ackermann (x y)
       (if (= x 0)
           (+ y 1)
         (ackermann (- x 1)
                    (if (= y 0)
                        1
                      (ackermann x (- y 1))))))
```

*donde un ejemplo de ejecución podría ser el siguiente:*

```
[7]> (load "~/lisp/CLisp/examples/ejemplo-ackermann")
;; Loading file /home/vilares/lisp/CLisp/examples/ejemplo-ackermann.cl ...
;; Loaded file /home/vilares/lisp/CLisp/examples/ejemplo-ackermann.cl
T
[8]> (ackermann 2 3)
9
```

**(WHEN forma {forma}*)**

Si el valor de la primera `forma` es distinto de (), `when` evalúa en secuencia las siguientes `formas`, devolviendo el valor de la última. En otro caso, devuelve `()`.

**Ejemplo 5.6** *Sea el fichero* ∼`/lisp/CLisp/examples/ejemplo-when.cl`*, de código:*

```
(defun ejemplo-when (booleano)
       (when booleano
             (print "Valor booleano positivo")))
```

*al que podemos aplicar la siguiente secuencia de evaluación:*

```
[9]> (load "~/lisp/CLisp/examples/ejemplo-when")
;; Loading file /home/vilares/lisp/CLisp/examples/ejemplo-when.cl ...
;; Loaded file /home/vilares/lisp/CLisp/examples/ejemplo-when.cl
T
[10]> (ejemplo-when t)

"Valor booleano positivo"
"Valor booleano positivo"
[11]> (ejemplo-when ())
NIL
```

### (UNLESS forma {forma}*)

Si el valor de la primera `forma` es igual a (), `unless` evalúa las siguientes `formas` en secuencia, devolviendo el valor de la última. En otro caso devuelve `()`.

**Ejemplo 5.7** *Sea el fichero* `~/lisp/CLisp/examples/ejemplo-unless.cl`*, de código:*

```
(defun ejemplo-unless (booleano)
       (unless booleano
             (print "Valor booleano negativo")))
```

*al que podemos aplicar la siguiente secuencia de evaluación:*

```
[12]> (load "~/lisp/CLisp/examples/ejemplo-unless")
;; Loading file /home/vilares/lisp/CLisp/examples/ejemplo-unless.cl ...
;; Loaded file /home/vilares/lisp/CLisp/examples/ejemplo-unless.cl
T
[13]> (ejemplo-unless ())

"Valor booleano negativo"
"Valor booleano negativo"
[14]> (ejemplo-unless t)
NIL
```

### (OR {forma}*)

Evalúa las `formas` en secuencia, hasta que alguna tiene un valor distinto de `()`, devolviendo dicho valor. En otro caso devuelve `()`:

```
[15]> (or (progn (print 1) ())
          (prog2 (print 2) (< 3 2) (print 3))
          (print 4)
          (print 5))

1
2
3
4
4
```

### (AND {forma}*)

Evalúa las `formas` en secuencia, hasta que alguna de ellas posee un valor `()`. En este caso devuelve `()`, sino devuelve el valor de la última `forma` evaluada:

```
[16]> (and (progn (print 1) ())
           (prog2 (print 2) (3 < 2) (print 3))
           (print 4)
           (print 5))

1
NIL
```

### (COND {(clave {forma}*)}* [{OTHERWISE | T} forma])

Se trata de la instrucción condicional más potente de Lisp, organizada en cláusulas de la forma`(clave` {`forma`}`*)`. La función `cond` evalúa en secuencia cada una de las formas `clave` de cada una de estas cláusulas hasta encontrar una con un valor distinto de `()`. En ese caso `cond` evalúa en secuencia el cuerpo {`forma`}`*` de la cláusula asociada a dicha `clave` y devuelve su valor. En otro caso devuelve `()`.

**Ejemplo 5.8** *Algunos ejemplos sencillos, que podemos realizar en interactivo directamente sobre el intérprete, son los siguientes:*

```
[17]> (cond ((progn (print 1) ())
             (print "Entrada 1"))
            ((prog1 (< 3 2) (print 2))
             (print "Entrada 2"))
            ((print 3) (print "Entrada 3") ()))

1
2
3
```

```
"Entrada 3"
NIL
[18]> (cond ((progn (print 1) ())
             (print "Entrada 1"))
            ((prog1 (< 3 2) (print 2))
             (print "Entrada 2"))
            (t (print "Entrada por defecto") ()))

1
2
"Entrada por defecto"
NIL
```

### (CASE forma {(clave {forma}*)}* [{OTHERWISE | T} forma])

El principio es el mismo expuesto en la función `cond`, con la salvedad de que aquí la selección de la cláusula `(clave {forma}*)` se hace con arreglo al valor de la forma inicial de acuerdo con el siguiente criterio:

- El `car` de la cláusula sin evaluar, si éste es un átomo.
- Los elementos del `car` de la cláusula, sin evaluar, si éste es una lista. En este caso, la comparación usa el predicado predefinido `member` para testar si el selector se encuentra en dicha lista.

Tras la selección, si ésta ha tenido lugar, se evalúa en secuencia el cuerpo de la cláusula y se devuelve el valor de la última de sus expresiones. Si la selección no tiene lugar, se devuelve `()`. Puede incluir una cláusula `otherwise`, siempre en último lugar, para capturar el tratamiento de casos no capturados por otras cláusulas. Éste último comportamiento puede simularse también incluyendo, siempre en último lugar, una cláusula cuya condición sea un `t`.

**Ejemplo 5.9** *Para ilustrar el funcionamiento de* `case`*, podemos teclear en interactivo algunas expresiones simples:*

```
[19]> (case 'rojo
        (verde 'espero)
        (rojo 'ok)
        (t 'imposible))
OK
[20]> (case 'rojo
        ((azul verde rojo) 'ok)
        ((rosa violeta) 'sospechoso)
        (t 'imposible))
```

```
OK
[21]> (case 'amarillo
         (verde 'espero)
         (rojo 'ok)
         (t 'imposible))
IMPOSIBLE
[22]> (case 'amarillo
         (verde 'espero)
         (rojo 'ok)
         (otherwise 'imposible))
IMPOSIBLE
[23]> (case 'amarillo
         (verde 'espero)
         (otherwise 'imposible)
         (rojo 'ok))

*** - CASE: la clausula OTHERWISE debe ser la Ã°ltima
Es posible continuar en los siguientes puntos:
ABORT          :R1      Abort debug loop
ABORT          :R2      Abort debug loop
ABORT          :R3      Abort main loop
```

### 5.4.2. Funciones no locales

Su uso está fuertemente recomendado, como alternativa al `goto` clásico. Utilizan una etiqueta definida de forma dinámica como referencia para un *escape*.

**(TAGBODY {etiqueta|forma}*)**

Define un entorno léxico de *escape* identificado por una o más `etiquetas`, que deben ser símbolos, y que no son evaluados. Evalúa en secuencia las `formas` y devuelve `nil`, salvo que un salto del control del intérprete mediante `go`, que pasamos a describir de inmediato, establezca lo contrario.

**(GO etiqueta)**

Devuelve la evaluación del intérprete al punto del cuerpo `tagbody` más interno que aparezca etiquetado con el símbolo `etiqueta`, utilizando `eql` como primitiva de comparación. De no existir tal punto, se produce un error.

**Ejemplo 5.10** *Consideremos la siguiente función incluida en el fichero* `~/lisp/CLisp/examples/ejemplo-bloques.cl`*:*

```
(defun ejemplo-tagbody-1 (i)
       (tagbody
          vuelta (print i)
              (setq i (+ 1 i))
              (when (< i 6) (go vuelta))))
```

*que podemos ahora ejecutar, después de cargarla en el intérprete:*

```
[108]> (load "~/lisp/CLisp/examples/ejemplo-bloques.cl")
;; Loading file /home/vilares/lisp/CLisp/examples/ejemplo-bloques.cl ...
;; Loaded file /home/vilares/lisp/CLisp/examples/ejemplo-bloques.cl
T
[109]> (ejemplo-tagbody-1 1)

1
2
3
4
5
NIL
```

### (BLOCK etiqueta {forma}*)

Define un entorno léxico de *escape* identificado por una `etiqueta`, que deben ser un símbolo, y que no es evaluado. Devuelve el valor de la evaluación de la secuencia de `formas`, salvo que una función `return-from`, especifique otro valor.

**Ejemplo 5.11** *Consideremos las siguientes funciones incluidas en el fichero* ∼`/lisp/CLisp/examples/ejemplo-bloques.cl`*:*

```
(defun ejemplo-block-1 (i j)
       (block vuelta
              (loop for l from i to j
                    do (print l)
                       (when (> l 4)
                             (return-from vuelta (print "Tenemos un 5")))))
       (print "Todo acabo"))

(defun ejemplo-block-2 (i j)
       (block vuelta
              (loop for l from i to j
                    do (print l)
                       (when (> l 4)
                             (return-from vuelta)))))
```

```
       (print "Todo acabo"))

(defun ejemplo-block-3 (i j)
       (block vuelta
              (loop for l from i to j
                    do (print l)
                       (when (> l 4)
                             (return))))
       (print "Todo acabo"))

(defun ejemplo-block-4 (i j)
       (block vuelta
              (loop for l from i to j
                    do (print l)
                       (when (> l 4)
                             (return (print "Tenemos un 5")))))
       (print "Todo acabo"))
```

*que podemos ahora ejecutar, después de cargarlas en el intérprete:*

```
[110]> (load "~/lisp/CLisp/examples/ejemplo-bloques.cl")
;; Loading file /home/vilares/lisp/CLisp/examples/ejemplo-bloques.cl ...
;; Loaded file /home/vilares/lisp/CLisp/examples/ejemplo-bloques.cl
T
[111]> (ejemplo-block-1 1 6)

1
2
3
4
5
"Tenemos un 5"
"Todo acabo"
"Todo acabo"
[112]> (ejemplo-block-2 1 6)

1
2
3
4
5
"Todo acabo"
"Todo acabo"
[113]> (ejemplo-block-3 1 6)
```

```
1
2
3
4
5
"Todo acabo"
"Todo acabo"
[114]> (ejemplo-block-4 1 6)

1
2
3
4
5
"Tenemos un 5"
"Todo acabo"
"Todo acabo"
```

**(CATCH etiqueta {forma}*)**

Define un entorno léxico identificado mediante una `etiqueta`, sobre el que podemos considerar una estrategia de salida no local de forma dinámica mediante `throw etiqueta`, que describimos inmediatamente. La etiqueta es evaluada, a diferencia de lo que ocurre con otras estrategias de definición de entornos léxicos, siendo el resultado de dicha evaluación el que identifica al entorno. Las expresiones se evalúan en secuencia mediante un `progn` implícito, salvo si una función de tipo `throw etiqueta`, descrita inmediatamente después, sea llamada en el transcurso de dicha evaluación. Para ello las etiquetas de `catch` y `throw` deben coincidir mediante un test de tipo `eq`.

**(THROW etiqueta forma)**

Se usa para salir del cuerpo de una función `catch`, devolviendo el resultado de evaluar `forma`, siempre y cuando las etiquetas de ambos coincidan mediante un test `eq`. Como en el caso de `catch`, la etiqueta es primero evaluada.

**Ejemplo 5.12** *Consideremos la siguiente función incluida en el fichero* `~/lisp/CLisp/examples/ejemplo-bloques.cl`*:*

```
(defun ejemplo-catch-throw-1 (i j)
       (catch 'salgo
```

```
            (loop for x from i to j
                  sum x
                  when (eq x 5)
                       do (throw 'salgo (print "Ejecutando throw"))
                  else do (print "No tenemos un 5")
                  end
                  finally (print "Finally en el loop")))
      (print "Todo acabo"))
```

*que podemos ahora ejecutar, después de cargarla en el intérprete:*

```
[115]> (load "~/lisp/CLisp/examples/ejemplo-bloques.cl")
;; Loading file /home/vilares/lisp/CLisp/examples/ejemplo-bloques.cl ...
;; Loaded file /home/vilares/lisp/CLisp/examples/ejemplo-bloques.cl
T
[116]> (ejemplo-catch-throw-1 1 6)

"No tenemos un 5"
"No tenemos un 5"
"No tenemos un 5"
"No tenemos un 5"
"Ejecutando throw"
"Todo acabo"
"Todo acabo"
```

## 5.5. Funciones de iteración

Los iteradores pueden considerarse como un caso particular de las funciones de control, sin embargo la complejidad de éstas en el caso de GNU CLISP, aconseja su tratamiento por separado. Al respecto es importante señalar algunas circunstancias que afectan tanto a la complejidad referida como al estilo de programación y consideración de este tipo de estructuras iterativas en la práctica:

- Este tipo de iteradores, asimilables a los existentes en los lenguajes imperativos, debieran considerarse en el estilo de programación Lisp como una funcionalidad a la que se recurre sólo de forma excepcional. De hecho, un diseño orientado a listas de las estructuras de datos debiera permitir que este tipo de funcionalidades sean asumidas por por las funciones de aplicación tipo `map`. Ello marcará el estilo propio de programación Lisp.

- Otros intérpretes Lisp anteriores, como por ejemplo LE-LISP, reducían este tipo de funciones a apenas un caso. El posterior proceso de

normalización del lenguaje Lisp, en la práctica reflejado en el proyecto COMMON LISP, ha desembocado en la fusión sintáctica de los iteradores de cada uno de los intérpretes previos.

El resultado es una sintaxis innecesariamente compleja y sólo aparentemente flexible. Desafortunadamente, todo ello ha servido como fuente de inspiración para la implementación de lenguajes de más reciente cuño, tales como PYTHON.

### (DOTIMES (i n [valor]) {forma}*)

Evalúa las `formas` en secuencia, para $i \in [0, n-1]$. Devuelve `()` o, si se proporciona, `valor`.

**Ejemplo 5.13** *Consideremos las siguientes dos funciones incluidas en el ejemplo* ∼`/lisp/CLisp/examples/ejemplo-loops.cl`*:*

```
(defun ejemplo-dotimes-1 (i num)
      (dotimes (i num)
               (print i)))

(defun ejemplo-dotimes-2 (i num)
      (dotimes (i num t)
               (print i)))
```

*ejecutémoslas, después de cargarlas en el intérprete:*

```
[26]> (load "~/lisp/CLisp/examples/ejemplo-loops.cl")
;; Loading file /home/vilares/lisp/CLisp/examples/ejemplo-loops.cl ...
;; Loaded file /home/vilares/lisp/CLisp/examples/ejemplo-loops.cl
T
[27]> (ejemplo-dotimes-1 1 5)

0
1
2
3
4
NIL
[28]> (ejemplo-dotimes-2 1 5)

0
1
2
3
4
T
```

### (DOLIST (i lista [valor]) {forma}*)

Evalúa las `formas` en secuencia para cada valor de *i*, que recorre cada elemento de la lista `lista`. Devuelve `()` o, si se proporciona, `valor`.

**Ejemplo 5.14** *Consideremos las siguientes dos funciones incluidas en el ejemplo* ∼`/lisp/CLisp/examples/ejemplo-loops.cl`*:*

```
(defun ejemplo-dolist-1 (lista)
       (dolist (x lista)
               (print x)))

(defun ejemplo-dolist-2 (lista)
       (dolist (x lista t)
               (print x)))
```

*ejecutémoslas, después de cargarlas en el intérprete:*

```
[29] (load "~/lisp/CLisp/examples/ejemplo-loops.cl")
;; Loading file /home/vilares/lisp/CLisp/examples/ejemplo-loops.cl ...
;; Loaded file /home/vilares/lisp/CLisp/examples/ejemplo-loops.cl
T
[30]> (ejemplo-dolist-1 '(1 2 3))

1
2
3
NIL
[31]> (ejemplo-dolist-2 '(1 2 3))

1
2
3
T
```

### (RETURN [forma])

Permite la salida desde el interior de un bucle, devolviendo el valor de la evaluación de `forma` si es que ésta se proporciona.

**Ejemplo 5.15** *Consideremos la siguiente función incluida en el ejemplo* ∼`/lisp/CLisp/examples/ejemplo-loops.cl`*:*

```
(defun ejemplo-return (i num)
       (dotimes (i num)
```

```
                (when (= i 4)
                      (return "Me salgo en el numero 4"))
                (format t "El numero ~D ~%" i)))
```

*ejecutémosla, después de cargarla en el intérprete:*

```
[32]> (load "~/lisp/CLisp/examples/ejemplo-loops.cl")
;; Loading file /home/vilares/lisp/CLisp/examples/ejemplo-loops.cl ...
;; Loaded file /home/vilares/lisp/CLisp/examples/ejemplo-loops.cl
T
[33]> (ejemplo-return 1 6)
El numero 0
El numero 1
El numero 2
El numero 3
"Me salgo en el numero 4"
```

**(DO ((i$_1$ inicial$_1$ incr$_1$) ... (i$_n$ inicial$_n$ incr$_n$) (test [forma]) {forma}*)**

Recuerda al clásico bucle `for` de los lenguajes imperativos como puede ser C. Evalúa en secuencia las `formas` de su cuerpo, incrementando el valor de los iteradores $i_j$ en los valores asociados $incr_j$, hasta que el test de terminación `test` sea positivo. En ese caso, devuelve el resultado de evaluar la `forma` asociada, si ésta se facilita. En otro caso, devuelve `()`.

Las variables de iteración son locales y se asignan simultáneamente. Si deseásemos asignarlas secuencialmente[1], bastaría con reemplazar `do` por `do*`. La función `do` aplica implícitamente un `progn`.

**Ejemplo 5.16** *Consideremos la siguiente función incluida en el fichero* ∼`/lisp/CLisp/examples/ejemplo-loops.cl`*:*

```
(defun ejemplo-do-1 ()
       (let ((c 0))
           (do ((a 1 (+ a 1))                    ; a = 1, 2, 3, ...
                (b '(1 10 3 2) (cdr b)))         ; b toma sucesivos CDRs
               ((null b) c)                      ; si b=(), devuelve c
              (setq c (+ c (expt (car b) a))))))) ; suma b^a * c

(defun ejemplo-do-2 ()
       (let ((c 0))
           (do ((a 1 (+ a 1))                    ; a = 1, 2, 3, ...
                (b '(1 10 3 2) (cdr b)))         ; b toma sucesivos CDRs
```

[1] esto es, si deseásemos que las variables de iteración fueran interdependientes.

```
             ((null b))                          ; si b=(), devuelve NIL
            (setq c (+ c (expt (car b) a))))))   ; suma b^a * c
```

*ejecutémoslas, después de cargarlas en el intérprete:*

```
[34]> (load "~/lisp/CLisp/examples/ejemplo-loops.cl")
;; Loading file /home/vilares/lisp/CLisp/examples/ejemplo-loops.cl ...
;; Loaded file /home/vilares/lisp/CLisp/examples/ejemplo-loops.cl
T
[35]> (ejemplo-do-1)
144
[36]> (ejemplo-do-2)
NIL
```

### (LOOP [claúsula loop]$^+$)

Ejecuta un bucle sobre el conjunto de claúsulas indicado[2], devolviendo `()` salvo que una claúsula de tipo `return` o `loop finish` establezca lo contrario.

La descripción de las claúsulas asociables a un `loop` está ligada a una sintaxis de extrema complejidad, no fácil comprensión y dudoso interés real. Todo ello aconseja su introducción vía ejemplos prácticos concretos, evitando formalizaciones que podrían complicar aún más el aprendizaje. Comenzaremos por describir someramente algunas de las cláusulas posibles:

```
DO {forma}*                              ; Evalua las formas en secuencia,
                                         ; mediante un PROGN implicito.
INITIALLY {forma}*                       ; Evalua las formas antes del
                                         ; loop, mediante un PROGN
                                         ; implicito.
FINALLY [DO] {forma}*                    ; Evalua las formas al acabar
                                         ; el loop, mediante un PROGN
                                         ; implicito.
FOR var [tipo] BEING THE HASH-VALUES     ; Bucle FOR sobre tablas [pares]
    {IN|OF} hash1 [USING hash2]          ; de hash.
FOR var [tipo] {FROM|DOWNFROM} inicio    ; Bucle FOR clasico.
        {TO|DOWNTO} fin  [BY paso]       ;
FOR var [tipo] {IN|ON} lista [BY paso]   ; Bucle FOR sobre listas.
REPEAT forma clausula {AND clausula}*    ; Bucle (incondicional) REPEAT
                                         ; clasico.
IF forma clausula {AND clausula}*        ; Evaluacion condicional.
   [ELSE clausula {AND clausula}*]       ;
   [END]                                 ;
ALWAYS forma clausula {AND clausula}*    ; Bucle ALWAYS clasico.
```

[2] al menos debe haber una.

```
NEVER forma clausula {AND clausula}*    ; Bucle NEVER clasico.
THEREIS forma clausula {AND clausula}*  ; Bucle THEREIS clasico.
UNLESS forma clausula {AND clausula}*   ; Bucle UNLESS clasico.
       [ELSE clausula {AND clausula}*]  ;
       [END]                            ;
UNTIL forma clausula {AND clausula}*    ; Bucle UNTIL clasico.
WHILE forma clausula {AND clausula}*    ; Bucle WHILE clasico.
COUNT forma [INTO var] [tipo]           ; Contabiliza las veces que forma
                                        ; no es ().
MAXIMIZE forma [INTO var] [tipo]        ; Guarda el valor de forma (numerico)
                                        ; mas alto
MINIMIZE forma [INTO var] [tipo]        ; Guarda el valor de forma (numerico)
                                        ; mas bajo.
SUM forma [INTO var] [tipo]             ; Suma todos los valores de forma.
APPEND forma [INTO lista]               ; Concatena los valores de forma en
                                        ; una lista. Los valores deben ser
                                        ; listas.
NCONC forma [INTO lista]                ; Concatena (fisicamente) los valores
                                        ; de forma en una lista. Los valores
                                        ; deben ser listas.
COLLECT forma [INTO lista]              ; Compila los valores de forma en
                                        ; una lista.
LOOP-FINISH                             ; Sale del loop mas interno.
NAMED etiqueta {clausula}*              ; Sale del loop nombrado. Debe
      RETURN-FROM etiqueta [forma]      ; seguir a LOOP inmediatamente.
                                        ; Devuelve el valor de forma.
                                        ; No ejecuta FINALLY.
RETURN [forma]                          ; Sale del bucle (mas interno) y
                                        ; devuelve el valor de forma.
                                        ; Equivale a DO (RETURN forma).
                                        ; No ejecuta FINALLY.
WITH var [tipo]=inicial                 ; Inicializa valores locales.
     {AND var [tipo]=inicial}*          ;
TAGBODY {etiqueta|forma}*               ; Define un entorno l\'exico de escape.
GO etiqueta                             ; Ejecuta un escape dentro de TAGBODY.
```

podemos, además, considerar algunos sinónimos para las palabras claves anteriores. Así, resumiendo, tenemos los siguientes grupos:

| | | |
|---|---|---|
| DO/DOING | FOR/AS | EACH/THE |
| HASH-KEY/HASH-KEYS | HASH-VALUE/HASH-VALUES | IF/WHEN |
| IN/ON | DOWNFROM/DOWNTO/ABOVE | UPFROM/UPTO/BELLOW |
| COUNT/COUNTING | MAXIMIZE/MAXIMIZING | MINIMIZE/MINIMIZING |
| APPEND/APPENDING | NCONC/NCONCING | COLLECT/COLLECTING |

Pasamos ahora a comentar más en detalle cada grupo de claúsulas, ilustrándolos con sencillos ejemplos prácticos.

**Iteración numérica.** Es, por esencia, la iteración clásica propia de los lenguajes imperativos.

```
(FOR var [tipo] {FROM|DOWNFROM} inicio {TO|DOWNTO} fin [BY
paso])
```

donde `paso` indica el valor del incremento en cada iteración, tomando por defecto el valor 1.

**Ejemplo 5.17** *Consideremos las siguientes funciones incluidas en el fichero* ∼`/lisp/CLisp/examples/ejemplo-loops.cl`*:*

```
(defun ejemplo-loop-1 (i j)
       (loop for l from i to j
             do (print l)))

(defun ejemplo-loop-2 (i j)
       (loop as l from i to j
             do (print l)))

(defun ejemplo-loop-3 (i j)
       (loop for l from i upto j
             do (print l)))

(defun ejemplo-loop-4 (i j)
       (loop for l from i below j
             do (print l)))

(defun ejemplo-loop-5 (i j)
       (loop for l from i downto j
             do (print l)))

(defun ejemplo-loop-6 (i j)
       (loop for l from i to j by 2
             do (print l)))

(defun ejemplo-loop-7 (i j)
       (loop for l from i upto j by 2
             do (print l)))

(defun ejemplo-loop-8 (i j)
```

```
       (loop for l from i below j by 2
             do (print l)))

(defun ejemplo-loop-9 (i j)
       (loop for l from i downto j by 2
             do (print l)))

(defun ejemplo-loop-10 (i j)
       (loop for l from j downto i
             do (print l)))

(defun ejemplo-loop-11 (i j)
       (loop for l from j downto i by 2
             do (print l)))
```

*podemos ahora ejecutar algunas de ellas, después de cargarlas en el intérprete, dejando el resto para su experimentación por parte del lector:*

```
[37]> (load "~/lisp/CLisp/examples/ejemplo-loops.cl")
;; Loading file /home/vilares/lisp/CLisp/examples/ejemplo-loops.cl ...
;; Loaded file /home/vilares/lisp/CLisp/examples/ejemplo-loops.cl
T
[38]> (ejemplo-loop-1 1 6)

1
2
3
4
5
6
NIL
[39]> (ejemplo-loop-2 1 6)

1
2
3
4
5
6
NIL
[40]> (ejemplo-loop-3 1 6)

1
2
```

```
3
4
5
6
NIL
[41]> (ejemplo-loop-4 1 6)

1
2
3
4
5
NIL
[42]>  (ejemplo-loop-5 6 1)

6
5
4
3
2
1
NIL
```

**Iteración sobre listas.** Extensión natural de la iteración numérica al caso de listas.

```
(FOR var {IN|ON} lista [BY paso])
```

donde `paso` indica el valor del incremento en cada iteración, tomando por defecto el valor `cdr`.

**Ejemplo 5.18** *Consideremos las siguientes funciones incluidas en el fichero* ∼`/lisp/CLisp/examples/ejemplo-loops.cl`*:*

```
(defun ejemplo-loop-12 (lista)
       (loop for l in lista
             do (print l)))

(defun ejemplo-loop-13 (lista)
       (loop for l in lista by 'cdr ; por defecto el step-fun es CDR
             do (print l)))

(defun ejemplo-loop-14 (lista)
       (loop for l in lista by 'cddr
```

```
            do (print l)))

(defun ejemplo-loop-15 (lista)
       (loop for l on lista         ; ON sinonimo de IN
             do (print l)))

(defun ejemplo-loop-16 (lista)
       (loop for l on lista by 'cdr ; por defecto el step-fun es CDR
             do (print l)))

(defun ejemplo-loop-17 (lista)
       (loop for l on lista by 'cddr
             do (print l)))
```

*podemos ahora ejecutar algunas de ellas, después de cargarlas en el intérprete, dejando el resto para su experimentación por parte del lector:*

```
[43]> (load "~/lisp/CLisp/examples/ejemplo-loops.cl")
;; Loading file /home/vilares/lisp/CLisp/examples/ejemplo-loops.cl ...
;; Loaded file /home/vilares/lisp/CLisp/examples/ejemplo-loops.cl
T
[44]> (ejemplo-loop-12 '(a b c d))

A
B
C
D
NIL
[45]> (ejemplo-loop-13 '(a b c d))

A
B
C
D
NIL
[46]> (ejemplo-loop-14 '(a b c d))

A
C
NIL
```

**Iteración incondicional.** Se corresponde con los bucles imperativos clásicos del tipo `repeat`.

```
(REPEAT forma cláusula {AND cláusula}*)
```

donde el valor de `forma` debe ser una expresión numérica que indica el número de evaluaciones que se aplican sobre la secuencia de `cláusulas` ligadas mediante operadores `AND`.

**Ejemplo 5.19** *Consideremos la siguiente función incluida en el fichero* ∼/lisp/CLisp/examples/ejemplo-loops.cl*:*

```
(defun ejemplo-loop-48 ()
       (let ((a 1))
           (loop repeat 10 do
                 (print (setq a (+ 1 a))))))
```

*que podemos ahora ejecutar, después de cargarla en el intérprete:*

```
[47]> (load "~/lisp/CLisp/examples/ejemplo-loops.cl")
;; Loading file /home/vilares/lisp/CLisp/examples/ejemplo-loops.cl ...
;; Loaded file /home/vilares/lisp/CLisp/examples/ejemplo-loops.cl
T
[48]> (ejemplo-loop-48)

2
3
4
5
6
7
8
9
10
11
NIL
```

**Iteración condicional.** Se corresponde con los bucles iterativos condicionales de los lenguajes imperativos clásicos.

```
(IF forma cláus {AND cláus}* [ELSE cláus {AND cláus}*] [END])
```

Si el valor de `forma` es `t` evalúa la primera secuencia de `cláusulas`. En otro caso, evalúa las `cláusulas` de la secuencia asociada al bloque `ELSE`. Devuelve `()` a su terminación.

**Ejemplo 5.20** *Consideremos la siguiente función incluida en el fichero* ∼/lisp/CLisp/examples/ejemplo-loops.cl*:*

```
(defun ejemplo-loop-53 (i j)
       (loop for x from i to j
             if (eq x 5)
                do (print x)
             else do (print "No tenemos un 5")
             end
             finally (print "Todo acabo")))
```

*que podemos ahora ejecutar, después de cargarla en el intérprete:*

```
[49]> (load "~/lisp/CLisp/examples/ejemplo-loops.cl")
;; Loading file /home/vilares/lisp/CLisp/examples/ejemplo-loops.cl ...
;; Loaded file /home/vilares/lisp/CLisp/examples/ejemplo-loops.cl
T
[50]> (ejemplo-loop-53 1 6)

"No tenemos un 5"
"No tenemos un 5"
"No tenemos un 5"
"No tenemos un 5"
5
"No tenemos un 5"
"Todo acabo"
NIL
```

`(ALWAYS forma cláusula {AND cláusula}*)`

Evalúa la secuencia de `cláusulas` siempre que todos los valores para `forma` sean `t`, devolviendo `t`. En otro caso, devuelve `()`. Forma de difícil asimilación.

**Ejemplo 5.21** *Consideremos la siguiente función incluida en el fichero* `~/lisp/CLisp/examples/ejemplo-loops.cl`*:*

```
(defun ejemplo-loop-54 (i j)
       (loop for x from i to j
             always (> x 4)
              do (print x)))
```

*que podemos ahora ejecutar, después de cargarla en el intérprete:*

```
[51]> (load "~/lisp/CLisp/examples/ejemplo-loops.cl")
;; Loading file /home/vilares/lisp/CLisp/examples/ejemplo-loops.cl ...
;; Loaded file /home/vilares/lisp/CLisp/examples/ejemplo-loops.cl
T
[52]> (ejemplo-loop-54 1 6)
```

```
NIL
[53]> (ejemplo-loop-54 4 6)
NIL
[54]> (ejemplo-loop-54 5 6)

5
6
T
```

`(NEVER forma cláusula {AND cláusula}*)`

Evalúa la secuencia de `cláusulas` salvo que `forma` tome un valor `t`, en cuyo caso devuelve `()`, terminando el bucle. En otro caso, cuando `forma` siempre se evalúe a `()`, devuelve `t`. Forma de difícil asimilación.

**Ejemplo 5.22** *Consideremos la siguiente función incluida en el fichero* ∼`/lisp/CLisp/examples/ejemplo-loops.cl`*:*

```
(defun ejemplo-loop-55 (i j)
       (loop for x from i to j
             never (> x 4)
               do (print x)))
```

*que podemos ahora ejecutar, después de cargarla en el intérprete:*

```
[55]> (load "~/lisp/CLisp/examples/ejemplo-loops.cl")
;; Loading file /home/vilares/lisp/CLisp/examples/ejemplo-loops.cl ...
;; Loaded file /home/vilares/lisp/CLisp/examples/ejemplo-loops.cl
T
[56]> (ejemplo-loop-55 1 8)

1
2
3
4
NIL
[57]> (ejemplo-loop-55 5 8)
NIL
[58]> (ejemplo-loop-55 1 4)

1
2
3
4
T
```

```
(THEREIS forma cláusula {AND cláusula}*)
```

Evalúa la secuencia de `cláusulas` hasta que `forma` tome un valor `t`, en cuyo caso devuelve `()`. En otro caso, cuando `forma` se evalúe alguna vez a `t`, devuelve `t`. Forma de difícil asimilación.

**Ejemplo 5.23** *Consideremos la siguiente función incluida en el fichero* ∼`/lisp/CLisp/examples/ejemplo-loops.cl`*:*

```
(defun ejemplo-loop-56 (i j)
       (loop for x from i to j
             thereis (> x 4)
               do (print x)))
```

*que podemos ahora ejecutar, después de cargarla en el intérprete:*

```
[59]> (load "~/lisp/CLisp/examples/ejemplo-loops.cl")
;; Loading file /home/vilares/lisp/CLisp/examples/ejemplo-loops.cl ...
;; Loaded file /home/vilares/lisp/CLisp/examples/ejemplo-loops.cl
T
[60]> (ejemplo-loop-56 1 4)

1
2
3
4
NIL
[61]> (ejemplo-loop-56 5 8)
T
[62]> (ejemplo-loop-56 1 8)

1
2
3
4
T
```

```
(UNLESS forma cláus {AND cláus}* [ELSE cláus {AND cláus}*]
[END])
```

Evalúa la primera secuencia de `cláusulas` a menos que `forma` tome un valor `t`. En otro caso, evalúa la secuencia de `cláusulas` asociada a la parte `else`. Devuelve `()` a su terminación.

**Ejemplo 5.24** *Consideremos las siguientes funciones incluidas en el fichero* ∼`/lisp/CLisp/examples/ejemplo-loops.cl`*:*

```
(defun ejemplo-loop-21 (i j)
       (loop for x from i to j
             unless (eq x 3)
                 do (print x)))

(defun ejemplo-loop-22 (i j)
       (loop for x from i to j
             unless (eq x 3)
                 do (print x)   ; DOING sinonimo de DO
             else do (print "Tenemos un 3")
             end                ; opcional (util en anidamientos)
             finally (print "Todo acabo")))
```

*que podemos ahora ejecutar, después de cargarla en el intérprete:*

```
[63]> (load "~/lisp/CLisp/examples/ejemplo-loops.cl")
;; Loading file /home/vilares/lisp/CLisp/examples/ejemplo-loops.cl ...
;; Loaded file /home/vilares/lisp/CLisp/examples/ejemplo-loops.cl
T
[64]> (ejemplo-loop-21 1 6)

1
2
4
5
6
NIL
[65]> (ejemplo-loop-22 1 6)

1
2
"Tenemos un 3"
4
5
6
"Todo acabo"
NIL
```

```
 (UNTIL forma cláusula {AND cláusula}*)
```

Evalúa la secuencia de `cláusulas` hasta que `forma` tome un valor `t`. Devuelve `()` a su terminación.

**Ejemplo 5.25** *Consideremos la siguiente función incluida en el fichero* `~/lisp/CLisp/examples/ejemplo-loops.cl`*:*

```
(defun ejemplo-loop-49 (i j)
       (loop for x from i to j
             until (> x 4)
               do (print x)))
```

*que podemos ahora ejecutar, después de cargarla en el intérprete:*

```
[66]> (load "~/lisp/CLisp/examples/ejemplo-loops.cl")
;; Loading file /home/vilares/lisp/CLisp/examples/ejemplo-loops.cl ...
;; Loaded file /home/vilares/lisp/CLisp/examples/ejemplo-loops.cl
T
[67]> (ejemplo-loop-49 1 6)

1
2
3
4
NIL
```

`(WHILE forma cláusula {AND cláusula}*)`

Evalúa en secuencia las `cláusulas` mientras que `forma` tome un valor `t`. Devuelve `()` a su terminación.

**Ejemplo 5.26** *Consideremos la siguiente función incluida en el fichero* `~/lisp/CLisp/examples/ejemplo-loops.cl`*:*

```
(defun ejemplo-loop-20 (i j)
       (loop for x from i to j
             while (< x 5)
                 do (print x)))
```

*que podemos ahora ejecutar, después de cargarla en el intérprete:*

```
[68]> (load "~/lisp/CLisp/examples/ejemplo-loops.cl")
;; Loading file /home/vilares/lisp/CLisp/examples/ejemplo-loops.cl ...
;; Loaded file /home/vilares/lisp/CLisp/examples/ejemplo-loops.cl
T
[69]> (ejemplo-loop-20 1 6)

1
2
3
4
NIL
```

**Iteración acumulativa sobre números.** Proporciona un conjunto de facilidades de tipo acumulativo sobre los resultados de las expresiones iteradas que, en cualquier caso, deben devolver valores numéricos.

