MISCELANEA DE EJERCICIOS – COMMON LISP

```
;;;---------------------------------------------------------------------
;;; PARTE I: EJERCICIOS BÁSICOS DE LISTAS
;;;---------------------------------------------------------------------

;;; En esta primera parte se verán algunos ejercicios en los que se definen
;;; funciones muy sencillas mediante recursión sobre listas. Algunas de
;;; estas funciones tienen su versión predefinida en Lisp.

;;; Como se observará, el "patrón" que siguen todas estas funciones es muy
;;; parecido:
;;;
;;;    (defun f
;;;      (... l ...)
;;;      (if (endp l)
;;;          <...>
;;;          <... (first l)
;;;               (f ... (rest l) ...) ...>))
;;;
;;; Es decir, usualmente el caso base se cumple cuando la lista es vacía, y
;;; en el caso recursivo se trata de operar de alguna manera sobre el
;;; primero de la lista y sobre la llamada recursiva que se obtiene
;;; aplicando la propia función que se define sobre el resto de la
;;; lista. Usaremos indistintamente CAR o FIRST, y del mismo modo usaremos
;;; indistintamente CDR o REST.


;;;**************************************************************************
;;; 1) Definir la función (longitud lst) que devuelva la longitud de la
;;; lista lst. Por ejemplo:
;;;
;;; > (longitud '(a b c d))
;;; 4
;;;**************************************************************************




;;;**************************************************************************
;;; 2) Escribir la función (cuadrados lst) que devuelva la lista de
;;; cuadrados de la lista numérica lst. Por ejemplo:
;;;
;;; > (cuadrados '(1 2 3 4))
;;; (1 4 9 16)
;;;**************************************************************************
```

```
;;;*************************************************************************
;;; 3) Escribir el predicado (suma-lista lst) que devuelva la suma de los
;;; elementos de la lista numérica lst. Por ejemplo:
;;;
;;; > (suma-lista '(1 2 3 4))
;;; 10
;;;*************************************************************************


;;;*************************************************************************
;;; 4) Definir la función (ultimo lst) que devuelva el último elemento de
;;; la lista lst. Supondremos que la lista de entrada nunca será vacía.
;;;
;;; Ejemplo de uso:
;;; > (ultimo '(1 2 3 4))
;;; 4
;;;*************************************************************************


;;;*************************************************************************
;;; 5) Definir la función (pertenece e c) que se verifique si la expresión
;;; e pertenece al conjunto c y devuelva la parte de C que comienza en la
;;; primera aparición de E. Por ejemplo:
;;;
;;; > (pertenece 'b '(a b c))
;;; (B C)
;;; > (pertenece 'b '(a (b) c))
;;; NIL
;;; > (pertenece '(b) '(a (b) c))
;;; ((B) C)
;;;*************************************************************************


;;;*************************************************************************
;;; 6) Definir el predicado (subconjunto c1 c2) que devuelva t si c1 es un
;;; subconjunto de c2 y nil e caso contrario. Por ejemplo,
;;;
;;; > (subconjunto nil '(a))
;;; T
;;; > (subconjunto '(a b) '(b a))
;;; T
;;; > (subconjunto '(a b) '((b) a))
;;; NIL
;;; > (subconjunto '((a b) c) '(c (b a) d))
;;; NIL
;;; > (subconjunto '((b a) c) '(c (b a) d))
```

```
;;; T
;;;*****************************************************************************


;;;*****************************************************************************
;;; 7) Definir el predicado
;;;        (igual-conjunto c1 c2)
;;; que devuelva T si los conjuntos c1 y c2 son iguales y nil, si no lo
;;; son.  Por ejemplo:
;;;
;;; > (igual-conjunto '(a b c) '(c a b))
;;; T
;;; > (igual-conjunto '((a b) c) '(c (a b) c))
;;; T
;;; > (igual-conjunto '((a b) c) '(a b c))
;;; NIL
;;;*****************************************************************************


;;;*****************************************************************************
;;; 8) Definir la función
;;;        (m-union c1 c2)
;;; que devuelva la unión de los conjuntos c1 y c2 (i.e. una lista sin
;;; elementos repetidos que contenga los elementos que están en c1 o en
;;; c2). Por ejemplo:
;;;
;;; > (m-union nil '(a b))
;;; (A B)
;;; > (m-union '(a (c b) e) '((c b) a d))
;;; (E (C B) A D)
;;;*****************************************************************************


;;;*****************************************************************************
;;; 9) Definir la función
;;;        (interseccion c1 c2)
;;; que devuelva la intersección de los conjuntos c1 y c2 (i. e. la lista
;;; de sus elementos comunes). Por ejemplo:
;;;
;;; > (interseccion '(a (c b) ((d))) '((c b) (a) ((d))))
;;; ((C B) ((D)))
;;;*****************************************************************************


;;;*****************************************************************************
```

```
;;; 10) En este ejercicio vamos a "comprimir" y "descomprimir" listas.
;;;
;;; Apartado (a).
;;; Definir la función (compresion lst) que devuelva la lista lst
;;; comprimida en el siguiente sentido:
;;; * Si el elemento x aparece n (n > 1) veces de manera consecutiva en l
;;;   sustituimos esas n ocurrencias por la lista (n x)
;;; * Si el elemento x es distinto de sus vecinos, entonces lo dejamos
;;;   igual
;;; Ejemplo:
;;;
;;; > (compresion '(1 1 1 2 1 3 2 4 4 6 8 8 8))
;;; ((3 1) 2 1 3 2 (2 4) 6 (3 8))
;;; > (compresion '(a a a b a c b d d f h h h))
;;; ((3 a) b a c b (2 d) f (3 h))
;;;
;;; Apartado (b).
;;; Definir la función (descompresion lst) que devuelva la lista lst
;;; descomprimida si se ha comprimido con el método del apartado anterior.
;;; Ejemplo:
;;;
;;; > (descompresion '((3 1) 2 1 3 2 (2 4) 6 (3 8)))
;;; (1 1 1 2 1 3 2 4 4 6 8 8 8)
;;;**************************************************************************




;;;--------------------------------------------------------------------
;;; PARTE II: RECURSIÓN PROFUNDA
;;;--------------------------------------------------------------------


;;; Una lista genérica es una lista cuyos elementos pueden ser átomos u
;;; otras listas genéricas. Un ejemplo de lista genérica es el siguiente:

;;; (0 (1 (2 (3 (4 5) 6) 7) 8) 9)

;;; Obviamente cualquier lista plana (compuesta únicamente por átomos) es a
;;; su vez una lista genérica.

;;; La recursión profunda es un proceso recursivo sobre listas genéricas,
;;; en el que la recursión va de izquierda a derecha recorriendo los
;;; elementos de la lista y además dentro de cada elemento, en el caso de
;;; que sea a su vez una lista genérica.

;;; En las definiciones por recursión profunda suele haber dos casos. El
;;; primero de ellos cuando la lista genérica argumento L es un átomo (esto
;;; no ocurre inicialmente sino dentro de las llamadas recursivas); este es
```

```
;;; un caso base. En este caso base suele ser necesario distinguir si el
;;; argumento es la lista vacía 'NIL' o no. El segundo caso se tiene cuando
;;; la lista L no es un átomo y por tanto es una lista; en este caso hay
;;; una recursión doble: la primera sobre el primer elemento de la lista L
;;; y la segunda sobre el resto de la lista L (es en estas llamadas
;;; recursivas donde puede darse el caso de que el argumento sea un
;;; átomo). Veamos una serie de ejercicios sobre recursión profunda.


;;;*****************************************************************************
;;; 11) Definir la función (pertenece-prof x l), que comprueba la
;;; pertenencia del elemento x a la lista genérica l.  Por ejemplo:
;;;
;;; > (pertenece-prof 5 '(0 (1 (2 (3 (4 5) 6) 7) 8) 9))
;;; T
;;; > (pertenece-prof 'a '(0 (1 (2 (3 (4 5) 6) 7) 8) 9))
;;; NIL
;;;*****************************************************************************


;;;*****************************************************************************
;;; 12) Definir la función (ocurrencias-prof x l), que cuenta el número de
;;; ocurrencias del elemento x en la lista genérica l. Por ejemplo:
;;;
;;; > (ocurrencias-prof 'a '(a (b (a (c (a a) b) d) a) f))
;;; 5
;;;*****************************************************************************


;;;*****************************************************************************
;;; 13) Definir la función (sustituye-prof x y l), que devuelve el
;;; resultado de reemplazar todas las ocurrencias del elemento x en la
;;; lista genérica l por el elemento y. Por ejemplo:
;;;
;;; > (sustituye-prof 'a 'x '(a (b (a (c (a a) b) d) a) f))
;;; (X (B (X (C (X X) B) D) X) F)
;;;*****************************************************************************


;;;*****************************************************************************
;;; 14) Definir la función (n-elementos l), que cuenta el número total de
;;; elementos atómicos en la lista genérica l (en cualquier nivel) . Por
;;; ejemplo:
;;;
;;; > (n-elementos '(a (b (a (c (a a) b) d) a) f))
;;; 10
```

```
;;;*************************************************************************


;;;*************************************************************************
;;; 15) Definir la función (profundidad l), que calcula la profundidad
;;; máxima de la lista genérica l. Por ejemplo:
;;;
;;; > (profundidad '(a (b (a (c (a a) b) d) a) f))
;;; 5
;;;*************************************************************************


;;;*************************************************************************
;;; 16) Definir la función (aplana l), que devuelve una lista plana cuyos
;;; elementos son los de la lista genérica l recorrida de izquierda a
;;; derecha y de arriba a abajo. Por ejemplo:
;;;
;;; > (aplana '(a (b (a (c (a a) b) d) a) f))
;;; (A B A C A A B D A F)
;;;*************************************************************************


;;;*************************************************************************
;;; 17) Definir la función (inversa l), que devuelve una lista genérica
;;; resultado de invertir los elementos de l y de todas las listas
;;; genéricas que puedan aparecer dentro de l. Esta función devuelve una
;;; lista genérica cuya estructura es simétrica de la original. Por
;;; ejemplo:
;;;
;;; > (inversa '(a (b (a (c (a a) b) d) a) f))
;;; (F (A (D (B (A A) C) A) B) A)
;;;*************************************************************************


;;;*************************************************************************
;;; 18) Definir la función (rellena lp lg) que devuelve una lista genérica
;;; con la misma estructura que la lista genérica lg en la que se han
;;; reemplazado los elementos atómicos por los elementos de la lista plana
;;; lp, en el orden en que estos aparecen en el resultado de aplanar lg. Se
;;; prescinde de los elementos de lp que sobren y en caso de faltar se
;;; dejan los originales de lp.  Por ejemplo:
;;;
;;; > (rellena '(1 2 3 4 5 6 7 8 9 10 11 12 13 14 15)
;;;            '(a (b (a (c (a a) b) d) a) f))
;;; (1 (2 (3 (4 (5 6) 7) 8) 9) 10)
```

```
;;; > (rellena '(1 2 3 4 5) '(a (b (a (c (a a) b) d) a) f))
;;; (1 (2 (3 (4 (5 A) B) D) A) F)
;;;*****************************************************************************



;;;-----------------------------------------------------------------------
;;; PARTE III: RECURSIÓN DE COLA
;;;-----------------------------------------------------------------------


;;; Una llamada recursiva se dice que es de tipo final si no aparece
;;; como argumento de ninguna función. Una definición recursiva de cola
;;; (o recursiva terminal) es aquella en la que todas las llamadas
;;; recursivas son de tipo final.

;;; Las funciones recursivas de cola son más eficientes que las que no lo
;;; son pues se pueden transformar de forma automática en procesos
;;; iterativos, que no necesitan pila de recursión. Esta transformación
;;; suele realizarse al compilar la función en el intérprete de Lisp.

;;; Por ejemplo, la siguiente definición de longitud no es recursiva de
;;; cola:

(defun longitud (l)
  (if (endp l)
      0
      (1+ (longitud (cdr l)))))

;;; Sin embargo, la siguiente versión SI es recursiva de cola:

(defun longitud (l &optional (acum 0))
  (if (endp l)
      acum
      (longitud (cdr l) (1+ acum))))


;;; Nótese que esta función longitud, aunque tiene dos argumentos, está
;;; diseñada para que la llamada inicial se realice sobre un único
;;; argumento (la lista a la que se le calcula la longitud). El segundo
;;; argumento es opcional y tiene valor por defecto igual a 0.

;;; Es decir, la usamos de la siguiente manera:

;;; > (longitud '(a b c d))
;;; 4

;;; La definición de una función recursiva de cola suele ser un poco más
```

