PRÁCTICA 0. PROGRAMACIÓN EN LISP

Fecha de publicación: 2014/01/22
NO ENTREGAR
Duración: 1 sesión de tutorías en laboratorio [2014/01/29] + 2 h. de trabajo Versión: 2014/01/22 18:00

La evaluación del código en el fichero no debe dar errores en Allegro 6.2 (recuerda 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

IMPORTANTE:

- **Puedes copiar el código ejemplo en el editor de Allegro y añadir comentarios. El fichero resultante puede servir como base para elaborar la memoria.**
- **Guarda a menudo los ficheros con los que estás trabajando.**
- En el editor Allegro 6.2 [CTRL + Z] sólo puede utilizarse para deshacer el último cambio.
- Después de un stack overflow es recomendable cerrar el entorno y volverlo a abrir.
- **Utiliza el comando LISP `trace` para depurar código**.

Ayuda:

- Se recomienda leer atentamente el documento Errores más frecuentes.
  Se penalizarán especialmente los errores que se cometan en la práctica en él 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 forma LISP sobre la expresión en la que está el cursor en una ventana de edición. [CUIDADO: SÓLO FUNCIONA PARA EXPRESIONES ENTRE PARÉNTESIS]
- Usa Ctrl+. para autocompletar la función que estás escribiendo.
- Con Ctrl+Shift+^; (también en el menú "Edit") puedes marcar / desmarcar como comentario el código seleccionado 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.

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 esta práctica) operaciones destructivas: `NCONC, NREVERSE, DELETE`. Usa las alternativas no destructivas: `APPEND, REVERSE, REMOVE`.
  - La ordenación `SORT` (destructiva) debe 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 tres 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 `IF`s 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 en mayúsculas 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 ..))
```

INSTRUCCIONES PARA ELABORAR PSEUDOCÓDIGO

El pseudocódigo que escribas debe describir las operaciones realizadas de forma que sea comprensible el procesamiento realizado a alto nivel. Es decir, debe estar a más alto nivel que LISP, debe ser próximo al lenguaje natural y describir la semántica del código, pero no debe presentar ambigüedades. No debe ser una traducción literal de LISP a castellano o a C.

Por ejemplo, el pseudocódigo (bastante detallado, puede que en la práctica no sea necesario tanto detalle) de una función que elimina los elementos de una lista que cumplan una condición dada (en LISP, la función remove) es.

Función: my-remove(lst, comp-p)
Entrada: lst [lista]
comp-p [predicado]
Salida: lista sin los elementos que cumplen la condición especificada por el predicado.
Pseudocódigo:

Si la lista es vacía, **entonces** evalúa a lst **; caso base**
en caso contrario, **; recursión**
Si el primer elemento de lst cumple la condición especificada por **comp-p**,
entonces descarta el primer elmento de lst y evalúa a la lista que resulta de **eliminar** del **resto** de lst los elementos que cumplan la condición especificada por comp-p.
en caso contrario,
construye una lista cuyo primer elemento es el **primer** elemento de lst, y cuyo resto sea el resultado de **eliminar** del **resto** de lst los elementos que cumplan la condición especificada por comp-p

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;  my-remove: elimina de una lista los elementos que
;;;              cumplen una determinada condición
;;;
;;;  RECIBE:  lst    [lista]
;;;           comp-p [predicado]
;;;
;;;  EVALÚA:  Una lista sin los elementos que cumplen la
;;;           condición especificada por el predicado comp-p
;;;

(defun my-remove (lst comp-p)
  (if (null lst)
      NIL                          ; base case
    (let ((elt-1 (first lst)))     ; let to avoid repeated code
      (if (funcall comp-p elt-1)   ; recursion
          (my-remove (rest lst) comp-p)
        (cons (first lst)
              (my-remove (rest lst) comp-p))))))

;;
;; EXAMPLES:
;;

(my-remove NIL #'oddp)       ;-> NIL    ; empty list case
(my-remove '(2 4 6) #'oddp)  ;-> (2 4 6); no elements removed
(my-remove '(3 4 5 6) #'oddp);-> (3 5)  ; some elements removed
(my-remove '(1 3 5 7) #'oddp);-> NIL    ; all elements removed
```

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).
8. Preguntad.