```
(COUNT forma [INTO num] [tipo])
```

Evalúa en `forma`, devolviendo el número de veces que dichas evaluaciones tienen un valor `t`. Opcionalmente puede indicarse el nombre de una variable `var` y su `tipo`, en al que almacenar dicho resultado.

**Ejemplo 5.27** *Consideremos las siguientes funciones incluidas en el fichero* ∼`/lisp/CLisp/examples/ejemplo-loops.cl`*:*

```
(defun ejemplo-loop-36 (i j)
       (loop for x from i to j
                 count x))

(defun ejemplo-loop-37 (i j)
       (loop for x from i to j
                 count x into resultado integer
                 finally (return resultado)))

(defun ejemplo-loop-38 (i j)
       (loop for x from i to j
                 count x into resultado
                 finally (return resultado)))
```

*que podemos ahora ejecutar, después de cargarlas en el intérprete:*

```
[70]> (load "~/lisp/CLisp/examples/ejemplo-loops.cl")
;; Loading file /home/vilares/lisp/CLisp/examples/ejemplo-loops.cl ...
;; Loaded file /home/vilares/lisp/CLisp/examples/ejemplo-loops.cl
T
[71]> (ejemplo-loop-36 1 6)
6
[72]> (ejemplo-loop-37 1 6)
6
[73]> (ejemplo-loop-38 1 6)
6
```

```
(MAXIMIZE forma [INTO var] [tipo])
```

Evalúa `forma`, que debe devolver un valor numérico, devolviendo su máximo. Opcionalmente puede indicarse el nombre de una variable `var` y su `tipo`, en al que almacenar dicho resultado.

**Ejemplo 5.28** *Consideremos las siguientes funciones incluidas en el fichero* ∼/lisp/CLisp/examples/ejemplo-loops.cl*:*

```
(defun ejemplo-loop-42 (i j)
       (loop for x from i to j
                 maximize x))

(defun ejemplo-loop-43 (i j)
       (loop for x from i to j
                 maximize x into resultado integer
                 finally (return resultado)))

(defun ejemplo-loop-44 (i j)
       (loop for x from i to j
                 maximize x into resultado
                 finally (return resultado)))
```

*que podemos ahora ejecutar, después de cargarlas en el intérprete:*

```
[74]> (load "~/lisp/CLisp/examples/ejemplo-loops.cl")
;; Loading file /home/vilares/lisp/CLisp/examples/ejemplo-loops.cl ...
;; Loaded file /home/vilares/lisp/CLisp/examples/ejemplo-loops.cl
T
[75]> (ejemplo-loop-42 1 6)
6
[76]> (ejemplo-loop-43 1 6)
6
[77]> (ejemplo-loop-44 1 6)
6
```

```
 (MINIMIZE forma [INTO var] [tipo])
```

Evalúa `forma`, que debe devolver un valor numérico, devolviendo su mínimo. Opcionalmente puede indicarse el nombre de una variable `var` y su `tipo`, en al que almacenar dicho resultado.

**Ejemplo 5.29** *Consideremos las siguientes funciones incluidas en el fichero* ∼/lisp/CLisp/examples/ejemplo-loops.cl*:*

```
(defun ejemplo-loop-39 (i j)
       (loop for x from i to j
                 minimize x))

(defun ejemplo-loop-40 (i j)
       (loop for x from i to j
```

```
                minimize x into resultado integer
                finally (return resultado)))

(defun ejemplo-loop-41 (i j)
       (loop for x from i to j
                minimize x into resultado
                finally (return resultado)))
```

*que podemos ahora ejecutar, después de cargarlas en el intérprete:*

```
[78]> (load "~/lisp/CLisp/examples/ejemplo-loops.cl")
;; Loading file /home/vilares/lisp/CLisp/examples/ejemplo-loops.cl ...
;; Loaded file /home/vilares/lisp/CLisp/examples/ejemplo-loops.cl
T
[79]> (ejemplo-loop-39 1 6)
1
[80]> (ejemplo-loop-40 1 6)
1
[81]> (ejemplo-loop-41 1 6)
1
```

```
 (SUM forma [INTO var] [tipo])
```

Evalúa `forma`, devolviendo la suma de los resultados[3] de la evaluación de cada iteración. Opcionalmente puede indicarse una variable `var` y su `tipo`, en la que almacenar dicho resultado.

**Ejemplo 5.30** *Consideremos las siguientes funciones incluidas en el fichero* ∼`/lisp/CLisp/examples/ejemplo-loops.cl`*:*

```
(defun ejemplo-loop-33 (i j)
       (loop for x from i to j
                sum x))

(defun ejemplo-loop-34 (i j)
       (loop for x from i to j
                sum x into resultado
                finally (return resultado)))

(defun ejemplo-loop-35 (i j)
       (loop for x from i to j
                sum x into resultado integer
                finally (return resultado)))
```

[3] que forzósamente deben ser numéricos.

*que podemos ahora ejecutar, después de cargarlas en el intérprete:*

```
[82]> (load "~/lisp/CLisp/examples/ejemplo-loops.cl")
;; Loading file /home/vilares/lisp/CLisp/examples/ejemplo-loops.cl ...
;; Loaded file /home/vilares/lisp/CLisp/examples/ejemplo-loops.cl
T
[83]> (ejemplo-loop-33 1 6)
21
[84]> (ejemplo-loop-34 1 6)
21
[85]> (ejemplo-loop-35 1 6)
21
```

**Iteración acumulativa sobre listas.** Proporciona un conjunto de facilidades de tipo acumulativo sobre los resultados de las expresiones iteradas, que son recolectados en forma de lista.

```
(APPEND forma [INTO lista])
```

Evalúa `forma`, devolviendo una copia de la concatenación[4] del resultado[5] de su evaluación en cada iteración. Opcionalmente puede indicarse una `lista`, en la que almacenar dicho resultado.

**Ejemplo 5.31** *Consideremos las siguientes funciones incluidas en el fichero* ∼`/lisp/CLisp/examples/ejemplo-loops.cl`*:*

```
(defun ejemplo-loop-27 (lista)
       (loop for l on lista
             append l))

(defun ejemplo-loop-29 (lista)
       (loop for l on lista
             append l into resultado
             finally (return resultado)))
```

*que podemos ahora ejecutar, después de cargarlas en el intérprete:*

```
[86]> (load "~/lisp/CLisp/examples/ejemplo-loops.cl")
;; Loading file /home/vilares/lisp/CLisp/examples/ejemplo-loops.cl ...
;; Loaded file /home/vilares/lisp/CLisp/examples/ejemplo-loops.cl
T
[87]> (ejemplo-loop-27 '((a b) (c) (d e)))
```

[4] la concatenación se realiza mediante la función `append`.

[5] que deben ser forzósamente una lista.

```
((A B) (C) (D E) (C) (D E) (D E))
[88]> (ejemplo-loop-28 '((a b) (c) (d e)))
((A B) (C) (D E) (C) (D E) (D E))
```

```
 (NCONC forma [INTO lista])
```

Evalúa `forma`, devolviendo una concatenación física[6] del resultado de su evaluación[7] en cada iteración. Opcionalmente puede indicarse una `lista`, en la que almacenar dicho resultado.

**Ejemplo 5.32** *Consideremos las siguientes funciones incluidas en el fichero* ∼`/lisp/CLisp/examples/ejemplo-loops.cl`*:*

```
(defun ejemplo-loop-30 (lista)
       (loop for l on lista
             nconc (list l)))

(defun ejemplo-loop-31 (lista)
       (loop for l on lista
             nconc (list l) into resultado
             finally (return resultado)))
```

*que podemos ahora ejecutar, después de cargarlas en el intérprete:*

```
[89]> (load "~/lisp/CLisp/examples/ejemplo-loops.cl")
;; Loading file /home/vilares/lisp/CLisp/examples/ejemplo-loops.cl ...
;; Loaded file /home/vilares/lisp/CLisp/examples/ejemplo-loops.cl
T
[90]> (ejemplo-loop-30 '(a b c d e))
((A B C D E) (B C D E) (C D E) (D E) (E))
[91]> (ejemplo-loop-31 '(a b c d e))
((A B C D E) (B C D E) (C D E) (D E) (E))
```

```
 (COLLECT forma [INTO lista])
```

Evalúa `forma`, devolviendo una lista conteniendo el resultado de la evaluación de cada iteración. Opcionalmente puede indicarse una `lista`, en la que almacenar dicho resultado.

**Ejemplo 5.33** *Consideremos las siguientes funciones incluidas en el fichero* ∼`/lisp/CLisp/examples/ejemplo-loops.cl`*:*

[6] la concatenación se realiza mediante la función `nconc`.

[7] que debe ser forzósamente una lista.

```
(defun ejemplo-loop-24 (lista)
       (loop for l on lista
             collect (length l)))

(defun ejemplo-loop-25 (lista)
       (loop for l on lista
             collect (length l) into resultado
             finally (return resultado)))
```

*que podemos ahora ejecutar, después de cargarlas en el intérprete:*

```
[92]> (load "~/lisp/CLisp/examples/ejemplo-loops.cl")
;; Loading file /home/vilares/lisp/CLisp/examples/ejemplo-loops.cl ...
;; Loaded file /home/vilares/lisp/CLisp/examples/ejemplo-loops.cl
T
[93]> (ejemplo-loop-24 '(a b c d e))
(5 4 3 2 1)
[94]> (ejemplo-loop-25 '(a b c d e))
(5 4 3 2 1)
```

**Inicialización de variables locales** Permite la definición de variables locales al `loop`, pudiendo también proceder a su inicialización a un valor y tipo determinados.

```
 (WITH vars [tipo]=iniciales {AND vars [tipo]=iniciales}*)
```

Inicializa la secuencia, ligada por operadores `AND`, de listas de variables locales al bucle a las listas de valores iniciales indicadas, eventualmente asignándole el tipo. En caso de tratarse de sólo una variable, no es necesaria la utilización de listas.

**Ejemplo 5.34** *Consideremos las siguientes funciones incluidas en el fichero* ∼`/lisp/CLisp/examples/ejemplo-loops.cl`*:*

```
(defun ejemplo-loop-63 (i j)
       (loop with (init) integer = '(1)
             for x from i to j
             do (print (+ x init))))

(defun ejemplo-loop-64 (i j)
       (loop with (init) = '(1)
             for x from i to j
             do (print (+ x init))))
```

```
(defun ejemplo-loop-65 (i j)
       (loop with init = 1
             for x from i to j
             do (print (+ x init))))

(defun ejemplo-loop-66 (i j)
       (loop with init integer = 1
             for x from i to j
             do (print (+ x init))))
```

*que podemos ahora ejecutar, después de cargarlas en el intérprete:*

```
[95]> (load "~/lisp/CLisp/examples/ejemplo-loops.cl")
;; Loading file /home/vilares/lisp/CLisp/examples/ejemplo-loops.cl ...
;; Loaded file /home/vilares/lisp/CLisp/examples/ejemplo-loops.cl
T
[96]> (ejemplo-loop-63 1 6)

2
3
4
5
6
7
NIL
[97]> (ejemplo-loop-64 1 6)

2
3
4
5
6
7
NIL
[98]> (ejemplo-loop-65 1 6)

2
3
4
5
6
7
NIL
[99]> (ejemplo-loop-66 1 6)
```

```
2
3
4
5
6
7
NIL
```

**Prólogos y epílogos** Permiten la definición de expresiones a evaluar antes o después de finalizar el bucle.

```
 (INITIALLY forma)
```

Evalúa `forma` como prólogo al bucle. Dicho prólogo precede a todo el código del mismo, excepto las asignaciones iniciales asociadas a `with` y `for`. Aplica una construcción `progn` implícita.

**Ejemplo 5.35** *Consideremos la siguiente función incluida en el fichero* ∼`/lisp/CLisp/examples/ejemplo-loops.cl`*:*

```
(defun ejemplo-loop-46 (i j)
       (loop for x from i to j
             initially (print "Comenzamos") ; Se ejecuta al
             do (print x)))                 ; principio del loop
```

*que podemos ahora ejecutar, después de cargarla en el intérprete:*

```
[100]> (load "~/lisp/CLisp/examples/ejemplo-loops.cl")
;; Loading file /home/vilares/lisp/CLisp/examples/ejemplo-loops.cl ...
;; Loaded file /home/vilares/lisp/CLisp/examples/ejemplo-loops.cl
T
[101]> (ejemplo-loop-46 1 6)

"Comenzamos"
1
2
3
4
5
6
NIL
```

```
 (FINALLY [DO] forma)
```

Evalúa `forma` como epílogo al bucle. Dicho epílogo sigue al final de la ejecución normal del proceso de iteración. Aplica una construcción `progn` implícita. Una cláusula incondicional puede seguir a este tipo de constructor.

**Ejemplo 5.36** *Consideremos la siguiente función incluida en el fichero* `~/lisp/CLisp/examples/ejemplo-loops.cl`*:*

```
(defun ejemplo-loop-47 (i j)
       (loop for x from i to j
             initially (print "Comenzamos") ; Se ejecuta al principio del loop
             finally (print "Terminamos")   ; Se ejecuta al final del loop
             do (print x)))
```

*que podemos ahora ejecutar, después de cargarla en el intérprete:*

```
[102]> (load "~/lisp/CLisp/examples/ejemplo-loops.cl")
;; Loading file /home/vilares/lisp/CLisp/examples/ejemplo-loops.cl ...
;; Loaded file /home/vilares/lisp/CLisp/examples/ejemplo-loops.cl
T
[103]> (ejemplo-loop-47 1 6)

"Comenzamos"
1
2
3
4
5
6
"Terminamos"
NIL
```

## 5.6. Ambitos y salidas no locales

Definen un ámbito lexical, facilitando los mecanismos para las salidas no locales del mismo.

### (NAMED etiqueta {cláus}* RETURN-FROM etiqueta [forma])

Define un entorno léxico para un bucle `loop` mediante `etiqueta`, del que podemos salir mediante `return-from`. En el proceso de salida se evalúa `forma`, devolviendo su valor. Debe seguir a `loop` inmediatamente. No ejecuta `finally`.

**Ejemplo 5.37** *Consideremos la siguiente función incluida en el fichero* `~/lisp/CLisp/examples/ejemplo-loops.cl`*:*

```
(defun ejemplo-loop-62 (i j)
       (loop named max for x from i to j
             do (print x)
                (when (eq x 5)
                      (return-from max (print "Llegue a 5")))
                finally (print "Todo acabo")))
```

*que podemos ahora ejecutar, después de cargarla en el intérprete:*

```
[104]> (load "~/lisp/CLisp/examples/ejemplo-loops.cl")
;; Loading file /home/vilares/lisp/CLisp/examples/ejemplo-loops.cl ...
;; Loaded file /home/vilares/lisp/CLisp/examples/ejemplo-loops.cl
T
[105]> (ejemplo-loop-62 1 6)

1
2
3
4
5
"Llegue a 5"
"Llegue a 5"
```

**(RETURN-FROM etiqueta [forma])**

Devuelve el valor de `forma`, y provoca la salida de cualquier bloque léxico identificado con el símbolo `etiqueta`, que no es evaluado. Ello, bien sea un bucle `loop` nombrado con `named`, bien un bloque `block`.

**(RETURN [forma])**

Devuelve el valor de `forma`, y provoca la salida del bloque léxico más interno que lo contiene, bien sea un bucle `loop` nombrado con `named`, bien un bloque `block`. Equivale a `(result-from nil [forma])`.

**(LOOP-FINISH)**

Permite salir del bucle `loop` más interno, devolviendo cualquier valor asociado a un constructor de acumulación. En caso de ser especificada, se ejecutarán las cláusulas `finally`, al contrario de lo que ocurre con los constructores `return` y `return-from`.

**Ejemplo 5.38** *Consideremos la siguiente función incluida en el fichero* ∼`/lisp/CLisp/examples/ejemplo-loops.cl`*:*

```
(defun ejemplo-loop-67 (i j)
       (loop for x from i to j
             sum x
             when (eq x 5)
                  do (loop-finish)
             else do (print "No tenemos un 5")
             end
             finally (print "Todo acabo")))
```

*que podemos ahora ejecutar, después de cargarla en el intérprete:*

```
[106]> (load "~/lisp/CLisp/examples/ejemplo-loops.cl")
;; Loading file /home/vilares/lisp/CLisp/examples/ejemplo-loops.cl ...
;; Loaded file /home/vilares/lisp/CLisp/examples/ejemplo-loops.cl
T
[107]> (ejemplo-loop-67 1 6)

"No tenemos un 5"
"No tenemos un 5"
"No tenemos un 5"
"No tenemos un 5"
"Todo acabo"
15
```

## 5.7. Funciones de aplicación

La aplicación de funciones es el método primario para la construción de programas Lisp, donde las operaciones se describen como la *aplicación* de una función a sus argumentos. Siguiendo el modelo funcional, las funciones Lisp son *funciones anónimas* que tambien denominamos *funciones lambda*. En ese sentido, las *funciones nombradas* no son mas que funciones lambda ligadas a un descriptor en la tabla de símbolos.

**(LAMBDA lambda-lista . cuerpo)**

donde `lambda-lista` debe ser una lista que especifica los nombres de los parámetros de la función[8]. El `cuerpo` puede, a partir de ese momento, referenciar los argumentos usando los nombres de los parámetros. El `cuerpo` consiste de un número de formas, posiblemente cero, que se evalúan en secuencia; devolviéndose la última como valor de la aplicación de la función `lambda`. En caso de que el cuerpo contenga cero formas, la aplicación devuelve `()`. En detalle, la sintaxis de una expresión lambda es la siguiente:

[8]cuando la función denotada por la lambda expresión se aplica a una lista de argumentos, éstos se asignan a los parámetros identificados en la `lambda-lista`.

```
(LAMBDA  (  {var}*
            [&optional {var | (var [forma-inicial [svar]])}*]
            [&rest var]
            [&key { var | ({var | (palabra-clave var)} [forma-inicial [svar]])
            [&aux {var | (var [forma-inicial])}*])
         {  forma }*)
```

El valor de una expresión `lambda`, sin aplicar, es ella misma. Al respecto, es necesario puntualizar algunos aspectos importantes:

- Cada elemento de una `lambda-lista` es bien un *especificador de parámetro* o bien una *palabra clave de lambda-lista*, estas últimas[9] comenzando por &, es el caso de `&optional`, `&rest`, `&key` y `&aux`.

- Las `var` o `svar` son símbolos que actúan como el nombre de una variable.

- Una `forma-inicial`es cualquier forma Lisp.

- Una `palabra-clave` debe ser un un símbolo de palabra clave, por ejemplo `:start`.

Una `lambda-lista` tiene cinco partes, cualquiera de las cuales puede ser vacía. Más en detalle, cada una de estas partes se refiere al siguiente conjunto de especificadores de parámetros:

- *Requeridos*. Se trata de todos los especificadores que aparecen en la lambda-lsita hasta la primera palabra clave de lambda-lista, si esta última existe.

- *Opcionales*. Si es que están presentes, su secuencia viene precedida por la palabra clave de lambda-lista `&optional`. Cada especificador opcional puede asociar un valor por defecto o no. En caso de asociarlo, el opcional y su correspondiente valor se presentan en forma de una par de lista `(opcional valor-por-defecto)`.

- *Restante*. Si es que está presente, precedido por la palabra clave de lambda-lista `&rest` y reúne en una lista al resto de los argumentos de la función. Debe ser seguido bien de otra *palabra clave de lambda-lista* bien del final de la `lambda-lista`.

[9] no se trata de palabras clave en el sentido usual y, de hecho, no se incluyen en la lista de palabras clave del lenguaje. Terminología confusa, se mantiene, como en tantos otros casos, por razones estrictamente históricas.

- *Palabras clave.* Si es que están presentes, su secuencia viene precedida por la palabra clave de lambda-lista `&key`. Cada palabra clave puede asociar un valor por defecto o no. En caso de asociarlo, la palabra clave y su correspondiente valor se presentan en forma de una par de lista `(palabra-clave valor-por-defecto)`.

  En las llamadas a la función `lambda` que luego realice el programador, la asignación de valores efectivos a las palabras clave[10] se realiza anteponiendo `:` a la palabra clave, seguida del valor que le deseamos asignar precedido por `#'`.

- *Variables auxiliares.* No se trata realmente de parámetros, sino de variables locales que se asignan secuencialmente. En este sentido, `&aux` desempeña el mismo papel que la función `let*`. Si la palabra clave de lambda-lista `&aux` está presente, entonces todos los especificadores que le siguen lo son de variables auxiliares.

  Cada variable auxiliar puede asociar un valor por defecto, resultante de evaluar una expresión Lisp. En caso de asociarlo, la palabra clave y su correspondiente valor se presentan en forma de un par de lista `(variable-auxiliar expresión-por-defecto)`. En caso de no explicitarlo, el valor asociado por defecto es `()`.

Obviamente, dado que una función nombrada no es más que una función anónima ligada al identificador que le da nombre, la descripción de especificadores que venimos de introducir es igualmente válida para estas últimas.

**Ejemplo 5.39** *Para ilustrar el funcionamiento de una función* `lambda` *nos limitaremos al caso más simple, sin utilizar otros especificadores que los de los parámetros requeridos. Los demás serán ilustrados sobre funciones nombradas.*
*Consideremos el siguiente código que ilustra la operación aritmética* `x * y` *dados dos argumentos numéricos* `x` *e* `y`*:*

```
(lambda (x y) (* x y))
```

*el resultado de interpretarla interactivamente es el siguiente:*

```
[117]> (lambda (x y) (* x y))
#<FUNCTION :LAMBDA (X Y) (* X Y)>
```

*esto es, ella misma, como ya habíamos comentado. Si queremos obtener su valor, por ejemplo, para* `x=3` *e* `y=2`*; tendremos que aplicar la expresión* `lambda` *sobre la lista de argumentos* `'(3 2)`

[10] en caso de no utilizar los valores por defecto.

```
[118]> (apply #'(lambda (x y) (* x y)) '(3 2))
6
```

Para ilustrar ahora el uso de los especificadores `&aux`, `&key`, `&optional` y `&rest` utilizaremos ejemplos de funciones nombradas.

**Ejemplo 5.40** *Consideremos el siguiente conjunto de funciones incluidas en el fichero* ∼`/lisp/CLisp/examples/ejemplo-aplicacion.cl`. *Comenzaremos por el caso de los argumentos opcionales:*

```
(defun ejemplo-opcional-1 (x &optional y)
       (format t "~&X is ~S" x)
       (format t "~&Y is ~S" y)
       (list x y))

(defun ejemplo-opcional-2 (dividendo &optional (divisor 2))
       (format t "~&~S ~A divisible por ~S"
               dividendo
               (if (zerop (rem dividendo divisor)) "es" "no es")
               divisor))
```

*que podemos ahora ejecutar, después de cargarlas en el intérprete:*

```
[118]> (load "~/lisp/CLisp/examples/ejemplo-aplicacion.cl")
;; Loading file /home/vilares/lisp/CLisp/examples/ejemplo-aplicacion.cl ...
;; Loaded file /home/vilares/lisp/CLisp/examples/ejemplo-aplicacion.cl
T
[119]> (ejemplo-opcional-1 3 5)
X is 3
Y is 5
(3 5)
[220]> (ejemplo-opcional-1 4)
X is 4
Y is NIL
(4 NIL)
[221]> (ejemplo-opcional-2 27 3)
27 es divisible por 3
NIL
[222]> (ejemplo-opcional-2 27)
27 no es divisible por 2
NIL
```

*Pasamos ahora al caso de la utilización de argumentos restantes en una función. Consideremos en este caso el siguiente ejemplo, incluido en el mismo fichero que el anterior:*

```
(defun ejemplo-resto-1 (&rest argumentos)
       (/ (reduce #'+ argumentos)
          (length argumentos)))
```

*que ejecutamos despues de cargarlo en el intérprete:*

```
[223]> (load "~/lisp/CLisp/examples/ejemplo-aplicacion.cl")
;; Loading file /home/vilares/lisp/CLisp/examples/ejemplo-aplicacion.cl ...
;; Loaded file /home/vilares/lisp/CLisp/examples/ejemplo-aplicacion.cl
T
[224]> (ejemplo-resto-1 1 2 3 4 5)
3
```

*Tocamos ahora el caso de los argumentos con carácter de palabras clave.*
*Consideremos el ejemplo siguiente:*

```
(defun ejemplo-palabra-clave-1 (nombre &key (tamano 'regular)
                                            (helado 'vainilla)
                                            (jarabe 'fresa)
                                            avellanas
                                            cerezas
                                            chocolate)
       (list 'helado
              (list 'para nombre)
              (list helado 'con jarabe 'jarabe)
              (list 'entradas '= (remove nil
                                                  (list (and avellanas 'avellanas)
                                                        (and cerezas 'cerezas)
                                                        (and chocolate 'chocolate))))))
```

*que ejecutamos despues de cargar el fichero donde lo hemos incluido:*

```
[225]> (load "~/lisp/CLisp/examples/ejemplo-aplicacion.cl")
;; Loading file /home/vilares/lisp/CLisp/examples/ejemplo-aplicacion.cl ...
;; Loaded file /home/vilares/lisp/CLisp/examples/ejemplo-aplicacion.cl
T
[226]> (ejemplo-palabra-clave-1 'Pepe)
(HELADO (PARA PEPE) (VAINILLA CON FRESA JARABE) (ENTRADAS = NIL))
[227]> (ejemplo-palabra-clave-1 'Sonia
                                :jarabe 'albaricoque
                                :avellanas t
                                :cerezas t)
(HELADO (PARA SONIA) (VAINILLA CON ALBARICOQUE JARABE)
 (ENTRADAS = (AVELLANAS CEREZAS)))
```

*Ilustraremos finalmente el caso de las variables auxiliares. Consideremos para ello el ejemplo que sigue:*

```
(defun ejemplo-variables-auxiliares-1 (&rest argumentos
                                       &aux (longitud (length argumentos)))
       (/ (reduce #'+ argumentos) longitud))
```

*que podemos ejecutar después de cargar el fichero que las incluye:*

```
[228]> (load "~/lisp/CLisp/examples/ejemplo-aplicacion.cl")
;; Loading file /home/vilares/lisp/CLisp/examples/ejemplo-aplicacion.cl ...
;; Loaded file /home/vilares/lisp/CLisp/examples/ejemplo-aplicacion.cl
T
[229]> (ejemplo-variables-auxiliares-1 1 2 3 4 5)
3
```

### 5.7.1. Funciones simples de aplicación.

Estas funciones se encuentran en el origen del funcionamiento básico del motor del intérprete, junto con **eval**.

#### (APPLY función arg args)

Devuelve como respuesta, el resultado de la aplicación de la `función` a la lista de argumentos cuyo `car` es `arg` y cuyo `cdr` es `args`. En este caso `función` ha de entenderse como la *interpretación funcional* de un símbolo, lo que conlleva la utilización de `function` o de su abreviatura `#'`.

**Ejemplo 5.41** *Podemos considerar un ejemplo sencillo, simplemente tecleando algunas expresiones en interactivo sobre el intérprete:*

```
[229]> (apply #'+ 1 '(2 3 4))
10
[230]> (cons 1 2)
(1 . 2)
[231]> (apply #'cons 1 '(2))
(1 . 2)
[232]> (apply 'cons 1 '(2))
(1 . 2)
[233]> (apply (function cons) 1 '(2))
(1 . 2)
[234]> (apply #'cons (+ 1 1) (list (+ 1 2)))
(2 . 3)
[235]> (apply #'list 1 '(2 3 4 5))
(1 2 3 4 5)
```

```
[236]> (apply 'list 1 '(2 3 4 5))
(1 2 3 4 5)
[237]> (apply #'list 1 (list 2 3 4 5))
(1 2 3 4 5)
[238]> (apply 'list 1 (list 2 3 4 5))
(1 2 3 4 5)
[239]> (apply (function list) 1 (list 2 3 4 5))
(1 2 3 4 5)
```

**Ejemplo 5.42** *Otro ejemplo sencillo es el de la implementación de la función* `funcall` *que luego pasamos a comentar. El código sería el siguiente:*

```
(defun mi-funcall (&rest args)
       (apply (car args) (cadr args) (cddr args)))
```

*que podemos, por ejemplo, bien interpretar en interactivo bien incluir en nuestro fichero* ∼`/lisp/CLisp/examples/ejemplo-aplicacion.cl`*, para luego cargarlo en el intérprete y evaluar sobre un ejemplo sencillo:*

```
[236]> (load "~/lisp/CLisp/examples/ejemplo-aplicacion.cl")
;; Loading file /home/vilares/lisp/CLisp/examples/ejemplo-aplicacion.cl ...
;; Loaded file /home/vilares/lisp/CLisp/examples/ejemplo-aplicacion.cl
T
[237]> (mi-funcall #'cons 1 2)
(1 . 2)
```

### (FUNCALL función {arg}*)

Devuelve como respuesta, el resultado de la aplicación de la función `función` a la lista de argumentos asociados. En este caso `función` ha de entenderse como la *interpretación funcional* de un símbolo, lo que conlleva la utilización de `function` o de su abreviatura `#'`.

**Ejemplo 5.43** *Podemos considerar un ejemplo sencillo, simplemente tecleando algunas expresiones en interactivo sobre el intérprete:*

```
[238]> (funcall #'+ 1 2 3 4)
10
[239]> (funcall '+ 1 2 3 4)
10
[240]> (funcall (function +) 1 2 3 4)
10
[241]> (cons 1 2)
(1 . 2)
[242]> (funcall #'cons 1 2)
```

```
(1 . 2)
[243]> (funcall 'cons 1 2)
(1 . 2)
[244]> (funcall (function cons) 1 2)
(1 . 2)
[245]> (funcall #'cons (+ 1 1) (+ 1 2))
(2 . 3)
[246]> (funcall #'list 1 2 3 4 5)
(1 2 3 4 5)
```

### 5.7.2. Funciones de aplicación de tipo map.

Permiten la aplicación repetida de una función a un conjunto de secuencias/listas que son tomadas como argumentos. La función de aplicación puede tener un número cualquiera de argumentos, y las aplicaciones se interrumpen cuando se consume una de las secuencias/listas tomadas como argumentos. Para aplicar argumentos *invariantes*, se pueden incluir estos en una lista circular[11]

**(MAP tipo función secuencia {secuencia}*)**

Devuelve una secuencia del `tipo` indicado, siendo necesario que el número de `secuencias` argumento sea igual a la aridad de la `función`. La secuencia obtenida es tal que su elemento i-ésimo es el resultado de la aplicación de la `función` sobre los i-ésimos componentes de las `secuencias` en el orden en el que aparecen en la llamada a `map`. La talla del resultado es la mínima de la del conjunto de `secuencias` argumento. Las `secuencias` no tienen porque ser del mismo tipo.

**Ejemplo 5.44** *Podemos considerar un ejemplo sencillo, simplemente tecleando algunas expresiones en interactivo sobre el intérprete:*

```
[226]> (map 'cons #'cons '(1 2 3) '(A B C D))
((1 . A) (2 . B) (3 . C))
[227]> (map 'cons 'cons '(1 2 3) '(A B C D))
((1 . A) (2 . B) (3 . C))
[228]> (map 'cons (function cons) '(1 2 3) '(A B C D))
((1 . A) (2 . B) (3 . C))
[229]> (map 'vector #'cons '(1 2 3) '(A B C D))
#((1 . A) (2 . B) (3 . C))
[230]> (map 'cons #'cons '(1 2 3) #(A B C D))
((1 . A) (2 . B) (3 . C))
```

[11] que en otros intérpretes Lisp, como LeLisp están predefinidos, pero que en Common Lisp habrá que construir explícitamente.

### (MAP-INTO secuencia función {secuencia}*)

La funcionalidad es la misma de `map`, con dos salvedades:

1. La primera es que el resultado de la evaluación de la `función` sobre el conjunto de `secuencias` finales se guarda, además, en la `secuencia` que sirve de primer argumento para `map-into`. Esta asignación es destructiva, de forma que el resultado de la i-ésima aplicación de la función, reemplaza al elemento en la posición i-ésima del primer argumento de `map-into`.
2. La segunda es que el tipo del resultado es el fijado por la propia naturaleza del primer argumento de la función `map-into`.

**Ejemplo 5.45** *Podemos considerar un ejemplo sencillo, simplemente tecleando algunas expresiones en interactivo sobre el intérprete:*

```
[229]> (defvar lista '(1 2))
LISTA
[230]> (defvar vector #(a b c d))
VECTOR
[231]> (map-into vector #'cons '(1 2) #(3 4 5))
#((1 . 3) (2 . 4) 5)
[232]> vector
#((1 . 3) (2 . 4) C D)
```

### (SOME predicado secuencia {secuencia}*)

Devuelve el primer valor, si existe, diferente de `()` resultado de la aplicación del `predicado` sobre las `secuencias` de argumentos. En otro caso devuelve `()`. La aplicación i-ésima es el resultado de la aplicación del `predicado` sobre los i-ésimos componentes de las `secuencias` en el orden en el que aparecen en la llamada a `some`. Los nombres de los

**Ejemplo 5.46** *Podemos ilustrar el funcionamiento de* `some`*, simplemente tecleando algunas expresiones en interactivo directamente sobre el intérprete:*

