PRACTICA 02 – COMMON LISP

```
;;; EJERCICIO 1
;;; Un número entero positivo se dice que es perfecto si la suma de
;;; sus divisores propios coincide con él mismo. Definir un predicado
;;; que, dado un entero positivo, determine si es o no un número
;;; perfecto. La función debe preguntar desde el teclado si se deben
;;; escribir o no en pantalla los divisores propios del número y
;;; actuar en consecuencia.

;;; Ejemplos:
(número-perfecto-p 6)  ; => T (ya que 6 = 1 + 2 + 3)
(número-perfecto-p 10) ; => NIL (ya que 10 /= 1 + 2 + 5)
(número-perfecto-p 28) ; => T (ya que 28 = 1 + 2 + 4 + 7 + 14)
;;; Fin de los ejemplos


;;; EJERCICIO 2
;;; Definir una función que, dados una matriz, y una fila y una
;;; columna de esa matriz, devuelva la matriz adjunta correspondiente;
;;; es decir, la matriz resultante de eliminar esas fila y columna.

;;; Ejemplos:
(defvar *M* (make-array '(4 6)
                        :initial-contents '((0 1 2 3 4 5)
                                            (6 7 8 9 10 11)
                                            (12 13 14 15 16 17)
                                            (18 19 20 21 22 23))))
; ==> *M* [#2A((0 1 2 3 4 5) (6 7 8 9 10 11) (12 13 14 15 16 17) (18 19 20 21 22 23))]
(defvar *M* nil)
; ==> *M* [#2A((0 1 2 3 4 5) (6 7 8 9 10 11) (12 13 14 15 16 17) (18 19 20 21 22 23))]
(defparameter *M-adjunta* (crear-matriz-adjunta *M* 0 0))
; ==> *M-ADJUNTA* [#2A((7 8 9 10 11) (13 14 15 16 17) (19 20 21 22 23))]
(defparameter *M-adjunta* (crear-matriz-adjunta *M* 2 2))
; ==> *M-ADJUNTA* [#2A((0 1 3 4 5) (6 7 9 10 11) (18 19 21 22 23))]
;;; Fin de los ejemplos


;;; EJERCICIO 3
;;; Las notas obtenidas a lo largo de un cuatrimestre por los alumnos
;;; de una asignatura de una titulación universitaria se guardan en
;;; una tabla hash en la que cada una de las entradas tiene por clave
;;; el nombre de uno de los alumnos y por valor una lista con las
;;; notas obtenidas por ese alumno.

;;; Definir una función que, dados una tabla hash del tipo anterior y
;;; la operación a realizar con las notas para calcular la nota final
;;; (como argumento clave y con valor por defecto la suma), devuelva
;;; una nueva tabla hash en la que cada una de las entradas tenga por
```

```
;;; clave el nombre de uno de los alumnos y por valor la nota final
;;; obtenida por el alumno (0 si no tiene ninguna nota).

;;; Ejemplos:
(defvar *notas-IA1*) ; ==> *NOTAS-IA1*
(setf *notas-IA1* (make-hash-table :test 'equal))
; ==> #<HASH-TABLE :TEST EQUAL :COUNT 0 ...>
(loop for alumno in '("José" "María" "Jesús") and
   lista-de-notas in '((1.0 2.0 3.0 4.0) (5.0 5.0) ())
   do (setf (gethash alumno *notas-IA1*) lista-de-notas))
; ==> NIL
(calcular-notas-finales *notas-IA1*)
; ==> #<HASH-TABLE :TEST EQUAL :COUNT 3 ...>
(calcular-notas-finales *notas-IA1* :calcular-nota-final #'max)
; ==> #<HASH-TABLE :TEST EQUAL :COUNT 3 ...>
;;; Fin de los ejemplos


;;; EJERCICIO 4
;;; Supongamos que tenemos un tablero infinito en el que las
;;; coordenadas de las casillas son números enteros. En él colocamos
;;; una serie de robots que recorrerán el tablero solamente avanzando
;;; y girando sobre sí mismos.

;;; Definir, usando estructuras, un nuevo tipo de dato que represente
;;; un robot colocado en el tablero y dos funciones que, aplicadas a
;;; uno de estos robots, hagan que avance y que gire, respectivamente.