;;; compleja que una que no lo sea. Una buena forma de realizar dichas
;;; definiciones consiste en transformar una definición recursiva que no
;;; sea de cola en una que sí lo sea (aunque esto no siempre es posible,
;;; por ejemplo cuando la definición dada tenga una recursión
;;; doble). Supongamos que partimos de una definición recursiva dada que no
;;; es recursiva de cola; para transformarla en una definición recursiva de
;;; cola se han de seguir las siguientes pautas:

;;; - La versión recursiva de cola usa un argumento adicional con respecto
;;; a la definición dada que sirve para acumular el resultado devuelto por
;;; la función. El valor por defecto de dicho acumulador suele coincidir
;;; con el valor devuelto por la versión recursiva dada en algún caso base.

;;; - La distinción de casos es la misma en ambas versiones.

;;; - El valor devuelto por la función recursiva de cola en los casos base
;;; se obtiene a partir del resultado devuelto por la definición dada y el
;;; acumulador.

;;; - Los argumentos de las llamadas recursivas varían de la misma forma en
;;; ambas versiones, a excepción del argumento adicional de la versión
;;; recursiva de cola, en el que se han de reflejar las operaciones que en
;;; la definición dada se evalúan sobre la llamada recursiva.

;;; - Las llamadas recursivas de tipo final dentro de la definición dada
;;; quedan exactamente igual en la versión recursiva de cola, en la que el
;;; acumulador no se modifica.

;;; En el ejemplo anterior de la versión recursiva de cola de longitud,
;;; podemos observar todas estas características. Recordemos:

```
(defun longitud (l &optional (acum 0))
  (if (endp l)
      acum
      (longitud (cdr l) (1+ acum))))
```


;;; - La versión recursiva de cola tiene un argumento opcional ACUM que
;;; sirve para acumular el resultado. El valor por defecto de dicho
;;; acumulador es 0, que es el valor devuelto por la versión recursiva dada
;;; en el caso base final.

;;; - La distinción de casos es la misma en ambas versiones.

;;; - El valor devuelto por la función recursiva de cola en el caso base es
;;; directamente el valor del argumento acumulador.

