PRACTICA 1. PROGRAMACIÓN EN **LISP**

Fecha de publicación: 02 de noviembre de 2011.
Versión: 2011/11/02

Forma de entrega: **Se realizará una entrega única al final del cuatrimestre por correo electrónico según las normas que se publicarán en la página web**: http://arantxa.ii.uam.es/~fdiez/docencia/11-12/IAextincion.html

- El código debe estar en un único fichero.
- La evaluación del código en el fichero no debe dar errores (recuerda, al hacer CTRL+A CTRL+E debe evaluar todo el código sin errores).

Material recomendado:

- En cuanto a estilo de programación LISP:
- http://www.cs.cmu.edu/Groups/AI/html/faqs/lang/lisp/part1/faq.html
- Como referencia de funciones LISP: http://www.lispworks.com/documentation/common-lisp.html

Ayuda:

- Se recomienda leer atentamente el documento Errores más frecuentes. Se penalizarán especialmente los errores que se cometan en la práctica aquí recogidos.
- En las prácticas usaremos la versión w IDE, ANSI ya que incluye el entorno de desarrollo con editor de textos y, por ser ANSI, no es sensible a mayúsculas o minúsculas.
- No utilizaremos las ventanas "Form" e "Inspect"
- Te será útil familiarizarte con las herramientas de *histórico de evaluaciones* que incluye la ventana "Debug Window".
- Si te pones en una línea que ya evaluaste en la "Debug Window" y pulsas Enter, se copiará automáticamente en la línea de *prompt* actual.
- Usa Ctrl+E para evaluar la línea sobre la que está el cursor en una ventana de edición.
- Usa Ctrl+. para autocompletar la función que estás escribiendo.
- Con Ctrl+Shift+^; (también en el menú "Edit") puedes poner en comentario (o descomentar) la fracción de código seleccionada en el editor.
- Con Ctrl+Shift+p, puedes tabular automáticamente el código seleccionado en formato estándar.
- Si escribes una función que esté definida (aunque sea tuya) y añades un espacio, en la esquina inferior izquierda de la ventana principal (en la que están Nuevo, Guardar, Cargar,...) podrás ver el prototipo y los parámetros que recibe.
- Si seleccionas un nombre de función y pulsas F1 se abrirá automáticamente la ayuda del entorno acerca de esa función.
- Si vas al menú "Run" verás que puedes trazar o poner un *breakpoint* en una función cuyo nombre tienes seleccionado. La traza la escribe el entorno en la "Debug Window" mientras que los *breakpoint* lanzan una ventana de mensaje y paran la evaluación momentáneamente.

OBJETIVO: El objetivo de esta práctica es la resolución de problemas sencillos mediante el lenguaje de programación LISP. Aparte de la corrección de los programas se valorará la utilización de un enfoque de programación funcional.

NOTA: Las funciones que se muestren por primera vez en el enunciado aparecerán en **negrita**. Asegúrate de parar ante cada una de ellas y repasar su funcionamiento en detalle mediante el enlace del manual de referencia de funciones LISP (indicado arriba).

REGLAS DE ESTILO LISP

Codifica utilizando estas reglas. Si tu código no las sigue, la valoración podría ser inferior a la máxima.

Adaptado de [http://www.cs.cmu.edu/Groups/AI/html/faqs/lang/lisp/part1/faq-doc-4.html]

- No escribas código C en LISP.
- Funciones
  - Escribe funciones cortas que realicen una única operación bien definida.
  - Utiliza nombres de funciones y variables que sean descriptivos. Es decir, que ilustren para qué son utilizadas y sirvan así de documentación.
- Formato
  - Utiliza adecuadamente la sangría.
  - No escribas líneas de más de 80 caracteres.
  - Utiliza los espacios en blanco para separar segmentos de código, pero no abuses de ellos

```
ERRÓNEO 1:
          (defun foo(x y)(let((z(+ x y 10)))(* z z)))

ERRÓNEO 2:
         (defun foo ( x  y )
           (let ( ( z (+ x y 10) ) )
             ( * z z )
             )
           )

CORRECTO:
          (defun foo (x y)
           (let ((z (+ x y 10)))
             (* z z)))
```