```
[233]> (some #'equal #(1 2 3 4) #(a b 3 c))
T
[234]> (some #'+ #(1 2 3 4) #(5 b 3 c))
6
[235]> (some #'(lambda (x y) (and (< x y) (* x y))) '(3 2) '(1 3))
6
[236]> (some #'(lambda (x y)
                       (and (< x y)
```

```
                          (prog1 t
                                 (format t "~D es menor que
                                           ~D ~%" x y) t)))
            '(4 5 2) '(1 2 3))
2 es menor que 3
T
```

**(EVERY predicado secuencia {secuencia}*)**

Evalúa el `predicado` sobre las `secuencias` de argumentos siguiendo el mismo procedimiento que en `some`. Devuelve `()` tan pronto como el valor de una de estas aplicaciones es `()`. En otro caso, devuelve `t`.

**Ejemplo 5.47** *Podemos ilustrar el funcionamiento de* `every`*, simplemente tecleando algunas expresiones en interactivo directamente sobre el intérprete:*

```
[237]> (every 'equal #(1 2 3 4) #(a b 3 c))
NIL
[238]> (every '+ #(1 2 3 4) #(5 6 7 8))
T
[239]> (every #'(lambda (x y)
                        (and (< x y)
                             (prog1 t
                                    (format t "~D es menor que
                                              ~D ~%" x y) t)))
             '(1 2 3) '(4 5 2))
1 es menor que 4
2 es menor que 5
NIL
```

**(NOTANY predicado secuencia {secuencia}*)**

Evalúa el `predicado` sobre las `secuencias` de argumentos siguiendo el mismo procedimiento que en `some`. Devuelve `()` tan pronto como el valor de una de estas aplicaciones es diferente de `()`. En otro caso, devuelve `t`.

**Ejemplo 5.48** *Podemos ilustrar el funcionamiento de* `notany`*, simplemente tecleando algunas expresiones en interactivo directamente sobre el intérprete:*

```
[240]> (notany 'equal #(1 2 3 4) #(a b c d))
T
[241]> (notany '+ #(1 2 3 4) #(5 6 7 8))
NIL
[242]> (notany #'(lambda (x y)
                         (and (< x y)
```

```
                              (prog1 t
                                     (format t "~D es menor que
                                               ~D ~%" x y) t)))
               '(4 5 2) '(1 2 3))
2 es menor que 3
NIL
```

**(NOTEVERY predicado secuencia {secuencia}*)**

Evalúa el `predicado` sobre las `secuencias` de argumentos siguiendo el mismo procedimiento que en `some`. Devuelve `t` tan pronto como el valor de una de estas aplicaciones es `()`. En otro caso, devuelve `()`.

**Ejemplo 5.49** *Podemos ilustrar el funcionamiento de* `notevery`*, simplemente tecleando algunas expresiones en interactivo directamente sobre el intérprete:*

```
[243]> (notevery 'equal #(1 2 3 4) #(a b c d))
T
[244]> (notevery '+ #(1 2 3 4) #(5 6 7 8))
NIL
[245]> (notevery #'(lambda (x y)
                           (and (< x y)
                                (prog1 t
                                       (format t "~D es menor que
                                                  ~D ~%" x y) t)))
                 '(1 2 3) '(4 5 2))
1 es menor que 4
2 es menor que 5
T
```

**(REDUCE función secuencia [:FROM-END {t|()}]**
**[:START inicio]**
**[:END final]**
**[:INITIAL-VALUE valor])**

Devuelve la combinación de la aplicación, a todos los elementos de `secuencia`, de la operación binaria[12] definida por la `función` y considerando una asociatividad por la izquierda. Por defecto la combinación se realizará de derecha a izquierda, salvo que `:from-end` tome un valor `t`. Los valores de `:start` y `:end`, de estar presentes, marcan las posiciones de `inicio` y `final` para las aplicaciones de la `función`. De estar presente, el `valor`

[12] nos referimos aquí por *operación binaria* a cualquier función que admita dos argumentos. En este sentido, por ejemplo, `+` es una operación binaria aunque su número de argumentos pueda ser mayor que dos.

de `:initial-value` se utilizará como primer argumento en la primera aplicación de la `función`.

**Ejemplo 5.50** *Podemos ilustrar el funcionamiento de* `reduce`*, tecleando algunas expresiones sobre el intérprete:*

```
[247]> (reduce #'(lambda (x y) (* x y)) '(3 2))
6
[248]> (reduce #'(lambda (x y) (* x y)) #(3 2))
6
[249]> (reduce #'- '(1 2 3 4))              ; (- (-(- 1 2) 3) 4) = -8
[250]> (reduce '- '(1 2 3 4))               ; (- (-(- 1 2) 3) 4) = -8
-8
[251]> (reduce #'- '(1 2 3 4) :from-end ()) ; (- (-(- 1 2) 3) 4) = -8
-8
[252]> (reduce #'- '(1 2 3 4) :from-end t)  ; (- 1 (- 2 (- 3 4))) = -2
-2
[253]> (reduce #'list '(1 2 3 4 5))         ; observar la aplicacion binaria
((((1 2) 3) 4) 5)
[254]> (reduce #'list '(2 3 4 5) :initial-value 1)
((((1 2) 3) 4) 5)
[255]> (reduce #'list '(1 2 3 4 5) :start 1 :end 3)
(2 3)
[256]> (reduce #'list '(1 2 3 4 5) :end 3)
((1 2) 3)
[257]> (reduce #'list '(1 2 3 4 5) :start 1)
(((2 3) 4) 5)
```

### (MAPCAR función lista {lista}*)

Aplica la función `función` a los `car` de listas de argumentos asociadas, luego a todos los `cadr` de dichas listas, luego a los `caddr`; hasta que se lee el fin de alguna de ellas. Devuelve la lista de respuestas de todas las aplicaciones realizadas.

**Ejemplo 5.51** *Podemos considerar un ejemplo sencillo, simplemente tecleando algunas expresiones en interactivo sobre el intérprete:*

```
[247]> (mapcar #'cons '(1 2 3) '(A B C D))
((1 . A) (2 . B) (3 . C))
[248]> (mapcar #'cons '(1 2 3) '())
NIL
```

### (MAPC función lista {lista}*)

Su funcionalidad es análoga a la de `mapcar`, con la salvedad de que el resultado de la evaluación es siempre la primera lista argumento.

**Ejemplo 5.52** *Podemos considerar un ejemplo sencillo, simplemente tecleando algunas expresiones en interactivo sobre el intérprete, para mostrar la funcionalidad de* `mapc` *y su diferencia con* `mapcar`*:*

```
[249]> (mapc #'(lambda (x y) (print (* x y))) (list 1 0 2) (list 3 4 5))

3
0
10
(1 0 2)
[250]> (mapcar #'(lambda (x y) (print (* x y))) (list 1 0 2) (list 3 4 5))

3
0
10
(3 0 10)
```

### (MAPCAN función lista {lista}*)

La funcionalidad es la misma que la de `mapcar`, con la diferencia de que a fortiori cada aplicación de la función debe devolver como respuesta una lista ó `()`. Devuelve como respuesta una lista resultado de la concatenación física, mediante el uso de `nconc`, de todas las respuestas parciales.

**Ejemplo 5.53** *Podemos considerar un ejemplo sencillo, simplemente tecleando algunas expresiones en interactivo sobre el intérprete, para mostrar la funcionalidad de* `mapcan` *y su diferencia con* `mapcar`*:*

```
[251]> (mapcan #'(lambda (x) (and (numberp x) (list x))) '(a 1 b c 3 4 d 5))
(1 3 4 5)
[252]> (mapcar 'list '(1 2 3) '(a b c))
((1 A) (2 B) (3 C))
[253]> (mapcan 'list '(1 2 3) '(a b c))
(1 A 2 B 3 C)
```

### (MAPLIST función lista {lista}*)

Aplica la función `función` a las listas de argumentos asociadas, luego a todos los `cdr` de dichas listas, luego a los `cddr`; hasta que se lee el fin de alguna de ellas. Devuelve la lista de respuestas de todas las aplicaciones realizadas.

**Ejemplo 5.54** *Podemos considerar un ejemplo sencillo, simplemente tecleando algunas expresiones en interactivo sobre el intérprete:*

```
[254]> (maplist #'cons '(1 2 3) '(A B C D))
(((1 2 3) A B C D) ((2 3) B C D) ((3) C D))
[255]> (maplist #'cons '(1 2 3) '())
NIL
```

### (MAPL función lista {lista}*)

Su funcionalidad es análoga a la de `maplist`, con la salvedad de que el resultado de la evaluación es siempre la primera lista argumento.

**Ejemplo 5.55** *Podemos considerar un ejemplo sencillo, simplemente tecleando algunas expresiones en interactivo sobre el intérprete, para mostrar la funcionalidad de* `mapl` *y su diferencia con* `maplist`*:*

```
[256]> (mapl #'(lambda (x y) (print (append x y)))
               (list 1 0 2) (list 3 4 5))

(1 0 2 3 4 5)
(0 2 4 5)
(2 5)
(1 0 2)
[257]> (maplist #'(lambda (x y) (print (append x y)))
                  (list 1 0 2) (list 3 4 5))

(1 0 2 3 4 5)
(0 2 4 5)
(2 5)
((1 0 2 3 4 5) (0 2 4 5) (2 5))
```

### (MAPCON función lista {lista}*)

La funcionalidad es la misma que la de `maplist`, con la diferencia de que a fortiori cada aplicación de la función debe devolver como respuesta una lista ó `()`. Devuelve como respuesta una lista resultado de la concatenación física, mediante el uso de `nconc`, de todas las respuestas parciales.

**Ejemplo 5.56** *Podemos considerar un ejemplo sencillo, simplemente tecleando algunas expresiones en interactivo sobre el intérprete, para mostrar la funcionalidad de* `mapcon` *y su diferencia con* `maplist`*:*

```
[258]> (maplist #'list '(a b))
(((A B)) ((B)))
[259]> (mapcon #'list '(a b))
((A B) (B))
```

## 5.8. Especificadores de tipo

Podemos definir nuevos especificadores de tipo de dos formas diferentes:

1. La primera pasa por definir un tipo estructurado mediante `defstruct`, lo que automáticamente conlleva que el nombre de la estructura se convierta en símbolo de un nuevo especificador de tipo.

2. La segunda pasa por utilizar la forma especial `deftype`, que permite definir nuevas abreviaturas para especificadores de tipo.

**(TYPEP objeto tipo)**

Verifica si `objeto` es de tipo `tipo`, donde `tipo` ha de ser símbolo asociado al nombre de un tipo aceptado en Common Lisp. Si el test es positivo, la respuesta es `t`, en otro caso será `()`.

**Ejemplo 5.57** *Tecleamos algunas expresiones en interactivo sobre el intérprete, para mostrar la funcionalidad de* `typep`*:*

```
[260]> (typep 5 'integer)
T
[261]> (typep '() 'cons)
NIL
[262]> (typep '(1 2) 'cons)
T
```

En cuanto a su uso, `typep` está estrechamente ligada a `coerce` y `deftype`, que presentamos a continuación.

**(SUBTYPEP subtipo tipo)**

Verifica si `subtipo` es un subtipo reconocible del tipo `tipo`. Devuelve dos valores:

- El primero de ellos es `t` en caso afirmativo, y `()` en otro caso.

- El segundo expresa la fiabilidad del primer diagnóstico, siendo `t` en caso afirmativo, y `()` en otro caso.

**Ejemplo 5.58** *Tecleamos algunas expresiones en interactivo sobre el intérprete, para mostrar la funcionalidad de* `subtypep`*:*

```
[263]> (subtypep 'integer 'symbol)
NIL ;
T
[264]> (subtypep 'integer 'cons)
NIL ;
T
[265]> (subtypep 'integer 'real)
T ;
T
[266]> (subtypep '(array single-float) '(array float))
T ;
T
```

### (DEFTYPE nombre lambda-lista {forma}*)

Su funcionamiento es similar al de `defmacro`. Cuando `nombre` es usado como una denominación de tipo, las `formas` del cuerpo son evaluadas en secuencia, constituyendo un especificador de tipo. La `lambda-lista` de argumentos puede incluir marcadores `&optional` y `&rest` en sus argumentos, siendo expandida de modo análogo a como ocurre en una `defmacro`, con la salvedad de que el valor por defecto para los parámetros opcionales es `*` y no `()`. Su uso está directamente relacionado con `typep`.

**Ejemplo 5.59** *Tecleamos algunas expresiones en interactivo sobre el intérprete, para mostrar la funcionalidad de* `deftype` *asociada a* `typep`*:*

```
[267]> (deftype nulo () '(satisfies null))
NULO
[268]> (typep '() 'nulo)
T
[269]> (typep 'a 'nulo)
NIL
 (deftype lista () '(or nulo cons))
LISTA
[270]> (typep '(1 2) 'lista)
T
[271]> (typep '(1 (A B) 2) 'lista)
T
[272]> (typep '() 'lista)
T
[273]> (typep 'a 'lista)
```

```
NIL
[274]> (typep '5 'lista)
NIL
```

*donde una lista de la forma* `(SATISFIES predicado)` *representa a todos aquellos objetos sobre los que* `predicado` *devuelve un valor* `t`*, cuando es llamado con uno de esos objetos como argumento. El* `predicado` *debe tener obligatoriamente un solo argumento.*

### (COERCE objeto tipo)

Intenta convertir, si posible, el `objeto` al `tipo` indicado. Si `objeto` es ya del `tipo` indicado, entonces devuelve dicho `objeto`. En otro caso, se permiten ciertos tipos de conversión:

- Si `tipo` es un tipo de secuencia[13], entonces `objeto` la conversión se realizará de forma natural, si posible.
- Si `tipo` es `character`, entonces las cadenas de longitud uno y símbolos referidos a nombres de un sólo carácter pueden ser convertidos.
- Si `tipo` es `float`, entonces los enteros pueden ser convertidos en versiones del lenguaje que soporten flotantes.

en cualquier otro caso `coerce` señala un error.

**Ejemplo 5.60** *Tecleamos algunas expresiones en interactivo sobre el intérprete, para mostrar la funcionalidad de* `deftype` *asociada a* `typep`*:*

```
[275]> (coerce #(a b c) 'list)
(A B C)
[276]> (coerce '(#\a #\b #\c) 'string)
"abc"
[277]> (coerce 1 'float)
1.0
[278]> (coerce 'a 'integer)

*** - COERCE: A no puede convertirse al tipo INTEGER
Es posible continuar en los siguientes puntos:
ABORT          :R1      Abort debug loop
ABORT          :R2      Abort debug loop
ABORT          :R3      Abort main loop
```

[13] cadena, lista, vector, ...

### (TYPE-OF objeto)

Devuelve el especificador de tipo del `objeto`.

**Ejemplo 5.61** *Tecleamos algunas expresiones en interactivo sobre el intérprete, para mostrar la funcionalidad de* `deftype` *asociada a* `typep`*:*

```
[279]> (typep (+ (expt 9 3) (expt 10 3)) 'integer)
T
[280]> (subtypep (type-of (+ (expt 1 3) (expt 12 3))) 'integer)
T ;
T
[281]> (type-of 'Felipe)
SYMBOL
[282]>  (type-of "Felipe")
(SIMPLE-BASE-STRING 6)
```

### (TYPECASE forma {(tipo {forma}*)}* [{OTHERWISE | T} forma])

Es una función condicional que escoge una de sus cláusulas asociadas, `(tipo {forma}*)`, examinando el tipo del objeto resultante de evaluar la forma inicial. Una vez seleccionada la cláusula, la evaluación es análoga a `case`. Puede incluir una cláusula `otherwise`, siempre en último lugar, para capturar el tratamiento de casos no capturados por otras cláusulas. Este último comportamiento puede simularse también incluyendo, siempre en último lugar, una cláusula cuya condición sea un `t`.

**Ejemplo 5.62** *Consideremos las siguientes funciones incluidas en el fichero* `~/lisp/CLisp/examples/ejemplo-tipos.cl`*. Comenzaremos por el caso de los argumentos opcionales:*

```
(defun ejemplo-tipos-1 (x)
       (typecase x
                 (number (print "Es un numero"))
                 (character (print "Es un caracter"))
                 (symbol (print "Es un simbolo"))
                 (cons (print "Es una lista"))
                 (otherwise (print "Otros casos"))))

(defun ejemplo-tipos-2 (x)
       (typecase x
                 (number (print "Es un numero"))
                 (character (print "Es un caracter"))
                 (symbol (print "Es un simbolo"))
```

```
                (cons (print "Es una lista"))
                (t (print "Otros casos"))))
```

*que podemos ahora ejecutar, después de cargarlas en el intérprete:*

```
[283]> (load "~/lisp/CLisp/examples/ejemplo-tipos.cl")
;; Loading file /home/vilares/lisp/CLisp/examples/ejemplo-tipos.cl ...
;; Loaded file /home/vilares/lisp/CLisp/examples/ejemplo-tipos.cl
T
[284]> (ejemplo-tipos-1 '(1 2))

"Es una lista"
"Es una lista"
[285]> (ejemplo-tipos-1 #(1 2))

"Otros casos"
"Otros casos"
[286]> (ejemplo-tipos-2 #(1 2))

"Otros casos"
"Otros casos"
```

## 5.9. Funciones sobre secuencias y listas

La utilización de listas como estructura básica de organización de datos en Lisp está al origen de un estilo propio de programación. De hecho, si algo ha propiciado la extensión de Lisp es, junto con su versatilidad en el campo del cálculo simbólico, su optimización en cuanto al manejo de dichas estructuras.

### 5.9.1. Funciones de acceso

Aceptan una lista como argumento, y permiten acceder diferentes posiciones de la misma.

**(CAR lista)**

Si `lista` es una lista, devuelve su primer elemento. En otro caso devuelve `()`.

**Ejemplo 5.63** *Podemos considerar un ejemplo sencillo, simplemente tecleando algunas expresiones en interactivo sobre el intérprete:*

```
[156]> (car '(1 2 3))
```

```
1
[157]> (car '((1 A) "pepe" 0.12))
(1 A)
[158]> (car (list (+ 1 2) (* 5 4) 'a))
3
[159]> (car '())
NIL
```

### (CDR lista)

Si `lista` es una lista, devuelve la misma sin su primer elemento. Si `lista` es la lista vacía, devuelve `()`.

**Ejemplo 5.64** *Podemos considerar un ejemplo sencillo, simplemente tecleando algunas expresiones en interactivo sobre el intérprete:*

```
[159]> (cdr '(1 2 3))
(2 3)
[160]> (cdr '((1 A) "pepe" 0.12))
("pepe" 0.12)
[161]> (cdr (list (+ 1 2) (* 5 4) 'a))
(20 A)
[162]> (cdr '())
NIL
```

### (C . . . R lista)

Se trata de combinaciones superpuestas de `car` y `cdr` con objeto de aliviar la carga de paréntesis.

**Ejemplo 5.65** *Podemos considerar un ejemplo sencillo, simplemente tecleando algunas expresiones en interactivo sobre el intérprete:*

```
[163]> (cadr '(1 2 3))
2
[164]> (cddr '(1 2 3))
(3)
[165]> (caddr '(1 2 3))
3
[166]> (cddddr '(1 A 2 B 3 C 4 D 5))
(3 C 4 D 5)
[167]> (cddddr '())
NIL
```

*normalmente el intérprete no permite más de cuatro niveles de imbricación, generando en otro caso un error de función no definida:*

```
[168]> (cdr (cddddr '(1 A 2 B 3 C 4 D 5)))
(C 4 D 5)
[169]> (cdddddr '(1 A 2 B 3 C 4 D 5))

*** - EVAL: la función CDDDDDR no está definida
Es posible continuar en los siguientes puntos:
USE-VALUE      :R1      Input a value to be used instead of (FDEFINITION 'CDDDDDR).
RETRY          :R2      Reintentar
STORE-VALUE    :R3      Input a new value for (FDEFINITION 'CDDDDDR).
ABORT          :R4      Abort debug loop
ABORT          :R5      Abort debug loop
ABORT          :R6      Abort debug loop
ABORT          :R7      Abort debug loop
ABORT          :R8      Abort main loop
```

**(NTHCDR número lista)**

Devuelve la sublista de `lista` que comienza en la n-ésima posición de la misma. Si `lista` no es una lista o su longitud es menor que `número`, devuelve `()`.

**Ejemplo 5.66** *Tecleando algunas expresiones en interactivo sobre el intérprete, para comprobar el comportamiento de* `nthcdr`*:*

```
[187]> (nthcdr 3 '(a b c d e))
(D E)
[188]> (nthcdr 6 '(a b c d e))
NIL
```

Una posible implementación de la función `nthcdr` en Lisp podría ser la siguiente:

```
(defun mi-nthcdr (numero lista)
       (if (<= numero 0)
           l
           (mi-nthcdr (- numero 1)
                      (cdr lista))))
```

**(ELT secuencia posición)**

Devuelve el elemento de la `secuencia` en la `posición` indicada.

**Ejemplo 5.67** *Para ilustrar el funcionamiento de* `elt`*, tecleamos algunas expresiones en interactivo sobre el intérprete:*

```
[200]> (elt "Hola que tal ?" 5)
#\q
[201]> (elt #(Hola que tal) 1)
QUE
[202]> (elt '(Hola que tal) 1)
QUE
```

**(SUBSEQ secuencia inicio :OPTIONAL final)**

Devuelve la subsecuencia de la `secuencia` argumento a partir de la posición `inicio`. Puede indicarse también la posición final.

**Ejemplo 5.68** *Para ilustrar el funcionamiento de* `subseq`*, tecleamos algunas expresiones en interactivo sobre el intérprete:*

```
[197]> (subseq "Hola que tal ?" 2)
"la que tal ?"
[198]>  (subseq "Hola que tal ?" 2 5)
"la "
```

### 5.9.2. Cálculo de longitud

Calculan la longitud de la secuencia argumento.

**(LENGTH secuencia)**

Devuelve la longitud de la `secuencia` argumento.

**Ejemplo 5.69** *Tecleamos algunas expresiones en interactivo sobre el intérprete, para comprobar el comportamiento de* `length`*:*

```
[189]> (length '())
0
[190]> (length '(1 2 3 4))
4
[191]> (length '((1 2) 3 (4 5 6)))
3
[192]> (length #(1 2 3))
3
```

**(LIST-LENGTH lista)**

Devuelve la longitud de la `lista`. La diferencia con `length` es que `list-length` devuelve `()` si la `lista` es circular, mientras que en esa situación `length` devolvería un error.

**Ejemplo 5.70** *Tecleamos algunas expresiones en interactivo sobre el intérprete, para comprobar el comportamiento de* `list-length`*, para ello echaremos mano de la función* `cirlist` *que genera una lista circular a partir de sus argumentos, y cuyo código es:*

```
(defun cirlist (&rest x)
       (rplacd (last x) x))
```

*supongamos esta función en* ∼`/lisp/CLisp/examples/cirlist.cl`*, que previamente hemos cargado, entonces:*

```
[192]> (load "~/lisp/CLisp/examples/cirlist")
;; Loading file /home/vilares/lisp/CLisp/examples/cirlist.cl ...
;; Loaded file /home/vilares/lisp/CLisp/examples/cirlist.cl
T
[193]> (length (list 1 2))
2
[194]> (list-length (list 1 2))
2
[195]> (list-length (cirlist 1 2))
NIL
[196]> (length (cirlist 1 2))

*** - LENGTH: A proper list must not be circular: #1=(2 1 . #1#)
Es posible continuar en los siguientes puntos:
ABORT          :R1      Abort debug loop
ABORT          :R2      Abort main loop
```

### 5.9.3. Funciones de búsqueda

Buscan en una secuencia para localizar uno o más elementos que satisfagan determinada condición.

**(FIND elemento secuencia [:FROM-END {t|()}] [:TEST test] [:TEST-NOT test] [:START inicio] [:END final] [:KEY función])**

Devuelve, si existe, el `elemento` más a la izquierda de la `secuencia` que verifica el `test` establecido. En otro caso devuelve `()`.

En cuanto a los argumentos opcionales, `:test`, determina el test de igualdad a aplicar y que, por defecto, es `eq`. Por su parte, `:test-not test` determina el test sobre cuya negación se aplicará la comparación. Por defecto la búsqueda se realizará de derecha a izquierda, salvo que `:from-end` tome un valor `t`. Los valores de `:start` y `:end`, de estar presentes, marcan las posiciones de `inicio` y `final` para las aplicaciones de la `función`.

Por defecto la búsqueda se realizará de derecha a izquierda, salvo que `:from-end` tome un valor `t`. Los valores de `:start` y `:end`, de estar presentes, marcan las posiciones de `inicio` y `final` para las aplicaciones de la `función`. Finalmente, `:key` modifica la parte del argumento sobre la que se aplica el test de igualdad. Por ejemplo, si el test es `eql`, en lugar de aplicar `(eql z x)`, usará `(eql (función z) (función x))`.

**Ejemplo 5.71** *Para ilustrar el funcionamiento de* `find`*, tecleamos algunas expresiones en interactivo sobre el intérprete:*

```
[197]> (find 3 '((a 1) (b 2) (b 3) (b 4) (c 5)) :key #'cadr)
(B 3)
[198]> (find '(b 3) '((a 1) (b 2) (b 3) (b 4) (c 5))
             :test #'equal)
(B 3)
[199]> (find '(b 3) '((a 1) (b 2) (b 3) (b 4) (c 5))
             :test-not #'equal)
(A 1)
[200]> (find '(b 3) '((a 1) (b 2) (b 3) (b 4) (c 5)))
NIL
[201]> (find 'b '((a 1) (b 2) (b 3) (b 4) (c 5))
             :key #'car :from-end t)
(B 4)
[202]> (find 'b '((a 1) (b 2) (b 3) (b 4) (c 5))
             :key #'car :from-end '())
(B 2)
[203]> (find '(b 3) '((a 1) (b 2) (b 3) (b 4) (c 5))
             :test #'equal :start 2 :end 4)
(B 3)
```

**(FIND-IF predicado secuencia [:FROM-END {t|()}] [:TEST test] [:TEST-NOT test] [:START inicio] [:END final] [:KEY función])**

Devuelve, si existe, el elemento más a la izquierda de la `secuencia` que verifica el `predicado` establecido. En otro caso devuelve `()`. En cuanto a los argumentos opcionales, la interpretación es la misma que la ya comentada para `find`.

**Ejemplo 5.72** *Para ilustrar el funcionamiento de* `find-if`*, tecleamos algunas expresiones en interactivo sobre el intérprete:*

```
[204]> (find-if #'numberp '((1 b) 2 a "hola"))
2
```

**(FIND-IF-NOT predicado secuencia** [**:FROM-END {t|()}**]
[**:TEST test**]
[**:TEST-NOT test**]
[**:START inicio**]
[**:END final**]
[**:KEY función**])

Devuelve, si existe, el elemento más a la izquierda de la `secuencia` que no verifica el `predicado` establecido. En otro caso devuelve `()`. En cuanto a los argumentos opcionales, la interpretación es la misma que la ya comentada para `find`.

**Ejemplo 5.73** *Para ilustrar el funcionamiento de* `find-if-not`*, tecleamos algunas expresiones en interactivo sobre el intérprete:*

```
[205]> (find-if-not #'numberp '((1 b) 2 a "hola"))
(1 B)
```

**(POSITION elemento secuencia** [**:FROM-END {t|()}**]
[**:TEST test**]
[**:TEST-NOT test**]
[**:START inicio**]
[**:END final**]
[**:KEY función**])

Devuelve, si existe, la posición del `elemento` más a la izquierda de la `secuencia` que verifica el `test` establecido. En otro caso devuelve `()`. En cuanto a los argumentos opcionales, la interpretación es la misma que la ya comentada para `find`.

**Ejemplo 5.74** *Para ilustrar el funcionamiento de* `position`*, tecleamos algunas expresiones en interactivo sobre el intérprete:*

```
[206]> (position 3 '((a 1) (b 2) (b 3) (b 4) (c 5))
                   :key #'cadr)
2
[207]> (position '(b 3) '((a 1) (b 2) (b 3) (b 4) (c 5))
                 :test #'equal)
2
[208]> (position '(b 3) '((a 1) (b 2) (b 3) (b 4) (c 5))
                 :test-not #'equal)
0
[209]> (position '(b 3) '((a 1) (b 2) (b 3) (b 4) (c 5)))
NIL
[210]> (position 'b '((a 1) (b 2) (b 3) (b 4) (c 5))
                 :key #'car :from-end t)
```

```
3
[211]> (position 'b '((a 1) (b 2) (b 3) (b 4) (c 5))
                :key #'car :from-end '())
1
[212]> (position '(b 3) '((a 1) (b 2) (b 3) (b 4) (c 5))
                :test #'equal :start 2 :end 4)
2
```

**(POSITION-IF predicado secuencia [:FROM-END {t|()}]**
**[:TEST test]**
**[:TEST-NOT test]**
**[:START inicio]**
**[:END final]**
**[:KEY función])**

Devuelve, si existe, la posición del elemento más a la izquierda de la `secuencia` que verifica el `predicado` establecido. En otro caso devuelve `()`. En cuanto a los argumentos opcionales, la interpretación es la misma que la ya comentada para `find`.

**Ejemplo 5.75** *Para ilustrar el funcionamiento de* `position-if`*, tecleamos algunas expresiones en interactivo sobre el intérprete:*

```
[213]> (position-if #'numberp '((1 b) 2 a "hola"))
1
```

**(POSITION-IF-NOT predicado secuencia [:FROM-END {t|()}]**
**[:TEST test]**
**[:TEST-NOT test]**
**[:START inicio]**
**[:END final]**
**[:KEY función])**

Devuelve, si existe, la posición del elemento más a la izquierda de la `secuencia` que no verifica el `predicado` establecido. En otro caso devuelve `()`. En cuanto a los argumentos opcionales, la interpretación es la misma que la ya comentada para `find`.

**Ejemplo 5.76** *Para ilustrar el funcionamiento de* `position-if-not`*, tecleamos algunas expresiones en interactivo sobre el intérprete:*

```
[215]> (position-if-not #'numberp '((1 b) 2 a "hola"))
0
[216]> (position-if-not #'consp '((1 b) 2 a "hola"))
1
```

**(COUNT elemento secuencia** [**:FROM-END {t|()}**]
[**:TEST test**]
[**:TEST-NOT test**]
[**:START inicio**]
[**:END final**]
[**:KEY función**])

Devuelve el número de veces que `elemento` aparece en `secuencia` verificando el `test` establecido. En cuanto a los argumentos opcionales, la interpretación es la misma que la ya comentada para `find`.

**Ejemplo 5.77** *Para ilustrar el funcionamiento de* `count`*, tecleamos algunas expresiones en interactivo sobre el intérprete:*

```
[217]> (count 3 '((a 1) (b 2) (b 3) (b 4) (c 3)) :key #'cadr)
2
[218]> (count '(b 3) '((a 1) (b 2) (b 3) (b 4) (c 5))
              :test #'equal)
1
[219]> (count '(b 3) '((a 1) (b 2) (b 3) (b 4) (c 5))
              :test-not #'equal)
4
[220]> (count '(b 3) '((a 1) (b 2) (b 3) (b 4) (c 5)))
0
[221]> (count 'b '((a 1) (b 2) (b 3) (b 4) (c 5))
              :key #'car :from-end t)
3
[222]> (count 'b '((a 1) (b 2) (b 3) (b 4) (c 5))
              :key #'car :from-end '())
3
[223]> (count '(b 3) '((a 1) (b 2) (b 3) (b 4) (c 5))
              :test #'equal :start 2 :end 4)
1
```

**(COUNT-IF predicado secuencia** [**:FROM-END {t|()}**]
[**:TEST test**]
[**:TEST-NOT test**]
[**:START inicio**]
[**:END final**]
[**:KEY función**])

Devuelve el número de veces que los elementos de la `secuencia` verifican el `predicado` señalado. En otro caso devuelve `()`. En cuanto a los argumentos opcionales, la interpretación es la misma que la ya comentada para `find`.

**Ejemplo 5.78** *Para ilustrar el funcionamiento de* `count-if`*, tecleamos algunas expresiones en interactivo sobre el intérprete:*

```
[224]> (count-if #'numberp '((1 b) 2 a "hola" 3))
2
```

**(COUNT-IF-NOT predicado secuencia** [**:FROM-END {t|()}**]
[**:TEST test**]
[**:TEST-NOT test**]
[**:START inicio**]
[**:END final**]
[**:KEY función**])

Devuelve el número de veces que los elementos de la `secuencia` no verifican el `predicado` señalado. En cuanto a los argumentos opcionales, la interpretación es la misma que la ya comentada para `find`.

**Ejemplo 5.79** *Para ilustrar el funcionamiento de* `count-if-not`*, tecleamos algunas expresiones en interactivo sobre el intérprete:*

```
[225]> (count-if-not #'numberp '((1 b) 2 a "hola"))
3
[226]> (count-if-not #'consp '((1 b) 2 "hola" (3 c)))
2
```

**(MISMATCH secuencia secuencia** [**:FROM-END {t|()}**]
[**:TEST test**]
[**:TEST-NOT test**]
[**:KEY función**]
[**:START1 inicio**]
[**:START2 inicio**]
[**:END1 final**]
[**:END2 final**])

Compara dos a dos los elementos de las `secuencias` argumento según el `test` facilitado. En caso de que la comparación sea positiva en todos los casos y las longitudes de las `secuencias` coincidan, la función devuelve `()`. En otro caso, devuelve la posición en la primera de las `secuencias` argumento correspondiente al elemento más a la izquierda en el que la comparación haya fallado.

En cuanto a los argumentos opcionales, se refiere a la definición del `test` de comparación positivo o negativo a considerar, y a las posiciones de `inicio` y `final` que delimiten la porción a comparar en cada `secuencia`. En cuanto al argumento `:from-end`, si está asignado a `t` implica la variación del orden de exploración, que así comenzaría por la derecha. Finalmente, `:key` modifica la parte del argumento sobre la que se aplica el test de igualdad. Por ejemplo, si el test es `eql`, en lugar de aplicar `(eql z x)`, usará `(eql (función z) (función x))`. Por defecto, el `test` utilizado es `eq`.

**Ejemplo 5.80** *Para ilustrar el funcionamiento de* `mismatch`*, tecleamos algunas expresiones en interactivo sobre el intérprete:*

```
[227]> (mismatch #((a 1) (2 c) (b 3) (b 4)) #((a 1) (2 c) (b 5) (d 6)))
0
[228]> (mismatch #((a 1) (2 c) (b 3) (b 4)) #((a 1) (2 c) (b 5) (d 6))
                 :test #'equal)
2
[229]> (mismatch #((a 1) (2 c) (b 3) (b 4)) #((a 1) (2 c) (b 5) (d 6))
                 :test #'equal :key #'car)
3
[230]> (mismatch #((a 1) (2 c) (b 3) (b 4)) #((a 1) (2 c) (b 5) (d 6))
                 :test #'equal :key #'car :from-end t)
4
```

**(SEARCH secuencia secuencia** [**:FROM-END {t|()}**]
[**:TEST test**]
[**:TEST-NOT test**]
[**:KEY función**]
[**:START1 inicio**]
[**:START2 inicio**]
[**:END1 final**]
[**:END2 final**])

Busca una subsecuencia de la segunda `secuencia` que coincida según `test` y elemento a elemento con la primera de las `secuencias` argumento. Si tal subsecuencia no existiera, devolverá `()`. En otro caso, devolverá la posición del elemento más a la izquierda en la segunda `secuencia` argumento que verifique el test de comparación. En cuanto a los argumentos opcionales, la interpretación es la misma que en `mismatch`.

**Ejemplo 5.81** *Para ilustrar el funcionamiento de* `search`*, tecleamos algunas expresiones en interactivo sobre el intérprete:*