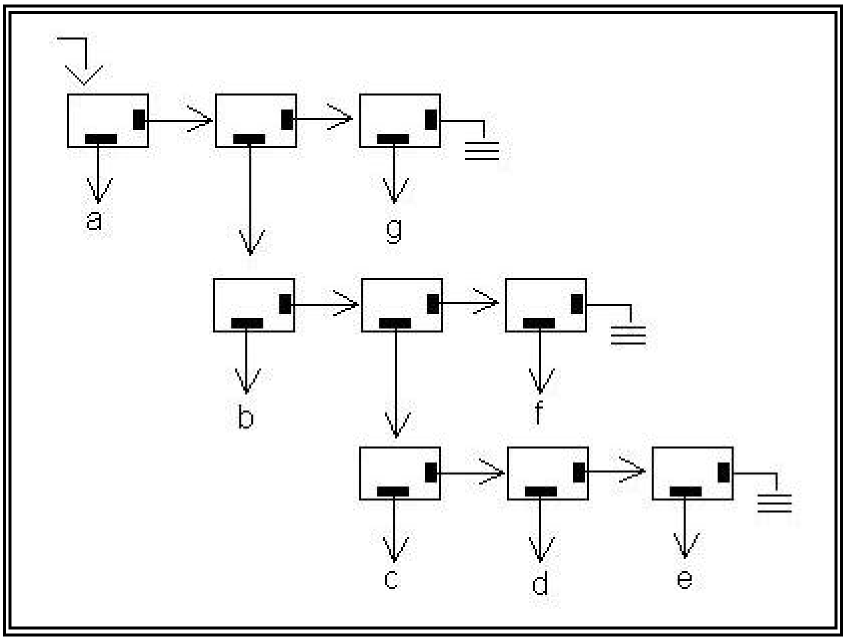
NOTA: Utiliza la instrucción `(trace <fn>)` para ver la secuencia de evaluaciones del intérprete de LISP. Para dejar de ver dicha secuencia, utiliza `(untrace <fn>)`

LISP I, EJEMPLOS

EJEMPLO 1: 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 esta práctica). Se trata de listas enlazadas simples. Cada nodo de la lista es una *celda cons*. Una celda cons está formada por dos 2 punteros. Uno de ellos, llamados históricamente *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).

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;  EJEMPLO 1
;;;

(print milista)
(setf milista nil)
(print milista)
(setf milista '(- 6 2))
(print milista)
(setf milista (- 6 2))
(print milista)
(setf milista '(8 (9 10) 11))
(car milista)
(first milista)
(cdr milista)
(rest milista)
(car (cdr milista))
```

```
(print milista)
(length milista)
(setf milista (cons 4 '(3 2)))
(print milista)
(setf milista (list '+ 4 3 2))
(print milista)
(eval milista)
(setf milista (append '(4 3) '(2 1)))
(print milista)
(reverse milista)
(print Milista)
```

PROGRAMACIÓN FUNCIONAL EN LISP

A pesar de que LISP es un lenguaje de programación multiparadigma, intentaremos circunscribirnos todo lo posible al paradigma funcional. En LISP, gracias a su sintaxis, código y datos son representados de la misma manera, mediante expresiones simbólicas (o s-exps) que son evaluadas por un intérprete para producir un valor.

- Las expresiones simbólicas de LISP están constituidas por:
  - Átomos: su valor es el propio átomo
  - Listas:
    - El primer elemento de la lista es interpretado como el nombre de una función.
    - El resto de elementos de la lista son interpretados como los argumentos de la función (notación prefija)
- La evaluación de expresiones en LISP es ansiosa (*eager evaluation*). Antes de evaluarse una llamada a una función se evalúan todos sus argumentos de izquierda a derecha. Los argumentos de entrada podrían ser a su vez expresiones que necesitan ser evaluadas.
- Como regla general, en el paradigma de programación funcional, la evaluación de una forma LISP no debe producir modificaciones en los parámetros recibidos.
  - Cuando como resultado de la evaluación de una expresión LISP se modifica el valor de alguno de los argumentos de entrada, se dice que es destructiva, por lo cual debe ser utilizada con precaución, ya que la evaluación puede producir efectos secundarios.
  - En esta práctica no se permite definir funciones destructivas.
- La base de la programación funcional es la transparencia referencial, es decir, que en un programa cualquier expresión pueda ser sustituida por el valor correspondiente resultando en un programa semánticamente equivalente al original. Para que un programa tenga la propiedad de la transparencia referencial debe estar libre de efectos secundarios.

EJEMPLO 2: EVALUACIÓN DE FORMAS LISP

Analiza el siguiente código y explica cómo se evalúa. La clave está en entender bien el funcionamiento de `quote`.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;  EJEMPLO 2
;;;

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

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

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

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

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

EJEMPLO 3: EFECTOS SECUNDARIOS

Analiza el siguiente código y explica cómo se evalúa. Centra tu atención en si las evaluaciones modifican o no sus argumentos de entrada.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;  EJEMPLO 3
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;  Uso de remove (no destructiva) / delete (destructiva)
;;;

(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*)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; Uso de cons  (no destructiva) / push (destructiva)
;;;        first (no destructiva) / pop  (destructiva)
;;;

(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*)

(first *dias-libres*)
(print *dias-libres*)

(pop *dias-libres*)
(print *dias-libres*)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;  Uso de sort (destructiva)
;;;
;;;  Ordenación de una lista de menor a mayor
;;;  por el valor absoluto
;;;  del cadr (= second) de sus elementos
;;;  versión no destructiva

(setf lst '((a -4) (b -3)  (c 1)  (d 9)))
```

```
(sort (copy-list lst)   ; copia solo el esqueleto de la lista
      #'(lambda(x y) (< (abs x) (abs y))) ; compara valor abs
      :key #'cadr)                        ; del cadr

(print lst)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;  Ordenación de una lista por el car de sus elementos
;;;  versión destructiva
;;;

(setf lst '((a -4) (b -3)  (c 1)  (d 9)))

(sort lst
      #'(lambda(x y) (< (abs x) (abs y))) ; compara valor abs
      :key #'cadr)                        ; del cadr

(print lst)
```