- Recomendaciones para codificar
  - No utilices `EVAL`.
  - Utiliza `APPLY` y `FUNCALL` sólo cuando sea necesario.
  - Utiliza recursión.
  - Aprende a usar funciones propias de LISP:
    - `MAPCAR` y `MAPCAN` (operaciones en paralelo)
    - `REDUCE` (recursión sobre los elementos de una lista)
  - No utilices (en este curso) operaciones destructivas: `NCONC, NREVERSE, DELETE`. Usa las alternativas no destructivas: `APPEND, REVERSE, REMOVE`. `MAPCAN` y la ordenación `SORT` (destructiva) puede ser utilizada con precaución.
  - No utilices funciones del tipo `C{A,D}R` con más de dos letras entre la `C` y la `R`. Son muy difíciles de leer.
  - Cuando se trata de una lista, es preferible utilizar `FIRST/REST/SECOND/THIRD` … en lugar de `CAR/CDR/CADR/CADDR` …
- Condicionales
  - Utiliza `WHEN` y `UNLESS` cuando la decisión tenga sólo una rama. En concreto `WHEN` y `UNLESS` se deben utilizar en lugar de un `IF` con 2 argumentos o un `IF` con 3 argumentos con algún argumento `NIL` o `T`.
  - Utiliza `IF` cuando la decisión tenga 2 ramas.
  - Utiliza `COND` cuando la decisión tenga una o más ramas.
  - Utiliza `COND` en lugar de `IF` y `PROGN`. En general, no utilices `PROGN` si se puede escribir el código de otra forma.

```
MAL:
  (IF (FOO X)
       (PROGN (PRINT "hi there") 23)
       34)
```

  - Utiliza `COND` en lugar de un conjunto de `IFs` anidados.
- Expresiones Booleanas

    - En caso de que debas escribir una expresión Booleana, utiliza operadores Booleanos: `AND, OR, NOT`, etc.
    - Evita utilizar condicionales para escribir expresiones Booleanas.
- Constantes y variables
    - Los nombres de variables globales deben estar en mayúsculas entre asteriscos.
      Ejemplo: `(DEFVAR *GLOBAL-VARIABLE*)`
    - Los nombres de constantes globales deben estar entre (+)
      Ejemplo: `(DEFCONSTANT +DOS+ 2)`
- En caso de que haya varias alternativas, utiliza la más específica.
    - No utilices `EQUAL` si `EQL` o `EQ` es suficiente.
    - No utilices `LET*` si es suficiente con `LET`.
    - Funciones
        - Si una función es un predicado, debe evaluar a `T/NIL`.
        - Si una función no es un predicado, debe evaluar a un valor útil.
        - Si una función no debe devolver ningún valor, por convención evaluará a `T`.
    - No utilices `DO` en los casos en los que se puede utilizar `DOTIMES` o `DOLIST`.
- Comenta el código.
    - Usa `;;;` para explicaciones generales sobre bloques de código.
    - Usa `;;` para explicaciones en línea aparte antes de una línea de código.
    - Usa `;` para explicaciones en la misma línea de código.
- Evita el uso de APPLY para aplanar listas.

```
(apply #'append list-of-lists)                ; Erróneo
```

  Utiliza REDUCE o MAPCAN en su lugar

```
(reduce #'append list-of-lists :from-end t)   ; Correcto
(mapcan #'copy-list list-of-lists)            ; Preferible
```

  Ten especial cuidado con llamadas del tipo

```
(apply f (mapcar ..))
```

ERRORES DE CODIFICACIÓN a EVITAR

1. Contemplad siempre el caso NIL como posible argumento.
2. Las funciones que codifiquéis pueden fallar, pero en ningún caso deben entrar en recursiones infinitas.
3. Cuando las funciones de LISP fallan, el intérprete proporciona un diagnóstico que debe ser leído e interpretado.

   Ejemplo:

```
>>  (rest 'a)
     Error: Attempt to take the cdr of A which is not listp.
     [condition type: TYPE-ERROR]
```

4. Haced una descomposición funcional adecuada. Puede que se necesiten funciones adicionales a las del enunciado.
5. Diseñad casos de prueba completos, no sólo los del enunciado. Describidlos antes de codificar.
6. Si ya existen, utilizad las funciones de LISP, en lugar de codificar las vuestras propias (a menos que se indique lo contrario).
7. Programad utilizando el sentido común (no intentando adivinar qué quiere el profesor). Preguntad si hay algo en el diseño de una función que no entendéis o que resulta chocante.