;;; - Los argumentos de la llamada recursiva varían de la misma forma en

```
;;; ambas versiones, a excepción del argumento adicional de la versión
;;; recursiva de cola que se modifica usando la función '1+', que es la
;;; función que se utiliza sobre la llamada recursiva en la definición
;;; dada.



;;;*****************************************************************************
;;; 19) Definir una versión recursiva de cola de la función (ocurrencias x
;;; l), que contabiliza el número de ocurrencias del elemento x dentro de
;;; la lista l. Por ejemplo:
;;;
;;; > (ocurrencias 3 '(a b 3 c d 3))
;;; 2
;;; > (ocurrencias 'x '(a b 3 c d 3))
;;; 0
;;;*****************************************************************************



;;;*****************************************************************************
;;; 20) Definir una versión recursiva de cola de la función (elimina x l),
;;; que elimina las ocurrencias del elemento x dentro de la lista l. Por
;;; ejemplo:
;;;
;;; > (elimina 3 '(a b 3 c d 3))
;;; (A B C D)
;;;*****************************************************************************



;;;*****************************************************************************
;;; 21) Definir una versión recursiva de cola de la función (ELIMINA-UNO X
;;; L), que elimina la primera ocurrencia del elemento X dentro de la lista
;;; L. Por ejemplo:
;;;
;;; > (elimina-uno 3 '(a b 3 c d 3))
;;; (A B C D 3)
;;;*****************************************************************************



;;;*****************************************************************************
;;; 22) Definir una versión recursiva de cola de la función (sustituye x y
;;; l), que devuelve el resultado de reemplazar todas las ocurrencias del
;;; elemento x en la lista l por el elemento y. Por ejemplo:
;;;
;;; > (sustituye 'a 'x '(a b a b a b a b a b))
;;; (X B X B X B X B X B)
```

```
;;;*************************************************************************


;;;*************************************************************************
;;; 23) Definir una versión recursiva de cola de la función (sustituye-uno
;;; x y l), que devuelve el resultado de reemplazar la primera ocurrencia
;;; del elemento x en la lista l por el elemento y. Por ejemplo:
;;;
;;; > (sustituye-uno 'a 'x '(b a b a b a b a b))
;;; (B X B A B A B A B)
;;;*************************************************************************


;;;*************************************************************************
;;; 24) Definir una versión recursiva de cola de la función (mezcla l1 l2),
;;; que recibe como argumentos dos listas numéricas ordenadas de menor a
;;; mayor y devuelve la mezcla ordenada de dichas listas.  Por ejemplo:
;;;
;;; > (mezcla '(1 3 4 6 7 8) '(0 2 5 9))
;;; (0 1 2 3 4 5 6 7 8 9)
;;;*************************************************************************


;;;*************************************************************************
;;; 25) Definir una versión recursiva de cola de la función (fibonacci n),
;;; que devuelve el valor del n-ésimo término de la sucesión de Fibonacci,
;;; calculado de la siguiente forma: f(0) = 1, f(1) = 1, f(n+2) =
;;; f(n)+f(n+1). Por ejemplo:
;;;
;;; > (fibonacci 5)
;;; 8
;;; > (fibonacci 25)
;;; 121393
;;;*************************************************************************


;;;*************************************************************************
;;; 26) Definir una versión recursiva de cola de la función (aplana l), que
;;; devuelve una lista plana cuyos elementos son los de la lista genérica l
;;; recorrida de izquierda a derecha y de arriba a abajo.
;;;
;;; > (aplana '(a (b (a (c (a a) b) d) a) f))
;;; (A B A C A A B D A F)
;;;*************************************************************************
```

```
;;;**************************************************************************
;;; 27) Definir una versión recursiva de cola de la función (tamaño l), que
;;; cuenta el número total de elementos en la lista genérica l.  Por
;;; ejemplo:
;;;
;;; > (n-elementos '(a (b (a (c (a a) b) d) a) f))
;;; 10
;;;**************************************************************************


;;;**************************************************************************
;;; 28) Definir una versión recursiva de cola de la función
;;; (ocurrencias-prof x l), que cuenta el número de ocurrencias del
;;; elemento x en la lista genérica l.  Por ejemplo:
;;;
;;; > (ocurrencias-prof 'a '(a (b (a (c (a a) b) d) a) f))
;;; 5
;;;**************************************************************************


;;;**************************************************************************
;;; 29) El método de Horner dado es un método eficiente (que puede ser
;;; implementado con una función recursiva de cola) para evaluar el valor
;;; de un polinomio (en una variable) para un x dado. Por ejemplo, el valor
;;; del polinomio a·X³+b·X²+c·X+d en un punto x se calcula
;;; x·(x·(a·x+b)+c)+d. Definir una función recursiva de cola (horner x
;;; coeficientes) que calcula el valor de un polinomio dado por sus
;;; coeficientes en un punto x.  Por ejemplo:
;;;
;;; > (horner 3 '(1 2 3 4))
;;; 58
;;;**************************************************************************


;;;--------------------------------------------------------------------
;;; PARTE IV: ITERACIÓN
;;;--------------------------------------------------------------------

;;; Existen varias formas de realizar procesos iterativos en loop. Antes de
;;; pasar a los ejercicios, comentaremos brevemente algunas de ellas.

;;; LOOP BÁSICO
;;; ===========
```

```
;;; La construcción loop es la forma más sencilla de iteración. Evalúa
;;; repetidamente el cuerpo de la expresión.

;;; loop {expresión}*

;;; Cada expresión del cuerpo, recorridas de izquierda a derecha, es
;;; evaluada. Cuando la última expresión es evaluada, vuelve de nuevo a la
;;; primera, indefinidamente. Una construcción loop nunca devuelve un
;;; valor. Su evaluación ha de interrumpirse explícitamente utilizando
;;; return o throw. Por ejemplo, la evaluación de la siguiente expresión
;;; entra en un bucle infinito que ha de ser interrumpido por el usuario:

;;; > (loop (+ 2 3))
;;; Ctrl-C Ctrl-C
;;; ** - Continuable Error
;;; EVAL: User break
;;; If you continue (by typing 'continue'): Continue execution
;;; Break 1> abort

;;; Un ejemplo del uso de return para salir de un bucle:
;;; > (let ((x 5)) (loop (setf x (+ x 1)) (when (= x 10) (return))))
;;; NIL
;;; > (let ((x 0)) (loop (setf x (+ x 1)) (when (= x 10) (return (* x 2)))))
;;; 20

;;; LOOP EXTENDIDO
;;; ==============

;;; Existe una forma extendida de loop. Los elementos de esta forma
;;; extendida, denominados clausulas, no son subexpresiones (es decir, no
;;; están delimitadas por paréntesis) y tiene su propia sintaxis. Se
;;; aconseja consultar el material de Lisp para más detalles sobre la
;;; sintaxis específica de la construcción loop extendida.

;;; Existen tres fases en la evaluación de una forma loop extendida.

;;; * Prólogo: Se evalúa antes de iniciar las iteraciones. Incluye la
;;;   asignación inicial de variables.

;;; * Cuerpo: Se evalúa en cada iteración. Se inicia con la condición de
;;;   parada, seguida por el cuerpo de la construcción y finaliza con la