```
[231]> (search "que" "hola, Â¿ que tal estas hoy ? Â¿ que dices ?")
8
[232]> (search '(0 1) '(2 4 6 1 3 5) :key #'oddp)
2
[233]> (search "que" "hola, Â¿ que tal estas hoy ? Â¿ que dices ?"
               :from-end t)
30
```

**(TAILP lista lista)**

Verifica si la primera `lista` argumento es, físicamente, una sublista de la segunda. Devuelve `t` en caso afirmativo, sino `()`.

**Ejemplo 5.82** *Para ilustrar el funcionamiento de* `tailp`*, tecleamos algunas expresiones en interactivo sobre el intérprete:*

```
[197]> (setq lista '(1 2 3 4 5))
[198]> (tailp '(1 2) lista)
NIL
[199]>  (tailp (nthcdr 2 lista) lista)
T
```

### 5.9.4. Funciones de ordenación y fusión

Este tipo de funciones puede alterar físicamente la estructura de sus argumentos.

**(SORT secuencia predicado [:KEY función])**

Ordena, y devuelve modificada físicamente, la `secuencia` mediante el `predicado` argumento. Este debe ser binario, y devolver un valor diferente de `()` cuando el primero de sus argumentos sea menor que el segundo en algún sentido apropiado; y `()` en otro caso. En caso de existir, el argumento `:key` es una `función` unaria que sirve de preprocesador de la `secuencia`, antes de que ésta sea ordenada por el `predicado` de forma efectiva. Dicha `función` se aplicará a cada uno de los elementos de la `secuencia`. Por defecto la `función` considerada es la identidad.

NOTA: La operación no es estable, en el sentido de que aquellos elementos considerados iguales por el `predicado` pueden no aparecer en el resultado en el mismo orden que en el argumento `secuencia`.

**Ejemplo 5.83** *Unas expresiones tecleadas en interactivo servirán para ilustrar el comportamiento de* `sort`*:*

```
[203]> (setq lista '(2 3 1))
(2 3 1)
[204]> (sort lista #'<)
(1 2 3)
[205]> lista
(1 2 3)
[206]> (sort '((1 5 7) (-3 6 9) (10 -7 32)) #'> :key #'car)
((10 -7 32) (1 5 7) (-3 6 9))
```

**(STABLE-SORT secuencia predicado [:KEY función])**

La funcionalidad es idéntica a la de `sort`, con la salvedad de que esta función es estable. Esto implica que si dos elementos con igual prioridad

de ordenación según `predicado`, aparecen correlativos en la `secuencia`, su orden se mantiene en la respuesta.

**(MERGE tipo secuencia secuencia predicado [:KEY función])**

Las `secuencias` son mezcladas, de acuerdo al orden determinado por el `predicado`, que ha de ser binario y devolver un valor diferente de `()` sii su primer argumento es estrictamente inferior al segundo en relación al orden considerado. El resultado es del `tipo` indicado, que debe ser un subtipo de `sequence`, siguiendo el modelo de la función `coerce`.

De estar presente, el argumento `:key` debe ser una función unaria, que se aplica previamente a cada uno de los elementos de las `secuencias` arguemnto antes de aplicar la ordenación dictada por `predicado`. El resultado es estable, en el sentido de `sort`.

NOTA: La función `merge` sólo devuelve una secuencia ordenada si ambas `secuencias` argumento lo están previamente. En otro caso, lo único que se garantiza es que no habrá un elemento de la primera `secuencia` que sea mayor que uno de la segunda en el resultado.

NOTA: Dependiendo del intérprete concreto, el función puede modificar o no físicamente las `secuencias` argumento.

**Ejemplo 5.84** *Unas expresiones tecleadas en interactivo servirán para ilustrar el comportamiento de* `merge`*:*

```
[207]> (setq test1 (list 1 6 3 4 7))
(1 6 3 4 7)
[208]> (setq test2 (vector 2 8 5))
(2 8 5)
[209]> (merge 'list test1 test2 #'<)
(1 2 6 3 4 7 8 5)
[210]> (setq test1 (list 1 3 4 6 7))
(1 3 4 6 7)
[211]> (setq test2 (vector 2 5 8))
#(2 5 8)
[212]> (merge 'list test1 test2 #'<)
(1 2 3 4 5 6 7 8)
[213]> (setq test1 '((1 2 3) (5 6 2) (-1 3 10)))
((1 2 3) (5 6 2) (-1 3 10))
[214]> (setq test2 '((3 2 1) (7 6 9) (-7 3 -15)))
((3 2 1) (7 6 9) (-7 3 -15))
[215]> (merge 'list test1 test2 #'< :key #'car)
((1 2 3) (3 2 1) (5 6 2) (-1 3 10) (7 6 9) (-7 3 -15))
```

### 5.9.5. Funciones de creación

En todos los casos aquí expuestos, el manejo de la memoria reservada para las listas es dinámico y automático; lo cual puede provocar la actuación del recogedor de la basura[14].

**(CONS forma forma)**

Construye una lista donde el valor de la primera forma actúa de `car` y el segundo de `cdr`, devolviéndola como resultado.

**Ejemplo 5.85** *Unas simples expresiones tecleadas en interactivo serán suficientes para ilustrar el comportamiento de* `cons`*:*

```
[192]> (cons 1 2)
(1 . 2)
[193]> (cons (+ 1 2) '(a b))
(3 A B)
[194]> (car (cons 1 2))
1
[195]> (cdr (cons 1 2))
2
[196]> (car (cons (+ 1 2) '(a b)))
3
[197]> (cdr (cons (+ 1 2) '(a b)))
(A B)
```

**(COPY-SEQ secuencia)**

Devuelve una copia, mediante `equalp`, de la `secuencia` argumento.

**(COPY-LIST lista)**

Devuelve una copia, mediante `equalp`, de la `lista` argumento.

**Ejemplo 5.86** *Para ilustrar el funcionamiento de* `copy-seq` *y* `copy-list`*, tecleamos algunas expresiones en interactivo directamente sobre el intérprete Lisp:*

```
[199]> (copy-seq "hola")
"hola"
[200]> (copy-seq #(h o l a))
#(H O L A)
[201]> (copy-seq '(h o l a))
```

[14] el algoritmo utilizado para liberar la memoria ocupada por datos inutilizables.

```
(H O L A)
[202]> (copy-list '(h o l a))
(H O L A)
```

### (LIST {forma}*)

Devuelve la lista cuyos elementos son el resultado de evaluar las `formas` que le sirven de argumento.

**Ejemplo 5.87** *Para ilustrar el funcionamiento de* `list`*, teclearemos algunas expresiones en interactivo, directamente sobre el intérprete:*

```
[198]> (list (+ 1 2) '(a b) "hola" '())
(3 (A B) "hola" ())
```

### (APPEND {lista}*)

Devuelve una copia de la concatenación de las `listas` que le sirven de argumento. El último de esos argumentos puede no ser lista, y en este caso dicho elemento se convierte en el fin de lista de la respuesta.

**Ejemplo 5.88** *Para ilustrar el funcionamiento de* `append`*, teclearemos algunas expresiones en interactivo, directamente sobre el intérprete:*

```
[199]> (append '(1 2) '(a b) 3)
(1 2 A B . 3)
[200]> (cdr (append '(1 2) '(a b) 3))
(2 A B . 3)
[201]> (append '(1 2))
(1 2)
[202]> (append)
NIL
[203]> (append 2)
2
[204]> (append '(1) 3)
(1 . 3)
```

### (NCONC {lista}*)

Concatena físicamente[15] todas las `listas` que le sirven de argumento, devolviendo el resultado correspondiente.

[15] lo que la diferencia claramente de `append`.

**Ejemplo 5.89**
```
[225]> (defvar lista-1 '(1 2))
LISTA-1
[226]> lista-1
(1 2)
[227]> (defvar lista-2 '(3 4))
LISTA-2
[228]> lista-2
(3 4)
[229]> (append lista-1 lista-2)
(1 2 3 4)
[230]> lista-1
(1 2)
[231]> lista-2
(3 4)
[232]> (nconc lista-1 lista-2)
(1 2 3 4)
[233]> lista-1
(1 2 3 4)
[234]> lista-2
(3 4)
```

Una posible implementación de la función **nconc** en Lisp es la que sigue:

```
(defun mi-nconc (lista1 lista2)
       (let ((lista1 lista1))
            (loop while (consp (cdr lista1))

                  ; Eliminamos fisicamente el primer
                  ; elemento de lista1

                  do (pop lista1))
            (rplacd lista1 lista2))
       lista1)
```

**(REVERSE secuencia)**

Devuelve una copia invertida del primer nivel de la `secuencia`.

**Ejemplo 5.90** *Para ilustrar el funcionamiento de* `reverse`*, teclearemos algunas expresiones en interactivo, directamente sobre el intérprete:*

```
[205]> (reverse '(a (b c) d))
(D (B C) A)
[206]> (reverse #(1 (2 3) 4)
#(4 (2 3) 1)
```

**(NREVERSE secuencia)**

Devuelve el mismo resultado que `reverse`, pero la inversión de la secuencia es física, un efectocolateral que supone una diferencia fundamental con `reverse`.

**Ejemplo 5.91** *Para ilustrar el funcionamiento de* `reverse`*, teclearemos algunas expresiones en interactivo, directamente sobre el intérprete:*

```
[205]> (defvar lista '(1 2 3))
LISTA
[206]> (defvar vector #(1 2 3))
VECTOR
[207]> (reverse lista)
(3 2 1)
[208]> (reverse vector)
(3 2 1)
[209]> lista
(1 2 3)
[210]> vector
#(1 2 3)
[211]> (nreverse lista)
(3 2 1)
[212]> (nreverse vector)
#(3 2 1)
[213]> lista
(3 2 1)
[214]> vector
#(3 2 1)
```

**(MAKE-SEQUENCE tipo talla [:INITIAL-ELEMENT elemento])**

Devuelve una secuencia del `tipo` y `talla` indicados, donde podemos especificar opcionalmente cuál es el `elemento` inicial. Por defecto este `elemento` es `()`.

**Ejemplo 5.92** *Ilustramos el funcionamiento de* `make-sequence`*, con algunas expresiones tecleadas en interactivo sobre el intérprete:*

```
[215]> (make-sequence 'cons 5)
(NIL NIL NIL NIL NIL)
[216]> (make-sequence 'cons 5 :initial-element 'a)
(A A A A A)
[217]> (make-sequence 'vector 5)
```

```
#(NIL NIL NIL NIL NIL)
[218]> (make-sequence 'vector 5 :initial-element 'a)
#(A A A A A)
[219]> (make-sequence '(vector float) 5 :initial-element 1)
#(1 1 1 1 1)
```

### (CONCATENATE tipo {secuencia}*)

Devuelve una el resultado de concatenar una copia de las `secuencias` que le sirven de argumento, cuyo `tipo` es el indicado y que siempre debe ser un subtipo de `sequence`. En caso de proporcionar sólo una `secuencia`, el resultado es una copia de la misma.

**Ejemplo 5.93** *Ilustramos el funcionamiento de* `concatenate`*, con algunas expresiones tecleadas en interactivo sobre el intérprete:*

```
[220]> (defvar lista-1 '(1 2))
LISTA-1
[221]> (defvar lista-2 '(3 4))
LISTA-2
[222]> (concatenate 'cons lista-1 lista-2)
(1 2 3 4)
[223]> (concatenate 'vector lista-1 lista-2)
#(1 2 3 4)
[224]> lista-1
(1 2)
[225]> lista-2
(3 4)
```

### (REVAPPEND lista lista)

Devuelve una copia de la concatenación de la inversa de la primera `lista` con la segunda. El último de esos argumentos puede no ser lista, y en este caso dicho elemento se convierte en el fin de lista de la respuesta. Funcionalmente `(revappend x y)` es equivalente a `(append (reverse x) y)`, pero resulta potencialmente más eficiente.

**Ejemplo 5.94** *Para ilustrar el funcionamiento de* `revappend`*, tecleáremos algunas expresiones en interactico, directamente sobre el intérprete:*

```
[199]> (revappend '(1 2) 3)
(2 1 . 3)
[200]> (revappend '(1 2) '(3 4))
(2 1 3 4)
```

### (NRECONC lista lista)

Devuelve una copia de la concatenación física de la inversa de la primera `lista` con la segunda. El último de esos argumentos puede no ser lista, y en este caso dicho elemento se convierte en el fin de lista de la respuesta. Funcionalmente `(nreconc x y)` es equivalente a `(nconc (nreverse x) y)`, pero resulta potencialmente más eficiente.

**Ejemplo 5.95** *Para ilustrar el funcionamiento de* `nreconc`*, teclearemos algunas expresiones en interactivo, directamente sobre el intérprete:*

```
[201]> (defvar lista-1 '(1 2))
LISTA-1
[202]> lista-1
(1 2)
[203]> (defvar lista-2 '(3 4))
LISTA-2
[204]> lista-2
(3 4)
[205]> (revappend lista-1 lista-2)
(2 1 3 4)
[206]> lista-1
(1 2)
[207]> lista-2
(3 4)
[208]> (nreconc lista-1 lista-2)
(2 1 3 4)
[209]> lista-1
(2 1 3 4)
[210]> lista-2
(3 4)
```

### (PUSH objeto lista)

Sitúa el `objeto` como el primer elemento de su segundo argumento, cuyo valor debe ser una lista. Devuelve la `lista` aumentada, que es modificada físicamente. En caso de que el último argumento no sea de tipo `cons`, devuelve una lista donde el `car` es el `objeto` y el `cdr` es el último argumento de `push`.

**Ejemplo 5.96** *Para ilustrar el funcionamiento de* `push`*, teclearemos algunas expresiones en interactico, directamente sobre el intérprete:*

```
[211]> (defvar lista '(3 4))
LISTA
```

```
[212]> lista
(3 4)
[213]> (push 2 lista)
(2 3 4)
[214]> lista
(2 3 4)
[215]> (defvar vector #(3 4))
VECTOR
[216]> vector
#(3 4)
[217]> (push 2 vector)
(2 . #(3 4))
[218]> vector
(2 . #(3 4))
```

**(PUSHNEW objeto lista [:TEST test] [:TEST-NOT test] ll [:KEY función])**

Si el `objeto` es ya miembro del segundo argumento, cuyo valor debe ser una lista, devuelve dicho argumento invariable. En otro caso el funcionamiento es análogo al de `push`. En relación a las *palabras clave de lambda lista*, su interpretación es la que sigue:

- `:key función`. Aplica la `función` especificada tanto a `objeto` como a la `lista`, previo a realizar el test comparativo. Por defecto, no se aplica ninguna `función`. Opción de difícil comprensión.
- `:test test`. Determina el `test` de igualdad a aplicar y que, por defecto, es `eql`.
- `:test-not test`. Determina el `test` sobre cuya negación se aplicará la comparación.

**Ejemplo 5.97** *Podemos considerar un ejemplo sencillo, simplemente tecleando algunas expresiones en interactivo sobre el intérprete. Comencemos por* `:test`*:*

```
[170]> (defvar lista '((c 3) (d 4) (e 5)))
LISTA
[171]> lista
((C 3) (D 4) (E 5))
[172]> (pushnew '(b 2) lista)
((B 2) (C 3) (D 4) (E 5))
[173]> lista
((B 2) (C 3) (D 4) (E 5))
```

```
[174]> (pushnew '(b 2) lista :test #'equal)
((B 2) (C 3) (D 4) (E 5))
[175]> lista
((B 2) (C 3) (D 4) (E 5))
[176]> (pushnew '(b 2) lista)       ; por defecto :test eql
((B 2) (B 2) (C 3) (D 4) (E 5))
[177]> lista
((B 2) (B 2) (C 3) (D 4) (E 5))
```

*Si pasamos ahora al caso de* `:key`*, y siguiendo con el ejemplo, podemos considerar:*

```
[174]> (pushnew '(b 6) lista :key #'car)
((B 2) (B 2) (C 3) (D 4) (E 5))
[175]> lista
((B 2) (B 2) (C 3) (D 4) (E 5))
[176]> (pushnew '(b 6) lista)
((B 6) (B 2) (B 2) (C 3) (D 4) (E 5))
[177]> lista
((B 6) (B 2) (B 2) (C 3) (D 4) (E 5))
```

*Finalmente, tocando a la palabra clave* `:test-not`*, sirvan de ejemplo las siguientes expresiones interpretadas interativamente:*

```
[178]> (pushnew '(b 6) lista :test-not #'eq)
((B 6) (B 2) (B 2) (C 3) (D 4) (E 5))
[179]> lista
((B 6) (B 2) (B 2) (C 3) (D 4) (E 5))
[180]> (pushnew '(b 6) lista :test #'eq)
((B 6) (B 6) (B 2) (B 2) (C 3) (D 4) (E 5))
[181]> lista
((B 6) (B 6) (B 2) (B 2) (C 3) (D 4) (E 5))
```

**(POP lista)**

Devuelve el `car` de su argumento, cuyo valor debe ser una lista, y modifica físicamente ésta última, eliminando ese `car`.

**Ejemplo 5.98** *Para ilustrar el funcionamiento de* `pop`*, teclearemos algunas expresiones en interactico, directamente sobre el intérprete:*

```
[211]> (defvar pila '(1 2 3 4))
PILA
[212]> pila
(1 2 3 4)
```

```
[213]> (pop pila)
1
[214]> pila
(2 3 4)
```

**(BUTLAST lista [n])**

Genera y devuelve una lista con los mismos elementos que el primer argumento, que no sufre efectos colaterales, de los que se excluyen los últimos `n` elementos. Por defecto `n=1`. Si la longitud de la lista es menor que `n`, devuelve `()`.

**Ejemplo 5.99** *Para ilustrar el funcionamiento de* `butlast`*, teclearemos algunas expresiones en interactico, directamente sobre el intérprete:*

```
[215]> (defvar lista '(a b c d))
LISTA
[216]> (butlast lista)
(A B C)
[217]> lista
(A B C D)
[218]> (butlast lista 2)
(A B)
[219]> lista
(A B C D)
```

**(NBUTLAST lista [n])**

La funcionalidad es idéntica a `butlast`, con la diferencia de que aquí `lista` sufre un efecto colateral, eliminándose físicmanete sus últimos `n` elementos.

**Ejemplo 5.100** *Para ilustrar el funcionamiento de* `nbutlast`*, teclearemos algunas expresiones en interactico, directamente sobre el intérprete, siguiendo el anterior ejemplo:*

```
[220]> lista
(A B C D)
[221]> (nbutlast lista)
(A B C)
[221]> lista
(A B C)
[223]> (butlast lista 2)
(A)
[224]> lista
(A)
```

**(LDIFF lista sublista)**

Devuelve el conjunto de elementos que en `lista` preceden a su `sublista`. La relación de inclusión ha de verificarse mediante el predicado `eq`, en otro caso devuelve `lista`.

**Ejemplo 5.101** *Para ilustrar el funcionamiento de* `ldiff`*, teclearemos algunas expresiones en interactico, directamente sobre el intérprete, siguiendo con el anterior ejemplo:*

```
[225]> (setq lista '(a b c d e))
(A B C D E)
[226]> (setq sublista (cddr lista))
(C D E)
[227]> (ldiff lista sublista)
(A B)
[228]> (ldiff lista '(c d e)))
(A B C D E)
Break 1 [228]>
*** - READ en
       #<INPUT CONCATENATED-STREAM #<INPUT STRING-INPUT-STREAM>
         #<IO TERMINAL-STREAM>>
      : un objeto no puede comenzar por #\)
Es posible continuar en los siguientes puntos:
ABORT          :R1      Abort debug loop
ABORT          :R2      Abort main loop
```

**(ENDP lista)**

Devuelve `t` si el argumento es `()`, y `()` en otro caso. Es la forma recomendada de testear el final de una estructura tipo lista.

**Ejemplo 5.102** *Para ilustrar el funcionamiento de* `endp`*, teclearemos algunas expresiones en interactico sobre el intérprete:*

```
[211]> (endp (list 1 2))
NIL
[212]> (endp '())
T
```

**(NTH número lista)**

Devuelve el elemento en la posición `número` de la `lista`, si es que existe. En otro caso, devuelve `()`. El `car` se supone que ocupa la posición cero.

**Ejemplo 5.103** *Para ilustrar el funcionamiento de* `endp`*, teclearemos algunas expresiones en interactico sobre el intérprete:*

```
[213]> (nth 0 '(1 2))
1
[214]> (nth 1 '(1 2))
2
[215]> (nth 0 '())
NIL
[216]> (nth 1 '())
NIL
[217]> (nth 3 '(1 2))
NIL
```

| | | |
|---|---|---|
| **(FIRST lista)** | **(SECOND lista)** | **(THIRD lista)** |
| **(FOURTH lista)** | **(FIFTH lista)** | **(SIXTH lista)** |
| **(SEVENTH lista)** | **(EIGHTH lista)** | **(NINETH lista)** |
| **(TENTH lista)** | **(REST lista)** | |

Devuelven el elemento correspondiente de `lista`, si éste existe. En otro caso devuelve `()`. Su uso se justifica, en relación a `nth` o funciones del tipo `caddr`, por su mayor legibilidad. La función `rest` es equivalente a `cdr`.

**Ejemplo 5.104** *Para ilustrar el funcionamiento de estas funciones, teclearemos algunas expresiones en interactico sobre el intérprete:*

```
[218]> (first '(1 2))
1
[219]> (second '(1 2))
2
[220]> (third '(1 2))
NIL
[221]> (rest '(1 2))
(2)
[222]> (last '(1 2 3 4) 1)
(4)
[223]> (last '(1 2 3 4) 2)
(3 4)
[224]> (last '(1 2 3 4) 3)
(2 3 4)
[225]> (last '(1 2 3 4) 4)
(1 2 3 4)
[226]> (last '(1 2 3 4) 5)
(1 2 3 4)
```

**(LAST lista [n])**

Devuelve los últimos `n cons` de la `lista`. Por defecto, devuelve el último de ellos.

**Ejemplo 5.105** *Ilustramos el funcionamiento de* `last`*, con algunas expresiones tecleadas en interactico sobre el intérprete:*

```
[222]> (last '(1 2 3 4))
(4)
[223]> (last '())
NIL
```

**(MAKE-LIST talla [:INITIAL-ELEMENT elemento])**

Devuelve una lista de la `talla` indicada, donde podemos especificar opcionalmente cuál es el `elemento` inicial. Por defecto este `elemento` es `()`.

**Ejemplo 5.106** *Ilustramos el funcionamiento de* `last`*, con algunas expresiones tecleadas en interactico sobre el intérprete:*

```
[224]> (make-list 5)
(NIL NIL NIL NIL NIL)
[225]> (make-list 5 :initial-element 'a)
(A A A A A)
```

**(FILL secuencia elemento [:START inicio] [:END final])**

La `secuencia` es modificada físicamente, y su valor devuelto, reemplazando las ocurrencias en porción de la `secuencia` delimitada mediante los valores opcionales `:start` y `:end`, por `elemento`. El valor por defecto para `:start` (resp. para `:end`) es 0 (resp. `(- (length secuencia) 1)`). La comparación se hace mediante `eq`.

**Ejemplo 5.107** *Ilustramos el funcionamiento de* `fill`*, con algunas expresiones tecleadas en interactico sobre el intérprete:*

```
[226]> (setq vector #(1 2 3 4 5))
#(1 2 3 4 5)
[227]> (fill vector (+ 3 3) :start 1 :end 3)
#(1 6 6 4 5)
```

**(REPLACE secuencia secuencia** [**:START1 inicio_1**]
[**:END1 final_1**])
[**:START2 inicio_2**]
[**:END2 final_2**])

Modifica físicamente, y devuelve, la primera `secuencia` mediante la copia de susesivos elementos de la segunda `secuencia`. Concretamente, reemplaza el intervalo de la primera `secuencia` entre :start1 y :end1, por el intervalo de la segunda delimitado por :start2 y :end2. Los valores por defecto para :start1 y :start2 (resp. para :end1 y :end2) son cero (resp. `(- (length secuencia) 1)`).

**Ejemplo 5.108** *Ilustramos el funcionamiento de* `replace`*, con algunas expresiones tecleadas en interactico sobre el intérprete:*

```
[226]> (defvar cadena1 "ab123fg")
CADENA1
[227]> (defvar cadena2 "cde")
CADENA2
[228]> (replace cadena1 cadena2 :start1 2 :end1 5)
"abcdefg"
[229]> cadena1
"abcdefg"
[230]> cadena2
"cde"
```

**(REMOVE elemento secuencia** [**:FROM-END {t|()}**]
[**:TEST test**]
[**:TEST-NOT test**]
[**:START inicio**]
[**:END final**]
[**:COUNT contador**]
[**:KEY función**])

Devuelve una copia de la `secuencia` de la que se han eliminado todos los elementos iguales, mediante `eql`, a `elemento`.

Los valores de `:start` y `:end`, de estar presentes, marcan las posiciones de `inicio` y `final` para la aplicación de la funcionalidad descrita. De estar presente, `:count` establece el número máximo de elementos a eliminar. Por defecto, la combinación se realizará de derecha a izquierda, salvo que `:from-end` tome un valor `t`. Obviamente, `:from-end` sólo tiene sentido en combinación con `:count`. En cuanto a `:test`, determina el test de igualdad a aplicar y que, por defecto, es `eq`. Por su parte, `:test-not test` determina el test sobre cuya negación se aplicará la comparación. Finalmente, `:key` modifica la parte de `x` sobre la que se aplica el test de igualdad. Por ejemplo, si el test es `eql`, en lugar de aplicar `(eql z x)`, usará `(eql z (función x))`.

**Ejemplo 5.109** *Ilustramos el funcionamiento de* `remove`*, con algunas expresiones tecleadas en interactico sobre el intérprete:*

```
[226]> (setq vector #(3 4))
#(3 4)
[227]> (setq lista (list 1 'a 2 3 '(1 2) #(3 4) 2 'a vector 'a))
(1 A 2 3 (1 2) #(3 4) 2 A #(3 4) A)
[228]> (remove 2 lista)
(1 A 3 (1 2) #(3 4) A #(3 4) A)
[229]> (remove #(3 4) lista)
(1 A 2 3 (1 2) #(3 4) 2 A #(3 4) A)
[230]> (remove vector lista)
(1 A 2 3 (1 2) #(3 4) 2 A A)
[231]> lista
(1 A 2 3 (1 2) #(3 4) 2 A #(3 4) A)
[232]>  (remove 'a lista :start 0 :end 7)
(1 2 3 (1 2) #(3 4) 2 A #(3 4) A)
[233]> (remove (list 1 2) lista)
(1 A 2 3 (1 2) #(3 4) 2 A #(3 4) A)
{234]> (remove (list 1 2) lista :test #'equal)
(1 A 2 3 #(3 4) 2 A #(3 4) A)
[235]> (remove 'a lista :from-end t :count 2)
(1 A 2 3 (1 2) #(3 4) 2 #(3 4))
[236]> (remove (list 1 2) lista :test-not #'equal)
((1 2))
[237]> (remove "a" "abab" :test-not #'eql)
""
[238]> (setq lista '((1 2) (3 2) (a b) (c d)))
((1 2) (3 2) (A B) (C D))
[239]> (remove '(2) lista :key #'cdr :test #'equal)
((A B) (C D))
```

Una posible implementación en Lisp de **remove**, sin considerar los argumentos opcionales, es la siguiente:

```
(defun mi-remove (elemento lista)
       (cond ((atom lista) lista)
             ((eql elemento (car lista))
              (mi-remove elemento (cdr lista)))
             (t (cons (car lista) (mi-remove elemento (cdr lista))))))
```

**(REMOVE-IF predicado secuencia** [**:FROM-END {t|()}**]
[**:START inicio**]
[**:END final**]
[**:COUNT contador**]
[**:KEY función**])

Devuelve una copia de la `secuencia` de la que se han eliminado todos los elementos que verifican `predicado`. La interpretación de los argumentos opcionales es análoga a la de `member-if`.

**Ejemplo 5.110** *Ilustramos el funcionamiento de* `remove-if`*, con algunas expresiones tecleadas en interactico sobre el intérprete:*

```
[240]> (setq lista '(1 2 'a "pepe" (1 2) 'b 3 (a b)))
(1 2 'A "pepe" (1 2) 'B 3 (A B))
[241]> (remove-if #'numberp lista)
('A "pepe" (1 2) 'B (A B))
```

**(REMOVE-IF-NOT predicado secuencia** [**:FROM-END {t|()}**]
[**:START inicio**]
[**:END final**]
[**:COUNT contador**]
[**:KEY función**])

Devuelve una copia de la `secuencia` de la que se han eliminado todos los elementos que no verifican `predicado`. La interpretación de los argumentos opcionales es análoga a la de `member-if-not`.

**Ejemplo 5.111** *Ilustramos el funcionamiento de* `remove-if`*, con algunas expresiones tecleadas en interactico sobre el intérprete:*

```
[242]> (setq lista '(1 2 'a "pepe" (1 2) 'b 3 (a b)))
(1 2 'A "pepe" (1 2) 'B 3 (A B))
[241]> (remove-if-not #'numberp lista)
(1 2 3)
```

**(REMOVE-DUPLICATES secuencia** [**:FROM-END {t|()}**]
[**:START inicio**]
[**:END final**]
[**:COUNT contador**]
[**:KEY función**])

Devuelve una copia de la `secuencia` de la que se han eliminado uno de los elementos de todos los pares consecutivos iguales según `eq`. La interpretación de los argumentos opcionales es análoga a la de `remove`.

**Ejemplo 5.112** *Ilustramos el funcionamiento de* `remove-duplicates`*, con algunas expresiones tecleadas en interactico sobre el intérprete:*

```
[242]> (setq lista '(1 2 2 2 3 'a 'a 3 'b 'a))
(1 2 'A 'A 3 'B 'A)
```

**(SUBSTITUTE elemento elemento secuencia** [**:FROM-END {t|()}**]
[**:START inicio**]
[**:END final**]
[**:COUNT contador**]
[**:KEY función**])

Devuelve una copia de la `secuencia` en la que todas las ocurrencias del `elemento` que actúa como segundo argumento son reemplazadas por el que actúa como primero usando `eq` como test comparativo. La interpretación de los argumentos opcionales es análoga a la de los de `remove`.

**Ejemplo 5.113** *Ilustramos el funcionamiento de* `substitute`*, con algunas expresiones tecleadas en interactico sobre el intérprete:*

```
[243]> (setq vector #(3 4))
#(3 4)
[244]> (setq lista (list 1 'a 2 3 '(1 2) #(3 4) 2 'a vector 'a))
(1 A 2 3 (1 2) #(3 4) 2 A #(3 4) A)
[245]> (substitute 4 2 lista)
(1 A 4 3 (1 2) #(3 4) 4 A #(3 4) A)
[246]> (substitute #(A A) vector lista :test #'equal)
(1 A 2 3 (1 2) #(3 4) 2 A #(A A) A)
```

**(SUBSTITUTE-IF elemento predicado secuencia** [**:FROM-END {t|()}**]
[**:START inicio**]
[**:END final**]
[**:COUNT contador**]
[**:KEY función**])

Devuelve una copia de la `secuencia` en la que todas las ocurrencias de elementos verificando el `predicado` son reemplazadas por el `elemento`. La interpretación de los argumentos opcionales es análoga a la de los de `substitute`.

**Ejemplo 5.114** *Ilustramos el funcionamiento de* `substitute-if`*, con algunas expresiones tecleadas en interactico sobre el intérprete:*

```
[246]> (setq vector #(3 4))
#(3 4)
[247]> (setq lista (list 1 'a 2 3 '(1 2) #(3 4) 2 'a vector 'a))
(1 A 2 3 (1 2) #(3 4) 2 A #(3 4) A)
[248]> (substitute-if 0 #'numberp lista)
(0 A 0 0 (1 2) #(3 4) 0 A #(3 4) A)
```

**(SUBSTITUTE-IF-NOT elemento predicado secuencia** [**:FROM-END {t|()}**]
[**:START inicio**]
[**:END final**]
[**:COUNT contador**]
[**:KEY función**])

Devuelve una copia de la `secuencia` en la que todas las ocurrencias de elementos que no verifican el `predicado` son reemplazadas por el `elemento`. La interpretación de los argumentos opcionales es análoga a la de los de `substitute`.

**Ejemplo 5.115** *Ilustramos el funcionamiento de* `substitute-if-not`*, con algunas expresiones tecleadas en interactico sobre el intérprete:*

```
[248]> (setq vector #(3 4))
#(3 4)
[249]> (setq lista (list 1 'a 2 3 '(1 2) #(3 4) 2 'a vector 'a))
(1 A 2 3 (1 2) #(3 4) 2 A #(3 4) A)
[250]> (substitute-if-not 0 #'numberp lista)
(1 0 2 3 0 0 2 0 0 0)
```

### 5.9.6. Funciones de modificación física

Estas funciones provocan *efectos colaterales* en sus argumentos, amén de devolver la respuesta indicada. Ello quiere decir que deben ser usadas con extremada precaución, sobre todo cuando nuestras estructuras de datos son compartidas.

El hecho de que Lisp permita modificar físicamente la representación de las listas quiere decir que el lenguaje ofrece el mismo poder de un lenguaje máquina.

**(DELETE elemento secuencia** [**:FROM-END {t|()}**]
[**:TEST test**]
[**:TEST-NOT test**]
[**:START inicio**]
[**:END final**]
[**:COUNT contador**]
[**:KEY función**])

Devuelve la `secuencia`, que se modifica físicamente, de la que se han eliminado todos los elementos iguales, mediante `eql`, a `elemento`. Es, por tanto, la versión destructiva de `remove`, teniendo la misma interpretación para sus argumentos opcionales.

**Ejemplo 5.116** *Ilustramos el funcionamiento de* `delete`*, con algunas expresiones tecleadas en interactico sobre el intérprete:*

```
[242]> (setq vector #(3 4))
#(3 4)
[243]> (setq lista (list 1 'a 2 3 '(1 2) #(3 4) 2 'a vector 'a))
(1 A 2 3 (1 2) #(3 4) 2 A #(3 4) A)
[244]> (delete 2 lista)
(1 A 3 (1 2) #(3 4) A #(3 4) A)
[245]> lista
(1 A 3 (1 2) #(3 4) A #(3 4) A)
[246]> (delete vector lista)
(1 A 3 (1 2) #(3 4) A A)
[247]> lista
(1 A 3 (1 2) #(3 4) A A)
```

Una posible implementación en Lisp de **delete**, sin considerar los argumentos opcionales, es la siguiente:

```
(defun mi-delete (elemento lista)
       (cond ((atom lista) lista)
             ((eql elemento (car lista))
              (mi-delete elemento (cdr lista)))
             (t (rplacd lista (mi-delete elemento (cdr lista))))))
```

**(DELETE-IF predicado secuencia** [**:FROM-END {t|()}**]
[**:START inicio**]
[**:END final**]
[**:COUNT contador**]
[**:KEY función**])

Devuelve la `secuencia`, modificada físicamente, de la que se han eliminado todos los elementos que verifican `predicado`. La interpretación de los argumentos opcionales es análoga a la de `member-if`, siendo su versión destructiva.

**Ejemplo 5.117** *Ilustramos el funcionamiento de* `remove-if`*, con algunas expresiones tecleadas en interactico sobre el intérprete:*

```
[248]> (setq lista '('a "pepe" (1 2) 'b 3 1 2 (a b)))
(1 2 'A "pepe" (1 2) 'B 3 (A B))
[249]> (delete-if #'numberp lista)
('A "pepe" (1 2) 'B (A B))
[250]> lista
('A "pepe" (1 2) 'B (A B))
```

NOTA: Parece haber un problema de funcionamiento con `delete-if`:

```
[251]> (setq lista '(1 2 'a "pepe" (1 2) 'b 3 (a b)))
(1 2 'A "pepe" (1 2) 'B 3 (A B))
[252]> (delete-if #'numberp lista)
('A "pepe" (1 2) 'B (A B))
[253]> lista
(1 2 'A "pepe" (1 2) 'B (A B))  ; esto es incongruente
```

**(DELETE-IF-NOT predicado secuencia [:FROM-END {t|()}] [:START inicio] [:END final] [:COUNT contador] [:KEY función])**

Devuelve la `secuencia`, modificada físicamente, de la que se han eliminado todos los elementos que no verifican `predicado`. La interpretación de los argumentos opcionales es análoga a la de `member-if-not`, siendo su versión destructiva.

**Ejemplo 5.118** *Ilustramos el funcionamiento de* `delete-if-not`*, con algunas expresiones tecleadas en interactico sobre el intérprete:*

```
[251]> (setq lista '(1 2 'a "pepe" (1 2) 'b 3  (a b)))
(1 2 'A "pepe" (1 2) 'B 3 (A B))
[252]> (delete-if-not #'numberp lista)
(1 2 3)
[253]> lista
(1 2 3)
```

NOTA: Parece haber un problema de funcionamiento con `delete-if-not`:

```
[254]> (setq lista '('a "pepe" (1 2) 'b 3 1 2 (a b)))
('A "pepe" (1 2) 'B 3 1 2 (A B))
[255]> (delete-if-not #'numberp lista)
(3 2 1)
[256]> lista
('A "pepe" (1 2) 'B 3 1 2)  ; esto es incongruente
```

**(DELETE-DUPLICATES secuencia [:FROM-END {t|()}] [:START inicio] [:END final] [:COUNT contador] [:KEY función])**

Devuelve una copia de la `secuencia` de la que se han eliminado, físicamente, uno de los elementos de todos los pares consecutivos iguales según `eq`. La interpretación de los argumentos opcionales es análoga a la de `remove-duplicates`.

**Ejemplo 5.119** *Ilustramos el funcionamiento de* `delete-duplicates`*, con algunas expresiones tecleadas en interactico sobre el intérprete:*

```
[242]> (setq lista '(1 2 2 2 3 'a 'a 3 'b 'a))
(1 2 'A 'A 3 'B 'A)
[243}> (delete-duplicates lista)
(1 2 'A 'A 3 'B 'A)
[244]> lista
(1 2 'A 'A 3 'B 'A)
```

**(NSUBSTITUTE elemento elemento secuencia [:FROM-END {t|()}] [:START inicio] [:END final] [:COUNT contador] [:KEY función])**

Devuelve la `secuencia`, modificada físicamente, en la que todas las ocurrencias del `elemento` que actúa como segundo argumento son reemplazadas por el que actúa como primero usando `eq` como test comparativo. La interpretación de los argumentos opcionales es análoga a la de los de `substitute`.

**Ejemplo 5.120** *Ilustramos el funcionamiento de* `nsubstitute`*, con algunas expresiones tecleadas en interactico sobre el intérprete:*

```
[245]> (setq vector #(3 4))
#(3 4)
[245]> (setq lista (list 1 'a 2 3 '(1 2) #(3 4) 2 'a vector 'a))
(1 A 2 3 (1 2) #(3 4) 2 A #(3 4) A)
[246]> (substitute 4 2 lista)
(1 A 4 3 (1 2) #(3 4) 4 A #(3 4) A)
[247]> lista
(1 A 4 3 (1 2) #(3 4) 4 A #(3 4) A)
[248]> (substitute #(A A) vector lista :test #'equal)
(1 A 2 3 (1 2) #(3 4) 2 A #(A A) A)
[249]> lista
(1 A 2 3 (1 2) #(3 4) 2 A #(A A) A)
```

**nsubstitute-if (NSUBSTITUTE-IF elemento predicado secuencia [:FROM-END {t|()}] [:START inicio] [:END final] [:COUNT contador] [:KEY función])**

Devuelve una copia de la `secuencia` en la que todas las ocurrencias de elementos verificando el `predicado` son reemplazadas por el `elemento`. La interpretación de los argumentos opcionales es análoga a la de los de `substitute`.

**Ejemplo 5.121** *Ilustramos el funcionamiento de* `substitute-if`*, con algunas expresiones tecleadas en interactico sobre el intérprete:*

```
[250]> (setq vector #(3 4))
#(3 4)
[251]> (setq lista (list 1 'a 2 3 '(1 2) #(3 4) 2 'a vector 'a))
(1 A 2 3 (1 2) #(3 4) 2 A #(3 4) A)
[252]> (nsubstitute-if 0 #'numberp lista)
(0 A 0 0 (1 2) #(3 4) 0 A #(3 4) A)
[253]> lista
(0 A 0 0 (1 2) #(3 4) 0 A #(3 4) A)
```

**(NSUBSTITUTE-IF-NOT elemento predicado secuencia [:FROM-END {t|()}] [:START inicio] [:END final] [:COUNT contador] [:KEY función])**

Devuelve la `secuencia`, modificada físicamente, en la que todas las ocurrencias de elementos que no verifican el `predicado` son reemplazadas por el `elemento`. La interpretación de los argumentos opcionales es análoga a la de los de `substitute-if-not`.

**Ejemplo 5.122** *Ilustramos el funcionamiento de* `substitute-if-not`*, con algunas expresiones tecleadas en interactico sobre el intérprete:*

```
[254]> (setq vector #(3 4))
#(3 4)
[255]> (setq lista (list 1 'a 2 3 '(1 2) #(3 4) 2 'a vector 'a))
(1 A 2 3 (1 2) #(3 4) 2 A #(3 4) A)
[256]> (substitute-if-not 0 #'numberp lista)
(1 0 2 3 0 0 2 0 0 0)
[257}> lista
(1 0 2 3 0 0 2 0 0 0)
```

**(RPLACA lista objeto)**

Reemplaza físicamente el `car` de `lista` por `objeto`. Devuelve la `lista` así modificada.

**Ejemplo 5.123** *Ilustramos el funcionamiento de* `rplaca`*, con algunas expresiones tecleadas en interactico sobre el intérprete:*

```
[258]> (setq lista '(a b c))
(A B C)
[259]> (rplaca lista '(x y))
```

```
((X Y) B C)
[260]> lista
((X Y) B C)
```

**(RPLACD lista objeto)**

Reemplaza físicamente el `cdr` de `lista` por `objeto`. Devuelve la `lista` así modificada.

**Ejemplo 5.124** *Ilustramos el funcionamiento de* `rplacd`*, con algunas expresiones tecleadas en interactico sobre el intérprete:*

```
[261]> (setq lista '(a b c))
(A B C)
[262]> (rplacd lista '(x y))
(A X Y)
[263]> lista
(A X Y)
```

### 5.9.7. Listas de asociación: A-listas

Las *listas de asociación* son listas formadas por pares de elementos de la forma `(clave . valor)`, el `car` de los cuales funciona como *clave* y el `cdr` como *valor* asociado a dicha clave. En una *A-lista*, no deben existir diferentes pares cuyo `car` sea coincidente mediante el predicado con el predicado de igualdad[16].

**(ACONS clave valor a-lista)**

Devuelve una A-lista resultado de añadir, sin efectos colaterales, a la `a-lista` el nuevo par `(clave . valor)`.

**Ejemplo 5.125** *Ilustramos el funcionamiento de* `acons`*, con algunas expresiones tecleadas en interactico sobre el intérprete:*

```
[264]> (defvar a-lista '((b . 11) (z . 40)))
A-LISTA
[265]> (acons 'a 10 a-lista)
((A . 10) (B . 11) (Z . 40))
[266]> a-lista
((B . 11) (Z . 40))
```

Un ejemplo de implementación de `acons` en Lisp es el que sigue:

[16] `eq`, `eql` o `equal` según la naturaleza de la función de manipulación utilizada en cada caso.

```
(defun mi-acons (clave valor a-lista)
       (cons (cons clave valor) a-lista))
```

**(PAIRLIS lista lista [a-lista])**

La descripción de los argumentos es la siguiente:

- La primera `lista` está compuesta de claves.
- La segunda `lista` está compuesta de valores.
- El argumento `a-lista` es opcional, en caso de incluirse debe ser una A-lista.

La función devuelve una nueva A-lista compuesta por a partir de las claves y valores asociadas a los dos primeros argumentos de `pairlis`. Si el último argumento `a-lista` aparece, se añade al final de la A-lista antes generada.

**Ejemplo 5.126** *Ilustramos el funcionamiento de* `pairlis`*, con algunas expresiones tecleadas en interactico sobre el intérprete:*

```
[267]> (pairlis '(arbre grand le) '(arbol gran el) '((vert . verde)))
((LE . EL) (GRAND . GRAN) (ARBRE . ARBOL) (VERT . VERDE))
```

Una posible implementación en Lisp para `pairlis` es la siguiente:

```
(defun mi-pairlis (claves valores &rest a-lista)
       (if (and (consp claves)
                (listp valores))
           (acons (car claves)
                  (car valores)
                  (mi-pairlis (cdr claves) (cdr valores) a-lista))
         a-lista))
```

**(ASSOC clave a-lista [:TEST test]**
**[:TEST-NOT test]**
**[:KEY función])**

De existir, devuelve el primer par `cons` cuyo `car` coincide con la `clave`, en función del `test` de igualdad considerado. En otro caso, devuelve `()`.

En cuanto a `:test`, determina el test de igualdad a aplicar y que, por defecto, es `eq`. Por su parte, `:test-not test` determina el test sobre cuya negación se aplicará la comparación. Finalmente, `:key` modifica la parte de `x` sobre la que se aplica el test de igualdad. Por ejemplo, si el test es `eql`, en lugar de aplicar `(eql z x)`, usará `(eql z (función x))`.

**Ejemplo 5.127** *Ilustramos el funcionamiento de* `assoc`*, con algunas expresiones tecleadas en interactico sobre el intérprete:*

```
[268]> (setq clave '(a))
(A)
[269]> (setq a-lista (pairlis (list clave 'b 'c)
                              '((0) 1 (d e)) '((1 . 2))))
((C D E) (B . 1) ((A) 0) (1 . 2))
[270]> (assoc 'd a-lista)
NIL
[271]> (assoc 'c a-lista)
(C D E)
[271]> (assoc '(a) a-lista)
NIL
[272]> (assoc clave a-lista)
((A) 0)
[273]> (assoc clave a-lista :test-not #'eql)
(C D E)
[274]> (assoc '(a) a-lista :test #'eql)
NIL
[275]> (assoc '(a) a-lista :test #'equal)
((A) 0)
[276]> (setq a-lista (pairlis (list clave '(b) '(c))
                              '((0) 1 (d e))
                              '((1 . 2))))
(((C) D E) ((B) . 1) ((A) 0) (1 . 2))
[277]> (assoc 'a a-lista :key #'car)
((A) 0)
```

Una posible implementación de la función `assoc` en Lisp, sin argumentos opcionales, sería:

```
(defun mi-assoc (clave a-lista)
       (cond ((atom a-lista) ())
             ((and (consp (car a-lista))
                   (eq (caar a-lista) clave))
              (cdar a-lista))
             (t (mi-assoc clave (cdr a-lista)))))
```

**(ASSOC-IF predicado a-lista [:KEY función])**

De existir, devuelve el primer par `cons` cuyo `car` verifica el test establecido por `predicado`. En otro caso, devuelve `()`. De estar presente, `:key` modifica la parte de `x` sobre la que se aplica el test de igualdad. Por ejemplo, si el test es `eql`, en lugar de aplicar `(eql z x)`, usará `(eql z (función x))`.

**Ejemplo 5.128** *Ilustramos el funcionamiento de* `assoc-if`*, con algunas expresiones tecleadas en interactico sobre el intérprete:*

```
[277]> (setq clave '(a))
(A)
[278]> (setq a-lista (pairlis (list clave '(b) '(c))
                              '((0) 1 (d e))
                              '((1 . 2))))
(((C) D E) ((B) . 1) ((A) 0) (1 . 2))
[279]> (assoc-if #'numberp a-lista)
(1 . 2)
```

**(ASSOC-IF-NOT predicado a-lista [:KEY función])**

De existir, devuelve el primer par `cons` cuyo `car` no verifica el test establecido por `predicado`. En otro caso, devuelve `()`. De estar presente, `:key` modifica la parte de `x` sobre la que se aplica el test de igualdad. Por ejemplo, si el test es `eql`, en lugar de aplicar `(eql z x)`, usará `(eql z (función x))`.

**Ejemplo 5.129** *Ilustramos el funcionamiento de* `assoc-if-not`*, con algunas expresiones tecleadas en interactico sobre el intérprete:*

```
[279]> (setq clave '(a))
(A)
[280]> (setq a-lista (pairlis (list clave '(b) '(c))
                              '((0) 1 (d e))
                              '((1 . 2))))
(((C) D E) ((B) . 1) ((A) 0) (1 . 2))
[281]> (assoc-if-not #'consp a-lista)
(1 . 2)
```

**(RASSOC valor a-lista [:TEST test]**
**[:TEST-NOT test]**
**[:KEY función])**

De existir, devuelve el primer par `cons` cuyo `cdr` coincide con el `valor`, en función del `test` de igualdad considerado. En otro caso, devuelve `()`. La interpretación de los argumentos opcionales es la misma que en el caso de `assoc`.

**Ejemplo 5.130** *Ilustramos el funcionamiento de* `rassoc`*, con algunas expresiones tecleadas en interactico sobre el intérprete:*

```
[282]> (setq clave '(a))
(A)
[283]> (setq a-lista (pairlis (list 'a 'b 'c)
                      (list '(0) 1 clave)
                      '((1 . 2))))
((C A) (B . 1) (A 0) (1 . 2))
```

```
[284]> (rassoc 1 a-lista)
(B . 1)
```

Una posible implementación de la función `rassoc` en Lisp, sin argumentos opcionales, sería:

```
(defun mi-rassoc (valor a-lista)
       (cond ((atom a-lista) ())
             ((and (consp (car a-lista))
                   (eq (cdar a-lista) valor))
              (car a-lista))
             (t (mi-rassoc valor (cdr a-lista)))))
```

**(RASSOC-IF predicado a-lista [:KEY función])**

De existir, devuelve el primer par `cons` cuyo `cdr` verifica el test establecido por `predicado`. En otro caso, devuelve `()`. La interpretación de los argumentos opcionales es la misma que en el caso de `assoc-if`.

**Ejemplo 5.131** *Ilustramos el funcionamiento de* `rassoc-if`*, con algunas expresiones tecleadas en interactico sobre el intérprete:*

```
[285]> (setq clave '(a))
(A)
[286]> (setq a-lista (pairlis (list 'a 'b 'c)
                              (list '(0) 1 clave)
                              '((1 . 2))))
((C A) (B . 1) (A 0) (1 . 2))
[287]> (rassoc-if #'numberp a-lista)
(B . 1)
```

**(RASSOC-IF-NOT predicado a-lista [:KEY función])**

De existir, devuelve el primer par `cons` cuyo `car` no verifica el test establecido por `predicado`. En otro caso, devuelve `()`. La interpretación de los argumentos opcionales es la misma que en el caso de `assoc-if-not`.

**Ejemplo 5.132** *Ilustramos el funcionamiento de* `rassoc-if-not`*, con algunas expresiones tecleadas en interactico sobre el intérprete:*

```
[288]> (setq clave '(a))
(A)
[289]> (setq a-lista (pairlis (list 'a 'b 'c)
                              (list '(0) 1 clave)
                              '((1 . 2))))
```

```
((C A) (B . 1) (A 0) (1 . 2))
[290]> (rassoc-if-not #'consp a-lista)
(B . 1)
```

**(COPY-ALIST a-lista)**

Devuelve una copia de la `a-lista`. Este objeto es idéntico al original mediante `equal`, pero no mediante `eq`.

**Ejemplo 5.133** *Para ilustrar el funcionamiento de* `copy-alist`*, tecleraremos algunas expresiones en interactico, directamente sobre el intérprete:*

```
[291]> (copy-alist (pairlis (list 'a 'b 'c)
                            (list '(0) 1 "hola")
                            '((1 . 2))))
((C . "hola") (B . 1) (A 0) (1 . 2))
```

### 5.9.8. Usando las listas como conjuntos

Incluímos aquí aquellas funciones que permiten que una lista sea tratada como un conjunto, lo que implica hacer abstracción del orden implícito a aquel tipo de secuencia.

**(ADJOIN elemento lista [:TEST test]**
**[:TEST-NOT test]**
**[:KEY función])**

Añade el `elemento` a `lista`, siempre y cuando no esté ya presente en la misma mediante el `test` de igualdad indicado, devolviendo una copia del resultado. En cuanto a los argumentos opcionales, `:test`, determina el test de igualdad a aplicar y que, por defecto, es `eq`. Por su parte, `:test-not test` determina el test sobre cuya negación se aplicará la comparación. Finalmente, `:key` modifica la parte de `x` sobre la que se aplica el test de igualdad. Por ejemplo, si el test es `eql`, en lugar de aplicar `(eql z x)`, usará `(eql (función z) (función x))`.

**Ejemplo 5.134** *Para ilustrar el funcionamiento de* `adjoin`*, tecleraremos algunas expresiones en interactico, directamente sobre el intérprete:*

```
[292]> (adjoin 2 '(1 2 3))
(1 2 3)
[293]> (adjoin 4 '(1 2 3))
(4 1 2 3)
[294]> (setq lista '((c 3) (d 4) (e 5)))
((c 3) (d 4) (e 5))
```

```
[295]> (adjoin '(c 3) lista :test #'equal)
((C 3) (D 4) (E 5))
[296]> (adjoin '(c 3) lista :test-not #'equal)
((C 3) (D 4) (E 5))
[297]> (adjoin '(c 3) lista :test #'eq)
((C 3) (C 3) (D 4) (E 5))
[298]> (adjoin '(c 3) lista :test-not #'eq)
((C 3) (D 4) (E 5))
[299]> (adjoin '(b 6) lista :key #'car)
((B 6) (C 3) (D 4) (E 5))
[300]> (adjoin '(c 6) lista :key #'car)
((C 3) (D 4) (E 5))
```

**(MEMBER elemento lista [:TEST test]**
**[:TEST-NOT test]**
**[:KEY función])**

En caso de encontrarlo, devuelve la `lista` a partir del primer `elemento` localizado que verifique el test de igualdad señalado por `:test` que, por defecto, es `eql`. En otro caso devuelve `()`. En relación a las *palabras clave de lambda lista*, y asumiendo una `lista` de la forma `(... x ...)`, estas son:

- `:key función`. Modifica la parte de `x` sobre la que se aplica el test de igualdad. Por ejemplo, si el test es `eql`, en lugar de aplicar `(eql z x)`, usará `(eql z (función x))`.
- `:test test`. Determina el `test` de igualdad a aplicar y que, por defecto, es `eql`.
- `:test-not test`. Determina el `test` sobre cuya negación se aplicará la comparación.

**Ejemplo 5.135** *Consideramos un ejemplo sencillo, simplemente tecleando algunas expresiones en interactivo sobre el intérprete. Comencemos por* `:test`*:*

```
[170]> (member '(b 2) '((1 a) (b 2) (c 3)))
NIL
[171]> (member '(b 2) '((1 a) (b 2) (c 3)) :test #'eq)
NIL
[172]> (member '(b 2) '((1 a) (b 2) (c 3)) :test #'equal)
((B 2) (C 3))
[173]> (member '(b 2) '((1 a) (b 2) (c 3)) :test #'eql)
NIL
```

*Si pasamos ahora al caso de* `:key`*, podemos considerar:*

```
[174]> (member 'b '((1 a) (b 2) (c 3)))
NIL
[175]> (member 'b '((1 a) (b 2) (c 3)) :key #'car)
((B 2) (C 3))
[176]> (member 'b '((1 a) (b 2) (c 3)) :key #'first)
((B 2) (C 3))
[177]> (member 'b '((1 a) (b 2) (c 3)) :key #'second)
NIL
[178]> (member '2 '((1 a) (b 2) (c 3)) :key #'second)
((B 2) (C 3))
```

*Finalmente, tocando a la palabra clave* `:test-not`*, sirvan de ejemplo las siguientes expresiones interpretadas interativamente:*

```
[179]> (member '(b 2) '((1 a) (b 2) (c 3)) :test #'eq)
NIL
[180]> (member '(b 2) '((1 a) (b 2) (c 3)) :test-not #'eql)
((1 A) (B 2) (C 3))
[181]> (member '(1 a) '((1 a) (b 2) (c 3)) :test #'equal)
((1 A) (B 2) (C 3))
[182]> (member '(1 a) '((1 a) (b 2) (c 3)) :test-not #'equal)
((B 2) (C 3))
```

**(MEMBER-IF predicado lista [:KEY función])**

En caso de encontrarlo, devuelve la `lista` a partir del primer ítem localizado que verifique el test de igualdad señalado por `predicado`. En otro caso devuelve `()`. La interpretación de la palabra clave `:key` es la misma que en el caso de `member`.

**Ejemplo 5.136** *Tecleando algunas expresiones en interactivo sobre el intérprete, para comprobar el comportamiento de* `nthcdr`*:*

```
[183]> (member-if #'numberp '(a b 1 c d))
(1 C D)
[184]> (member-if #'numberp '((a b) (1 c) (d e)) :key #'first)
((1 C) (D E))
```

**(MEMBER-IF-NOT predicado lista [:KEY función])**

En caso de encontrarlo, devuelve la `lista` a partir del primer ítem localizado que verifique la negación deltest de igualdad señalado por `predicado`. En otro caso devuelve `()`. La interpretación de la palabra clave `:key` es la misma que en el caso de `member`.

**Ejemplo 5.137** *Tecleando algunas expresiones en interactivo sobre el intérprete, para comprobar el comportamiento de* `nthcdr`*:*

```
[185]> (member-if-not #'atom '(a b (1 2) c d))
((1 2) C D)
[186]> (member-if-not #'atom '((a b) ((1 2) 3) (c d)) :key #'first)
(((1 2) 3) (C D))
```

**(UNION lista lista [:TEST test] [:TEST-NOT test] [:KEY función])**

Devuelve una copia del resultado de la unión de las dos `listas` argumento, evitando duplicaciones entre éstas. No hay garantía de que se mantenga el orden original de los elementos en las `listas` argumento. En caso de que alguna de estas `listas` incluya de partida alguna duplicación, no está garantizado que el resultado no la incluya también. En cuanto a los argumentos opcionales, la interpretación es la misma que en el caso de `adjoin`.

**Ejemplo 5.138** *Para ilustrar el funcionamiento de* `union`*, teclearemos algunas expresiones en interactico, directamente sobre el intérprete:*

```
[301]> (union '(a b c) '(f a d))
(B C F A D)
[302]> (union '(a a b c) '(f a d))
(B C F A D)
[303]> (setq test1 '((c 3) (d 4) (e 5)))
((C 3) (D 4) (E 5))
[304]> (setq test2 '((c 3) (f 6) (e 7)))
((C 3) (F 6) (E 7))
[305]> (union test1 test2 :test #'equal)
((D 4) (E 5) (C 3) (F 6) (E 7))
[306]> (union test1 test2 :test #'eq)
((C 3) (D 4) (E 5) (C 3) (F 6) (E 7))
[307]> (union test1 test2)
((C 3) (D 4) (E 5) (C 3) (F 6) (E 7))
[308]> (union test1 test2 :key #'car)
((D 4) (C 3) (F 6) (E 7))
```

**(NUNION lista lista [:TEST test] [:TEST-NOT test] [:KEY función])**

La funcionalidad es la misma que la de `union`, pero en este caso las `listas` argumento (la primera) son alteradas físicamente.

**Ejemplo 5.139** *Para ilustrar el funcionamiento de* `nunion`*, teclearemos algunas expresiones en interactico, directamente sobre el intérprete:*

```
[310]> (setq test1 '((c 3) (d 4) (e 5)))
((C 3) (D 4) (E 5))
[311]> (setq test2 '((c 3) (f 6) (e 7)))
((C 3) (F 6) (E 7))
[312]> (union test1 test2)
((C 3) (D 4) (E 5) (C 3) (F 6) (E 7))
[313]> test1
((C 3) (D 4) (E 5))
[314]> test2
((C 3) (F 6) (E 7))
[315]> (nunion test1 test2)
((C 3) (D 4) (E 5) (C 3) (F 6) (E 7))
[316]> test1
((C 3) (D 4) (E 5) (C 3) (F 6) (E 7))
[317]> test2
((C 3) (F 6) (E 7))
```

**(INTERSECTION lista lista [:TEST test] [:TEST-NOT test] [:KEY función])**

Devuelve una copia del resultado de la intersección de las dos `listas` argumento. No hay garantía de que se mantenga el orden original de los elementos en las `listas` argumento. En caso de que alguna de estas `listas` incluya de partida alguna duplicación, no está garantizado que el resultado no la incluya también. En cuanto a los argumentos opcionales, la interpretación es la misma que en el caso de `adjoin`.

**Ejemplo 5.140** *Para ilustrar el funcionamiento de* `intersection`*, teclearemos algunas expresiones en interactico, directamente sobre el intérprete:*

```
[318]> (setq test1 '((c 3) (d 4) (e 5)))
((C 3) (D 4) (E 5))
[319]> (setq test2 '((c 3) (f 6) (e 7)))
((C 3) (F 6) (E 7))
[320]> (intersection test1 test2 :test #'equal)
((C 3))
[321]> (intersection test1 test2 :test #'eq)
NIL
[322]> (intersection test1 test2 :key #'car)
((C 3) (E 5))
```

**(NINTERSECTION lista lista [:TEST test]**
**[:TEST-NOT test]**
**[:KEY función])**

La funcionalidad es la misma que la de `intersection`, pero en este caso las `listas` argumento (la primera) son alteradas físicamente. En cuanto a los argumentos opcionales, la interpretación es la misma que en el caso de `adjoin`.

**Ejemplo 5.141** *Para ilustrar el funcionamiento de* `nintersection`*, teclearemos algunas expresiones en interactico, directamente sobre el intérprete:*

```
[323]> (setq test1 '((c 3) (d 4) (e 5)))
((C 3) (D 4) (E 5))
[324]> (setq test2 '((c 3) (f 6) (e 7)))
((C 3) (F 6) (E 7))
[325]> (intersection test1 test2 :test #'equal)
((C 3))
[326]> test1
((C 3))
[327]> test2
((C 3) (F 6) (E 7))
```

**(SET-DIFFERENCE lista lista [:TEST test]**
**[:TEST-NOT test]**
**[:KEY función])**

Devuelve una copia del resultado de la eliminación de la primera `lista` argumento de aquellos que aparecen en la segunda. No hay garantía de que se mantenga el orden original de los elementos en las `listas` argumento. En cuanto a los argumentos opcionales, la interpretación es la misma que en el caso de `adjoin`.

**Ejemplo 5.142** *Para ilustrar el funcionamiento de* `set-difference`*, teclearemos algunas expresiones en interactico, directamente sobre el intérprete:*

```
[328]> (setq test1 '((c 3) (d 4) (e 5)))
((C 3) (D 4) (E 5))
[329]> (setq test2 '((c 3) (f 6) (e 7)))
((C 3) (F 6) (E 7))
[330]> (set-difference test1 test2 :test #'equal)
((D 4) (E 5))
[331]> (set-difference test1 test2 :test #'eq)
((C 3) (D 4) (E 5))
```

```
[332]> (set-difference test1 test2 :key #'car)
((D 4))
[333]> (set-difference '(1 2 2 3 4 2 5) '(2 6 7))
(1 3 4 5)
```

**(NSET-DIFFERENCE lista lista [:TEST test]**
**[:TEST-NOT test]**
**[:KEY función])**

La funcionalidad es la misma que la de `set-difference`, pero en este caso las `listas` argumento (la primera) son alteradas físicamente. En cuanto a los argumentos opcionales, la interpretación es la misma que en el caso de `adjoin`.

**Ejemplo 5.143** *Para ilustrar el funcionamiento de* `nset-difference`*, teclearemos algunas expresiones en interactico, directamente sobre el intérprete:*

```
[334]> (setq test1 '(a b a a c d))
(A B A A C D)
[335]> (setq test2 '(a d))
(A D)
[336]> (set-difference test1 test2)
(B C)
[337]> test1
(A B A A C D)
[338]> test2
(A D)
[339]> (nset-difference test1 test2)
(B C)
[340]> test1
(A B C)
[341]> test2
(A D)
```

**(SET-EXCLUSIVE-OR lista lista [:TEST test]**
**[:TEST-NOT test]**
**[:KEY función])**

Devuelve una lista conteniendo los elementos que sólo aparecen en una de las `listas` argumento. No hay garantía de que se mantenga el orden original de los elementos en las `listas` argumento. En cuanto a los argumentos opcionales, la interpretación es la misma que en el caso de `adjoin`.

**Ejemplo 5.144** *Para ilustrar el funcionamiento de* `set-exclusive-or`*, teclearemos algunas expresiones en interactico, directamente sobre el intérprete:*

```
[342]> (setq test1 '((c 3) (d 4) (e 5)))
((C 3) (D 4) (E 5))
[343]> (setq test2 '((c 3) (f 6) (e 7)))
((C 3) (F 6) (E 7))
[344]> (set-exclusive-or test1 test2 :test #'equal)
((D 4) (E 5) (F 6) (E 7))
[345]> (set-exclusive-or test1 test2 :test #'eq)
((C 3) (D 4) (E 5) (C 3) (F 6) (E 7))
[346]> (set-exclusive-or test1 test2 :key #'car)
((D 4) (F 6))
[347]> (set-exclusive-or '(1 2 2 3 4 2 5) '(2 6 7))
(1 3 4 5 6 7)
```

**(NSET-EXCLUSIVE-OR lista lista [:TEST test] [:TEST-NOT test] [:KEY función])**

La funcionalidad es la misma que la de `set-exclusive-or`, pero en este caso las `listas` argumento podrían ser alteradas físicamente. En cuanto a los argumentos opcionales, la interpretación es la misma que en el caso de `adjoin`.

**Ejemplo 5.145** *Para ilustrar el funcionamiento de* `nset-exclusive-or`*, teclearemos algunas expresiones en interactico, directamente sobre el intérprete:*

```
[348]> (setq test1 '(a b a a c d))
(A B A A C D)
[349]> (setq test2 '(a d e f))
(A D E F)
[350]> (set-exclusive-or test1 test2)
(B C E F)
[351]> test1
(A B A A C D)
[352]> test2
(A D E F)
[353]> (nset-exclusive-or test1 test2)
(B C E F)
[354]> test1
(A B A A C D)
[355]> test2
(A D E F)
```

**(SUBSETP lista lista [:TEST test] [:TEST-NOT test] [:KEY función])**

Es un predicado que devuelve `t` si todo elemento de la primera `lista` argumento aparece también en la segunda. En otro caso devuelve `()`.

**Ejemplo 5.146** *Para ilustrar el funcionamiento de* `set-exclusive-or`*, teclearemos algunas expresiones en interactico, directamente sobre el intérprete:*

```
[356]> (setq test1 '((c 3) (d 4) (e 5)))
((C 3) (D 4) (E 5))
[357]> (setq test2 '((c 3) (f 6) (e 7) (d 4) (e 5)))
((C 3) (F 6) (E 7) (D 4) (E 5))
[358]> (subsetp  test1 test2)
NIL
[359]> (subsetp  test1 test2 :test #'equal)
T
```

## 5.10. Funciones sobre símbolos

### 5.10.1. Funciones de acceso a los valores de los símbolos

**(boundp symb)**

Testea si el símbolo **symb** tiene algún valor asociado como variable. En caso afirmativo devuelve **t**, y () en otro caso.

Ejemplo:

```
? (boundp 'hola)
= ()
? (defvar hola "Hola")
= hola
? (boundp 'hola)
= t
```

### 5.10.2. Funciones que modifican los valores de los símbolos

Permiten definir variables y constantes globales, así como modificar su valor.

**(DEFVAR símbolo [forma [cadena]])**

Aún cuando las variables no necesitan ser declaradas en Lisp, es una buena práctica el hacerlo en el caso de las globales. Este el objeto fundamental de `defvar`, que asigna la evaluación de la `forma` a `símbolo`, devolviendo dicho `símbolo` como respuesta. El argumento `forma` puede ser omitido, en

cuyo caso por defecto toma el valor `()`. En cuanto a `cadena`, se trata de una cadena de caracteres opcional que describe la naturaleza de la nueva variable global.

Una vez declarada así una variable, la única forma de cambiar su valor es mediante `setq`. Una nueva llamada a `defvar` sobre una variable global ya declarada, no modifica su valor.

**Ejemplo 5.147** *Ilustramos el funcionamiento de* `defvar`*, con algunas expresiones tecleadas en interactico sobre el intérprete:*

```
[1]>(defvar variable 2  "Esto es un ejemplo de variable")
VARIABLE
[2]> variable
2
[3]> (setq variable 3)
3
[4]> variable
3
[5]> (defvar variable 4)
VARIABLE
[6]> variable
3                       ; la nueva llamada no ha cambiado el valor
```

**(DEFPARAMETER símbolo forma [cadena])**

Su funcionamiento es similar de `defvar`, salvo que se usa para declarar variables globales que no cambian durante la ejecución del programa. Por esta razón, en este caso el argumento `forma` es obligatorio. Sin embargo esta diferencia con `defvar` es sólo teórica. En la práctica, si podríamos modificar su valor mediante un `setq`. También, a diferencia de `defvar`, una redefinición mediante `defparameter` si resulta efectiva.

**Ejemplo 5.148** *Ilustramos el funcionamiento de* `defparameter`*, con algunas expresiones tecleadas en interactico sobre el intérprete:*

```
[7]> (defparameter parametro 4 "Esto es un parametro")
PARAMETRO
[8]> parametro
4
[9]> (setq parametro 3)          ; esto no deberia ser posible
3
[10]> parametro
3
[11]> (defparameter parametro 5)
PARAMETRO
```