1ª Semana: LISP I, EJEMPLOS

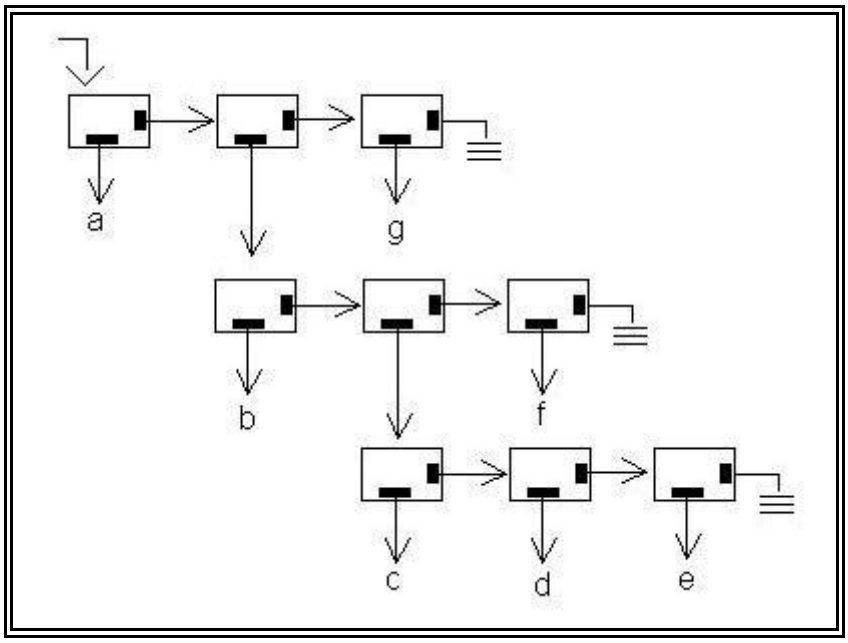
En esta primera semana vamos a realizar la evaluación del código que se facilita en cada apartado y debemos analizar, contrastar y explicar el resultado obtenido en cada caso.

EVALUACION: En los ejercicios de la primera semana sólo se deben realizar todos y resolver las dudas que surjan. **No es necesario entregar nada**. Es importante que no paséis a los ejercicios de la segunda semana sin entender realmente los conceptos básicos de esta primera.

APARTADO 1.0: ESTRUCTURA DE LOS DATOS LISTA EN LISP

La estructura de datos utilizada para almacenar tanto datos como el propio código fuente en LISP es la lista (también se pueden guardar datos de otras formas, como arrays o tablas hash, pero no van a ser utilizados para estas prácticas). Se trata de listas enlazadas. Cada nodo de la lista es una *celda cons*. Una celda cons está formada por dos 2 punteros. Uno de ellos, comúnmente llamado *car* (Contents of Address Register) apunta al dato que tenemos en esa posición de la lista. El otro, llamado *cdr* (Contents of Decrement Register), apunta al siguiente elemento de la lista (*nil* en caso de que la lista haya terminado).

A continuación puedes ver la estructura que tiene en memoria la lista `(a (b (c d e) f) g)`.



Evalúa las siguientes líneas en el intérprete de LISP (una por una).

```
milista

(setf milista nil)

milista

(setf milista '(- 6 2))

milista

(setf milista (- 6 2))
```

```
milista

(setf milista '(8 (9 10) 11))

(car milista)

(first milista)

(cdr milista)

(rest milista)

(car (cdr milista))

milista

(length milista)

(setf milista (cons 4 '(3 2)))

milista

(setf milista (list '+ 4 3 2))

milista

(setf milista (append '(4 3) '(2 1)))

milista

(reverse milista)

milista
```

RECUERDA: Debes evaluar el código anterior y poder explicar en detalle qué está pasando.

APARTADO 1.1: PROGRAMACIÓN FUNCIONAL

LISP es un lenguaje orientado hacia la programación funcional, en contraste con la programación procedural a la que estamos acostumbrados en otros lenguajes como C.
La base de la programación funcional es el uso del retorno que dan las funciones como parámetro de una función superior. Antes de evaluarse una función se evalúan todos sus parámetros de entrada de izquierda a derecha, los cuales pueden ser a su vez otras evaluaciones de funciones.
Las evaluaciones de funciones LISP deben devolver un único valor sin producir modificaciones en los parámetros recibidos. Cuando una función modifica los parámetros que recibe se dice que es una función destructiva y debe ser utilizada con cuidado (no se permite definir funciones destructivas).
En la segunda de las siguientes líneas de código LISP se evalúan una serie de funciones anidadas produciendo un resultado final en el que han sido necesarias determinadas transformaciones de una lista inicial sin que haya por ello ninguna modificación en la misma. Nótese que el programador no ha necesitado la creación de variables auxiliares para almacenar los valores intermedios.

```
(setf milista '(1 2 3 4 5 6 7 8 9 10))
(list (length milista)
      (apply #'+ (cons 100 milista))
      (mapcar #'(lambda (x y) (list x y))
        milista (reverse milista)))
(print milista)
```

RECUERDA: Evalúa en el entorno el código anterior. Observar el resultado y traza detalladamente el orden de evaluación que utiliza el intérprete LISP para obtener el resultado final.