;;;   actualización de las variables.

;;; * Epílogo: Se valúa al finalizar las iteraciones. Termina con el valor
;;;   que devolverá la construcción.

;;; Por ejemplo:
;;; > (loop for x from 0 to 3 do (format t "~a" x))
```

```
;;; 0123
;;; NIL


;;; La primera cláusula de la expresión anterior es: for x from 0 to
;;; 9. Dicha cláusula hace que se asigne el valor 0 en el prólogo, que se
;;; compare el valor de x con 9 al iniciar el cuerpo y que se incremente en
;;; uno el valor de x al finalizar el cuerpo. La segunda cláusula es: do
;;; (format t "~a" x). Nótese que el valor de la expresión anterior es NIL,
;;; y que (0 1 2 3) es simplemente lo escrito por pantalla, como efecto
;;; colateral de la evaluación de la expresión.

;;; Más ejemplos:
;;; > (loop for x = 8 then (/ x 2) until (< x 1) do (format t "~a" x))
;;; 8421
;;; NIL
;;; > (loop for x from 1 to 3 and y from 11 to 13
;;;         do (format t "~a" (list x y)))
;;; (1 11)(2 12)(3 13)
;;; NIL

;;; Nótese que las expresiones como las anteriores, de la forma (loop
;;; ... do ...) toman como valor el de la última expresión que se evalúa en
;;; la última vuelta. Usualmente, en este tipo de bucles el valor obtenido
;;; no es importante: lo importante es el efecto colateral. Tampoco debemos
;;; confundir el "do" que aparece en loop con la función "do" que se
;;; comentará más adelante.


;;; Sin embargo, hay algunas construcciones loop en los que el valor
;;; obtenido es lo relevante. Por ejemplo, algunas cláusulas nos permiten
;;; acumular valores:

;;; > (loop for x in '(1 2 3 4) collect (* 2 x))
;;; (2 4 6 8)
;;; >

;;; En el prólogo se asocia a x el valor 1 y a un acumulador el valor
;;; '(). En el cuerpo se comprueba si quedan más elementos en la lista, se
;;; añade el valor de x al acumulador, y se le asigna a x un nuevo valor de
;;; la lista. En el epílogo se devuelve el valor del acumulador.

;;; Otro ejemplo:
;;; > (loop for x in '(1 2 3 4 5 6 7 8 9)
;;;         if (evenp x) collect x into pares
;;;         else collect x into impares
;;;         finally (return (list pares impares)))
;;; ((2 4 6 8) (1 3 5 7 9))
```

```
;;; DO y DO*
;;; ========


;;; En contraste con loop, las construcciones do y do* proporcionan una
;;; potente herramienta para el cálculo repetido de variables:

;;; do ({var | (var [ini [paso]])}*)
;;;    (fin-test {res}*)
;;;    cuerpo*

;;; Una construcción do tiene un número arbitrario de variables. Dichas
;;; variables toman un valor inicial y son actualizadas en cada iteración
;;; de forma paralela según se especifique. Cuando se verifica la condición
;;; de parada la evaluación termina devolviendo el valor especificado.

;;; La forma general de una construcción do* es exactamente la misma, la
;;; diferencia estriba en que las sucesivas asignaciones se realizan de
;;; forma secuencial.

;;; Si se omite el valor inicial de una de las variables, por defecto
;;; utiliza el valor nil. Si se omite la expresión paso el valor de la
;;; variable no se actualiza en cada iteración.

;;; Las variables utilizadas recuperan sus valores originales al finalizar
;;; la evaluación de la construcción.

;;; La condición de parada va acompañada de un número arbitrario de
;;; expresiones. Al inicio de cada iteración, tras recalcular las
;;; variables, se evalúa la condición de parada, si resulta distinta de
;;; nil, se evalúan las expresiones que la acompañan de izquierda a
;;; derecha. La construcción do devuelve como valor el de la última
;;; expresión. En caso de no existir ninguna, el valor por defecto es NIL.

;;; Si la condición de parada es nil se evalúan las expresiones del cuerpo
;;; de la construcción antes de iniciar una nueva iteración.

;;; Por ejemplo, el siguiente uso de do nos permite recorrer un vector y
;;; actualizar aquellas componentes que estén a NIL con el valor 0.

;;; > (defparameter vector (vector 1 2 nil 3 4 nil nil 5 nil))
;;; VECTOR
;;; > vector
;;; #(1 2 NIL 3 4 NIL NIL 5 NIL)
;;; > (do ((n (length vector))
;;;       (i 0 (+ 1 i)))
;;;      ((= i n))
;;;    (when (null (aref vector i))
```

```
;;;       (setf (aref vector i) 0)))
;;; NIL
;;; > vector
;;; #(1 2 0 3 4 0 0 5 0)

;;; En el siguiente ejemplo, usamos do para calcular el último elemento
;;; de una lista

;;; > (defparameter lista (list 1 2 3 4 5 6 7 8))
;;; LISTA
;;; > (do ((x lista (cdr x)) (antx nil x)) ((null x) antx))
;;; (8)
;;; > lista
;;; (1 2 3 4 5 6 7 8)

;;; DOLIST y DOTIMES
;;; ================

;;; Las construcciones dolist y dotimes evalúan el cuerpo una vez por cada
;;; uno de los valores que toma la variable. La construcción dolist recorre
;;; los valores de una lista y dotimes los enteros entre 0 y n-1 para un
;;; entero positivo n.

;;; El valor de la construcción puede especificarse, por defecto será nil.

;;; dolist (var lista [result])
;;;        cuerpo*

;;; dotimes (var cont [result])
;;;         cuerpo*

;;; Ejemplos:
;;; > (+ (dolist (x '(a b c d e) 10) (format t "~a " x)) 5)
;;; A B C D E
;;; 15
;;; > (dotimes (k 3 k) (format t "Iteracion: ~a~%" k))
;;; Iteracion: 0
;;; Iteracion: 1
;;; Iteracion: 2
;;; 3
;;; >

;;; ITERACIÓN CON MAPCAR y SIMILARES
;;; --------------------------------

;;; Las siguientes funciones realizan una especie de iteración al aplicar
;;; una función, sucesivamente, a distintos elementos de una lista. El
;;; resultado es una lista con los distintos valores obtenido.
```

```
;;; mapcar función lista &rest listas
;;; maplist función lista &rest listas
;;; mapc función lista &rest listas
;;; mapl función lista &rest listas
;;; mapcan función lista &rest listas
;;; mapcon función lista &rest listas

;;; El primer argumento de estas funciones es siempre una función, los
;;; restantes son listas. La función debe tener tantos argumentos como
;;; listas aparezcan.

;;; La función mapcar aplica la función a los primeros elementos de las
;;; listas, los segundos, ... y así sucesivamente. La iteración termina al
;;; llegar el último elemento de la lista más corta.


;;; > (mapcar #'/ '(1 2 3 4) '(1 2 3 4 5 6))
;;; (1 1 1 1)
;;; > (mapcar #'abs '(3 -4 2 -5 -6))
;;; (3 4 2 5 6)
;;; > (mapcar #'cons '(a b c) '(1 2 3))
;;; ((a . 1) (b . 2) (c . 3))

;;; La función maplist aplica la función a las listas y a los sucesivos
;;; restos de dichas listas.

;;; > (maplist #'(lambda (x) (cons 'foo x)) '(a b c d))
;;; ((foo a b c d) (foo b c d) (foo c d) (foo d))
;;; > (maplist #'(lambda (x) (if (member (car x) (cdr x)) 0 1))
;;;           '(a b a c d b c))
;;; (0 0 1 0 1 1 1)

;;; Las funciones mapl y mapc son como mapcar y maplist, respectivamente,
;;; pero no acumulan los resultados obtenidos. Estas funciones, por tanto,
;;; se utilizan cuando el interés está en los efectos colaterales más que
;;; en los valores calculados. El valor devuelto es la primera de las
;;; listas recibidas.

;;; Las funciones mapcan y mapcon son como mapcar y maplist,
;;; respectivamente, pero el resultado lo construyen utilizando nconc en
;;; lugar de list.

;;; > (mapcan #'(lambda (x) (when (numberp x) (list x))) '(a 1 b c 3 4 d 5))
;;; (1 3 4 5)

;;; Recuérdese que la función nconc es destructiva.

;;; Las funciones utilizadas como argumento deben ser aceptables para la
;;; función apply, por tanto no pueden ser macros o formas especiales.
```