```
[12]> parametro                    ;con defvar esto no era posible
5
```

### (DEFCONSTANT símbolo forma [cadena])

Su funcionamiento es similar al de `defparameter`, usándose para declarar variables globales que no cambian durante la ejecución del programa. A diferencia de `defparameter`, en este caso el intérprete sí tiene en cuenta las hopótesis semánticas oportunas, siendo imposible modificar el valor inicial mediante `setq`, `defvar` o `defparameter`.

**Ejemplo 5.149** *Ilustramos el funcionamiento de* `defconstant`*, con algunas expresiones tecleadas en interactico sobre el intérprete:*

```
[13]> (defconstant constante 5 "Esto es una constante")
CONSTANTE
[14]> constante
5
[15]> (setq constante 2)

*** - SETQ: CONSTANTE is a constant, may not be used as a variable
Es posible continuar en los siguientes puntos:
USE-VALUE      :R1      Input a value to be used instead.
ABORT          :R2      Abort main loop
```

### (SETQ {símbolo valor}*)

Asigna la secuencia de `símbolos` a la secuencia de evaluaciones de las `formas`. Devuelve el `valor` de la última de las asignaciones realizadas. La función `setq` se puede utilizar en tres contextos diferentes:

- Declaración de variables globales. Dicho uso debe ser evitado, reservándolo para la función `defvar`.
- Reinicialización del valor de una variable global.
- Reinicialización del valor de una variable local.

**Ejemplo 5.150** *Ilustramos el funcionamiento de* `setq`*, con algunas expresiones tecleadas en interactico sobre el intérprete:*

```
[16]> (setq global-1 1)
1
[17]> global-1
1
[18]> (defvar global-2 2)
```

```
GLOBAL-2
[19]> global-2
2
[20]> (setq global-2 3)
3
[21]> global-2
3
[22]> (defun asignacion-local (valor)
             (let ((valor-local valor))
                  (format t "Valor local original= ~D ~%" valor-local)
                  (setq valor-local 2)
                  (format t "Valor local nuevo = ~D" valor-local)
                  ()))
ASIGNACION-LOCAL
[23]> (asignacion-local 1)
Valor local original= 1
Valor local nuevo = 2
NIL
```

### (PSETQ {símbolo forma}*)

La funcionalidad es la misma que en `setq` son una salvedad. Mientras que en `setq` la asignación de las evaluaciones de las `formas` se realiza en secuencia, aquí se hace en paralelo. Devuelve el valor `()`.

**Ejemplo 5.151** *Ilustramos el funcionamiento de* `psetq`*, con algunas expresiones tecleadas en interactico sobre el intérprete:*

```
[24]> (defvar a 1)
A
[25]> (defvar b 2)
B
[26]> a
1
[27]> b
2
[28]> (setq a b b a)
2
[29]> a
2
[30]> b
2
[31]> (setq a 1 b 2)
2
```

```
[32]> a
1
[33]> b
2
[34]> (psetq a b b a)
NIL
[35]> a
2
[36]> b
1
```

**(SET símbolo forma)**

La funcionalidad es la misma de `setq`, salvo que aquí el `símbolo` no se evalúa y que `set` no puede alterar el valor de una variable local.

**Ejemplo 5.152** *Ilustramos el funcionamiento de* `set`*, con algunas expresiones tecleadas en interactico sobre el intérprete:*

```
[37]> (set 'global-1 1)
= 1
[38]> global-1
1
[39]> (defvar global-2 2)
GLOBAL-2
[40]> global-2
2
[41]> (set 'global-2 3)
3
[42]> global-2
3
[43]> (defun asignacion-local (valor)
        (let ((valor-local valor))
             (format t "Valor local inicial= ~D ~%" valor-local)
             (set (quote valor-local) 2)
             (format t "Valor local final= ~D" valor-local)
             ()))
ASIGNACION-LOCAL
[44]>(asignacion-local 1)
Valor local inicial= 1
Valor local final= 1
NIL
```

**(MAKUNBOUND símbolo)**

Provoca la desafección de su valor a la variable de nombre `símbolo`.

**Ejemplo 5.153** *Ilustramos el funcionamiento de* `makunbound`*, con algunas expresiones tecleadas en interactico sobre el intérprete:*

```
[45]> (setq a 1)
1
[46]> (makunbound a)
A
[47]> a

*** - SYSTEM::READ-EVAL-PRINT: variable A has no value
Es posible continuar en los siguientes puntos:
USE-VALUE      :R1      Input a value to be used instead of A.
STORE-VALUE    :R2      Input a new value for A.
ABORT          :R3      Abort debug loop
ABORT          :R4      Abort debug loop
ABORT          :R5      Abort debug loop
ABORT          :R6      Abort debug loop
ABORT          :R7      Abort debug loop
ABORT          :R8      Abort debug loop
ABORT          :R9      Abort debug loop
ABORT          :R10     Abort debug loop
ABORT          :R11     Abort debug loop
ABORT          :R12     Abort debug loop
ABORT          :R13     Abort debug loop
ABORT          :R14     Abort debug loop
ABORT          :R15     Abort debug loop
ABORT          :R16     Abort debug loop
ABORT          :R17     Abort debug loop
ABORT          :R18     Abort debug loop
ABORT          :R19     Abort debug loop
ABORT          :R20     Abort debug loop
ABORT          :R21     Abort debug loop
ABORT          :R22     Abort main loop
[48]> (setq a 2)
2
[49]> a
2
```

### (SETF {puntero forma}*)

Asigna, y devuelve, a cada `puntero` la alocación de memoria del resultado de evaluar la `forma` asociada. La asignación es secuencial.

**Ejemplo 5.154** *Ilustramos el funcionamiento de* `setf`*, con algunas expresiones tecleadas en interactico sobre el intérprete:*

```
[59]> (setf a (make-array 3))
#(NIL NIL NIL)
[60]> a
#(NIL NIL NIL)
[61]> (defvar b (make-array 5))
#(NIL NIL NIL NIL NIL)
[62]> (setf c b)
#(NIL NIL NIL NIL NIL)
[63]> c
#(NIL NIL NIL NIL NIL)
```

### (PSETF {puntero forma}*)

Idéntica a `setf`, salvo que la asignación es aquí en paralelo y que devuelve siempre `()`.

**Ejemplo 5.155** *Ilustramos el funcionamiento de* `setf`*, con algunas expresiones tecleadas en interactico sobre el intérprete:*

```
[64]> (setq a (make-array 3))
#(NIL NIL NIL)
[61]> (setq b (make-array 5))
#(NIL NIL NIL NIL NIL)
[62]> a
#(NIL NIL NIL)
[63]> b
#(NIL NIL NIL NIL NIL)
[64]> (setf a b b a)
NIL
[65]> a
#(NIL NIL NIL NIL NIL)
[66]> b
#(NIL NIL NIL NIL NIL)
[67]> (setq a (make-array 3) b (make-array 5))
#(NIL NIL NIL NIL NIL)
[68]> a
#(NIL NIL NIL)
[69]> b
#(NIL NIL NIL NIL NIL)
[70]> (psetf a b b a)
NIL
[71]> a
#(NIL NIL NIL NIL NIL)
[72]> b
#(NIL NIL NIL)
```

### 5.10.3. Acceso a la definición de las funciones

Permiten testear la asociación de símbolos a funciones, así como recuperar ésta o modificarla físicamente.

**(FUNCTION símbolo)**

Devuelve la interpretación funcional de de `símbolo`, asociado al nombre de una función global. El `símbolo` no es evaluado.

**Ejemplo 5.156** *Ilustramos el funcionamiento de* `function`*, con algunas expresiones tecleadas en interactico sobre el intérprete:*

```
[53]> (function print)
#<SYSTEM-FUNCTION PRINT>
```

**(FDEFINITION símbolo)**

Idéntica a `function`, salvo en el hecho de que `símbolo` si es aquí evaluado.

**Ejemplo 5.157** *Ilustramos el funcionamiento de* `fdefinition`*, con algunas expresiones tecleadas en interactico sobre el intérprete:*

```
[54]> (fdefinition 'print)
#<SYSTEM-FUNCTION PRINT>
```

**(SYMBOL-FUNCTION símbolo)**

Idéntica a `fdefinition`, salvo que admite el `símbolo` asociado a cualquier función, no necesariamente global.

**Ejemplo 5.158** *Ilustramos el funcionamiento de* `symbol-function`*, con algunas expresiones tecleadas en interactico sobre el intérprete:*

```
[55]> (symbol-function 'print)
#<SYSTEM-FUNCTION PRINT>
```

**(FBOUNDP símbolo)**

Devuelve `t` si `símbolo` está asociado a una función de carácter global; sino devuelve `()`. Evalúa `símbolo`.

**Ejemplo 5.159** *Ilustramos el funcionamiento de* `symbol-function`*, con algunas expresiones tecleadas en interactico sobre el intérprete:*

```
[56]> (fboundp 'print)
T
```

**(FMAKUNBOUND símbolo)**

Provoca la desafección de su definición de la función de nombre `símbolo`.

**Ejemplo 5.160** *Ilustramos el funcionamiento de* `fmakunbound`*, con algunas expresiones tecleadas en interactico sobre el intérprete:*

```
[49]> (defun hola () (print "Hola mundo"))
HOLA
[50]> (fdefinition 'hola)
#<FUNCTION HOLA NIL (DECLARE (SYSTEM::IN-DEFUN HOLA))
  (BLOCK HOLA (PRINT "Hola mundo"))>
[51]> (fmakunbound 'hola)
HOLA
[52]> (fdefinition 'hola)

*** - FDEFINITION: la función HOLA no está¡ definida
Es posible continuar en los siguientes puntos:
USE-VALUE      :R1      Input a value to be used instead of (FDEFINITION 'HOLA)
RETRY          :R2      Reintentar
STORE-VALUE    :R3      Input a new value for (FDEFINITION 'HOLA).
ABORT          :R4      Abort debug loop
ABORT          :R5      Abort debug loop
ABORT          :R6      Abort debug loop
ABORT          :R7      Abort debug loop
ABORT          :R8      Abort debug loop
ABORT          :R9      Abort debug loop
ABORT          :R10     Abort debug loop
ABORT          :R11     Abort debug loop
ABORT          :R12     Abort debug loop
ABORT          :R13     Abort debug loop
ABORT          :R14     Abort debug loop
ABORT          :R15     Abort debug loop
ABORT          :R16     Abort debug loop
ABORT          :R17     Abort debug loop
ABORT          :R18     Abort debug loop
ABORT          :R19     Abort debug loop
ABORT          :R20     Abort debug loop
ABORT          :R21     Abort main loop
```

**(SPECIAL-FORM-P símbolo)**

Devuelve `t` si `símbolo` está asociado a una función de carácter global; sino devuelve `()`. Evalúa `símbolo`.

**Ejemplo 5.161** *Ilustramos el funcionamiento de* `symbol-function`*, con algunas expresiones tecleadas en interactico sobre el intérprete:*

```
[57]> (special-form-p 'print)
NIL
[58]> (special-form-p 'catch)
T
```

**(DEFINE-MODIFY-MACRO símbolo puntero lambda-lista función [cadena])**

Genera una macro identificable mediante `símbolo`, siendo un `puntero` su primer argumento y el resto de argumentos identificables por `lambda-lista`. La nueva macro redefine la aplicación de `función` a la lista completa de argumentos, de manera que el valor resultante es asignado a `puntero`.

**Ejemplo 5.162** *Ilustramos el funcionamiento de* `define-modify-macro`*, con algunas expresiones tecleadas en interactico sobre el intérprete:*

```
[59]> (setq x '(1) y '(2) z '(3))
(3)
[60]> (define-modify-macro appendf (&rest args) append "Concatenacion con
       reasignacion fisica del primer argumento")
APPENDF
[61]> (appendf x y z)
(1 2 3)
[62]> x
(1 2 3)
[63]> y
(2)
[64]> z
(3)
```

**(DEFSETF función función [cadena])**

Permite redefinir `setf` sobre cualquier puntero resultante de la evaluación de una forma con la primera `función` argumento. La redefinición se hace mediante la segunda `función` argumento. Como restricción, el primer argumento no puede ser una `función` ya predefinida por el sistema.

**Ejemplo 5.163** *Ilustramos el funcionamiento de* `defset`*, con algunas expresiones tecleadas en interactico sobre el intérprete. La idea en este caso es mostrar que una forma* `(defsetf symbol-value set)` *establece* `(setf (symbol-value a) 5` *con* `(set a 5)`*.*

```
[65]> (defun my-symbol-value (s) (symbol-value s))
TOTO
[66]> (setq a 5)
5
[67]> a
5
[68]> (defsetf my-symbol-value set)
TOTO
[69]> (setf (my-symbol-value 'a) 3) ; se expande como (set 'a 3)
3
[70]> a
3
```

## 5.11. Funciones sobre caracteres

Manipulan los códigos internos, considerados por el intérprete, de los caracteres. Esta codificación no tiene porque corresponderse forzósamente con la codificación ASCII.

**(CHAR-CODE carácter)**

Devuelve el código interno del `carácter`.

**(CODE-CHAR código)**

Devuelve el carácter con el `código` interno indicado, si existe. En otro caso devuelve `()`.

**(CHARACTER objeto)**

Transforma, si es posible, y devuelve el `objeto` transformado a carácter.

**(CHAR-UPCASE carácter)**

Convierte, si posible, y devuelve el `carácter` a mayúscula.

**(CHAR-DOWNCASE carácter)**

Convierte, si posible, y devuelve el `carácter` a mayúscula.

**Ejemplo 5.164** *Una posible implementación de* `character` *viene dada por:*

```
(defun mi-character (objeto) (coerce objeto) 'character)
```

**Ejemplo 5.165** *Ilustraremos el funcionamiento de las funciones sobre caracteres con una secuencia de llamadas tecleadas directamente sobre el intérprete:*

```
[71]> (char-code #\A)
65
[72]> (code-char (char-code #\A))
#\A
[73]> (code-char 5000)
#\ETHIOPIC_SYLLABLE_SEBATBEIT_FWA
[74]> (char-upcase #\a)
#\A
[75]> (char-downcase #\A)
#\a
```

## 5.12. Funciones sobre cadenas de caracteres

Una *cadena de caracteres* es una colección de caracteres accesible por su índice, el cual es un número natural. En Lisp, la representación externa de una cadena es de la forma:

"*xxxxxxxxxxxxx*"

siendo su tipo `string`.

**(MAKE-STRING talla [:INITIAL-ELEMENT elemento] [:ELEMENT-TYPE subtipo-carácter])**

Devuelve una cadena con la `talla` establecida y, opcionalmente, donde sus elementos se inicializan a `elemento` que debe ser de tipo carácter u, opcionalmente, de un `subtipo` de `carácter`.

**Ejemplo 5.166** *Ilustraremos el funcionamiento de* `make-string` *con algunas llamadas tecleadas directamente sobre el intérprete:*

```
[76]> (make-string 5 :initial-element #\a)
"aaaaa"
[77]> (make-string 5 :initial-element #\a :element-type 'base-char)
"aaaaa"
```

**(STRING argumento)**

Convierte cualquier `argumento` de tipo carácter, `nil`, símbolo o cadena de caracteres en una cadena de caracteres.

**Ejemplo 5.167** *Ilustraremos el funcionamiento de* `string` *con algunas llamadas tecleadas directamente sobre el intérprete:*

```
[76]> (string 'a)
"A"
[77]> (string #\a)
"a"
[78]> (string "hola")
"hola"
[79}> (string '())
"NIL"
```

**(STRING-UPCASE cadena [:START inicio] [:END final])**

Convierte, si posible, y devuelve la `cadena` en mayúsculas. Opcionalmente pueden indicarse las posiciones `inicial` y `final` para dicha conversión.

**(NSTRING-UPCASE cadena [:START inicio] [:END final])**

La funcionalidad es idéntica a la de `string-upcase`, pero provoca la modificación física de la `cadena` argumento.

**(STRING-DOWNCASE cadena [:START inicio] [:END final])**

Convierte, si posible, y devuelve la `cadena` en minúsculas. La interpretación de los argumentos opcionales es la misma que en `string-upcase`.

**(NSTRING-DOWNCASE cadena [:START inicio] [:END final])**

La funcionalidad es idéntica a la de `string-downcase`, pero provoca la modificación física de la `cadena` argumento.

**(STRING-CAPITALIZE cadena [:START inicio] [:END final])**

Genera y devuelve una copia de la `cadena` argumento, en la que todas las palabras comienzan por mayúscula y el resto de los caracteres son minúsculos. La interpretación de los argumentos opcionales es la misma que en `string-upcase`.

**(NSTRING-CAPITALIZE cadena [:START inicio] [:END final])**

La funcionalidad es idéntica a la de `string-capitalize`, pero provoca la modificación física de la `cadena` argumento.

**(STRING-TRIM secuencia cadena)**

Devuelve una copia de la eliminación de todos los caracteres presentes en la `secuencia`, del principio y final de la `cadena` argumento.

**(STRING-LEFT-TRIM secuencia cadena)**

Devuelve una copia de la eliminación de todos los caracteres presentes en la `secuencia`, del principio de la `cadena` argumento.

**(STRING-RIGHT-TRIM secuencia cadena)**

Devuelve una copia de la eliminación de todos los caracteres presentes en la `secuencia`, del final de la `cadena` argumento.

**(STRING= cadena cadena [:START1 inicio]) [:END1 final]) [:START1 inicio]) [:END2 final])**

Devuelve `t` si las porciones indicadas de las `cadenas` argumento son idénticas. En otro caso, devuelve `()`. Por defecto los valores de `inicio` y `final` se fijan a las posiciones inicial y final de cada cadena.

**(STRING/= cadena cadena [:START1 inicio]) [:END1 final]) [:START1 inicio]) [:END2 final])**

Devuelve `()` si las porciones indicadas de las `cadenas` argumento son idénticas. En otro caso, devuelve `t`. Por defecto los valores de `inicio` y `final` se fijan a las posiciones inicial y final de cada cadena.

**(STRING< cadena cadena** [**:START1 inicio**])
[**:END1 final**])
[**:START1 inicio**])
[**:END2 final**])
**(STRING> cadena cadena** [**:START1 inicio**])
[**:END1 final**])
[**:START1 inicio**])
[**:END2 final**])
**(STRING<= cadena cadena** [**:START1 inicio**])
[**:END1 final**])
[**:START1 inicio**])
[**:END2 final**])
**(STRING>= cadena cadena** [**:START1 inicio**])
[**:END1 final**])
[**:START1 inicio**])
[**:END2 final**])

Comparan lexicográficamente las dos `cadenas` argumento, siendo el resultado `()` a menos que la primera `cadena` sea menor, mayor, menor o igual, mayor o igual a la segunda `cadena`. Si la condición se satisface, devuelven la primera posición en la que las dos `cadenas` difieren. Por defecto los valores de `inicio` y `final` se fijan a las posiciones inicial y final de cada cadena.

| | |
|---|---|
| **(STRING-EQUAL cadena cadena** | **[:START1 inicio])** |
| | **[:END1 final])** |
| | **[:START1 inicio])** |
| | **[:END2 final])** |
| **(STRING-NOT-EQUAL cadena cadena** | **[:START1 inicio])** |
| | **[:END1 final])** |
| | **[:START1 inicio])** |
| | **[:END2 final])** |
| **(STRING-LESSP cadena cadena** | **[:START1 inicio])** |
| | **[:END1 final])** |
| | **[:START1 inicio])** |
| | **[:END2 final])** |
| **(STRING-NOT-LESSP cadena cadena** | **[:START1 inicio])** |
| | **[:END1 final])** |
| | **[:START1 inicio])** |
| | **[:END2 final])** |
| **(STRING-GREATERP cadena cadena** | **[:START1 inicio])** |
| | **[:END1 final])** |
| | **[:START1 inicio])** |
| | **[:END2 final])** |
| **(STRING-NOT-GREATERP cadena cadena** | **[:START1 inicio])** |
| | **[:END1 final])** |
| | **[:START1 inicio])** |
| | **[:END2 final])** |

Las funcionalidades son ideénticas, respectivamente, a las de `string=`, `string/=`, `string<`, `string>=`, `string>`, `string<=`. La única diferencia estriba en que en este caso se ignoran las distinciones entre mayúsculas y minúsculas en las comparaciones lexicográficas.

**Ejemplo 5.168** *Ilustraremos el funcionamiento de las funciones sobre cadenas de caracteres con una secuencia de llamadas tecleadas directamente sobre el intérprete:*

```
[79]> (setq test "hola Â¿ que tal estas ?")
"hola Â¿ que tal estas ?"
[80]> (string-upcase test)
"HOLA Â¿ QUE TAL ESTAS ?"
[81]> (string-downcase (string-upcase test))
"hola Â¿ que tal estas ?"
[82]> (string-upcase test :start 7 :end 10)
"hola Â¿ QUE tal estas ?"
[83]> (string-upcase test :start 7)
"hola Â¿ QUE TAL ESTAS ?"
[84]> (string-upcase test :end 10)
```

```
"HOLA Â¿ QUE tal estas ?"
[85]> (string-capitalize test)
"Hola Â¿ Que Tal Estas ?"
[86]> test
"hola Â¿ que tal estas ?"
[87]> (nstring-upcase test)
"HOLA Â¿ QUE TAL ESTAS ?"
[88]> test
"HOLA Â¿ QUE TAL ESTAS ?"
[89]> (nstring-capitalize test)
"Hola Â¿ Que Tal Estas ?"
[90]> test
"Hola Â¿ Que Tal Estas ?"
[91]> (string-trim "abc" "abcaakaaakabcaaa")
"kaaak"
[92]> (string-trim '(#\Space #\Tab #\Newline) " estas loco
        ")
"estas loco"
[93]> (string-left-trim "abc" "abcaakaaakabcaaa")
"kaaakabcaaa"
[94]> (string-right-trim "abc" "abcaakaaakabcaaa")
"abcaakaaak"
[95]> (string= "hola" "Hola")
NIL
[96]> (string= "hola" "hola")
T
[97]> (string/= "hola" "hola")
NIL
[98]> (string< "hola" "holas")
4
[99]> (string< "abcd" "holas")
0
[100]> (string> "hola que tal" "hola")
4
[101]> (string>= "hola que tal" "hola")
4
[102]> (string-not-lessp "hola que tal" "hola")
4
[103]> (string-not-lessp "Hola que tal" "hola")
4
[104]> (string>= "Hola que tal" "hola")
NIL
```

**(SUBSTRING cadena inicio [final])**

Devuelve una copia de la subcadena de la `cadena` que comienza y termina en als posiciones indicadas. Si no se indica la posición `final`, por defecto su valor es el de la longitud de la `cadena` menos 1.

**Ejemplo 5.169** *Ilustraremos el funcionamiento de* `substring` *con algunas llamadas tecleadas directamente sobre el intérprete:*

```
[105]> (substring "hola que tal ?" 5 9)
"que "
[106]> (substring "hola que tal ?" 5)
"que tal ?"
```

## 5.13. Funciones sobre tableros

Aunque Lisp no es un lenguaje expresamente concebido para el tratamiento numérico, proporciona una serie de utilidades de base para el manejo de tableros. Técnicamente, hablaremos de *tablero* para referirnos a una colección de objetos accessible por su índice, que debe ser de tipo natural[17]. En COMMON LISP, la representación externa de un tablero es $\#(s_1 s_2 \ldots s_n)$.

**(MAKE-ARRAY dimensiones** [**:ELEMENT-TYPE tipo**]
[**:INITIAL-ELEMENT elemento**]
[**:INITIAL-CONTENTS secuencia**]
[**:ADJUSTABLE {t|()}**]
[**:FILL-POINTER {t|()|longitud}**]
[**:DISPLACED-TO**] **tablero**
[**:DISPLACED-INDEX-OFFSET desplazamiento**])

Genera, y devuelve, un tablero cuyas `dimensiones` han de facilitarse en forma de una lista de números enteros, cada cual inferior al valor `array-dimension-limit`. El producto de tales `dimensiones` debe, además, ser inferior a `array-total-size-limit`.

En relación a los argumentos opcionales, podemos indicar el `tipo` de los elementos del tablero, así como su `elemento` inicial común a todos ellos. Por su parte `:initial-contents` asigna los valores a cada uno de los elementos del tablero a partir de una `secuencia`. En cuanto a `:adjustable`, si su valor es diferente a `()`, permite *a posteriori* alterar la talla del tablero.

La opción `:fill-pointer`, hace referencia al uso de un puntero de relleno para completar el tablero. Si se especifica y es diferente de `()`, implica que el tablero será unidimensional y sus elementos inicializados al `valor` indicado por `:initial-element` hasta la `longitud` indicada. En el caso de usarse `t`

[17] esto es, la numeración de los vectores comienza en cero.

como opción, la longitud será la del tablero, siendo el valor por defecto para este argumento `()`.

Finalmente, la combinación de argumentos `:displaced-to` y `:displaced-index-offset` permiten la compartición de contenidos con el `tablero` indicado, en el que se aplica el `desplazamiento` considerado.

**Ejemplo 5.170** *Ilustraremos el funcionamiento de* `substring` *con algunas llamadas tecleadas directamente sobre el intérprete:*

```
[107]> (make-array '(2 3 4))
#3A(((NIL NIL NIL NIL) (NIL NIL NIL NIL) (NIL NIL NIL NIL))
    ((NIL NIL NIL NIL) (NIL NIL NIL NIL) (NIL NIL NIL NIL)))
[108]> (make-array '(2 3 4) :element-type 'base-char :initial-element #\a)
#3A(("aaaa" "aaaa" "aaaa") ("aaaa" "aaaa" "aaaa"))
[109]> (make-array '(2 3 4) :element-type 'base-char :initial-contents
                  '(("abcd" "efgh" "dsft") ("errw" "dadf" "jhgj")))
#3A(("abcd" "efgh" "dsft") ("errw" "dadf" "jhgj"))
[110]> (make-array 6 :element-type 'character
                     :initial-element #\a
                     :fill-pointer 3)
"aaa"
[111]> (make-array 6 :element-type 'character
                     :initial-element #\a
                     :fill-pointer t)
"aaaaaa"
[112]> (make-array 6 :element-type 'character
                     :initial-element #\a)
"aaaaaa"
[113]> (make-array '(2 3) :element-type 'character
                          :initial-element #\a
                          :fill-pointer '()) ; permite multidimensionalidad
#2A("aaa" "aaa")
[114]> (setq tablero (make-array '(4 3) :initial-contents '((a b c)
                                                            (1 2 3)
                                                            (E F G)
                                                            (4 5 6))))
#2A((A B C) (1 2 3) (E F G) (4 5 6))
[115]> (setq desplazado (make-array 8 :displaced-to tablero
                                      :displaced-index-offset 2))
#(C 1 2 3 E F G 4)
[116]>  (setf (aref desplazado 0) H)
H
[117]> desplazado
```

```
#(H 1 2 3 E F G 4)
[118]> tablero
#2A((A B H) (1 2 3) (E F G) (4 5 6))
```

### (AREF tablero $\mathbf{n}_1 \ldots \mathbf{n}_m$)

Accede, y devuelve, el elemento almacenado en la célula $n_1 \ldots n_m$ del `tablero`.

**Ejemplo 5.171** *Ilustraremos el funcionamiento de* `aref` *con algunas llamadas tecleadas directamente sobre el intérprete:*

```
[119]> (setq tablero #(a b c d))
#(a b c d)
[120]> (aref tablero 2)
C
[121]> (setf (aref tablero 2) H)
H
[122]> tablero
#(A B H D)
```

### (VECTOR objeto$_1$ ... objeto$_m$)

Genera, y devuelve, un tablero unidimensional cuyos elementos se inicializan a la lista de `objetos` proporcionada.

**Ejemplo 5.172** *Ilustraremos el funcionamiento de* `vector` *con algunas llamadas tecleadas directamente sobre el intérprete:*

```
[123]> (vector 1 2 3 4)
#(1 2 3 4)
[124]> (vector #(1 2) #(3 4) #(5 6))
#(#(1 2) #(3 4) #(5 6))
```

### (VECTOR-POP vector)

Decrementa físicamente en una unidad el puntero de relleno de `vector`, si éste existe, devolviendo el valor del nuevo puntero de relleno. Obviamente el argumento debe ser un tablero unidimensional, esto es, un vector.

**Ejemplo 5.173** *Ilustraremos el funcionamiento de* `vector-pop` *con algunas llamadas tecleadas directamente sobre el intérprete:*

```
[125]> (setq tablero (make-array 6 :element-type 'character
                                   :initial-element #\a
```

```
                                    :fill-pointer 3))
"aaa"
[126]> (fill-pointer tablero) ; puntero de relleno original
3
[127]> (vector-pop tablero)
#\a
[128]> (fill-pointer tablero) ; nuevo puntero de relleno
2
[129]> tablero
"aa"
```

### (VECTOR-PUSH elemento vector)

Incrementa físicamente en una unidad el puntero de relleno de `vector`, asignándo el nuevo `elemento` al antiguo puntero. El tablero argumento debe ser unidimensional, esto es, un vector. Devuelve el puntero de relleno inicial.

**Ejemplo 5.174** *Ilustraremos el funcionamiento de* `vector-push` *con algunas llamadas tecleadas directamente sobre el intérprete:*

```
[130]> (setq tablero (make-array 6 :element-type 'character
                                    :initial-element #\a
                                    :fill-pointer 3))
"aaa"
[131]> (fill-pointer tablero) ; puntero de relleno original
3
[132]> (vector-push #\b tablero)
3
[133]> (fill-pointer tablero) ; nuevo puntero de relleno
4
[134]> tablero
"aaab"
```

### (VECTOR-PUSH-EXTEND elemento vector [desplazamiento])

La funcionalidad es análoga a la de `vector-push`, salvo que aquí si el puntero de relleno no fuese suficiente para albergar al nuevo `elemento`, el `vector` será ajustado por extensión en un valor mínimo igual al `desplazamiento` indicado. Evidentemente, el `vector` debe haber sido declarado como ajustable.

**Ejemplo 5.175** *Ilustraremos el funcionamiento de* `vector-push-extend` *con algunas llamadas tecleadas directamente sobre el intérprete:*

```
[135]> (setq tablero (make-array 4 :element-type 'character
                                   :initial-element #\a
                                   :adjustable t
                                   :fill-pointer 3))
"aaa"
[136]> (fill-pointer tablero)             ; puntero de relleno original
3
[137]> (vector-push-extend #\b tablero 1) ; aun no sobrepasamos talla original
3
[138]> (fill-pointer tablero)             ; nuevo puntero de relleno
4
[139]> (vector-push-extend #\c tablero 1) ; sobrepasamos talla original
4
[140]> (fill-pointer tablero)             ; nuevo puntero de relleno
5
[139]> tablero
"aaabc"
```

**(SVREF vector posición)**

Accede, y devuelve, la celda en la `posición` indicada en el vector simple `vector`.

**Ejemplo 5.176** *Ilustraremos el funcionamiento de* `svref` *con algunas llamadas tecleadas directamente sobre el intérprete:*

```
[125]> (setq vector (vector 'a 'b 'c 'd))
#(A B C D)
[126]> (svref vector 2)
C
[127]> (setf (svref vector 2) 'H)
H
[128]> vector
#(A B H D)
```

**(ARRAY-RANK tablero)**

Devuelve el número de dimensiones del `tablero`.

**Ejemplo 5.177** *Ilustraremos el funcionamiento de* `array-rank` *con algunas llamadas tecleadas directamente sobre el intérprete:*

```
[129]> (array-rank (vector #(1 2 3) #(4 5 6) #(7 8 9)))
1
[130]> (array-rank (vector 1 2 3))
```

```
1
[131]> (array-rank (make-array '(4 3) :initial-contents '((a b c)
                                                          (1 2 3)
                                                          (E F G)
                                                          (4 5 6))))
2
```

**(ARRAY-DIMENSION tablero número)**

Devuelve la dimensión indicada por `número` en el `tablero`.

**Ejemplo 5.178** *Ilustraremos el funcionamiento de* `array-dimension` *con algunas llamadas tecleadas directamente sobre el intérprete:*

```
[132]> (array-dimension (vector #(1 2 3) #(4 5 6) #(7 8 9)) 0)
3
[133]> (array-dimension (vector 'a 'b 'c 'd) 0)
4
[134]> (array-dimension (make-array '(4 3) :initial-contents '((a b c)
                                                               (1 2 3)
                                                               (E F G)
                                                               (4 5 6)))
                        0)
4
[135]> (array-dimension (make-array '(4 3) :initial-contents '((a b c)
                                                               (1 2 3)
                                                               (E F G)
                                                               (4 5 6)))
                        1)
3
```

**(ARRAY-DIMENSIONS tablero)**

Devuelve una lista con las dimensiones del `tablero`.

**Ejemplo 5.179** *Ilustraremos el funcionamiento de* `array-dimensions` *con algunas llamadas tecleadas directamente sobre el intérprete:*

```
[136]> (array-dimensions (vector #(1 2 3) #(4 5 6) #(7 8 9)))
(3)
[137]> (array-dimensions (vector 'a 'b 'c 'd))
(4)
[138]> (array-dimensions (make-array '(4 3) :initial-contents '((a b c)
                                                                (1 2 3)
                                                                (E F G)
```

```
                                                          (4 5 6))))
(4 3)
```

### (ARRAY-TOTAL-SIZE tablero)

Devuelve el número total de elementos en el `tablero`.

**Ejemplo 5.180** *Ilustraremos el funcionamiento de* `array-total-size` *con algunas llamadas tecleadas directamente sobre el intérprete:*

```
[139]> (array-total-size (vector #(1 2 3) #(4 5 6) #(7 8 9)))
3
[140]> (array-total-size (vector 'a 'b 'c 'd))
4
[141]> (array-total-size (make-array '(4 3) :initial-contents '((a b c)
                                                                (1 2 3)
                                                                (E F G)
                                                                (4 5 6))))
12
```

### (ARRAY-IN-BOUNDS-P tablero $\mathbf{n}_1 \ldots \mathbf{n}_m$)

Predicado que devuelve `t` si los índices indicados están dentro de los límites del `tablero`. En otro caso devuelve `()`.

**Ejemplo 5.181** *Ilustraremos el funcionamiento de* `array-in-bounds-p` *con algunas llamadas tecleadas directamente sobre el intérprete:*

```
[142]> (array-in-bounds-p (vector #(1 2 3) #(4 5 6) #(7 8 9)) 0)
T
[143]> (array-in-bounds-p (vector 'a 'b 'c 'd) 4)
NIL
[144]> (array-in-bounds-p (make-array '(4 3) :initial-contents '((a b c)
                                                                 (1 2 3)
                                                                 (E F G)
                                                                 (4 5 6)))
                          2 3)
NIL
[145]> (array-in-bounds-p (make-array '(4 3) :initial-contents '((a b c)
                                                                 (1 2 3)
                                                                 (E F G)
                                                                 (4 5 6)))
                          2 2)
T
```

**(ARRAY-ROW-MAJOR-INDEX tablero $\mathbf{n}_1 \ldots \mathbf{n}_m$)**

Devuelve, en la ordenación por filas[18], el orden del elemento del `tablero` en la posición indicada.

**Ejemplo 5.182** *Ilustraremos el funcionamiento de* `array-row-major-index` *con algunas llamadas tecleadas directamente sobre el intérprete:*