PREGUNTA: ¿Por qué se ha utilizado `copy-list` y no `copy-tree` en la versión no destructiva del uso de `sort`?

EJEMPLO 4: LISTAS OBTENIDAS POR MODIFICACIÓN DE OTRAS LISTAS

MUY IMPORTANTE: Se debe programar utilizando un estilo funcional (recursión, uso funciones de orden superior como `mapcar`).
Aunque se pueda obtener el mismo resultado 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).
Debemos seguir esta norma de programación en las prácticas de la asignatura, a menos que se indique expresamente lo contrario.
MAPCAR es una función que realiza una transformación sobre cada uno de los elementos de la lista o listas que toma como argumentos pero no se debe usar como método de procesamiento secuencial.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;  EJEMPLO 4.1
;;;
;;;  Multiplica por un número todos los elementos
;;;  de una lista
;;;

(defun multiplica-num-lst (num lst)
  (mapcar #'(lambda (x) (* num x)) lst))

(multiplica-num-lst 2 '(1 2 3 4))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;  EJEMPLO 4.2
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;  Genera una lista a partir de otra que contenga sólo
;;;  los valores superiores a un valor dado
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
```

```
;;;  Implementación con remove-if (preferida)
;;;

(defun sublista-superiores-0 (valor lista)
  (remove-if
   #'(lambda (x) (<= x valor))
   lista))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;  Implementación con mapcar
;;;

(defun sublista-superiores-1 (valor lista)
  (remove nil (mapcar #'(lambda (elem)
                          (if (> elem valor) elem))
                lista)))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;  Implementación con mapcan
;;;

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

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;  Implementación con recursión
;;;

(defun sublista-superiores-3 (valor lista)
  (cond
   ((null lista)   lista)
   ((> (first lista) valor)
    (cons (first lista)
          (sublista-superiores-3 valor (rest lista))))
   (T (sublista-superiores-3 valor (rest lista)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;  Implementación con dolist
;;;

(defun sublista-superiores-4 (valor lista)
  (reverse
   (let ((retorno nil))
     (dolist (elem lista retorno)
       (if (> elem valor)
           (setf retorno (cons elem retorno)))))))

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

(sublista-superiores-0
  5
  (multiplica-num-lst 2 *lista-numeros*)))

(sublista-superiores-0 5 *lista-numeros*))
(sublista-superiores-1 5 *lista-numeros*))
(sublista-superiores-2 5 *lista-numeros*))
(sublista-superiores-3 5 *lista-numeros*))
(sublista-superiores-4 5 *lista-numeros*))
```

IMPORTANTE: Analiza la implementación de las funciones anteriores. ¿Qué ventajas e inconvenientes tiene cada una? Para sublista-superiores-4, 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.

EJEMPLO 5: ÁMBITO DE VISIBILIDAD DE VARIABLES EN FUNCIONES

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;  EJEMPLO 5.1
;;;

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

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

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

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

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

(suma2 a)
(print a)
(print resultado)
```

MUY IMPORTANTE: no utilizar SETF dentro de `mapcar`. En caso de que se quiera evitar evaluar más de una vez una misma expresión puede vincularse su valor con una variable mediante un `let`.