;;; Aunque en este apartado se verán algunos ejercicios con mapcar y
;;; similares, se verán con más detalle en el apartado de programación de
;;; segundo orden.

;;;***
;;; 30) Definir una función (sin utilizar recursión), factorial, para
;;; calcular el factorial de un número natural.
;;;***

;;;***
;;; 31) Definir una versión iterativa de la función longitud del ejercicio
;;; 1).
;;;***

;;;***
;;; 32) Definir una versión iterativa de la función pertenece del ejercicio
;;; 5)
;;;***

;;;***
;;; 33) Definir una función inversa que, sin utilizar reverse ni recursión,
;;; construya la inversa de una lista.
;;;***

;;;***
;;; 34) Definir una versión iterativa de la función elimina del ejercicio
;;; 20)
;;;***

;;;***
;;; 35) Definir un versión iterativa de la función sustituye del ejercicio
;;; 22)
;;;***

;;;***

```
;;; 36) Definir una función iterativa que nos permita concatenar dos
;;; listas.
;;;*****************************************************************



;;;*****************************************************************
;;; 37) Definir una función, no recursiva, que sume los elementos de una
;;; lista.
;;;*****************************************************************



;;;*****************************************************************
;;; 38) Escribir una versión iterativa de la función cuadrados del
;;; ejercicio 3)
;;;*****************************************************************



;;;*****************************************************************
;;; 39) Escribe una función (asocia-cuadrado l) que dada una lista de
;;; números, devuelva una lista de sublistas donde cada sublista
;;; corresponde a un número de la lista argumento junto con su cuadrado.
;;; Por ejemplo:
;;;
;;; > (asocia-cuadrado '(5 2 3 14))
;;; ((5 25) (2 4) (3 9) (14 196))
;;;*****************************************************************



;;;*****************************************************************
;;; 40) Definir una función que compruebe si todos los elementos de una
;;; lista simbólica, no vacía, son iguales. Por ejemplo:
;;;
;;; > (iguales '(3 3 3 3 4 3))
;;; NIL
;;; > (iguales '(3 3 3 3 3 3))
;;; T
;;;*****************************************************************



;;;*****************************************************************
;;; 41) Escribe una función (dibuja-caja n1 n2) para dibujar una caja de
;;; dimensiones especificadas por parámetros.  Por ejemplo:
;;;
;;; > (dibuja-caja 8 3)
```

```
;;; ++++++++
;;; ++++++++
;;; ++++++++
;;; NIL
;;;*****************************************************************************



;;;*****************************************************************************
;;; 42)  Definir una función (perfectos N M P) que recibiendo como
;;; entrada:

;;; - dos números naturales N y M (con N menor o igual que M)
;;; - y un símbolo de función P definido previamente como función de un
;;;   argumento numérico que devuelve T o NIL (este argumento debe ser un
;;;   argumento clave de nombre :PROPIEDAD)

;;; devuelve como salida los números perfectos que hay entre N y M
;;; (ambos inclusive) que verifiquen P. Un número se dice perfecto si es
;;; igual a la suma de todos sus divisores (distintos de él,
;;; obviamente).

;;; La salida debe ser como la que a continuación aparece en los
;;; ejemplos.

;;; Ejemplos:
;;; > (defun par (x) (= 0 (mod x 2)))
;;; PAR
;;; > (perfectos 3 200 :propiedad 'par)
;;;  El 6 es perfecto y sus divisores son 1 2 3.
;;;  El 28 es perfecto y sus divisores son 1 2 4 7 14.
;;; NIL
;;; > (perfectos 3 500 :propiedad 'par)
;;;  El 6 es perfecto y sus divisores son 1 2 3.
;;;  El 28 es perfecto y sus divisores son 1 2 4 7 14.
;;;  El 496 es perfecto y sus divisores son 1 2 4 8 16 31 62 124 248.
;;; NIL
;;; > (defun todos (x) t)
;;; TODOS
;;; > (perfectos 3 500 :propiedad 'todos)
;;;  El 6 es perfecto y sus divisores son 1 2 3.
;;;  El 28 es perfecto y sus divisores son 1 2 4 7 14.
;;;  El 496 es perfecto y sus divisores son 1 2 4 8 16 31 62 124 248.
;;; NIL
;;; > (defun otra (x) (= 0 (mod x 7)))
;;; OTRA
;;; > (perfectos 3 500 :propiedad 'otra)
;;;  El 28 es perfecto y sus divisores son 1 2 4 7 14.
;;; NIL
```

```
;;;*************************************************************************


;;;*************************************************************************
;;; 43) Definir una versión iterativa de la función subconjunto del
;;; ejercicio 6)
;;;*************************************************************************


;;;-------------------------------------------------------------------
;;; PARTE V: ARRAYS, ESTRUCTURAS Y TABLAS HASH
;;;-------------------------------------------------------------------

;;; Como en la inmensa mayoría de los lenguajes de programación, Lisp
;;; dispone de un tipo de datos array (matrices). Las principales funciones
;;; que tratan con arrays en Lisp son:

;;; - Creación: MAKE-ARRAY
;;; - Acceso a los componentes de un array: AREF
;;; - Modificación de los componentes de un array: (SETF (AREF ...) ...)
;;; - Dimensiones de una matriz: ARRAY-DIMENSIONS

;;; Las estructuras en Lisp permiten agrupar en un tipo de datos compuesto
;;; diversos datos, al estilo de los registros de una base de datos. Las
;;; principales funciones que tratan con estructuras son:

;;; - Definición del patrón de la estructura: DEFSTRUCT
;;; - El acceso a los campos de una estructura se realiza a través de los
;;;   mismos nombres de los campos precedidos de un prefijo que por defecto
;;;   es el nombre de la estructura.
;;; - La modificación de los valores de los campos de una estructura se
;;;   realiza con (SETF (<NOMBRE-ACCESOR-CAMPO> ...) ....).

;;; Por último las tablas hash nos permiten almacenar eficientemente
;;; parejas (CLAVE VALOR) sin necesidad de saber el tamaño de la tabla de
;;; antemano y con tiempos de acceso y modificación casi constantes (cuando
;;; hay un número suficiente de entradas). Las funciones principales que
;;; tratan con tablas hash son:

;;; - Construcción: MAKE-HASH-TABLE
;;; - Acceso al valor asociado a una clave: GETHASH
;;; - Insertar nuevas entradas o modificar existentes: (SETF (GETHASH ...)..)
;;; - Borrado de una entrada: REMHASH

;;; Consultar el material de Lisp para más detalles sobre todas estas
;;; funciones.
```

```
;;;*************************************************************************
;;; 44) La multiplicación escalar de un número k por una matriz A=(a_ij) es
;;; la matriz k.A= (k.a_ij). Definir una función (lista-suma-escalar A k)
;;; que recibiendo como entrada un número k>0 y una matriz A, devuelva la
;;; lista (1.A 1.A+2.A 1.A+2.A+3.A ..... 1.A+2.A+...+k.A). Por
;;; ejemplo:
;;;
;;; > (defparameter *a* (make-array '(3 3)
;;;                         :initial-contents '((1 0 0) (0 1 0) (0 0 1))))
;;; *A*
;;; > (lista-suma-escalar 6 *a*)
;;; (#2A((1 0 0) (0 1 0) (0 0 1)) #2A((3 0 0) (0 3 0) (0 0 3))
;;;  #2A((6 0 0) (0 6 0) (0 0 6)) #2A((10 0 0) (0 10 0) (0 0 10))
;;;  #2A((15 0 0) (0 15 0) (0 0 15)) #2A((21 0 0) (0 21 0) (0 0 21)))
;;;*************************************************************************



;;;*************************************************************************
;;; 45) Decimos que el elemento a(i,j) de una matriz A es un punto de silla
;;; si es el máximo de los elementos de la fila i y el mínimo de los
;;; elementos de la columna j. Es posible probar que una matriz cuyos
;;; elementos son todos distintos tiene un único punto de silla.
;;;
;;; Definir una función SILLA que recibiendo como entrada una matriz A
;;; cuadrada de números naturales distintos, devuelva el par (i j) tal que
;;; el elemento a(i,j) es un punto de silla de A. Ejemplos:
;;;
;;; > (defparameter *b*
;;;    (make-array '(3 3) :initial-contents '((1 2 3) (4 5 6) (7 8 9))))
;;; *B*
;;; > (silla *b*)
;;; (0 2)
;;; > (defparameter *c* (make-array '(4 4) :initial-contents
;;;                 '((1 4 3 2) (9 8 7 6) (5 10 11 13) (12 14 15 16))))
;;; *C*
;;; > (silla *c*)
;;; (0 1)
;;;*************************************************************************



;;;*************************************************************************
;;; 46) Definir una función CICLO-MATRIZ que recibiendo como entrada una
;;; matriz almacenada en una variable A:
;;;
;;; /                   \
```

```
;;; | a(1)(1) ... a(1)(n) |
;;; | ..................... |
;;; | ..................... |
;;; | a(m)(1) ... a(m)(n) |
;;; \                     /
;;;
;;; tenga como efecto colateral que una vez que se ejecute (CICLO-MATRIZ A)
;;; el valor de A queda cambiado de la siguiente manera:
;;;
;;; /                       \
;;; | a(m)(1) ..... a(m)(n) |
;;; | a(1)(1) ..... a(1)(n) |
;;; | ..................... |
;;; | ..................... |
;;; | a(m-1)(1) ... a(m-1)(n) |
;;; \                       /
;;;
;;; NOTA: el valor que devuelva (CICLO-MATRIZ A) no es relevante. Lo
;;; importante es que después de ejecutarse el valor de A cambie como se ha
;;; indicado.
;;;
;;; Usando CICLO-MATRIZ, definir una función CICLO-MATRIZ-N que recibiendo
;;; como entrada un número natural n>=0, y una variable A que almacena una
;;; matriz, aplique a la matriz A la función CICLO-MATRIZ n veces (si n es
;;; 0, A no cambia).
;;;
;;; Ejemplos:
;;; > (defparameter *mat1*
;;;     (make-array '(3 3) :initial-contents '((1 2 3) (4 5 6) (7 8 9))))
;;; *MAT1*
;;; > (ciclo-matriz *mat1*)
;;; ...
;;; > *mat1*
;;; #2A((7 8 9) (1 2 3) (4 5 6))
;;; > (defparameter *mat2*
;;;    (make-array '(4 4) :initial-contents '((1 1 1 1) (2 2 2 2)
;;;                                           (3 3 3 3) (4 4 4 4))))
;;; *MAT2*
;;; > (ciclo-matriz-n 3 *mat2*)
;;; ...
;;; > *mat2*
;;; #2A((2 2 2 2) (3 3 3 3) (4 4 4 4) (1 1 1 1))
;;; > (ciclo-matriz-n 3 *mat2*)
;;; ...
;;; > *mat2*
;;; #2A((3 3 3 3) (4 4 4 4) (1 1 1 1) (2 2 2 2))
;;;*************************************************************************
```

```
;;;*************************************************************************
;;; 47) Definir una función APLICA-SUMA que recibiendo como entrada una
;;; matriz numérica cuadrada y una lista de símbolos de función (f_1
;;; ... f_n), devuelva la suma de los resultados de aplicar cada f_i a b_i
;;; para 1 <= i <= n, (donde b_i es la suma de los elementos de la columna
;;; i) mas la suma de aplicar cada f_i a c_i, para 1 <= i <= n, (donde c_i
;;; es la suma de los elementos de la fila i). Cada f_i es una función
;;; numérica de un argumento. Ejemplos:
;;;
;;; > (defparameter *a1*
;;;     (make-array '(3 3) :initial-contents '((1 1 1) (2 2 2) (1 6 13))))
;;; *A1*
;;; > (aplica-suma *a1* (list 'sqrt 'sqrt 'sqrt))
;;; 17.65367
;;; > (defparameter *a2*
;;;       (make-array '(2 2) :initial-contents '((1 2) (3 4))))
;;; *A2*
;;; > (aplica-suma  *a2* (list 'sin 'sqrt))
;;; 4.4795585
;;;*************************************************************************


;;;*************************************************************************
;;; 48) Definir una función (copia-tabla tabla) que haga una copia de una
;;; tabla HASH dada. Ejemplo:
;;;
;;; > (defparameter *t* (make-hash-table))
;;; *T*
;;; > (setf (gethash 'a *t*) 1 (gethash 'b *t*) 2
;;;         (gethash 'c *t*) 3 (gethash 'd *t*) 4)
;;;
;;; > (defparameter *t-copia* (copia-tabla *t*))
;;; *T-COPIA*
;;; ;; Modificamos *t-copia*
;;; > (setf (gethash 'a *t-copia*) 8)
;;; ;; Y *t-copia* queda modificado pero *t* no.
;;; > *t*
;;; #S(HASH-TABLE EQL (D . 4) (C . 3) (B . 2) (A . 1))
;;; > *t-copia*
;;; #S(HASH-TABLE EQL (A . 8) (B . 2) (C . 3) (D . 4))
;;;*************************************************************************


;;;*************************************************************************
;;; 49) Definir una función (filtra-tabla tabla pred) tal que dada una
;;; tabla y un predicado binario, construya una nueva tabla con las
```

```
;;; entradas (CLAVE VALOR) de la tabla original que verifiquen pred. Por
;;; ejemplo:
;;;
;;; > (defparameter *t* (make-hash-table))
;;; *T*
;;; > (setf (gethash 'a *t*) 1 (gethash 'b *t*) 2
;;;         (gethash 'c *t*) 3 (gethash 'd *t*) 4)
;;;
;;; > (defparameter *t-filtrada*
;;;                 (filtra-tabla *t* #'(lambda (k v) (evenp v))))
;;; *T-FILTRADA*
;;; > *t-filtrada*
;;; #S(HASH-TABLE EQL (B . 2) (D . 4))
;;;*****************************************************************************



;;;*****************************************************************************
;;; 50) Definir una función (tiene-valor? clave tabla) que recibe una clave
;;; y una tabla hash y comprueba si existe algún valor asignado a esa clave
;;; en la tabla. Ejemplo:
;;;
;;; > (defparameter *t* (make-hash-table))
;;; *T*
;;; > (setf (gethash 'a *t*) 1 (gethash 'b *t*) 2
;;;         (gethash 'c *t*) nil (gethash 'd *t*) 4)
;;;
;;; > (tiene-valor? 'a *t*)
;;; T
;;; > (tiene-valor? 'h *t*)
;;; NIL
;;; > (tiene-valor? 'c *t*)
;;; T
;;;*****************************************************************************



;;;*****************************************************************************
;;; 51) Representar la agenda de un teléfono móvil mediante una estructura
;;; con dos campos: uno para almacenar una tabla hash con el teléfono de
;;; cada persona y el grupo al que pertenece (si existe) y otro campo para
;;; incluir la lista de grupos existentes. Definir también una función de
;;; impresión para dicha estructura. Por defecto la lista de grupos es
;;; (amigos familia clientes trabajo otros).  Ejemplo:
;;;
;;; > (defparameter *agenda* (make-agenda))
;;; *AGENDA*
;;; > *agenda*
;;; Agenda:
```

```
;;; Grupos:
;;;   (AMIGOS FAMILIA CLIENTES TRABAJO OTROS)
;;;*****************************************************************************



;;;*****************************************************************************
;;; 52) Definir funciones que permitan incluir teléfonos, actualizarlos,
;;; borrar entradas y añadir nuevos grupos a una agenda del tipo de la
;;; definida en el ejercicio 51). En concreto:

;;; - (inserta-entrada nomb datos agenda) que añade una nueva entrada
;;;   correspondiente al nombre nomb con los datos dados. Si hubiera ya una
;;;   entrada con ese nombre, sustituye la información por los nuevos
;;;   datos.
;;; - (borra-entrada nomb agenda) que borra toda la información de la
;;;   persona nomb en la agenda.
;;; - (inserta-grupo grupo agenda) que incluye un nuevo grupo en la lista
;;;   de grupos.
;;;
;;; Ejemplo:
;;;
;;; > (defparameter *agenda* (make-agenda))
;;; *AGENDA*
;;; > (inserta-entrada 'juan '(672913244 amigos) *agenda*)
;;; Agenda:
;;; JUAN 672913244 (AMIGOS)
;;; Grupos:
;;;   (AMIGOS FAMILIA CLIENTES TRABAJO OTROS)
;;;
;;; > (inserta-entrada 'maria '(632127688 familia) *agenda*)
;;; Agenda:
;;; MARIA 632127688 (FAMILIA)
;;; JUAN 672913244 (AMIGOS)
;;; Grupos:
;;;   (AMIGOS FAMILIA CLIENTES TRABAJO OTROS)
;;;
;;; > (inserta-entrada 'luis '(667321121 trabajo) *agenda*)
;;; Agenda:
;;; LUIS 667321121 (TRABAJO)
;;; MARIA 632127688 (FAMILIA)
;;; JUAN 672913244 (AMIGOS)
;;; Grupos:
;;;   (AMIGOS FAMILIA CLIENTES TRABAJO OTROS)
;;;
;;; > (inserta-entrada 'teletaxi '(954443298) *agenda*)
;;; Agenda:
;;; TELETAXI 954443298 (SIN GRUPO)
;;; LUIS 667321121 (TRABAJO)
```

```
;;; MARIA 632127688 (FAMILIA)
;;; JUAN 672913244 (AMIGOS)
;;; Grupos:
;;;   (AMIGOS FAMILIA CLIENTES TRABAJO OTROS)
;;;
;;; > (inserta-grupo 'emergencias *agenda*)
;;; Agenda:
;;; TELETAXI 954443298 (SIN GRUPO))
;;; LUIS 667321121 (TRABAJO)
;;; MARIA 632127688 (FAMILIA)
;;; JUAN 672913244 (AMIGOS)
;;; Grupos:
;;;   (EMERGENCIAS AMIGOS FAMILIA CLIENTES TRABAJO OTROS)
;;;
;;; > (inserta-entrada 'teletaxi '(954443298 emergencias) *agenda*)
;;; Agenda:
;;; TELETAXI 954443298 (EMERGENCIAS)
;;; LUIS 667321121 (TRABAJO)
;;; MARIA 632127688 (FAMILIA)
;;; JUAN 672913244 (AMIGOS)
;;; Grupos:
;;;   (EMERGENCIAS AMIGOS FAMILIA CLIENTES TRABAJO OTROS)
;;;
;;; > (borra-entrada 'luis *agenda*)
;;; Agenda:
;;; TELETAXI 954443298 (EMERGENCIAS)
;;; MARIA 632127688 (FAMILIA)
;;; JUAN 672913244 (AMIGOS)
;;; Grupos:
;;;   (EMERGENCIAS AMIGOS FAMILIA CLIENTES TRABAJO OTROS)
;;;**************************************************************************



;;;**************************************************************************
;;; 53) Definir una función (mostrar-telefonos-grupo grupo agenda) que
;;; muestre todos los teléfonos de un grupo de una agenda del tipo de la
;;; definida en el ejercicio 51). Ejemplo (con la agenda del ejercicio
;;; anterior)
;;;
;;; > (mostrar-telefonos-grupo 'familia *agenda*)
;;; Agenda(FAMILIA):
;;; MARIA 632127688 (FAMILIA)
;;;**************************************************************************



;;;--------------------------------------------------------------------
```

```
;;; PARTE VI: ENTRADA-SALIDA
;;;---------------------------------------------------------------------

;;; Un "canal" (o "stream") es un objeto Lisp que representa el lugar donde
;;; se produce la entrada y/o salida de caracteres. Usualmente, y por
;;; defecto, el canal de salida se asimila con la salida por pantalla y el
;;; de entrada con la entrada mediante teclado. Sin embargo, es posible
;;; asignar ficheros a canales para que funcionen tanto de entrada como de
;;; salida. La función OPEN abre un fichero para entrada y/o salida, a la
;;; vez que crea un canal lógico que apunta a dicho fichero físico. Una vez
;;; abierto un fichero, las operaciones de entrada y/o salida de caracteres
;;; para ese fichero han de hacerse referenciando al canal que tiene
;;; asignado. La orden CLOSE sirve para cerrar un canal.

;;; Por comodidad, es a veces muy útil usar la macro WITH-OPEN-FILE, que
;;; permite escribir bloques de código Lisp que efectúan operaciones de
;;; entrada y/o salida en un canal dado sin realizar OPEN y CLOSE
;;; explícitamente.

;;; La función básica para efectuar salida a un canal dado es FORMAT. Su
;;; sintaxis general es (FORMAT canal cadena-de-control [argumentos]*).  La
;;; cadena de control es un patrón de cadena de caracteres (conteniendo
;;; caracteres y "directivas"), que junto con los argumentos sirve para
;;; construir la cadena de caracteres que se escribirá en el canal de
;;; salida (consultar las numerosas directivas de FORMAT en el material de
;;; Lisp). En general, la cadena de caracteres que FORMAT imprime NO es su
;;; valor, sino un efecto colateral (su valor es NIL). Esto debe quedar
;;; claro, para no confundir la posible salida que tenga una función con el
;;; valor que devuelva. Sólo en el caso particular de que canal sea NIL,
;;; FORMAT no efectúa salida alguna sino que devuelve como valor la cadena
;;; de caracteres construida.  Otras funciones útiles de salida son PRINT,
;;; PRINC ó PRIN1 (consultar el manual).

;;; Las funciones de entrada más importantes son READ-LINE, READ y
;;; READ-CHAR:
;;; - En general, READ-LINE toma tres argumentos (opcionales): un canal, un
;;;   booleano que indica si devolver o no error cuando se alcance el final
;;;   de un fichero, y un tercero que indica qué devolver si se llega al
;;;   final del fichero y el segundo argumento es NIL. READ-LINE devuelve
;;;   como valor todos los caracteres del canal de entrada hasta la primera
;;;   aparición de nueva línea.
;;; - READ tiene los mismos argumentos que READ-LINE, pero devuelve como
;;;   valor la primera expresión Lisp leída del canal de entrada.
;;; - Igualmente, READ-CHAR lee el primer carácter del canal de entrada.

;;; En todos los casos, es importante remarcar que lo leído del canal de
;;; entrada es el valor devuelto por las funciones.
```

```
;;;*************************************************************************
;;; 54) Definir una función (crear-nuevo-registro agenda) que permita crear
;;; un nuevo registro de manera interactiva en una agenda del tipo de la
;;; definida en el ejercicio 51). Tanto el nombre como el número de
;;; teléfono deben introducirse por teclado. El grupo debe seleccionarse de
;;; la lista de grupos de la agenda.  En el caso de que ya exista en la
;;; agenda el nombre proporcionado por el usuario, debe escribir un mensaje
;;; de error.
;;;
;;; Ejemplo:
;;;
;;; > (defparameter *agenda* (make-agenda))
;;; *AGENDA*
;;; > (crea-nuevo-registro *agenda*)
;;; Nombre: Juan
;;; Telefono: 664567831
;;; Grupo?(S/N): s
;;; 1) AMIGOS
;;; 2) FAMILIA
;;; 3) CLIENTES
;;; 4) TRABAJO
;;; 5) OTROS
;;; Elige una opcion (1-5): 1
;;; Agenda:
;;; JUAN 664567831 (AMIGOS)
;;; Grupos:
;;;   (AMIGOS FAMILIA CLIENTES TRABAJO OTROS)
;;;
;;; > (crea-nuevo-registro *agenda*)
;;; Nombre: Luis
;;; Telefono: 634212110
;;; Grupo?(S/N): n
;;; Agenda:
;;; LUIS 634212110 (SIN GRUPO))
;;; JUAN 664567831 (AMIGOS)
;;; Grupos:
;;;   (AMIGOS FAMILIA CLIENTES TRABAJO OTROS)
;;;
;;; > (crea-nuevo-registro *agenda*)
;;; Nombre: Antonio
;;; Telefono: 611739969
;;; Grupo?(S/N): s
;;; 1) AMIGOS
;;; 2) FAMILIA
;;; 3) CLIENTES
;;; 4) TRABAJO
;;; 5) OTROS
;;; Elige una opcion (1-5): 9
```