```
[146]> (setq tablero (make-array '(4 3) :initial-contents '((a b c)
                                                            (1 2 3)
                                                            (E F G)
                                                            (4 5 6))))
#2A((A B C) (1 2 3) (E F G) (4 5 6))
[147]> (array-row-major-index tablero 2 2)  ; elemento "G"
8
[148]> (array-row-major-index tablero 3 2)  ; elemento "6"
11
```

**(ROW-MAJOR-AREF tablero posición)**

Devuelve, en la ordenación por filas[19], el elemento del `tablero` en la `posición` indicada.

**Ejemplo 5.183** *Ilustraremos el funcionamiento de* `row-major-aref` *con algunas llamadas tecleadas directamente sobre el intérprete:*

```
[149]> (setq tablero (make-array '(4 3) :initial-contents '((a b c)
                                                            (1 2 3)
                                                            (E F G)
                                                            (4 5 6))))
#2A((A B C) (1 2 3) (E F G) (4 5 6))
[150]> (row-major-aref tablero 8)   ; elemento "G"
G
[151]> (row-major-aref tablero 11)  ; elemento "6"
6
[152]> (setf (row-major-aref tablero 11) 7)
7
[153]> tablero
#2A((A B C) (1 2 3) (E F G) (4 5 7))
```

**(ADJUSTABLE-ARRAY-P tablero)**

[18] comenzando la numeración por 0.
[19] comenzando la numeración por 1.

Predicado que devuelve `t` si el `tablero` argumento es ajustable, sino devuelve `()`.

**Ejemplo 5.184** *Ilustraremos el funcionamiento de* `adjustable-array-p` *con algunas llamadas tecleadas directamente sobre el intérprete:*

```
[154]> (setq tablero1 (make-array '(4 3) :initial-contents '((a b c)
                                                             (1 2 3)
                                                             (E F G)
                                                             (4 5 6))))
#2A((A B C) (1 2 3) (E F G) (4 5 6))
[155]> (adjustable-array-p tablero1)
NIL
[156]> (setq tablero2 (make-array '(4 3) :initial-contents '((a b c)
                                                             (1 2 3)
                                                             (E F G)
                                                             (4 5 6))
                                         :adjustable t))
#2A((A B C) (1 2 3) (E F G) (4 5 6)
[157]> (adjustable-array-p tablero2)
T
```

### (ARRAY-HAS-FILL-POINTER-P tablero)

Predicado que devuelve `t` si el `tablero` argumento posee punteros de relleno. En otro caso, devuelve `()`.

**Ejemplo 5.185** *Ilustraremos el funcionamiento de* `array-has-fill-pointer-p` *con algunas llamadas tecleadas directamente sobre el intérprete:*

```
[154]> (setq tablero1 (make-array 6 :element-type 'character
                                    :initial-element #\a
                                    :fill-pointer 3))
"aaa"
[155]> (array-has-fill-pointer-p tablero1)
T
[156]> (setq tablero2 (make-array 6 :element-type 'character
                                    :initial-element #\a
                                    :fill-pointer t))
"aaaaaa"
[157]> (array-has-fill-pointer-p tablero2)
T
[158]> (setq tablero3 (make-array 6 :element-type 'character
                                    :initial-element #\a))
```

```
"aaaaaa"
[159]>  (array-has-fill-pointer-p tablero3)
NIL
```

**(ADJUST-ARRAY tablero dimensiones**
**[:ELEMENT-TYPE tipo]**
**[:INITIAL-ELEMENT elemento]**
**[:INITIAL-CONTENTS secuencia]**
**[:FILL-POINTER {t|()|valor}]**
**[:DISPLACED-TO tablero]**
**[:DISPLACED-INDEX-OFFSET desplazamiento])**

Devuelve, y modifica físicamente, el `tablero`, que debe ser ajustable, redimensionado a las `dimensiones` establecidas. El significado de los argumentos opcionales es el mismo que en `make-array`.

**Ejemplo 5.186** *Ilustraremos el funcionamiento de* `adjust-array` *con algunas llamadas tecleadas directamente sobre el intérprete:*

```
[158]> (setq tablero (make-array '(4 3) :initial-contents '((a b c)
                                                            (1 2 3)
                                                            (E F G)
                                                            (4 5 6))
                                        :adjustable t))
#2A((A B C) (1 2 3) (E F G) (4 5 6)
[159]> (adjust-array tablero '(2 2))
#2A((A B) (1 2))
[160]> tablero
#2A((A B) (1 2))
[161]> (adjust-array tablero '(5 5) :initial-element 'a)
#2A((A B A A A) (1 2 A A A) (A A A A A) (A A A A A) (A A A A A))
[162]> tablero
#2A((A B A A A) (1 2 A A A) (A A A A A) (A A A A A) (A A A A A))
[163]> (adjust-array tablero '(2 2 3))

*** - ADJUST-ARRAY: no es posible cambiar el rango 2 de la matriz
       #2A((A B A A A)
           (1 2 A A A)
           (A A A A A)
           (A A A A A)
           (A A A A A))
      : (2 2 3)
Es posible continuar en los siguientes puntos:
ABORT          :R1      Abort debug loop
ABORT          :R2      Abort debug loop
```

```
ABORT           :R3      Abort debug loop
ABORT           :R4      Abort debug loop
ABORT           :R5      Abort debug loop
ABORT           :R6      Abort debug loop
ABORT           :R7      Abort debug loop
ABORT           :R8      Abort debug loop
ABORT           :R9      Abort main loop
```

**(ARRAY-DISPLACEMENT tablero)**

En el caso de que el argumento sea un `tablero` desplazado, devuelve los valores de las correspondientes opciones `:displaced-to` y `:displaced-index-offset`. En otro caso, devuelve `()`.

**Ejemplo 5.187** *Ilustraremos el funcionamiento de* `array-displacement` *con algunas llamadas tecleadas directamente sobre el intérprete:*

```
[164]> (setq tablero (make-array '(4 3) :initial-contents '((a b c)
                                                            (1 2 3)
                                                            (E F G)
                                                            (4 5 6))))
#2A((A B C) (1 2 3) (E F G) (4 5 6))
[165]> (setq desplazado (make-array 8 :displaced-to tablero
                                      :displaced-index-offset 2))
#(C 1 2 3 E F G 4)
[166]> (array-displacement desplazado)
#2A((A B C) (1 2 3) (E F G) (4 5 6)) ;
2
```

## 5.14. Funciones aritméticas

La mayoría de los dialectos Lisp distinguen al menos tres tipos de aritmética:

- Aritmética genérica.
- Aritmética entera.
- Aritmética de punto flotante.

Sin embargo, nosotros sólo abordaremos aquí la artimética genérica, con objeto de reducir la talla del curso.

### 5.14.1. Conversiones de tipo

**(FLOAT forma [prototipo])**

Convierte el valor de la `forma` argumento, que debe ser un número real, en flotante. El argumento opcional debe ser un número, e indica el formato al que la función realizará la conversión de su primer argumento.

**Ejemplo 5.188** *Ilustraremos el funcionamiento de* `float` *con algunas llamadas tecleadas directamente sobre el intérprete:*

```
[167]> (float 3)
3.0
[168]> (float 3/4)
0.75
[169]> (float 3/4 1.0d0)
0.75d0
```

**(RATIONAL forma)**

Convierte el valor de la `forma` argumento, que debe ser un número real, en racional. El argumento opcional debe ser un número, e indica el formato al que la función realizará la conversión de su primer argumento. Considera que el valor del argumento es completamente exacto.

**Ejemplo 5.189** *Ilustraremos el funcionamiento de* `rational` *con algunas llamadas tecleadas directamente sobre el intérprete:*

```
[170]> (rational 3)
3
[171]> (rational 3/4)
3/4
[172]> (rational 3.1)
6501171/2097152
```

**(RATIONALIZE forma)**

La funcionalidad es idéntica a la de `rational`, salvo que aquí elintérprete estima que el valor del argumento como número real es exacto sólo en la precisión de la repesentación, pudiendo devolver cualquier número considerado la mejor aproximación posible del argumento.

**Ejemplo 5.190** *Ilustraremos el funcionamiento de* `rationalize` *con algunas llamadas tecleadas directamente sobre el intérprete:*

```
[173]> (rationalize 3)
3
[174]> (rationalize 3/4)
3/4
[175]> (rationalize 3.1)
31/10
[176]> (rational 3.1)
6501171/2097152
[177]> (rationalize 3.15)
63/20
[178]> (rational 3.15)
6606029/2097152
```

**(NUMERATOR forma)**

**(DENOMINATOR forma)**

Devuelven el numerador (resp. denominador) del resultado de la evaluación de la `forma` argumento, que debe ser un número racional. El denominador siempre es un entero positivo.

**Ejemplo 5.191** *Ilustraremos el funcionamiento de estas funciones con algunas llamadas tecleadas directamente sobre el intérprete:*

```
[179]> (denominator 3/4)
4
[180]> (numerator 3/4)
3
[181]> (numerator -3)
-3
[182]> (denominator 3)
1
```

**(FLOOR forma [divisor])**

**(CEILING forma [divisor])**

**(TRUNCATE forma [divisor])**

**(ROUND forma [divisor])**

En su forma unaria más simple, todas ellas transforman el valor de `forma` (que debe ser un número) a un entero. La diferencia estriba en el algoritmo usado para ello. Así:

- `floor` utiliza un truncamiento hacia el infinito positivo. Esto es, devuelve el mayor entero que sea inferior al argumento.
- `ceiling` utiliza un truncamiento hacia el infinito negativo. Esto es, devuelve el menor entero que sea superior al argumento.
- `truncate` utiliza un truncamiento hacia el cero. Esto es, devuelve un entero del mismo signo que el argumento y cuyo valor absoluto es el mayor de los menores posibles en relación al del argumento.
- `round` hace un redondeo hacia el entero más próximo, sea negativo o positivo.

Como segundo valor, devuelven la diferencia de la respuesta primera en relación al argumento.
En caso de usarse el argumento opcional, éste actúa como `divisor` del argumento antes de aplicar las funcionalidades ya comentadas.

**Ejemplo 5.192** *Ilustraremos el funcionamiento de estas funciones con algunas llamadas tecleadas directamente sobre el intérprete:*

```
[183]> (floor -3.567)
-4 ;
0.4330001
[184]> (ceiling -3.567)
-3 ;
-0.5669999
[185]> (truncate -3.567)
-3 ;
-0.5669999
[186]> (round -3.567)
-4 ;
0.4330001
[187]> (floor -3.567 2)
-2 ;
0.4330001
[188]> (ceiling -3.567 3)
-1 ;
-0.56700003
[189]> (truncate -3.567 4)
0 ;
-3.567
[190]> (round -3.567 5)
-1 ;
1.433
```

**(FFLOOR forma [divisor])**

**(FCEILING forma [divisor])**

**(FTRUNCATE forma [divisor])**

**(FROUND forma [divisor])**

El funcionamiento es el mismo que el de `floor`, `ceiling`, `truncate` y `fround`; pero en este caso el truncamiento se expresa siempre en forma de flotante.

**Ejemplo 5.193** *Ilustraremos el funcionamiento de estas funciones con algunas llamadas tecleadas directamente sobre el intérprete:*

```
[191]> (round -7 2)
-4 ;
1
[192]> (fround -7 2)
-4.0 ;
1
```

**(MOD forma divisor)**

**(REM forma divisor)**

Devuelven el valor de la `forma` módulo el `divisor`, y del resto de dividir el valor de `forma` entre el `divisor`; respectivamente.

**Ejemplo 5.194** *Ilustraremos el funcionamiento de estas funciones con algunas llamadas tecleadas directamente sobre el intérprete:*

```
[193]> (mod 13 4)
1
[194]> (rem 13 4)
1
[195]> (mod -13 5)
2
[196]> (rem -13 5)
-3
```

**(COMPLEX real [compleja])**

Devuelve un número complejo, con las partes `real` e `imaginaria` indicadas.

**Ejemplo 5.195** *Ilustraremos el funcionamiento de* `complex` *con algunas llamadas tecleadas directamente sobre el intérprete:*

```
[197]> (complex 1)
1
[198]> (complex 1 2)
#C(1 2)
```

**(CONJUGATE forma)**

Devuelve el conjugado del valor de `forma`, que debe ser un número complejo.

**Ejemplo 5.196** *Ilustraremos el funcionamiento de* `conjugate` *con algunas llamadas tecleadas directamente sobre el intérprete:*

```
[197]> (conjugate (complex 3 4))
#C(3 -4)
```

**(REALPART forma)**

**(IMAGPART forma)**

Devuelve las partes real e imaginaria del valor de `forma`, que debe ser un número complejo.

**Ejemplo 5.197** *Ilustraremos el funcionamiento de* `realpart` *e* `imagpart` *con algunas llamadas tecleadas directamente sobre el intérprete:*

```
[199]> (realpart (complex 1 2))
1
[200]> (imagpart (complex 1 2))
2
```

**(RANDOM forma [estado])**

Genera y devuelve un valor aleatorio a partir del valor (numérico) de `forma`. El argumento opcional `estado` indica un punto de partida para el cálculo del número aleatorio. Dicho punto de partido viene dado por la constante `*random-state*`.

**Ejemplo 5.198** *Ilustraremos el funcionamiento de* `random` *con algunas llamadas tecleadas directamente sobre el intérprete:*

```
[201]> (random 2)
1
[202]> (random 3.4)
1.9287555
[203]> (random 3.4)
2.7538328
[204]> (random 3.4 (make-random-state))
2.2486277
```

**(MAKE-RANDOM-STATE [estado])**

Genera y devuelve un nuevo estado de tipo `random-state`. Si el argumento opcional es `()`, devuelve una copia del actual `random state`. Si se trata de estado, devuelve una copia del mismo. Si el argumento opcional es `t`, entonces devuelve un nuevo estado. Por defecto, el argumento opcional es `()`.

**Ejemplo 5.199** *Ilustraremos el funcionamiento de* `make-random-state` *con algunas llamadas tecleadas directamente sobre el intérprete:*

```
[205]> *random-state*
#S(RANDOM-STATE
   #*1100111101100001110110001110110110001010111111101111000001011000)
[206]> (make-random-state t)
#S(RANDOM-STATE
   #*0110011101101111010001011000101110000100010100011001111010001001)
[207]> (make-random-state t)
#S(RANDOM-STATE
   #*0110011111001110001000110111101111111100001010010100000110001001)
[208]> *random-state*
#S(RANDOM-STATE
   #*1100111101100001110110001110110110001010111111101111000001011000)
[209]> (make-random-state)
#S(RANDOM-STATE
   #*1100111101100001110110001110110110001010111111101111000001011000)
[210]> (make-random-state *random-state*)
#S(RANDOM-STATE
   #*1100111101100001110110001110110110001010111111101111000001011000)
```

**(RANDOM-STATE-P forma)**

Predicado que devuelve `t` siel resultado de la evaluación de la `forma` es un `random-state`. En otro caso devuelve `()`.

**Ejemplo 5.200** *Ilustraremos el funcionamiento de* `random-state-p` *con algunas llamadas tecleadas directamente sobre el intérprete:*

```
[211]> (random-state-p *random-state*)
T
[212]> (random-state-p (make-random-state *random-state*))
T
```

### 5.14.2. Funciones de la aritmética genérica

Implementan las operaciones aritméticas más usuales.

**(+ $\mathbf{n}_1 \ldots \mathbf{n}_m$)**

Devuelve el resultado de la suma de los números $n_1 \ldots n_m$.

**(- $\mathbf{n}_1 \ldots \mathbf{n}_m$)**

Devuelve el resultado de la resta $n_1 - n_2 - \ldots - n_m$.

**(1+ n)**

Devuelve el valor de `n + 1`.

**(1- n)**

Devuelve el valor de `n - 1`.

**(INCF forma [forma])**

Devuelve, y modifica físicamente, el valor del puntero resultado de evaluar la primera `forma` argumento; incrementandola en el valor de la segunda si ésta se facilita. En otro caso el incremento es de 1.

**(DECF forma [forma])**

Devuelve, y modifica físicamente, el valor del puntero resultado de evaluar la primera `forma` argumento; decrementandola en el valor de la segunda si ésta se facilita. En otro caso el decremento es de 1.

**Ejemplo 5.201** *Ilustraremos el funcionamiento de* `incf` *y* `decf` *con algunas llamadas tecleadas directamente sobre el intérprete:*

```
[213]>  (setq n 0)
0
[214]> (incf n)
1
```

```
[215]> (incf n)
2
[216]> (incf n 3)
5
[217]> (decf n 3)
2
[218]> n
2
```

**(GCD $\mathbf{n}_1 \ldots \mathbf{n}_m$)**

Devuelve el mayor común divisor de sus argumentos.

**(LCM $\mathbf{n}_1 \ldots \mathbf{n}_m$)**

Devuelve el menor común múltiplo de sus argumentos.

**(ABS n)**

Devuelve el valor absoluto de su argumento.

**($*$ $\mathbf{n}_1 \ldots \mathbf{n}_m$)**

Devuelve el resultado de la multiplicación de los números $n_1 \ldots n_m$.

**(/ $\mathbf{n}_1 \ldots \mathbf{n}_m$)**

Devuelve el resultado de la división de $\mathbf{n}_1$ por $\mathbf{n}_2$, luego por $\mathbf{n}_3$ ...

**(MAX $\mathbf{n}_1 \ldots \mathbf{n}_m$)**

Devuelve el máximo de sus argumentos.

**(MIN $\mathbf{n}_1 \ldots \mathbf{n}_m$)**

Devuelve el mínimo de sus argumentos.

**(EXP n)**

Devuelve el valor de $e^n$.

**(EXPT base exponente)**

Devuelve el valor de $\texttt{base}^{\texttt{exponente}}$.

**(LOG n [base])**

Devuelve el valor de $\texttt{log}_{\texttt{base}}\texttt{n}$. Por defectola `base` es `e`.

**(SQRT n)**

Devuelve el valor de la raíz cuadrada de `n`.

**(ISQRT n)**

Devuelve el valor de la raíz cuadrada entera de `n`, siendo `n` un número entero.

**(SIGNUM n)**

Devuelve -1, 0 ó 1 según el argumento sea negativo, cero o positivo.

**(SIN n)** **(COS n)** **(TAN n)** **(ASIN n)** **(ACOS n)** **(ATAN n [m])**
**(SINH n)** **(COSH n)** **(TANH n)** **(ASINH n)** **(ACOSH n)** **(ATANH n [m])**

Implementan las funciones trigonométricas clásicas y de las hiperbólicas. En caso de facilitarse el segundo argumento, la función `atan` devuelve el valor `atan n/m`.

### 5.14.3. Predicados de la aritmética genérica

Implementan los predicados clásicos de igualdad y desigualdad, sobre un número finito de argumentos.

**($= \mathbf{n}_1 \ldots \mathbf{n}_m$)**

Devuelve `t` si los números $\mathbf{n}_1 \ldots \mathbf{n}_m$ son iguales. Sino devuelve ().

**($>= \mathbf{n}_1 \ldots \mathbf{n}_m$)**

Devuelve `t` si $\mathbf{n}_1$ es mayor o igual que $\mathbf{n}_2$ que es mayor o igual que $\mathbf{n}_3 \ldots$. Sino devuelve `()`.

**($> \mathbf{n}_1 \ldots \mathbf{n}_m$)**

Devuelve `t` si $\mathbf{n}_1$ es mayor que $\mathbf{n}_2$ que es mayor que $\mathbf{n}_3 \ldots$. Sino devuelve `()`.

**($<= \mathbf{n}_1 \ldots \mathbf{n}_m$)**

Devuelve `t` si $\mathbf{n}_1$ es menor o igual que $\mathbf{n}_2$ que es menor o igual que $\mathbf{n}_3 \ldots$.

Sino devuelve `()`.

$(< \mathbf{n}_1 \ldots \mathbf{n}_m)$

Devuelve `t` si $\mathbf{n}_1$ es menor que $\mathbf{n}_2$ que es menor que $\mathbf{n}_3 \ldots$. Sino devuelve `()`.

## 5.15. Funciones del sistema

### 5.15.1. Gestión del tiempo.

El intérprete COMMON LISP representa el tiempo de tres formas diferentes: *tiempo decodificado*, *tiempo universal* y *tiempo interno*. Las dos primeras representaciones se usan para representar el tiempo asociado al calendario y su precisión tiene una holgura de 1 segundo, mientras que la tercera se asocia a la medida del tiempo de computación del propio intérprete. En este último caso la precisión es de una fracción de segundo y viene especificada por la constante del sistema `internal-time-units-per-second`. El tiempo decodificado sólo se usa para indicaciones absolutas de tiempo, mientras que las otras dos también pueden ser relativas.

**(GET-DECODED-TIME)**

Devuelve la hora en formato decodificado: segundos, minutos, hora, día, mes, año, día de la semana[20], un booleano indicando si el horario de verano está activo y la zona horaria[21].

**Ejemplo 5.202** *Ilustraremos el funcionamiento de* `get-decoded-time` *con un ejemplo en interactivo:*

```
[55]> (get-decoded-time)
50 ;
44 ;
15 ;
25 ;
9 ;
2012 ;
1 ;
T ;
-1
```

[20] el 0 es el lunes y el 6 es el domingo.

[21] indicada por el decalage horario medido en horas hacia el oeste de Greenwich.

**(GET-UNIVERSAL-TIME)**

Devuelve la hora en formato universal, mediante un número entero.

**Ejemplo 5.203** *Ilustraremos el funcionamiento de* `get-universal-time` *con un ejemplo en interactivo:*

```
[56]> (get-universal-time)
3557570505
```

**(DECODE-UNIVERSAL-TIME hora-universal [zona-horaria])**

Devuelve la hora en formato decodificado la hora universal. Si no se especifica la `zona horaria`, toma por defecto la actual del sistema.

**Ejemplo 5.204** *Ilustraremos su funcionamiento con un ejemplo en interactivo:*

```
[57]> (decode-universal-time (get-universal-time))
46 ;
5 ;
16 ;
25 ;
9 ;
2012 ;
1 ;
T ;
-1
[58]> (decode-universal-time (get-universal-time) 3)
15 ;
6 ;
11 ;
25 ;
9 ;
2012 ;
1 ;
NIL ;
3
```

**(ENCODE-UNIVERSAL-TIME segundos minutos horas día mes año [zona-horaria])**

Codifica la hora universal a partir de los argumentos.

**Ejemplo 5.205** *Ilustraremos su funcionamiento con un ejemplo en interactivo:*

```
[59]> (encode-universal-time 37 22 16 25 9 2012 -1)
3557575357
```

### (GET-INTERNAL-RUN-TIME)

Devuelve el tiempo de ejecución en formato interno, mediante un número entero. Es dependiente de la implementación del intérprete y puede depender de los ciclos COPU, de la hora real o de cualquier otra magnitud.

**Ejemplo 5.206** *Ilustraremos su funcionamiento con un ejemplo en interactivo:*

```
[57]> (get-internal-run-time)
20000
```

### (GET-INTERNAL-REAL-TIME)

Devuelve la hora en formato interno, mediante un número entero. Es dependiente de la implementación del intérprete.

**Ejemplo 5.207** *Ilustraremos su funcionamiento con un ejemplo en interactivo:*

```
[58]> (get-internal-real-time)
1348584288974346
```

### (SLEEP n)

Desactiva el intérprete durante `n` segundos. Devuelve `()`.

**Ejemplo 5.208** *Ilustraremos el funcionamiento de* `sleep` *con un ejemplo en interactivo:*

```
[59]> (sleep 5)
NIL
```

### (TIME forma)

Devuelve el tiempo CPU, en segundos, usado por el intérprete para evaluar la `forma` que sirve de argumento.

**Ejemplo 5.209** *Dadas las funciones siguientes, que cargaremos en el intérprete,*

```
(defmacro subp-1 (x y)
     `(+ (* 2 ,x) (* 3 ,y)))

(defun subp-2 (x y)
       (+ (* 2 x) (* 3 y)))
```

```
(defun main-1 (x)
       (subp-1 x 3))

(defun main-2 (x)
       (subp-2 x 3))
```

*podemos considerar las llamadas interactivas:*

```
[60]> (time (loop for l from 1 to 100000 do (main-1 4)))
Real time: 0.176827 sec.
Run time: 0.17 sec.
Space: 9120 Bytes
NIL
[61]> (time (loop for l from 1 to 100000 do (main-2 4)))
Real time: 0.20292 sec.
Run time: 0.2 sec.
Space: 9120 Bytes
NIL
```

### 5.15.2. El recogedor de la basura

Las zonas de la memoría del sistema que contienen objetos Lisp, se gestionan de forma dinámica. Ello quiere decir que cuando una de dichas zonas está saturada, el *recogedor de la basura* o *garbage collector*, es llamado de forma automática por el sistema con el objeto de liberar y dejar disponibles aquellas partes ocupadas por objetos no referenciados por ninguna variable del programa actualmente en memoria.

#### (GC)

Llama explícitamente al *recogedor de la basura*, liberando las zonas de memoria en desuso.

**Ejemplo 5.210** *Ilustraremos el funcionamiento de* `gc` *con un ejemplo en interactivo:*

```
[62]> (gc)
5610616 ;
1402654 ;
190288 ;
2 ;
379320 ;
50000
```

No es una función estandard de Lisp. En algunos casos, el recogedor de la basura actúa de forma continuada, sin interrumpir la interpretación. En otros casos no es así, y ésta se interrumpe.

# Capítulo 6

# Entradas y salidas

En Lisp, las operaciones básicas de entrada/salida, se realizan a través de *canales*[1] secuenciales, a nivel de caracteres, líneas o formas. En este sentido, una línea en Lisp, viene delimitada por los caracteres:

```
\#CR
\#LF
```

Los canales están asociados, bien a ficheros, bien a terminales y sus valores se almacenan en las siguientes variables:

| | | |
|---|---|---|
| ∗standard-input∗ | ∗standard-output∗ | ∗error-output∗ |
| ∗query-io∗ | ∗debug-io∗ | ∗terminal-io∗ |
| ∗trace-output∗ | | |

Cuando un fichero se abre, automáticamente se un canal que sirve de nexo de comunicación de aquel con el entorno Lisp, de modo que las operaciones sobre el ficherose reflejan en operaciones sober el canal asociado. En este contexto, la acción de *cerrar* el canal rompe esa asociación dando por terminadas las transacciones sobre el fichero.

**(OPEN fichero** [**:DIRECTION dirección**]
[**:ELEMENT-TYPE tipo**]
[**:IF-EXISTS acción**]
[**:IF-DOES-NOT-EXIST acción**]
[**:EXTERNAL-FORMAT esquema**])

Devuelve un canal asociado a la apertura del `fichero`, donde la interpretación de los argumentos es la siguiente:

- El parámetro del argumento `:direction` expresa la `dirección` del flujo de datos en el canal. Puede tomar los valores `:input`, `:output`, `:io` y `probe` según el flujo sea de entrada, salida, ambos o sin dirección.

[1] en terminología anglosajona, *sreams*.

En este último caso el canal se abre para cerrarse de forma inmediata, siendo su utilidad fundamental la de verificar si un fichero existe. Por defecto el valor es `:input`.

- El parámetro del argumento `:element-type` especifica el `tipo` de la unidad de transacción para el canal. Puede tomar los siguientes valores, cuya interpretación detallada puede verse en el Manual de Referencia de Common Lisp: `character`, `base-character`, `(unsigned-byte` *n*`)`[2], `unsigned-byte`, `(signed-byte` *n*`)`[3], `signed-byte`, `bit`, `(mod` *n*`)`[4], `:default`[5]. Por defecto el valor es `character`.

- El parámetro del argumento `:if-exists` especifica la `acción` a aplicar si la `dirección` es `:output` o `io`, y además el fichero existe. Si la `dirección` es `:input` o `:probe`, el parámetro es simplemente ignorado. Los posibles valores del parámetro son los siguientes:

  - Si el valor es `:error`, se está señalando un error. Es el valor por defecto si la versión del `fichero` especificado no está calificada como `:newest`.
  - Si el valor es `:new-version`, crea un nuevo fichero con el mismo nombre del indicado en argumento, pero con un número de versión más largo. Es el valor por defecto si la versión del fichero está calificada como `:newest`.
  - Si el valor es `:rename`, renombra el `fichero` existente y crea uno nuevo con el nombre asociado a aquel.
  - Si el valor es `:rename-and-delete`, renombra el `fichero` existente y borra el original. Luego crea uno nuevo con el asociado a éste último.
  - Si el valor es `:overwrite`, usa el `fichero` actual modificándolo de forma destructiva. El puntero de lectura/escritura es inicialmente colocado al inicio de dicho `fichero`.
  - Si el valor es `:append`, usa el `fichero` actual situando el puntero de lectura/escritura al final del mismo.
  - Si el valor es `:supersede`, reemplaza el actual fichero. La diferencia con `:new-version` es que aquel generaba un fichero del mismo nombre pero con un número de versión más largo, mientras que aquí no es así.
  - Si el valor es `nil`, no se crea ningún fichero ni canal, devolviendo `nil` para indicar fallo.

[2] el argumento indica la talla.
[3] el argumento indica la talla.
[4] un número no negativo menor que el argumento.
[5] el sistema lo asigna de modo automático.

- El parámetro del argumento `:if-does-not-exist` especifica la `acción` a realizar en caso de que el fichero con el `nombre` especificado no exista. Los posibles valores del parámetro son los siguientes:

  - Si el valor es `:error`, se está señalando un error. Es el valor por defecto si la `dirección` es `:input`, o si el argumento `:if-exists` es `:overwrite` o `append`.
  - Si el valor es `:create`, genera un nuevo fichero con el nombre del argumento para luego proceder sobre él como si ya hubiera existido. Es el valor por defecto si la `dirección` es `:output` o `io`, y el valor del argumento `:if-exists` no es `overwrite` ni `append`.
  - Si el valor es `nil`, no se crea ningún fichero ni canal, devolviendo `nil` para indicar fallo. Es el valor por defecto si la `dirección` es `:probe`.

- El parámetro del argumento `:external-format` especifica un `esquema` de representacion para representar los caracteres en los ficheros. Por defecto toma el valor `:default`, lo que debería dar soporte a `base-character`. Este argumento puede ser especificado si la `dirección` es `:input`, `:output` o `:io`.

Un vez finalizadas las operaciones, y a diferencia de `with-open-file`, debe llamarse explícitamente a la función `close` para cerrar el canal abierto.

**Ejemplo 6.1** *Para ilustrar el funcionamiento de* `open`*, podemos teclear en interactivo algunas expresiones simples:*

```
[3]> (open 'fichero :direction :probe) ; veo si fichero existe (no es asi)
NIL
[4]> (setq canal (open 'fichero :if-does-not-exist :create)) ; creo/abro fichero
#<INPUT BUFFERED FILE-STREAM CHARACTER #P"fichero" @1>
[5]> (close canal)                                          ; cierro canal
T
[6]> (open 'fichero :direction :probe) ; veo si fichero existe (ahora si)
#<CLOSED INVALID BUFFERED FILE-STREAM CHARACTER #P"fichero">
[7]> (setq canal (open 'fichero :direction :output))
#<IO BUFFERED FILE-STREAM CHARACTER #P"fichero" @1>
[8]> (format canal "Escribo un par ~%de
                    lineas de prueba~%")
NIL
[9]> (do ((linea (read-line canal)
          (read-line canal nil 'eof)))
         ((eq linea 'eof) "He leido el fin de fichero")
       (format t "~&*** ~A~%" linea))
```

```
*** - READ: el flujo de entrada
      #<IO BUFFERED FILE-STREAM CHARACTER #P"fichero" @1> ha alcanzado su
      final
Es posible continuar en los siguientes puntos:
ABORT          :R1      Abort debug loop
ABORT          :R2      Abort debug loop
ABORT          :R3      Abort main loop
[10]> (close canal)
T
[10]> (setq canal (open 'fichero :direction :input))
#<INPUT BUFFERED FILE-STREAM CHARACTER #P"fichero" @1>
[11]> (do ((linea (read-line canal)
           (read-line canal nil 'eof)))
          ((eq linea 'eof) "He leido el fin de fichero")
        (format t "~&*** ~A~%" linea))
*** Escribo un par
*** de lineas de prueba
"He leido el fin de fichero"
[12]> (close canal)
T
[13]> (setq canal (open 'fichero :direction :output :if-exists :append))
#<OUTPUT BUFFERED FILE-STREAM CHARACTER #P"fichero">
[14]> (format canal "y otro par ~%para acabar~%")
NIL
[15]> (close canal)
T
[16]> (setq canal (open 'fichero :direction :input))
#<INPUT BUFFERED FILE-STREAM CHARACTER #P"fichero" @1>
[17]> (do ((linea (read-line canal)
           (read-line canal nil 'eof)))
          ((eq linea 'eof) "He leido el fin de fichero")
        (format t "~&*** ~A~%" linea))
*** Escribo un par
*** de lineas de prueba
*** y otro par
*** para acabar
"He leido el fin de fichero"
[18]> (close canal)
T
```

**(WITH-OPEN-FILE (canal fichero {opción}*) {declaración}* {forma}*)**

Evalúa las `formas` del cuerpo mediante un `progn` implícito mientras la

variable `canal` se asigna a un canal de entrada/salida asociado al `fichero`. Las `opciones` se evalúan y usan como argumentos para la función `open` que abrirá el `canal`. Cuando el control del intérprete abandona el cuerpo, el `canal` se cierra de forma automática. Devuelve `()`.

**Ejemplo 6.2** *Para ilustrar el funcionamiento de* `with-open-file`*, podemos teclear en interactivo algunas expresiones simples:*

```
[8]> (with-open-file (canal 'fichero :direction :output :if-exists :supersede)
                     (format canal "Escribo un par ~%de lineas de prueba~%"))
NIL
[9]> (with-open-file (canal 'fichero)          ; por defecto :direction :input
                     (do ((linea (read-line canal)
                                 (read-line canal nil 'eof)))
                         ((eq linea 'eof) "He leido el fin de fichero")
                       (format t "~&*** ~A~%" linea)))
*** Escribo un par
*** de lineas de prueba
"He leido el fin de fichero"
[10]> (with-open-file (canal 'fichero :direction :output :if-exists :append)
                      (format canal "y otro par ~%para acabar~%"))
NIL
[11]> (with-open-file (canal 'fichero)          ; por defecto :direction :input
                      (do ((linea (read-line canal)
                                  (read-line canal nil 'eof)))
                          ((eq linea 'eof) "He leido el fin de fichero")
                        (format t "~&*** ~A~%" linea)))