APARTADO 1.2: EVALUACIÓN DE PARÁMETROS DE FUNCIÓN

```
1.2.1) (setf a (+ 2 3))
     (eval a)

1.2.2) (setf b '(+ 2 3))
     (eval b)

1.2.3) (setf c (quote (+ 2 3)))
     (eval c)

1.2.4) (setf d (list 'quote '(+ 2 3)))
     (eval d)
     (eval (eval d))

1.2.5) (setf d (list 'quote (+ 2 3)))
     (eval d)
     (eval (eval d))
```

IMPORTANTE: Evalúa el código anterior y explica qué está pasando en cada caso. La clave está en entender bien el funcionamiento de las nuevas funciones.

APARTADO 1.3: EFECTOS INDIRECTOS

```
1.3.1)(setf *lista-amigos* '(jaime maria julio))

    (setf *lista-amigos-masculinos* (remove 'maria *lista-amigos*))

    (print *lista-amigos*)

    (setf *lista-amigos-masculinos* (delete 'maria *lista-amigos*))

    (print *lista-amigos*)

1.3.2) (setf *dias-libres* '(domingo))

     (cons 'sabado *dias-libres*)

     (print *dias-libres*)

     (push 'sabado *dias-libres*)

     (print *dias-libres*)

     (setf *dias-libres* (cons 'viernes *dias-libres*))

     (print *dias-libres*)
```

IMPORTANTE: Evalúa el código anterior y explica qué está pasando. Centra tu atención en la destructividad o no de cada conjunto de instrucciones

APARTADO 1.4: LISTAS OBTENIDAS POR MODIFICACIÓN DE OTRAS LISTAS

```
1.4.1) (defun multiplica-2 (lista)
       (mapcar #'(lambda (x) (* 2 x)) lista))

1.4.2) (defun sublista-superiores-1 (valor lista)
        (remove nil (mapcar #'(lambda (elem)
                                (if (> elem valor) elem))
                      lista)))

1.4.3)(defun sublista-superiores-2 (valor lista)
       (mapcan #'(lambda (elem)
                   (if (> elem valor) (list elem)))
         lista))
```

MUY IMPORTANTE: Aunque se pueda obtener el mismo resultado por medio de la programación con enfoque funcional y sin él, no se aceptará programación basada en manipular el "estado" (i.e. los valores de las variables en la memoria) (es decir traducido directamente de C). No se permite utilizar bucles (DO, DOLIST, DOTIMES, LOOP, etc). Se debe programar utilizando un estilo funcional (recursión, procesamiento en paralelo con `mapcar`). Debemos seguir esta norma de programación durante TODAS las prácticas de la asignatura, a menos que se indique expresamente lo contrario.

1.4.3) [Implementación **NO FUNCIONAL - No utilizar**]

```
(defun sublista-superiores-3 (valor lista)
  (let ((retorno nil))
    (dolist (elem lista retorno)
      (if (> elem valor)
          (setf retorno (cons elem retorno))))))
```

1.4.4) [Implementación **FUNCIONAL**]

```
(defun sublista-superiores-4 (valor lista)
  (cond
   ((null lista)   lista)
   ((> (car lista) valor)
    (cons (car lista)
          (sublista-superiores-4 valor (cdr lista))))
   ( T (sublista-superiores-4 valor (cdr lista)))))

1.4.5) (setf *lista-numeros* '(1 2 3 4 5 6 7 8 9 10))

     (setf *lista-numeros-pares-superiores-5*
        (sublista-superiores-2 5 (multiplica-2 *lista-numeros*)))

      (print *lista-numeros*)
```

IMPORTANTE: Analiza la implementación de las funciones anteriores. ¿Qué ventajas e inconvenientes tiene cada una? Para el caso del apartado 4.5, analiza también las ventajas de la programación funcional respecto a una hipotética versión en código C o Java. La clave está en distinguir las distintas implementaciones y analizar las características y el funcionamiento de cada una.