;;; Ejemplos:
(defparameter *robot1*
  (colocar-robot :coordenada-x 0 :coordenada-y 0))
; ==> *ROBOT1* [Robot en (0, 0) mirando al este]
(avanzar-robot *robot1*) ; ==> Robot en (1, 0) mirando al este
(avanzar-robot *robot1* 5) ; ==> Robot en (6, 0) mirando al este
(girar-robot *robot1* :sentido-agujas-reloj t)
; ==> Robot en (6, 0) mirando al sur
(defparameter *robot2* (clonar-robot *robot1*))
; ==> *ROBOT2* [Robot en (6, 0) mirando al sur]
(avanzar-robot *robot1*) ; ==> Robot en (6, -1) mirando al sur
(girar-robot *robot2* :sentido-agujas-reloj nil)
; ==> Robot en (6, 0) mirando al este
;;; Fin de los ejemplos


;;; EJERCICIO 5

;;; El 8-puzle es un tablero cuadrado con nueve posiciones, de las
;;; cuales 8 están rellenas con una baldosa numerada y la restante
;;; está hueca. Cualquier baldosa adyacente al hueco puede deslizarse
```

```
;;; hacia él, creando una nueva posición hueca. El objetivo del puzle
;;; consiste en, comenzando a partir de una configuración arbitraria
;;; de baldosas, deslizar estas hasta llegar a la siguiente
;;; configuración:

;;;                         +---+---+---+
;;;                         | 1 | 2 | 3 |
;;;                         +---+---+---+
;;;                         | 8 |   | 4 |
;;;                         +---+---+---+
;;;                         | 7 | 6 | 5 |
;;;                         +---+---+---+

;;; A continuación se definen los parámetros y funciones que permiten
;;; realizar búsquedas ciega sobre este problema.

;;; Abre y compila los ficheros auxiliares-busqueda.lisp y
;;; b-anchura.lisp, realiza una búsqueda en anchura evaluando la
;;; expresión (busqueda-en-anchura) y corrige los errores que hay en
;;; el código hasta que la búsqueda sea capaz de encontrar una
;;; solución al problema.

(defparameter *estado-inicial* #2A((2 8 3) (1 6 4) (7 H 5)))

(defun es-estado-final (estado)
  (eq estado #2A((1 2 3) (8 H 4) (7 6 5))))

(defstruct (operador (:constructor crea-operador)
                     (:print-function
                      (lambda (operador canal profundidad)
                          (format canal
                                  (cond ((= (operador-abajo operador) -1) "arriba")
                                      ((= (operador-abajo operador) 1) "abajo")
                                      ((= (operador-derecha operador) -1) "izquierda")
                                      (t "derecha")))))))
  abajo derecha)

(defparameter *operadores*
  (list (crea-operador :abajo 0 :derecha -1)
        (crea-operador :abajo 0 :derecha 1)
        (crea-operador :abajo -1 :derecha 0)
        (crea-operador :abajo 1 :derecha 0)))

(defun copiar-estado (estado)
  (let ((nuevo-estado (make-array '(3 3))))
    (loop for i below 3
       do (loop for j below 3
             do (setf (aref nuevo-estado i j)
                      (aref estado i j))))))
```

```
(defun buscar-hueco (estado)
  (loop named búsqueda for i below 3
     do (loop for j below 3
              do (when (eq (aref estado i j) 'H)
                   (return-from búsqueda (cons i j))))))

(defun aplica (operador estado)
  (let ((nuevo-estado (copiar-estado estado))
        (coordenadas-hueco (buscar-hueco estado))
        (fila-hueco (car coordenadas-hueco))
        (nueva-fila-hueco
         (+ fila-hueco (operador-abajo operador)))
        (columna-hueco (cdr coordenadas-hueco))
        (nueva-columna-hueco
         (+ columna-hueco (operador-derecha operador))))
    (when (and (<= 0 nueva-fila-hueco 2)
               (<= 0 nueva-columna-hueco 2))
      (setf (aref nuevo-estado fila-hueco columna-hueco)
            (aref nuevo-estado nueva-fila-hueco nueva-columna-hueco))
      (setf (aref nuevo-estado nueva-fila-hueco nueva-columna-hueco)
            'H)
      nuevo-estado)))


;;; EJERCICIO 6
;;; Trabajamos de nuevo con el problema del 8 puzle del ejercicio
;;; anterior. Realiza todas las búsquedas estudiadas en teoría,
;;; definiendo los elementos necesarios para las búsquedas informadas.

;;; Considera también la versión del 8 puzle donde la configuración
;;; final es:

;;;                          +---+---+---+
;;;                          |   | 1 | 2 |
;;;                          +---+---+---+
;;;                          | 3 | 4 | 5 |
;;;                          +---+---+---+
;;;                          | 6 | 7 | 8 |
;;;                          +---+---+---+
```