*** Escribo un par
*** de lineas de prueba
*** y otro par
*** para acabar
"He leido el fin de fichero"
[12]> (with-open-file (canal 'fichero :direction :output)
                      (format canal "Me he cargado el contenido anterior~%"))
NIL
[13]> (with-open-file (canal 'fichero)          ; por defecto :direction :input
                      (do ((linea (read-line canal)
                                  (read-line canal nil 'eof)))
                          ((eq linea 'eof) "He leido el fin de fichero")
                        (format t "~&*** ~A~%" linea)))
*** Me he cargado el contenido anterior
"He leido el fin de fichero"
```

**(FILE-LENGTH canal)**

Devuelve, de existir, la longitud del fichero cuyo `canal` asociado es el argumento. En otro caso devuelve `()`.

La unidad de longitud es la asociada al argumento `:element-type` especificado en el momento de la apertura del `canal`, tal como se indicó en la descripción de la función `open`.

**Ejemplo 6.3** *Para ilustrar el funcionamiento de* `file-length`*, podemos teclear en interactivo algunas expresiones simples:*

```
[14]> (with-open-file (canal 'fichero :direction :output
                                      :if-does-not-exist :create
                                      :if-exists :overwrite)
                      (format canal "Escribo un par ~%de lineas de prueba~%")
                      (format t "Longitud del fichero: ~D ~%"
                                (file-length canal)))
Longitud del fichero: 36
NIL
```

**(FILE-POSITION canal [posición])**

Devuelve, de existir, la posición actual del puntero de lectura/escritura del fichero cuyo `canal` asociado es el primer argumento argumento. Si se proporciona la `posición`, ésta se asigna al puntero, devolviendo su nuevo valor. En otro caso devuelve `()`. La unidad de longitud es la misma considerada por `file-length`.

**Ejemplo 6.4** *Para ilustrar el funcionamiento de* `file-position`*, podemos teclear en interactivo algunas expresiones simples:*

```
[14]> (with-open-file (canal 'fichero :direction :io
                                      :if-does-not-exist :create
                                      :if-exists :overwrite)
                      (format canal "Escribo un par ~%de
                                     lineas de prueba~%")
                      (format t "Posicion actual puntero: ~D ~%"
                                (file-position canal))
                      (format t "Posicion actual puntero: ~D ~%"
                                (file-position canal 0))
                      (format canal "Corrijo")
                      (format t "Posicion actual puntero: ~D ~%"
                                (file-position canal))
                      (file-position canal 0)
                      (do ((linea (read-line canal)
```

```
                                    (read-line canal nil 'eof)))
                              ((eq linea 'eof) "He leido el fin de fichero")
                            (format t "~&*** ~A~%" linea)))
Posicion actual puntero: 36
Posicion actual puntero: 0
Posicion actual puntero: 7
*** Corrijo un par
*** de lineas de prueba
"He leido el fin de fichero"
```

**(READ-LINE [canal] [eof-error-p] [valor-eof] [recursivo-p])**

Lee una línea desde el canal de entrada. Si `eof-error-p` es cierto (valor por defecto), entonces se señalará un error al alcanzar el `eof`, sino simplemente se devuelve el `valor de eof`. Cuando `recursivo-p` es `t`, entonces estamos indicando que esta llamada será embebida en otra de lectura de más alto nivel. Los valores por defecto de los argumentos son `*standar-input*`, `t`, `()` y `()`; respectivamente. Devuelve `()`.

**Ejemplo 6.5** *Para ilustrar el funcionamiento de* `read-line`*, podemos teclear en interactivo algunas expresiones simples:*

```
[15]> (with-open-file (canal 'fichero :direction :io
                                      :if-does-not-exist :create
                                      :if-exists :overwrite)
                      (format canal "Escribo un par ~%de lineas de prueba~%")
                      (file-position canal 0)
                      (read-line canal))
"Escribo un par " ;
NIL
[16]> (setq valor-eof "He leÃdo el EOF")
"He leÃdo el EOF"
[17]> (with-open-file (canal 'fichero :direction :io
                                      :if-does-not-exist :create
                                      :if-exists :overwrite)
                      (format canal "Escribo un par ~%de lineas de prueba~%")
                      (read-line canal () valor-eof))
"He leÃdo el EOF" ;
T
[18]> (read-line)
Estoy poniendo a prueba READ-LINE sobre la entrada estandar
"Estoy poniendo a prueba READ-LINE sobre la entrada estandar" ;
NIL
```

**(READ-CHAR [canal] [eof-error-p] [valor-eof] [recursivo-p])**

Lee y devuelve un carácter desde el canal de entrada. La interpretación de los argumentos es la misma de `read-line`.

**Ejemplo 6.6** *Para ilustrar el funcionamiento de* `read-char`*, podemos teclear en interactivo algunas expresiones simples:*

```
[19]> (with-open-file (canal 'fichero :direction :io
                                      :if-does-not-exist :create
                                      :if-exists :overwrite)
                      (format canal "Escribo un par ~%de lineas de prueba~%")
                      (file-position canal 0)
                      (read-char canal))
#\E
```

**(WRITE objeto [:STREAM canal] [:ESCAPE {t | ()}]**
**[:RADIX {t | ()}] [:BASE base]**
**[:CIRCLE {t | ()}] [:PRETTY {t | ()}]**
**[:LEVEL profundidad] [:LENGTH longitud]**
**[:CASE {:upcase | downcase | capitalize}]**
**[:GENSYM {t | ()}] [:ARRAY {t | ()}]**
**[:READABLY {t | ()}] [:RIGHT-MARGIN {entero | ()}]**
**[:MISER-WIDTH {t | ()}] [:LINES {entero | ()}]**
**[:PPRINT-DISPATCH tabla])**

La representación de salida del `objeto` se escribe en el canal de salida especificado por el argumento `:stream`, siendo éste por defecto `*standar-output*`. El resto de argumentos opcionales especifican valores de control en la generación de la representación de salida:

- Valor por defecto `*print-escape*`. Establece si los caracteres de `escape` se hacen visibles o no. En el primer caso el valor del argumento debe ser `t`, en el segundo `()`.
- Valor por defecto `*print-radix*`. Controla la salida de racionales. Si es `t`, la salida incluye un especificador que indica el `radio` son el que se representa un número racional.
- Valor por defecto `*print-base*`. Controla la `base` de representación numérica.
- Valor por defecto `*print-circle*`. Controla la representación de estructuras circulares. En caso de ser `()`, la salida sigue simplemente un curso recursivo, sin detectar circularidades.
- Valor por defecto `*print-pretty*`. Da acceso, en caso de ser `t` al `pretty-printer` del intérprete.

- Valor por defecto `*print-level*`. Controla el nivel de `profundidad` de la representación.
- Valor por defecto `*print-length*`. Controla el nivel de `longitud` de la representación.
- Valor por defecto `*print-case*`. Controla la presencia de mayúsculas y minúsculas en la salida. Puede tomar los valores `:upcase`, `:downcase` o `:capitalize`, asegurando la salida en mayúsculas, minúsculas o mezcla de ambas.
- Valor por defecto `*print-gensym*`. Controla si el símbolo # se imprime antes de los símbolos. Si su valor es `()`, dicho símbolo no aparece.
- Valor por defecto `*print-array*`. Controla el formato de salida de los `array`. Si es `()`, entonces todos los contenidos de un `array`, salvo aquellos que sean de tipo `string`, no se representarán; a la vez que la salida es representada en un formato específico.
- Valor por defecto `*print-readably*`. Controla el formato de salida de algunos objetos Lisp. Específicamente, si es `t`, se procede como si `*print-escape*`, `*print-array*` y `*print-gensym*` fueran también `t`, y como si `*print-length*`, `*print-level*` y `*print-lines*` fueran `()`.
- Valor por defecto `*print-right-margin*`. Si su valor es diferente a `()`, ha de ser un número `entero` que especifica el margen derecho de la salida en término de unidades para el `pretty-printer`[6].
- Valor por defecto `*print-miser-width*`. Si su valor es diferente de `()`, el `pretty-printer` cambia a un estilo más compacto de salida denominado `estilo avaro`.
- Valor por defecto `*print-lines*`. Si su valor es diferente de `()`, ha de tratarse de un número `entero`, que marca el límite de líneas de salida para una expresión en el `pretty-printer`.
- Valor por defecto `*print-pprint-dispatch*`. Su valor es dependiente de la implementación y viene fijada por una `tabla`.

Devuelve la representación del `objeto`.

**Ejemplo 6.7** *Para ilustrar el funcionamiento de* `write`*, podemos teclear en interactivo algunas expresiones simples:*

[6] esto es, cuando `print-pretty` posee el valor `t`.

```
[20]> (write #\A)
#\A
#\A
[21]> (write #\A :escape ())
A
#\A
[22]> (write 5/4 :radix t)
#10r5/4
5/4
[23]> (write 5/4 :radix t :base 2)
#b101/100
5/4
[24]> (defun cirlist (&rest x) (rplacd (last x) x))
CIRLIST
[25]> (write (cirlist 1 2) :circle t)
#1=(2 1 . #1#)
[26]> (write '(let ((a 1) (b 2)) (+ a b)) :pretty t) ; parece no funcionar
(LET ((A 1)
      (B 3))
     (+ A B))
NIL
[27]> (write (cirlist 1 2) :length 20)
(2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 ...)
[28]> (write '(((1 2 (3)) 4 (5)) 6 ((2) 1 ((3 4 (5))))) :level 2)
((# 4 #) 6 (# 1 #))
((# 4 #) 6 (# 1 #))
[29]> (write '(a B c D e F) :case :upcase)
(A B C D E F)
(A B C D E F)
[30]> (write '(a B c D e F) :case :downcase)
(a b c d e f)
(A B C D E F)
[31]> (write '(a B c D e F) :case :capitalize) ; parece no funcionar
(a B c D e F)
(A B C D E F)
[32]> (gensym)
#:G11507
[33]> (write (gensym) :gensym ())
G11508
#:G11508
[34]> (write (vector 1 "hola" 2) :array ())
#<ARRAY T (3) #x0003629124D0>
#(1 "hola" 2)
[35]> (write (vector 1 "hola" 2))
```

```
#(1 "hola" 2)
#(1 "hola" 2)
```

**(PRIN1 objeto [canal])**

Da salida, y devuelve, por `canal` al `objeto`. Por defecto el segundo argumento es la salida estandar.

**(PRINT objeto [canal])**

Idéntica a `prin1`, con la salvedad de que la representación del `objeto` está precedida por un `newline`.

**(PPRINT objeto [canal])**

Idéntica a `print`, con la salvedad de que la salida de restreo se omite, y se utiliza el `pretty-printer`.

**(PRINC objeto [canal])**

Idéntica a `prin1`, con la salvedad de que la representación del `objeto` no incluye caracteres de `escape`.

**Ejemplo 6.8** *Para ilustrar el funcionamiento de* `prin1`*,* `print`*,* `pprint` *y* `princ`*, teclearemos en interactivo algunas expresiones:*

```
[1]> (prin1 '(let ((a 1) (b 2)) (+ a b)))
(LET ((A 1) (B 2)) (+ A B))
(LET ((A 1) (B 2)) (+ A B))
[2]> (print '(let ((a 1) (b 2)) (+ a b)))

(LET ((A 1) (B 2)) (+ A B))
(LET ((A 1) (B 2)) (+ A B))
[3]> (pprint '(let ((a 1) (b 2)) (+ a b)))

(LET ((A 1) (B 2)) (+ A B))

[4]> (princ '\#a)
#A
|#A|
[5]> (print '\#a)

|#A|
|#A|
```

```
[6]> (pprint '\#a)

|#A|
```

**(PRIN1-TO-STRING objeto)**

**(PRINC-TO-STRING objeto)**

**(WRITE-TO-STRING objeto [:STREAM canal] [:ESCAPE {t | ()}]
[:RADIX {t | ()}] [:BASE base]
[:CIRCLE {t | ()}] [:PRETTY {t | ()}]
[:LEVEL profundidad] [:LENGTH longitud]
[:CASE {:upcase | downcase | capitalize}]
[:GENSYM {t | ()}] [:ARRAY {t | ()}])**

Las funcionalidades son, respectivamente, las mismas que las de `prin1`, `princ` y `write`. La diferencia es que aquí la salida se guarda en forma de cadena, que se devuelve como resultado.

**Ejemplo 6.9** *Para ilustrar el funcionamiento de* `write-to-string`, `prin1-to-string` *y* `princ-to-string`, *teclearemos en interactivo algunas expresiones:*

```
[7]> (prin1-to-string '(let ((a 1) (b 2)) (+ a b)))
"(LET ((A 1) (B 2)) (+ A B))"
```

**(FORMAT destino cadena-control {argumento}*)**

Genera salidas formateadas a partir de la `cadena de control`. Si el `destino` es una cadena, `t` o un canal; entonces devuelve `()`. En otro caso, el resultado es una cadena. La salida se dirige hacia el `destino`, y si este es `t` indicará la salida estandar. Los `argumentos` lo son para la `cadena de control`. La `cadena de control` incluye directivas de formato (precedidas por ∼), algunas de las cuales resumimos en la tabla siguiente:

| | |
|---|---|
| ∼A | El objeto se representa sin caracteres de `escape`, como con `princ` |
| ∼S | El objeto se representa con caracteres de `escape` |
| ∼D | El objeto, un número entero, se representa en formato decimal |
| ∼B | El objeto, un número entero, se representa en formato binario |
| ∼O | El objeto, un número entero, se representa en formato octal |
| ∼X | El objeto, un número entero, se representa en formato hexadecimal |
| ∼C | El objeto, un carácter, se representa como tal |
| ∼F | El objeto, un punto flotante, se representa como tal |
| ∼E | El objeto, un real exponencial, se representa como tal |
| ∼G | El objeto, un real, se representa como punto flotante, exponencial o formato fijo |
| ∼$ | El objeto, un real, se representa en formato fijo |
| ∼% | Representa un carácter `#\Newline` |
| ∼& | A menos que estemis a principio de línea, representa un carácter `#\Newline` |
| ∼∼ | Representa una ∼ |
| ∼T | Representa una tabulación |
| ∼ ∗ | Ignora el siguiente argumento. ∼ $n*$ ignora los `n` siguientes |
| ∼W | El argumento, cualquier objeto, es representado como con `write` |

**Ejemplo 6.10** *Para ilustrar el funcionamiento de* `format`*, teclearemos en interactivo algunas expresiones directamente sobre el intérprete:*

```
[8]> (format nil "La respuesta es ~D." 5)
"La respuesta es 5."
[9]> (format nil "La respuesta es ~3D." 5)
"La respuesta es   5."
[10](format t "La respuesta es ~D." 5)
"La respuesta es 5."
NIL
[11]> (format nil "El ~D en binario ~B." 5 5)
"El 5 en binario 101."
[12]> (format nil "El ~D en exponencial ~E." 5 5)
"El 5 en exponencial 5.0E+0."
[13]> (format nil "El ~D en formato fijo ~$." 5 5)
"El 5 en formato fijo 5.00."
[14]> (format nil "Una tabulacion en medio ~T y una nueva linea ~% luego")
"Una tabulacion en medio   y una nueva linea
 luego"
[15]> (format nil "Un vector ~W" (vector 1 2 3))
"Un vector #(1 2 3)"
```

# Capítulo 7

# Paquetes: utilización y generación.

Un problema común a alguno de los primeros entornos Lisp era el uso de un único espacio de nombres lo que, entre otras cosas, impedía la consideración de estrategias de programación modular primero y orientada a objetos después. Los `package` vienen a dar respuesta a esta cuestión.

Técnicamente, un `package` es una estructura de datos que establece una correspondencia entre cadenas y símbolos. El `package` actual es, por definición, el contenido en un momento dado de la variable `*package*`, que podemos recuperar simplemente interrogando al sistema:

```
[1]> *package*
#<PACKAGE COMMON-LISP-USER>
```

o cambiar mediante la función `in-package`

```
[2]> (in-package "COMMON-LISP-USER") ; en este caso la dejo como estaba ...
#<PACKAGE COMMON-LISP-USER>
[3]> *package*                       ; ... la prueba
#<PACKAGE COMMON-LISP-USER>
```

En la práctica, podemos referirnos a símbolos en un `package` simplemente precediendo al mismo del nombre de éste último. Así, por ejemplo, podemos referirnos símbolo `bienvenida` del `package` de nombre `saludo`, siguiendo el modelo que pasamos a ilustrar:

```
[4]> (make-package 'saludo)
#<PACKAGE SALUDO>
[5]> (setq saludo::bienvenida "hola")
"hola"
[6]> saludo::bienvenida
"hola"
```

Los símbolos de un `package` pueden ser *internos* o *externos*. En el primer caso, el símbolo es accesible desde su propio `package`, a la vez que propiedad de éste en lo relativo a las condiciones de acceso, lo que restringe su uso al `package` propietario. En el segundo caso, dicho acceso símbolo es considerado como público y, por tanto, accesible desde otros `packages`. La mayoría de los símbolos son generados como internos, pudiendo convertirse en externos mediante una declaración explícita de tipo `export`.

```
[7]> (use-package 'saludo)    ; heredamos todo simbolo externo de 'SALUDO
T
[8]> (intern "adios" 'saludo) ; ADIOS simbolo interno de 'SALUDO
SALUDO::|adios| ;
:INTERNAL
[9]> (find-symbol "adios")    ; veo si ADIOS es accesible desde *package*
NIL ;
NIL
[10]> (export (find-symbol "adios" 'saludo) 'saludo) ; exporto ADIOS desde 'SAL
T
[11]> (find-symbol "adios")    ; ahora ADIOS si es accesible desde *package*
|adios| ;
:INHERITED
```

Podemos ademÃ¡s forzar que un símbolo sea accesible directamente, no sólo por herencia en un paquete:

```
[12]> (shadow "adios" *package*); aseguramos que ADIOS sea interno a *package*
T
[13]> (find-symbol "adios")     ; y e aqui la prueba
|adios| ;
:INTERNAL
```

De la misma forma, un símbolo externo de otro `package` puede hacerse interno a uno dado mediante una declaración de tipo `import`:

```
[14]> (import 'common-lisp::car        ; importo 'CAR desde 'COMMON-LISP
              (make-package 'temporal  ; al package TEMPORAL
                            :use nil)) ; no heredamos de ningun package
T
[15]> (find-symbol "CAR" 'temporal)    ; en efecto, CAR ha sido importado
CAR ;
:INTERNAL
[16]> (find-symbol "CDR" 'temporal)    ; no es el caso de CDR, que no fue impor
NIL ;
NIL
```

Podemos también desvincular un símbolo interno a un `package` dado mediante la función `unintern`:

```
[15]> (unintern  (find-symbol "CAR" 'temporal) 'temporal)
T
[16]> (find-symbol "CAR" 'temporal)   ; CAR ya no es interna al package TEMPORAL
NIL ;
NIL
```

Igualmente podemos hacer que un símbolo externo a un `package`, deje de serlo para convertirse en interno al mismo.

```
[17]> (unexport (find-symbol "adios") ; ADIOS deja de ser externa
                'saludo)              ; al package SALUDO ...
T
[18]> (find-symbol "adios")           ; ... la prueba
NIL ;
NIL
```

También podemos identificar todas las localizaciones en `packages` para un símbolo dado:

```
[19]>  (intern "CDR"                   ; declaro CDR como interno al
               (make-package 'temporal ; package TEMPORAL
                             :use
                             nil))
TEMPORAL::CDR ;
NIL
[20]> (find-all-symbols 'cdr)          ; CDR esta asociado a TEMPORAL y COMMON-LISP
(CDR TEMPORAL::CDR)
```

y también iterar sobre los símbolos de un `package` dependiendo de si son internos o externos:

```
[21]> (intern "CAR" 'temporal)
TEMPORAL::CAR ;
NIL
[22]> (export (intern "CONS" 'temporal) 'temporal)
T
[23]> (let ((lista))
        (do-symbols
            (s (find-package 'temporal)) ; itero accesibles en TEMPORAL
            (push s lista))
        lista)
(TEMPORAL:CONS TEMPORAL::CDR TEMPORAL::CAR)
```

```
[24]> (let ((lista))
         (do-external-symbols
             (s (find-package 'temporal)) ; itero externos en TEMPORAL
             (push s lista))
         lista)
(TEMPORAL:CONS)
[25]> (let ((lista))
         (do-all-symbols
             (s lista)                    ; itero en todo package registrado
             (when (eq (find-package 'temporal) (symbol-package s))
                   (push s lista)))
         lista)
(TEMPORAL:CONS TEMPORAL::CDR TEMPORAL::CAR)
```

Amén de ello, podemos renombrar y asociar nuevos pseudónimos a los packages ya creados

```
[26]> (rename-package 'saludo       ; renombro el package SALUDO
                      'salut        ; como SALUT
                      'hi)          ; con el pseudonimo HI
#<PACKAGE SALUT>
[27]> (find-symbol "adios" 'salut)  ; y ADIOS es por tanto interna a SALUT
SALUT::|adios| ;
:INTERNAL
[28]> (find-symbol "adios" 'hi)     ; y tambien a HI
SALUT::|adios| ;
:INTERNAL
[29]> (find-symbol "adios" 'saludo) ; pero no a SALUDO que ya no existe

*** - FIND-SYMBOL: There is no package with name "SALUDO"
Es posible continuar en los siguientes puntos:
USE-VALUE      :R1      Input a value to be used instead.
ABORT          :R2      Abort debug loop
ABORT          :R3      Abort main loop
[30]> (package-nicknames 'SALUT)    ; recupero los pseudonimos del package SALU
("HI")
```

También podemos detectar las interdependencias entre packages:

```
[31]> (package-use-list 'salut)     ; los package usados por SALUT
(#<PACKAGE COMMON-LISP>)
[32]>  (package-used-by-list 'salut); y los que usan al package SALUT
(#<PACKAGE COMMON-LISP-USER>)
[33]> (list-all-packages)           ; Asi como el total de packages en este ins
```

```
(#<PACKAGE GSTREAM> #<PACKAGE GRAY> #<PACKAGE I18N> #<PACKAGE SOCKET>
 #<PACKAGE FFI> #<PACKAGE SCREEN> #<PACKAGE CUSTOM> #<PACKAGE EXT>
 #<PACKAGE CLOS> #<PACKAGE CS-COMMON-LISP-USER> #<PACKAGE CS-COMMON-LISP>
 #<PACKAGE CHARSET> #<PACKAGE KEYWORD> #<PACKAGE SYSTEM>
 #<PACKAGE COMMON-LISP-USER> #<PACKAGE COMMON-LISP> #<PACKAGE EXPORTING>
 #<PACKAGE POSIX> #<PACKAGE REGEXP> #<PACKAGE READLINE> #<PACKAGE XLIB>
 #<PACKAGE XPM> #<PACKAGE BDB> #<PACKAGE LINUX>
 #<PACKAGE COMMON-LISP-CONTROLLER> #<PACKAGE ASDF-UTILITIES> #<PACKAGE ASDF>
 #<PACKAGE TEMPORAL> #<PACKAGE SALUT>)
```

y, por supuesto, eliminarlos:

```
[34]> (delete-package (find-package 'salut)); elimino el package SALUT ...
T
[35]> (find-package 'salut)                 ; ... la prueba
NIL
[36]> (find-package 'hi)                    ; mismo resultado con el pseudonimo
NIL
```

Aunque, sin duda, la mejor forma de definir un `package` es usando la macro `defpackage`, lo que nos permite condensar la mayor parte de las funcionalidades antes comentadas:

```
[37]> (defpackage "MI-PACKAGE"
                  (:nicknames "MIPKG" "MI-PKG")
                  (:use "COMMON-LISP")
                  (:shadow "CAR" "CDR")
                  (:export "EQ" "CONS" "FROBOLA"))
#<PACKAGE MI-PACKAGE>
```

Por defecto, Common Lisp proporciona algunos `packages` con el sistema, estos son:

1. El `package user`, que se carga en el momento de lanzar el intérprete.
2. El `package common-lisp` contiene las primitivas de ANSI Common Lisp. Sus símbolos externos incluyen todas las funciones visibles al usuario y variables globales presentes en el referido sistema, tales como `car`, `cdr` o `package`. Su pseudónimo es `cl`.
3. El `package common-lisp-user` es, por defecto, el `package` actual en el momento de cargar el intérprete.
4. El `package keyword` contiene todas las palabras clave usadas por las funciones Lisp, tanto si son definidas por el usuario como si son del propio sistema.

5. El `package system` está reservado a la implementación del propio sistema. Usa el pseudónimo `sys`.

# Capítulo 8

# Módulos

Un *módulo* es un subsistema Lisp que se carga a partir de uno o más ficheros. Normalmente se carga como una única unidad, independientemente del número de ficheros implicados. En este sentido, un módulo puede consistir de uno o más `packages`. La lista de módules actualmente cargadas por el intérprete es recuperable a través de la variable `*modules*`.

```
[30]> (require "regexp") ; cargo el package REGEXP en *MODULES*
T
[31]> (provide "regexp") ; incluyo REGEXP en *MODULES*
("ASDF" "berkeley-db" "clx" "linux" "readline" "regexp" "syscalls" "i18n")
[32]>  *modules*         ; visualizo *MODULES*
("ASDF" "berkeley-db" "clx" "linux" "readline" "regexp" "syscalls" "i18n")
```

# Capítulo 9

# Programación orientada a objetos

Los intérpretes COMMON LISP incluyen una extensión orientada a objetos habitualmente conocida por las siglas CLOS y basada en los conceptos de clase, funciones genéricas, métodos, herencia y polimorfismo.

## 9.1. Clases

Hay dos tipos de clases, denominadas `t` y `standard-object`. Las del primer tipo no tienen superclases asociadas, mientras que las del segundo si. La clase `t` es, de hecho,una superclase de cualquier clase con excepción de si misma. Las clases del tipo `standard-object` son instancias de la clase `standard-class`.

Las clases se representan por objetos que son, en si mismos, instancias de clases. La clase a la que pertenece la clase de un objeto se denomina *metaclase*, y determina tanto el tipo de mecanismo de herencia como la representación de sus instancias. El sistema proporciona una metaclase por defecto, que denominamos `standrad-class`, y que es apropiada para la matoría de propósitos prácticos.

En similares términos podemos referirnos a las funciones genéricas y a los métodos, instancias de las clases `standard-generic-function` y `standard-method`, respectivamente.

**(DEFCLASS clase ({superclase}*) ({campo}*) [{opción}*]**

donde el primer argumento da nombre a la `clase` definida, mientras el segundo señala las `superclases` que pretendemos asignarle. En cuanto a las `ranuras`, estas responden a la gramática siguiente:

**campo** ::= **nombre**
| **(nombre descriptor)**

**descriptor** ::= **{:READER función}***
| **{:WRITER función}***
| **{:ACCESSOR función}***
| **{:ALLOCATION función}***
| **{:INITARG nombre}***
| **{:INITFORM forma}***
| **{:TYPE tipo}***
| **{:DOCUMENTATION cadena}***

**función** ::= **{símbolo}***
| **{SETF símbolo }***

donde el significado de los descriptores es el siguiente:

- Descriptor `:documentation`, una cadena comentando la clase.
- Descriptor `:allocation`, posee dos valores posibles:
  - El valor `:instance` indica que se trata de una `campo` local a cada instancia de la clase.
  - El valor `:class` indica que se trata de una `campo` compartida por todas las instancias de la clase.
- Descriptor `:initarg`, usada como palabra clave por la función `make-instance` para proporcionar el valor de este campo.
- Descriptor `:initform`, forma evaluada al momento de la creación de una instancia para proporcionar el valor para este campo.
- Descriptor `:reader`, símbolo que da nombre a un método usado para recuperar el valor de este campo en cada instancia.
- Descriptor `:writer`, símbolo que da nombre a un método usado para asignar el valor de este campo en cada instancia.
- Descriptor `:accessor`, símbolo que da nombre a un método usado para recuperar o asignar el valor de este campo en cada instancia.
- Descriptor `:type`, el tipo de valores permitido en este campo.

En relación a las `opciones`, vienen definidas por:

**opción** ::= **(:DEFAULT-INITARGS . lista)**
| **(:DOCUMENTATION cadena)**
| **(:METACLASS nombre)**

donde en relación a los descriptores:

- Descriptor `:default-initargs`, asocia una `lista` de valores alternados de `initarg` y valores por defecto para los mismos.
- Descriptor `:documentation`, asocia una `cadena` documentando la clase.
- Descriptor `:metaclass`, asocia un símbolo no nulo representando la metaclase que a la que pretendemos ligar la estructura definida.

**(MAKE-INSTANCE clase . argumentos)**

Devuelve una instancia de `clase`, asociada a la lista inicial de `argumentos` proveída.

**(CLASS-OF objeto)**

Devuelve la clase de la que `objeto` es instancia.

## 9.2. Polimorfismo y herencia: métodos y funciones genéricas

Un método es una función asociada como una propiedad a una o varias clases, siendo la invocación y construcción de su lambda-lista asociada idéntica a las de aquellas, con la sola diferencia de que hemos de explicitar cada una de esas clases. Para ello utilizaremos una notación de par `(argumento clase)`. EL concepto clásico de herencia se aplica, por tanto, a los métodos como propiedades que son de una clase. Los métodos son construidos, en la filosofía del *"despacho de mensajes"* heredada de LeLisp, por lo que son independentes de las clases en lo que a su invocación se refiere por parte del usuario.

Podemos, también, definir métodos por encima de clases individuales. En este caso los métodos se asocian a otros métodos con el mismo nombre, más que a una clase determinada. De esta forma, si para una clase determinada no existiera un método específico, aplicaríamos entonces la *función genérica* asociada a dicho método. Aunque, en principio, cualquier función no asociada a una clase concreta es una función genérica, éstas pueden definirse de forma explícita con al ayuda de `defgeneric`, lo que nos permite asociarle una cadena de `documentación`.

**Ejemplo 9.1** *Supongamos las siguientes funciones definida como ejemplo en el fichero* ∼`/lisp/CLisp/examples/ejemplo-oo.cl`*, que ilustran las funciones anteriormente introducidas:*

```
; Una simple jerarquia ANIMAL

(defclass animal ()
   ((nombre :type string :reader nombre :initarg :nombre)
    (peso :type decimal :accessor peso :initarg :peso)
    (cobertura :type symbol :reader cobertura :initarg :cobertura)
    (crias-viables :type integer :accessor crias-viables :initarg :crias-viable

(defclass mamifero (animal)
   ((cobertura :initform 'pelo)
    (lactancia :type decimal  ; periodo lactancia en meses
               :accessor lactancia
               :initarg :lactancia)))

(defclass ave (animal)
   ((cobertura :initform 'pluma)
    (puesta :type integer     ; numero puesta huevos
            :accessor puesta
            :initarg :puesta)))

(defclass pinguino (ave)
   ((puesta :initform 3)))

; Metodos asociados a la jerarquia ANIMAL antes definida

(defmethod movilidad ((especimen pinguino))
    "Los pingÃ¼inos no vuelan, sino que nadan"
    'nadando)

(defmethod movilidad ((especimen ave))
    "Las aves vuelan"
    'volando)

; Funcion generica MOVILIDAD. Se asocia al propio metodo MOVILIDAD, de forma
; que se aplica a cualquier objeto para el que no se haya definido un
; metodo MOVILIDAD especifico.

(defgeneric movilidad (especimen)
    (:documentation "Devuelve el modo de desplazamiento del animal ESPECIMEN")
    (:method (especimen)
        "Los mamÃferos caminan"
        'andando))

; Jerarquia adicional sobre la clase ANIMAL, con objeto de experimentar
```

```
; la herencia multiple

(defclass carnivoro (animal)
   ((crias-viables :initform 2)
    (territorio :type decimal       ; radio terriorio caza en Km
                :accessor territorio
                :initarg :territorio)))

(defclass herbivoro (animal)
   ((crias-viables :initform 1)))

; Clases obtenidas por herencia mÃºltiple

(defclass gato (mamifero carnivoro) nil)

(defclass canario (ave herbivoro) nil)

(defclass murcielago (mamifero carnivoro) nil)

; Metodos asociados a mas de una clase

(defmethod se-lo-come ((predador carnivoro) (presa herbivoro))
    "Determina si el predador se come a la presa, como regla por defecto,
     en función de su peso"
    (< (peso presa) (* 1.25 (peso predador))))

(defmethod se-lo-come ((predador herbivoro) (presa animal)) nil)

(defmethod se-lo-come ((predador carnivoro) (presa carnivoro))
   "Determina si el predador se come a la presa, como regla por defecto,
    en función de su peso"
    (< (peso presa) (peso predador)))
```

*definiendo una jerarquía simple, y no poco imaginativa, en el mundo animal. Ilustramos ahora brevemente su comportamiento, tecleando algunas expresiones directamente sobre el intérprete:*

```
[1]> (load "~/lisp/CLisp/examples/ejemplo-oo")
;;  Loading file /home/vilares/lisp/CLisp/examples/ejemplo-oo.cl ...
;;  Loaded file /home/vilares/lisp/CLisp/examples/ejemplo-oo.cl
T

[2]> (setf piolin (make-instance 'canario :nombre "Piolin" :peso 0.1))
#<CANARIO #x000334693BF0>
```

```
[3]> (setf silvestre (make-instance 'gato :nombre "Silvestre" :peso 5 :territor
#<GATO #x000334694658>
[4]> (setf vampus (make-instance 'murcielago :nombre "Vampus" :peso 0.12))
#<MURCIELAGO #x000334694EB0>
[5]> (setf ernesto (make-instance 'pinguino :nombre "Ernesto" :peso 20))
#<PINGUINO #x000334BA19B8>
[6]> (class-of ernesto)         ; recupero la clase del objeto
#<STANDARD-CLASS PINGUINO>
[7]> (class-of piolin)
#<STANDARD-CLASS CANARIO>
[8]> (nombre ernesto)           ; recupero su campo NOMBRE
"Ernesto"
[9]> (cobertura ernesto)        ; recupero su campo COBERTURA
PLUMA
[10]> (typep ernesto 'pinguino)  ; verifico si el tipo del objeto es PINGUINO
T
[11]> (nombre silvestre)
"Silvestre"
[12]> (cobertura silvestre)
PELO
[13]> (typep silvestre 'pinguino)
NIL
[14]> (typep silvestre 'mamifero)
T
[15]> (typep silvestre 'animal)
T
[16]> (typep silvestre 'ave)
NIL
[17]> (typep silvestre 'gato)
T
[18]> (peso piolin)              ; heredado desde ANIMAL
0.1
[19]> (crias-viables piolin)     ; heredado desde ANIMAL, inicializado en HERBIV
1
[20]> (peso silvestre)           ; heredado desde ANIMAL
5
[21]> (territorio silvestre)     ; heredado desde CARNIVORO
2
[22]> (crias-viables silvestre) ; heredado desde ANIMAL, inicializado en CARNIV
2
[23]> (movilidad piolin)         ; llamo metodo MOVILIDAD de AVES
VOLANDO
[24]> (movilidad ernesto)        ; llamo metodo MOVILIDAD de AVES que son PINGUI
NADANDO
```

```
[25]> (movilidad silvestre)     ; llamo funcion generica MOVILIDAD
ANDANDO
[26]> (se-lo-come piolin silvestre) ; un HERBIVORO no puede comer un CARNIVORO
NIL
[27]> (se-lo-come silvestre vampus) ; un CARNIVORO grande puede comer otro CARNIVORO
T
[28]> (se-lo-come vampus silvestre) ; un CARNIVORO pequeño no puede comer otro CARNIVORO
NIL
```

# Índice alfabético