APARTADO 1.5: VISIBILIDAD DE VARIABLES EN FUNCIONES

```
1.5.1)
(defun suma1a ()
  (setf a (+ a 1)))

(setf a 3)
(suma1a)
(print a)
```

```
1.5.2)
(defun suma1 (numero)
  (setf numero (+ 1 numero)))

(suma1 a)
(print a)
(print (boundp 'numero))

1.5.3)
(defun suma2 (numero)
  (setf resultado (+ 2 numero)))

(suma2 a)
(print a)

(print resultado)
```

MUY IMPORTANTE: no utilizar nunca SETF dentro de una función o dentro de un `mapcar`. En caso de que sea necesario evitar repetir código definid un cierre léxico con un `let`

1.5.4) [Implementación **NO FUNCIONAL** y mal uso de **SETF - No utilizar**]

```
(defun cuenta-elementos-1 (lista)
  (let ((contador 0))
    (dolist (elem lista contador)
       (setf contador (+ 1 contador)))))

(cuenta-elementos-1 '(1 2 3 4))
(print (boundp 'contador))
```

IMPORTANTE: Evalúa el código anterior y explica qué está pasando. Haz un examen crítico de la implementación enfocado en la visibilidad de las variables.

1.5.5) Implementación con mapcar

```
 (defun cuenta-elementos-2 (lst)
   (apply #'+ (mapcar #'(lambda(x) 1) lst)))

 (cuenta-elementos-2 '(1 2 3 4))
```

1.5.6) Implementación recursiva

```
 (defun cuenta-elementos-3 (lst)
  (if (null lst) 0 (+ 1 (cuenta-elementos-3 (rest lst)))))

 (cuenta-elementos-3 '(1 2 3 4))
```

NOTA: Utiliza la instrucción `(trace <fn>)` para ver la secuencia de evaluaciones del intérprete de LISP. Para dejar de ver dicha secuencia, utiliza `(untrace <fn>)`

APARTADO 1.6: ACCESO Y MODIFICACIÓN DE LOS ELEMENTOS DE UNA LISTA

```
1.6.1) (setf lista '(1 2 3 4 5 6 7))
     (defun elimina-segundo (lista)
       (cond
        ((and (consp lista) (> (length lista) 1))
         (setf (cdr lista) (cddr lista)))))

     (elimina-segundo lista)
     (print lista)

1.6.2) (defun modifica-elemento (lista posicion nuevo-valor)
       (cond
```

```
        ((and (consp lista) (> (length lista) posicion))
          (setf (nth posicion lista) nuevo-valor))))

(modifica-elemento lista 2 300)
(print lista)

1.6.3) (defun elimina-primero (lista)
       (cond
         ((consp lista)
          (setf lista (cdr lista)))))

     (elimina-primero lista)
     (print lista)

         1. 6.4) (setf lista (cdr lista))
                 (print lista)
```

IMPORTANTE: Evalúa el código anterior y explica qué está pasando. Haz un análisis crítico de la implementación. Comenta las ventajas e inconvenientes que puede tener la programación destructiva.

APARTADO 1.7: CREACIÓN DE LISTAS

```
1.7.1)
(defun creaLista1 (numElems incremento)
  (let (retorno)
    (dotimes (i numElems retorno)
      (setf retorno (append retorno (list (* i incremento)))))))

1.7.2)
(defun creaLista2 (numElems incremento)
  (let (retorno)
    (dotimes (i numElems (reverse retorno))
      (setf retorno (cons (* i incremento) retorno)))))

1.7.3)
(defun creaLista3 (numElems incremento &optional (desde 0))
  (unless (= 0 numElems)
    (cons desde (creaLista3 (- numElems 1)
                            incremento
                            (+ incremento desde)))))

(time (creaLista1 1000 5))
(time (creaLista2 1000 5))
(time (creaLista3 1000 5))
```

IMPORTANTE: Evalúa el código anterior y explica qué está pasando. Haz un examen crítico de la implementación. ¿Qué versión es más eficiente? ¿Por qué?

APARTADO 1.8: RECORRIDO DE UNA LISTA

RECORDATORIO: `incf` no es una función sino una macro y modifica su argumento. Una macro no es más que una regla de reescritura, a modo de un `#define` de "C". El interprete cuando ve `(incf ref delta)` automáticamente lo interpreta como `(setf ref (+ ref delta))`.