Analiza el siguiente código y explica cómo se evalúa. Haz un estudio crítico de la implementación enfocado en la visibilidad de las variables.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;  EJEMPLO 5.2
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; cuenta-elementos: Implementación recursiva
;;;

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

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

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; cuenta-elementos: Implementación con mapcar
;;;
```

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

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

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; cuenta-elementos: Implementación con dolist
;;;

(defun cuenta-elementos-3 (lista)
  (let ((contador 0))
    (dolist (elem lista contador)
      (incf contador))))

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

EJEMPLO 6: MODIFICACIÓN DE LOS ELEMENTOS DE UNA LISTA

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;  EJEMPLO 6
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;  Escribe una función que, aplicada sobre una lista
;;;  evalúa a otra lista en la que el primer elemento de
;;;  la lista original ha sido eliminado
;;;

(defun elimina-primero (lista)
  (rest lista))

(setf lista '(1 2 3 4 5 6 7))
(elimina-primero lista)
(print lista)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;    Escribe una función que, aplicada sobre una lista
;;;    evalúa a otra lista en la que el segundo elemento
;;;    de la lista original ha sido eliminado
;;;

(defun elimina-segundo (lista)
  (cons (first lista)
        (rest (rest lista))))

(setf lista '(1 2 3 4 5 6 7))
(elimina-segundo lista)
(print lista)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;   Escribe una función que, aplicada sobre una lista
;;;   evalúa a otra lista en la que el elemento que ocupa
;;;   la posición n en la lista original ha sido eliminado
;;;

(defun elimina-n (lista n)
  (if (<= n 1)
      (rest lista)
    (cons (first lista)
```

```
          (elimina-n (rest lista) (- n 1)))))

(setf lista '(1 2 3 4 5 6 7))
(elimina-n lista 4)
(print lista)
```

EJEMPLO 7: PARÁMETROS OPCIONALES

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; EJEMPLO 7:
;;; Genera una lista de valores de un tamaño n
;;; desde un valor dado con incrementos especificado
;;;

(defun creaLista (n incremento &optional (desde 0))
  (unless (= n 0)
    (cons desde
          (creaLista (- n 1)
                     incremento
                     (+ desde incremento)))))

(creaLista 10 2)     ;; lista de pares
(creaLista 10 2 1)   ;; lista de impares
```

EJEMPLO 8: RECORRIDO DE UNA LISTA

RECORDATORIO: `incf` no es una función sino una macro y modifica su argumento. Una macro se expande en tiempo de preprocesamiento. La llamada a la macro de LISP (`incf ref delta`) se expande en la expresión (`setf ref (+ ref delta)`).

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

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; EJEMPLO 8.1:
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; Suma los elementos de una lista: impl. recursiva
;;;

(defun suma-1 (lista)
  (if (null lista)
        0
    (+ (first lista)
       (suma-1 (rest lista)))))

(setf lst '(1 2 3 4))
(suma-1 lst)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; Suma los elementos de una lista: impl.con apply
;;;

(defun suma-2 (lista)
  (apply #'+ lista))

(setf lst '(1 2 3 4))
(suma-2 lst)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; EJEMPLO 8.2:
;;; Genera una lista con los números pares de otra lista
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; extrae-pares: Implementación con remove-if [preferida]
;;;

(defun extrae-pares (lista)
  (remove-if #'oddp lista))

(extrae-pares '(1 2 4 5 0 6 8 7 4))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; extrae-pares: Implementación recursiva
;;;

(defun extrae-pares-recursiva (lista)
  (unless  (null lista)
    (let ((primero (first lista)))
      (if (evenp primero)
          (cons primero (extrae-pares-recursiva (rest lista)))
        (extrae-pares-recursiva (rest lista))))))

(extrae-pares-recursiva '(1 2 4 5 0 6 8 7 4))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; extrae-pares: Implementación con mapcar
;;;

(defun extrae-pares-mapcar (lista)
  (remove NIL
          (mapcar #'(lambda (x)
                      (when (evenp  x) x))
            lista)))

(extrae-pares-mapcar '(1 2 4 5 0 6 8 7 4))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; extrae-pares: Implementación con mapcan
;;;

(defun extrae-pares-mapcan (lista)
  (mapcan #'(lambda (x)
              (when (evenp  x) (list x)))
            lista))

(setf milista (crealista 1000 5))

(time (extrae-pares-recursiva milista))
(time (extrae-pares-mapcar milista))
(time (extrae-pares-mapcan milista))

(compile 'extrae-pares-recursiva)
(compile 'extrae-pares-mapcar)
(compile 'extrae-pares-mapcan)

(time (extrae-pares-recursiva milista))
(time (extrae-pares-mapcar milista))
(time (extrae-pares-mapcan milista))
```