;;; Elige una opcion (1-5): 213
;;; Elige una opcion (1-5): 3
;;; Agenda:
;;; ANTONIO 611739969 (CLIENTES)
;;; LUIS 634212110 (SIN GRUPO))
;;; JUAN 664567831 (AMIGOS)
;;; Grupos:
;;;   (AMIGOS FAMILIA CLIENTES TRABAJO OTROS)
;;; > (crea-nuevo-registro *agenda*)
;;; Nombre: Luis
;;; ERROR: entrada ya existente
;;; NIL
;;;**

;;;**
;;; 55) Definir una función (editar-entrada nombre agenda) que permita
;;; editar de manera interactiva la entrada correspondiente a un nombre en
;;; una agenda del tipo de la definida en el ejercicio 51). Si el nombre no
;;; correspondiese a ninguna entrada de la agenda se devolverá un error.
;;; Ejemplo:
;;;
;;; > *agenda*
;;; Agenda:
;;; ANTONIO 611739969 (CLIENTES)
;;; LUIS 634212110 (SIN GRUPO))
;;; JUAN 664567831 (AMIGOS)
;;; Grupos:
;;;   (AMIGOS FAMILIA CLIENTES TRABAJO OTROS)
;;;
;;; > (editar-entrada 'Antonio *agenda*)
;;;   1. Nuevo nombre
;;;   2. Nuevo numero
;;;   3. Nuevo grupo
;;; Elija una opcion: 1
;;; Nombre: Toni
;;; Agenda:
;;; TONI 611739969 (CLIENTES)
;;; LUIS 634212110 (SIN GRUPO))
;;; JUAN 664567831 (AMIGOS)
;;; Grupos:
;;;   (AMIGOS FAMILIA CLIENTES TRABAJO OTROS)
;;;
;;; > (editar-entrada 'Luis *agenda*)
;;;   1. Nuevo nombre
;;;   2. Nuevo numero
;;;   3. Nuevo grupo
;;; Elija una opcion: 2

```
;;; Número:  661214965
;;; Agenda:
;;; TONI 611739969 (CLIENTES)
;;; LUIS 661214965 (SIN GRUPO))
;;; JUAN 664567831 (AMIGOS)
;;; Grupos:
;;;   (AMIGOS FAMILIA CLIENTES TRABAJO OTROS)
;;;
;;; > (editar-entrada 'Juan *agenda*)
;;;   1. Nuevo nombre
;;;   2. Nuevo numero
;;;   3. Nuevo grupo
;;; Elija una opcion: 3
;;; Elige grupo:
;;; 1) AMIGOS
;;; 2) FAMILIA
;;; 3) CLIENTES
;;; 4) TRABAJO
;;; 5) OTROS
;;; Elige una opcion (1-5): 4
;;; Agenda:
;;; TONI 611739969 (CLIENTES)
;;; LUIS 661214965 (SIN GRUPO))
;;; JUAN 664567831 (TRABAJO)
;;; Grupos:
;;;   (AMIGOS FAMILIA CLIENTES TRABAJO OTROS)
;;;
;;; > (editar-entrada 'Alberto *agenda*)
;;; ERROR: entrada inexistente
;;; NIL
;;;**************************************************************************



;;;**************************************************************************
;;; 56) Definir una función (crear-grupo agenda) que permita introducir un
;;; nuevo grupo en la lista de grupos de la agenda.  Ejemplo:
;;;
;;; > *agenda*
;;; Agenda:
;;; TONI 611739969 (CLIENTES)
;;; LUIS 661214965 (SIN GRUPO))
;;; JUAN 664567831 (TRABAJO)
;;; Grupos:
;;;   (AMIGOS FAMILIA CLIENTES TRABAJO OTROS)
;;;
;;; > (crea-nuevo-grupo *agenda*)
;;; Nombre del nuevo grupo: proveedores
;;; Agenda:
```

```
;;; TONI 611739969 (CLIENTES)
;;; LUIS 661214965 (SIN GRUPO))
;;; JUAN 664567831 (TRABAJO)
;;; Grupos:
;;;   (PROVEEDORES AMIGOS FAMILIA CLIENTES TRABAJO OTROS)
;;;
;;; > (editar-entrada 'Luis *agenda*)
;;;   1. Nuevo nombre
;;;   2. Nuevo numero
;;;   3. Nuevo grupo
;;; Elija una opcion: 3
;;; Elige grupo:
;;; 1) PROVEEDORES
;;; 2) AMIGOS
;;; 3) FAMILIA
;;; 4) CLIENTES
;;; 5) TRABAJO
;;; 6) OTROS
;;; Elige una opción (1-6): 1
;;; Agenda:
;;; TONI 611739969 (CLIENTES)
;;; LUIS 661214965 (PROVEEDORES)
;;; JUAN 664567831 (TRABAJO)
;;; Grupos:
;;;   (PROVEEDORES AMIGOS FAMILIA CLIENTES TRABAJO OTROS)
;;;**************************************************************************



;;;**************************************************************************
;;; 57) Definir una función aspa que recibe como entrada dos parámetros
;;; clave :caracter y :altura, el primero de ellos para designar a una
;;; cadena con un sólo carácter y el segundo para designar un número
;;; natural impar mayor o igual que 3. El efecto de la función es escribir
;;; en pantalla una cruz en forma de aspa construida con el carácter dado.
;;; Por defecto el parámetro :caracter es "o" Ejemplos:
;;;
;;; > (aspa :altura 7)
;;; o     o
;;;  o   o
;;;   o o
;;;    o
;;;   o o
;;;  o   o
;;; o     o
;;; NIL
;;; > (aspa :altura 17 :caracter "X")
;;; X               X
;;;  X             X
```

```
;;;  X         X
;;;   X       X
;;;    X      X
;;;     X    X
;;;      X   X
;;;       X X
;;;        X
;;;       X X
;;;      X   X
;;;     X     X
;;;    X       X
;;;   X         X
;;;  X           X
;;; X             X
;;; NIL
;;;*****************************************************************************
```

```
;;;*****************************************************************************
;;; 58) Definir la función (escribe-arbol-horizontal l) tal que recibiendo
;;; una lista l que representa una expresión aritmética en forma prefija,
;;; la escribe por pantalla en forma de árbol (en horizontal).
;;;
;;; Ejemplo:
;;;
;;; > (escribe-arbol-horizontal '(+ (* 3 a) (- b 2) (* (- b) (/ c (* d d)))))
;;; +
;;;     *
;;;         3
;;;         A
;;;     -
;;;         B
;;;         2
;;;     *
;;;         -
;;;             B
;;;         /
;;;             C
;;;             *
;;;                 D
;;;                 D
;;;*****************************************************************************
```

```
;;;*****************************************************************************
;;; 59) Definir escribe-arbol-horizontal-lineas, una versión mejorada de
```

```
;;; escribe-arbol-horizontal (ejercicio anterior) de manera que se escriban
;;; líneas que conecten cada nodo del árbol con sus hijos.
;;;
;;; Por ejemplo:
;;;
;;; > (escribe-arbol-horizontal-lineas
;;;         '(+ (* 3 a) (- b 2) (* (- b) (/ c (* d d)))))
;;; +
;;; |-----*
;;; |     |-----3
;;; |     |-----A
;;; |------
;;; |     |-----B
;;; |     |-----2
;;; |-----*
;;; |     |------
;;; |     |     |-----B
;;; |     |-----/
;;; |     |     |-----C
;;; |     |     |-----*
;;; |     |     |     |-----D
;;; |     |     |     |-----D
;;; NIL
;;;*****************************************************************************
```

```
;;;*****************************************************************************
;;; 60) Definir una función (mi-cat fichero) similar al comando cat de
;;; unix. Es decir, escribe por pantalla el contenido de un fichero de
;;; texto.
;;;*****************************************************************************
```

```
;;;*****************************************************************************
;;; 61) Definir una función (mi-grep cadena fichero) similar al comando
;;; grep de unix (sin uso de patrones). Es decir, escribe por pantalla las
;;; líneas de fichero en las que ocurre cadena (junto con el número de
;;; línea).
;;;
;;; Por ejemplo, si buscamos la cadena "iterativa" en un fichero en el que
;;; estuvieran los 50 primeros ejercicios de Lisp anteriores, obtenemos:
;;;
;;; > (mi-grep "iterativa" "ejercicios-lisp-resueltos-50.lsp")
;;; Linea 803: ;;; 31) Definir una versión iterativa de la función longitud del ejercicio
;;;                          ^^^^^^^^
;;; Linea 810: ;;; 32) Definir una versión iterativa de la función pertenece del ejercicio
;;;                          ^^^^^^^^
```

```
;;; Linea 824: ;;; 34) Definir una versión iterativa de la función elimina del ejercicio
;;;                              ^^^^^^^^^
;;; Linea 831: ;;; 35) Definir un versión iterativa de la función sustituye del ejercicio
;;;                              ^^^^^^^^^
;;; Linea 838: ;;; 36) Definir una función iterativa que nos permita concatenar dos
;;;                              ^^^^^^^^^
;;; Linea 852: ;;; 38) Escribir una versión iterativa de la función cuadrados del
;;;                               ^^^^^^^^^
;;; Linea 941: ;;; 43) Definir una versión iterativa de la función subconjunto del
;;;                              ^^^^^^^^^
;;;*****************************************************************************



;;;*****************************************************************************
;;; 62) Definir una función (evalua fichero) que evalua las expresiones
;;; Lisp de un fichero y muestra los valores por pantalla.  Por ejemplo, si
;;; el fichero expresiones.lsp contiene lo siguiente:
;;;
(+ 2 3)
(append '(a b c) '(d e f))
(log 2)
;;;
;;; entonces:
;;; > (evalua "expresiones.lsp")
;;; El valor de la expresion (+ 2 3) es 5
;;; El valor de la expresion (APPEND '(A B C) '(D E F)) es (A B C D E F)
;;; El valor de la expresion (LOG 2) es 0.6931472
;;;*****************************************************************************



;;;*****************************************************************************
;;; 63) Definir una función (codifica-decodifica fichero1 fichero2) que
;;; codifique un fichero de texto fichero1 cambiando cada carácter por el
;;; carácter cuyo código ascii resulta de restar 255 menos el código ascii
;;; del carácter original. El fichero resultante debe ser fichero2.  Por
;;; ejemplo, si el fichero quijote.txt contiene:
;;;
;;; En un lugar de la Mancha
;;; de cuyo nombre no quiero acordarme.
;;;
;;; entonces después de hacer
;;; > (codifica-decodifica "quijote.txt" "quijote-cod.txt")
;;; el fichero "quijote-cod.txt" contiene el fichero codificado
;;; convenientemente (no lo reproducimos aquí, ya que es ininteligible).
;;;
;;; Si ahora volvemos a hacer
;;; > (codifica-decodifica "quijote-cod.txt" "quijote-cod-cod.txt")
```

```
;;; entonces, por la propia definición de la codificación, se tiene que
;;; el fichero "quijote-cod-cod.txt" tiene exactamente el mismo
;;; contenido que "quijote.txt"
;;;*****************************************************************************



;;;*****************************************************************************
;;; 64) Supongamos que tenemos definida una estructura tabla de la
;;; siguiente manera:
;;;
(defstruct (tabla (:print-function escribe-tabla))
  nombres datos)
;;;
;;; En esta estructura, datos va a contener un array numérico representando
;;; una tabla con varias filas columnas y en nombres están los nombres de
;;; las columnas.
;;;
;;; Definir la función de impresión escribe-tabla que escribe por pantalla
;;; la tabla de datos en columnas, cada una con su nombre y con una anchura
;;; de 10. Cada número debe escribirse con dos decimales.
;;;
;;; Por ejemplo:
;;;
;;; > (make-tabla :nombres '("Haber" "Debe" "Saldo")
;;;               :datos (make-array '(6 3) :initial-element 23.456))
;;;     Haber       Debe       Saldo
;;; ------------------------------------
;;;     23.46      23.46      23.46
;;;     23.46      23.46      23.46
;;;     23.46      23.46      23.46
;;;     23.46      23.46      23.46
;;;     23.46      23.46      23.46
;;;     23.46      23.46      23.46
;;;*****************************************************************************



;;;-----------------------------------------------------------------------
;;; PARTE VII: PROGRAMACIÓN DE SEGUNDO ORDEN
;;;-----------------------------------------------------------------------

;;; Como se ha visto ya en muchos ejercicios anteriores, en Lisp las
;;; funciones son un tipo de dato más, y por tanto pueden tanto recibirse
;;; como argumento de entrada a una función como devolverse como valor.

;;; Para referirnos a una función que no tiene un nombre podemos usar una
;;; expresión lambda. Una expresión lambda es una lista que contiene el
```

;;; símbolo "lambda" seguida de una lista de parámetros más un "cuerpo" con
;;; cero o más expresiones. Por ejemplo, la expresión (LAMBDA (X Y) (* X X
;;; Y)) es la función que dados dos argumentos obtiene la multiplicación
;;; del cuadrado del primero por el segundo. Esta expresión podría usarse
;;; como dato de entrada para otra función, y también podría ser el valor
;;; que devolviera otra función.