Se proponen las siguientes funciones para contar los elementos de una lista:

```
1.8.1) (defun f1 (lista)
       (let ((contador 0) (numElems (length lista)))
       (dotimes (i numElems contador)
```

```
          (incf contador i))))

(f1 '(1 2 3 4))
(f1 '(1 2 3 7))
```

```
1.8.2)
    (defun f2 (lista)
       (let ((contador 0) (num-elems (length lista)))
       (dotimes (i num-elems contador)
          (incf contador (nth i lista)))))
```

```
1.8.3)
    (defun f3 (lista)
       (let ((contador 0))
       (dolist (elem lista contador)
         (incf contador elem))))
```

MUY IMPORTANTE: **MAPCAR** es una función que realiza una transformación en paralelo sobre cada uno de los elementos de la lista o listas que toma como argumentos. No se debe usar como método de procesamiento secuencial.

1.8.4) [INCF dentro de MAPCAR - **No utilizar. ERROR GRAVE**]

```
(defun f4 (lista)
  (let ((contador 0))
   (mapcar #'(lambda (x) (incf contador x)) lista)
    contador))
```

1.8.5)

```
(defun f5 (lista)
  (cond
   ((null lista) 0)
   ( T (+ (car lista) (f5 (cdr lista))))))
```

1.8.6)

```
(defun f6 (lista)
  (apply #'+ lista))
```

1.8.7)

```
(defun f7 (lista)
  (eval (cons '+ lista)))
```

Ahora vamos a ver tres versiones para crear una lista con los números pares de otra lista de números.

1.8.8) versión RECURSIVA

```
(defun f9 (lista)
  (cond
   ((null lista) nil)
   (t (if (evenp (car lista)) (cons (car lista) (f9 (cdr lista)))
        (f9 (cdr lista))))))
```

1.8.9) versión REMOVE NIL

```
(defun f8 (lista)
  (remove nil (mapcar #'(lambda (x) (if (evenp x) x)) lista)))
```

1.8.10) versión con MAPCAN [PREFERIBLE]

```
(defun f10 (lista)
  (mapcan #'(lambda (x) (if (evenp x) (list x))) lista))
```

IMPORTANTE: Evalúa el código anterior y explica qué está pasando. Haz un análisis crítico de las distintas implementaciones. En las 5 primeras funciones recorremos la lista para sumar sus elementos. ¿Cuál es la más eficiente? En la 6ª y 7ª aplicamos la función + a la lista. ¿Cómo crees que suma los

elementos esta función? Las tres últimas versiones nos proporcionan el mismo resultado para la misma lista de números de entrada. Analiza el enfoque de cada una teniendo en cuenta sus ventajas y desventajas. Para todas ellas, utiliza las líneas de código que te proponemos más adelante para hacer las pruebas. Puedes hacer experimentos con distintas longitudes de lista para fundar mejor tu análisis.
Nota: después de un stack overflow es recomendable cerrar el entorno y volverlo a abrir.
Nota: `compile` crea la versión compilada de una función.

```
(setf milista (crealista2 10000 5))

(time (f2 milista))
(time (f3 milista))
(time (f4 milista))
(time (f5 milista))
(time (f6 milista))
(time (f7 milista))

(compile 'f3)
(time (f3 milista))

(setf milista (crealista2 1000 1))
(time (f8 milista))
(time (f9 milista))
(time (f10 milista))
```

APARTADO 1.9 [entregar, 3 puntos]:

Consideremos las siguientes implementaciones de funciones que comprueban si un determinado objeto pertenece a una lista

```
(defun our-member-equal-1 (obj lst)
  (unless (null lst)
    (or (equal (first lst) obj)
      (our-member-equal-1 obj (cdr lst)))))


(defun our-member-equal-2 (obj lst)
  (unless (null lst)
    (eval (cons 'or (mapcar #'(lambda (x) (equal obj x))
                      lst)))))
```

9.1 Escribe el pseudocódigo para la función `our-member-equal-1`.

9.2 Explica por qué la implementación realizada en `our-member-equal-1` es preferible a la implementación `our-member-equal-2`.

9.3 Considera la función

```
(defun our-member-equal-3 (obj lst)
    (or (equal (first lst) obj)
      (our-member-equal-3 obj (rest lst))))
```

Compara las evaluaciones

```
>> (our-member-equal-3 NIL NIL)
>> (our-member-equal-3 NIL '(NIL))
```

con

```
>> (our-member-equal-1 NIL NIL)
>> (our-member-equal-1 NIL '(NIL))
```

¿Por qué es incorrecta la implementación our-member-equal-3?