;;; Otra manera, quizás la más frecuente, de introducir funciones en Lisp
;;; es asignándolas a un nombre (un símbolo) mediante DEFUN. Es lo que
;;; llamamos el valor funcional de un símbolo. Todo símbolo situado en la
;;; primera posición de una expresión es evaluado tomado su valor
;;; funcional.

;;; La forma especial (FUNCTION <fn>) nos devuelve el valor funcional de
;;; <fn> interpretando <fn> como si estuviera en la primera posición de una
;;; expresión. Normalmente, <fn> podría ser tanto una expresión lambda como
;;; un símbolo de función previamente definido (predefinido o con
;;; DEFUN). Usamos #' como abreviatura de FUNCTION, del mismo modo que
;;; usamos ' como abreviatura de QUOTE.

;;; La función (SYMBOL-FUNCTION S) obtiene el valor funcional de un símbolo
;;; S. Nótese la diferencia entre FUNCTION y SYMBOL-FUNCTION: por ejemplo
;;; #'+, (FUNCTION +) y (SYMBOL-FUNCTION '+) son equivalentes.

;;; Existen numerosas funciones predefinidas en Lisp que reciben como
;;; entrada otras funciones. Destacamos las siguientes:

;;; - (FUNCALL FN A1 ... AN) aplica la función FN a los argumentos
;;;   A1,..,AN.
;;; - (APPLY FN LIST-ARGS) aplica la función FN a los elementos de la lista
;;;   LIST-ARGS.
;;; - (MAPCAR FN L1 ... Ln): si FN es una función de n argumentos y cada Li
;;;   es de la forma (ai1 ... aim) entonces devuelve la lista cuyo k-ésimo
;;;   elemento es (FN a1k ... ank).
;;; - (REDUCE FN L): si FN es una función de dos argumentos y L es la lista
;;;   (a1 ... an), devuelve (FN ... (FN (FN a1 a2) a3) ... an).

;;; Estas funciones que suelen recibir otras funciones como argumentos de
;;; entrada se denominan funciones de segundo orden. Existen otras muchas
;;; funciones predefinidas de segundo orden como SOME, EVERY, REMOVE-IF,
;;; COUNT-IF, FIND-IF, POSITION-IF,... Consultar el material de Lisp para
;;; detalles sobre estas funciones.

;;;***
;;; 65) El método de búsqueda dicotómica de la raíz de una función en un
;;; intervalo consiste en:

```
;;;
;;; Supongamos que tenemos una función f definida en un intervalo [a,b]
;;; (tal que f(a) y f(b) tienen distinto signo) y un número épsilon.  Sea
;;; c=(a+b)/2.
;;;
;;; - Si b - a<épsilon, devolvemos c.
;;;
;;; - Si f(c)=0, c es la raíz buscada y devolvemos c.
;;;
;;; - Si f(a) y f(c) tienen distinto signo, repetir el proceso en el
;;;   intervalo [a,c].
;;;
;;; - Si f(b) y f(c) tienen distinto signo, repetir el proceso en el
;;;   intervalo [c,b].
;;;
;;; Definir un procedimiento DICOTOMIA que recibiendo como entrada un
;;; símbolo de función f (previamente definida como una función numérica de
;;; un argumento), y tres números a, b y épsilon tales que a<b, épsilon>0 y
;;; f(a) de distinto signo que f(b), busque una raíz de la función f en el
;;; intervalo [a,b] con un error máximo de épsilon.
;;;
;;; Ejemplos:
;;; > (defun raizdos (x) (- (* x x) 2))
;;; RAIZDOS
;;; > (dicotomia 'raizdos 1.0 4.0 0.000001)
;;; 1.4142135
;;; > (dicotomia 'sin 2.0 4.0 0.000001)
;;; 3.1415925
;;;
;;;*****************************************************************************



;;;*****************************************************************************
;;; 66) Definir una función (suma-abs-dif l1 l2) que, dadas dos listas
;;; numéricas de igual longitud l1, l2, calcule la suma de las diferencias
;;; en valor absoluto de los elementos de l1 y l2. Ejemplo:
;;;
;;; > (suma-abs-dif '(2 3 4 5) '(3 2 1 4))
;;; 6
;;;*****************************************************************************



;;;*****************************************************************************
;;; 67) Definir una función (suma-pos-dif l1 l2) que, dadas dos listas
;;; numéricas de igual longitud l1, l2, calcule la suma de las diferencias
;;; de los elementos de l1 y l2 que sean positivas.  Ejemplo:
;;;
```

```
;;; > (suma-pos-dif '(2 3 4 5) '(3 2 1 4))
;;; 5
;;;*****************************************************************************


;;;*****************************************************************************
;;; 68) Definir un procedimiento (elimina-ocurrencias x l), que reciba un
;;; dato x, y devuelva una función que al recibir una lista l, devuelva la
;;; lista resultante de eliminar en l todos los elementos de primer nivel
;;; iguales a x.  Ejemplo:
;;;
;;; > (funcall (elimina-ocurrencias 1) '(1 2 (1) 3 1 5))
;;; (2 (1) 3 5)
;;;*****************************************************************************


;;;*****************************************************************************
;;; 69) Definir un procedimiento (minima-distancia l) que reciba como
;;; argumento una lista de puntos del plano, y devuelva la mínima distancia
;;; existente entre dos puntos cualesquiera de dicha lista. Ejemplo:
;;;
;;; > (minima-distancia '((0 0) (0 1) (1 0) (1 1)))
;;; 1
;;; > (minima-distancia '((0 0) (3 1) (1 0) (1 1) (-1 0)))
;;; 1
;;;*****************************************************************************


;;;*****************************************************************************
;;; 70) Una matriz numérica cuadrada es un cuadrado mágico si cumple la
;;; siguiente propiedad: las sumas de cada una de sus filas, columnas y dos
;;; diagonales principales coinciden.

;;; Representamos una matriz numérica como una lista cuyos elementos son
;;; las filas de la matriz, en forma de listas.
;;; Por ejemplo, '((2 9 4) (7 5 3) (6 1 8)).

;;; Definir un procedimiento (es-cuadrado-magico l) que reciba una matriz
;;; numérica y compruebe si es o no un cuadrado mágico.  Ejemplos:
;;;
;;; > (es-cuadrado-magico '((2 9 4) (7 5 3) (6 1 8)))
;;; T
;;; > (es-cuadrado-magico '((1 2 3) (4 5 6) (7 8 9)))
;;; NIL
;;;*****************************************************************************
```

```
;;;*************************************************************************
;;; 71) Definir una función (cuenta-segun-pred pred) que reciba un
;;; predicado, y devuelva una función que reciba una lista y calcule el
;;; número de elementos de la lista que verifican el predicado.  Por
;;; ejemplo:
;;;
;;; > (funcall (cuenta-segun-pred #'listp) '(1 2 (1) (a 1) 5))
;;; 2
;;;*************************************************************************



;;;*************************************************************************
;;; 72) Consideremos la siguiente definición:
;;;
(defun patron (f g)
  (lambda (ls) (apply f (mapcar g ls))))
;;;
;;; Obsérvese que patrón es una función que recibe dos funciones, f y g, y
;;; devuelve otra función. Esta función actúa sobre una lista ls de la
;;; forma siguiente:
;;;
;;;    * en primer lugar, se obtiene la lista que resulta de aplicar la
;;;      función g a cada elemento de ls
;;;    * luego, se le aplica la función f a los elementos de la lista
;;;      resultante anterior
;;;
;;; Usar patrón para definir las siguientes funciones:
;;;
;;; (a) La longitud de una lista.
;;; (b) La función filtra-numeros que obtiene la lista de números de una
;;;     lista dada.
;;; (c) En general, una función para filtrar los elementos de una lista que
;;;     verifican un predicado.
;;; (d) La función cuenta-segun-pred del ejercicio anterior.
;;;*************************************************************************



;;;*************************************************************************
;;; 73) Definir un procedimiento todos-verifican que reciba un predicado, y
;;; devuelva un procedimiento que reciba una lista y compruebe si todos los
;;; elementos de la lista verifican el predicado.  Ejemplo:
;;;
;;; > (funcall (todos-verifican #'listp) '(1 2 (1) (a 1) 5))
;;; NIL
;;; > (funcall (todos-verifican #'listp) '(((1 (2))) (1) (a 1)))
```

;;; T
;;;**

;;;**
;;; 74) Definir un procedimiento (linea-sudoku n) que reciba un número
;;; natural n, y devuelva un procedimiento que reciba una lista numérica y
;;; compruebe si es una posible línea de un sudoku de tamaño n (es decir,
;;; una permutación de los n primeros números naturales). Ejemplos:
;;;
;;; > (funcall (linea-sudoku 9) '(6 8 5 1 3 2 9 7 4))
;;; T
;;; > (funcall (linea-sudoku 9) '(6 8 5 1 3 12 9 7 4))
;;; NIL
;;; > (funcall (linea-sudoku 9) '(6 8 9 1 3 2 9 7 4))
;;; NIL
;;; > (funcall (linea-sudoku 9) '(6 8 9 1 3 2 7 4))
;;; NIL
;;;**

;;;**
;;; 75) Definir interseccion-m una versión de interseccion (vista en el
;;; ejercicio 9) que admita un número arbitrario de argumentos. Por
;;; ejemplo:
;;;
;;; > (interseccion-m '(1 b (a) c) '(2 b (a)) '(3 c 2 (a)) '(b (a)))
;;; ((A))
;;;**