APARTADO 1.10: INVERTIR EL ORDEN DE LOS ELEMENTOS DE UNA LISTA

[Adaptado de "On LISP", P. Graham, disponible en la red http://www.paulgraham.com/onlisp.html]

Compara la función:

```
(defun bad-reverse (lst)
  (let* ((len (length lst))
         (ilimit (truncate (/ len 2))))
    (do ((i 0 (1+ i))
         (j (1- len) (1- j)))
        ((>= i ilimit))
      (rotatef (nth i lst) (nth j lst)))))
```

y la función

```
(defun good-reverse (lst)
    (good-reverse-aux lst nil))

(defun good-reverse-aux (lst acc)
  (if (null lst)
      acc
    (good-reverse-aux (cdr lst)
                      (cons (car lst) acc))))
```

¿Cuál es la respuesta del intérprete y cuáles son los efectos de las siguientes evaluaciones?

```
>> (setf lst '(1 2 3 4))
>> (bad-reverse lst)
>>  lst

>> (setf lst '(1 2 3 4))
>> (good-reverse lst)
>>  lst
```

Utilizando esta información, enumera las razones (hay varias) por las que `good-reverse` es correcta y `bad-reverse` no es correcta desde el punto de vista de la programación funcional.

2ª semana: LISP II, EJERCICIOS

Estos ejercicios deben ser entregados al final del cuatrimestre

Evaluación: Para evaluar el trabajo de la segunda semana se tendrá en cuenta el correcto funcionamiento del código entregado y el estilo de programación.

- No se puede utilizar `setf` o similares.
- Ninguna función debe ser destructiva.
- No se puede utilizar iteración directa (`dolist, do, dotimes`...).
- Utiliza `mapcar, mapcan` o recursión.
- No utilices `mapcar` en caso de que no sea estrictamente necesario recorrer toda la lista.
- Reutiliza código.

2.1 Secuencia de Fibonacci [1.5 puntos]

Los números de Fibonacci se generan mediante la recursión

F(0) = 0
F(1) = 1
F(n)=F(n-1)+F(n-2) si n > 1

2.1.1 Implementa de forma recursiva una función para generar los números de Fibonacci.

```
Prototipo: (defun numero-fibonacci(n)    ...)

Ejemplo:
>> (numero-fibonacci 8)
    21
```

2.1.2 Implementa de forma recursiva una función para generar la sucesión de n números de Fibonacci. Por razones de eficiencia, esta función **NO DEBE UTILIZAR la anterior**

- `(secuencia-fibonacci 0)` evalúa a `()`
- `(secuencia-fibonacci 1)` evalúa a `(0)`
- `(secuencia-fibonacci 2)` evalúa a `(0 1)`
- `(secuencia-fibonacci 3)` evalúa a `(0 1 1)`
- `(secuencia-fibonacci 4)` evalúa a `(0 1 1 2)`
- `(secuencia-fibonacci 9)` evalúa a `(0 1 1 2 3 5 8 13 21)`

```
Prototipo: (defun secuencia-fibonacci(n)    ...)

Ejemplo:
>> (secuencia-fibonacci 9)
   (0 1 1 2 3 5 8 13 21)
```

2.2 Desviación estándar [1.5 puntos]

Implementa (1) de forma recursiva y (2) con `mapcar` sendas funciones que calcule la desviación estándar de un conjunto de N valores reales almacenados en una lista

$$(x_1\ x_2\ ...\ x_N);\qquad <x> = \frac{1}{N}\sum_{n=1}^{N} x_n;\qquad stdev = \sqrt{\frac{1}{N}\sum_{n=1}^{N}\left(x_n - <x>\right)^2}$$

Nota: Puede que sea necesario definir funciones auxiliares para realizar el cálculo.

```
Prototipos: (defun stdev-mapcar (lst)...)
            (defun stdev-recursive (lst)...)
```

2.3 recursive-inverse-bubble-sort [1 punto]

Considere el siguiente código:

```
(defun coloca-primero-lst (lst)
  (unless (null lst)
    (let ((primero (first lst))
          (resto1 (coloca-primero-lst (rest lst))))
      (if (null resto1)
          (list primero)
        (let ((segundo (first resto1))
              (resto2  (rest resto1)))
          (if (< primero segundo)
              (cons primero (cons segundo resto2))
            (cons segundo (cons primero  resto2)))))))))
```

que coloca el elemento de menor tamaño en la primera posición de la lista.

Se debe implementar de forma recursiva la función `recursive-inverse-bubble-sort`, utilizando la función `coloca-primero-lst`. Observa que el primer elemento de la salida de `(coloca-primero-lst lst)` está en la posición correcta. Sólo queda ordenar el resto.

```
Prototipo:
 (defun recursive-inverse-bubble-sort (lista) ...)

Ejemplo:
>  (recursive-inverse-bubble-sort '(3 1 5 2 4))
(1 2 3 4 5)
```

2.4 Relaciones [2 puntos]

IMPORTANTE:

- El código de este apartado será de utilidad en sucesivos ejercicios.
- Considera siempre el caso de que las listas puedan ser vacías.

2.4.1 Combinar un elemento con una lista: Definir una función que combine un elemento dado con todos los elementos de una lista

```
Prototipo: (defun combina-elt-lst (elt lst) ...)

Ejemplo:
>> (combina-elt-lst 'a '(1 2 3))
   ((a 1) (a 2) (a 3))
```

2.4.2 Producto cartesiano de dos listas: Diseñar una función que calcule el producto cartesiano de dos listas.

```
Prototipo: (defun combina-lst-lst (lst1 lst2) ...)

Ejemplo:
>> (combina-lst-lst '(a b c) '(1 2))
```

```
   ((A 1) (A 2) (B 1) (B 2) (C 1) (C 2))
```

2.5 Operaciones con conjuntos [2 puntos]

Un conjunto es una colección de elementos distintos en la que el orden en que se encuentran los elementos no es importante. Se puede representar un conjunto mediante una lista, siempre y cuando las operaciones sobre la lista tengan en cuenta que el orden no es relevante.
En los ejercicios que siguen no se pueden utilizar funciones de librería LISP existentes para el manejo de conjuntos (ADJOIN, SET-DIFFERENCE, UNION, INTERSECTION **y similares). Para prácticas posteriores sí podremos utilizarlas. Se admite que, según la implementación realizada, pueda variar el orden de ordenación de los elementos en las listas que representan los conjuntos resultantes.**

2.5.1 Escribe una función que permita determinar si un elemento pertenece a un conjunto

```
Prototipo: (defun my-member-p (elt cjto) ...)
Ejemplo:
>> (my-member-p '(L 1 5.3) '((L 1 5.3) (Q) P (L 6.2) (A 3 6.8)))
T
>> (my-member-p '() '((L 1 5.3) (Q) P (L 6.2) (A 3 6.8)))
NIL
```

2.5.2 Escribe una función que determine si una lista es un conjunto, suponiendo que los objetos distintos se representan mediante un símbolo distinto

```
Prototipo: (defun my-set-p (lst) ...)
Ejemplos:
>> (my-set-p nil)
T
>> (my-set-p '(L 1 5.3))
T
>> (my-set-p '((L 1 5.3) Q (Q) P (L 6.2) (A 3 6.8)))
T
>> (my-set-p 'a)
NIL
>> (my-set-p '(a b c a))
NIL
```

2.5.3 Escribe una función que permita eliminar un elemento de un conjunto

```
Prototipo: (defun my-remove (elt lst) ...)
Ejemplo: >> (my-remove '(L 1 5.3) '( (L 1 5.3) Q (Q) (L 6.2) P (A 3 6.8)))
            (Q (Q) P (L 6.2)(A 3 6.8))
```

2.5.4 Escribe una función que permita incluir un elemento en un conjunto

```
Prototipo:  (defun my-adjoin (elt cjto) ...)
Ejemplos:
>> (my-adjoin '(L 1 5.3) '((Q) (L 6.2) P (A 3 6.8)))
   ((Q) P (L 6.2) (A 3 6.8) (L 1 5.3))
>> (my-adjoin '((C 8 8.9) A (P Q)) '((Q) (L 6.2) P (A 3 6.8)))
   ((Q) (L 6.2) P (A 3 6.8) ((C 8 8.9) A (P Q)))
```

2.5.5 Escribe una función que permita realizar la unión de dos conjuntos

```
Prototipo:  (defun my-union (cjtoa cjtob) ...)
Ejemplos:
>> (my-union '((Q) (L 6.2) P (A 3 6.8)) '())
   ((A 3 6.8) (L 6.2) P (Q))
>> (my-union '((Q) (L 6.2) P (A 3 6.8)) '(R (B 3 6.8) (L 6.2)))
   (R (B 3 6.8) (A 3 6.8) P (L 6.2) (Q)) ;;; en cualquier orden
```

2.5.6 Escribe una función que permita realizar la intersección de dos conjuntos

```
Prototipo:  (defun my-intersection (cjtoa cjtob) ...)
Ejemplos:
>> (my-intersection '((Q) (L 6.2) P (A 3 6.8)) '())
   NIL
>> (my-intersection '((Q) (L 6.2) P (A 3 6.8)) '( P  R (B 3 6.8) (L 6.2)))
   (P (L 6.2))
```