Nota: `compile` crea la versión compilada de una función para ser ejecutada por una máquina virtual.

EJEMPLO 9: RECURSIÓN DE COLA

Considera las dos implementaciones del factorial

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;   EJEMPLO 9
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;  Factorial recursivo:
;;;   caso base:  0! = 1
;;;   recursión:  n! = n(n-1)!
;;;

(defun factorial-recursivo (n)
  (when (and (integerp n) (>= n  0 ))
    (factorial-recursivo-aux n)))

(defun factorial-recursivo-aux (n)
  (if (= n 0)
      1
    (* n (factorial-recursivo-aux (- n 1))))))

(factorial-recursivo 3.14)  ;-> NIL
(factorial-recursivo -3)    ;-> NIL
(factorial-recursivo 0)     ;-> 1
(factorial-recursivo 5)     ;-> 120

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;  Factorial recursivo con recursión de cola
;;;

(defun factorial-recursivo-cola (n)
  (when (and (integerp n) (>= n  0 ))
    (factorial-recursivo-cola-aux n 1)))

(defun factorial-recursivo-cola-aux (n acumulador)
  (if (= n 0)
      acumulador
    (factorial-recursivo-cola-aux (- n 1) (* n acumulador))))

(factorial-recursivo-cola 3.14) ;-> NIL
(factorial-recursivo-cola -3)   ;-> NIL
(factorial-recursivo-cola 0)    ;-> 1
(factorial-recursivo-cola 5)    ;-> 120
```

¿Es más eficiente `factorial-recursivo-cola` o `factorial-recursivo`?
¿Cómo se podría hacer más eficiente `factorial-recursivo-cola`?

EJEMPLO 10: Diferencias entre eql y equal (identidad e igualdad estructural)

Explica brevemente las diferencias entre `eql` y `equal`

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;  EJERCICIO: EQL, EQUAL
;;;
```

```
(setf a 3)
(eql a 3)
(equal a 3)

(setf lst '(1 2 3))
(eql lst '(1 2 3))
(equal lst '(1 2 3))
```

EJEMPLO 11: Función reduce

La expresión `(reduce #'* '(1 2 3 4))` equivale a `(* (* (* 1 2) 3) 4)` y, por lo tanto, computa el factorial de 4.
Propón otro ejemplo práctico de uso de la función `reduce` definida como:

$$(\text{reduce } \#'\text{fn } '(a_1\ a_2\ a_3 \ldots a_n)) \equiv (\text{fn} \ldots (\text{fn } (\text{fn } 'a_1\ 'a_2)\ 'a_3) \ldots 'a_n)$$

EJEMPLO 12: Determina si un elemento pertenece a un conjunto

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

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;  EJERCICIO: our-member
;;;

(defun our-member-1 (obj lst &optional (comparator #'equal))
  (unless (null lst)
    (or (funcall comparator (first lst) obj)
      (our-member-1 obj (cdr lst) comparator))))

(our-member-1 'a NIL #'eql)           ;-> NIL
(our-member-1 'a '(b c a d f) #'eql)  ;-> T
(our-member-1 'h '(b c a d f) #'eql)  ;-> NIL

(our-member-1
  'a '((b 1) (c 2) (a 3) (d 5) (f 6))
  #'(lambda (x y)
          (eql (first x) y)))         ;-> T
```

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

Considera la función

```
(defun our-member-2 (obj lst &optional (comparator #'equal))
   (or (funcall comparator (first lst) obj)
     (our-member-equal-2 obj (rest lst) comparator)))
```

Compara las evaluaciones

```
>> (our-member-2 NIL NIL)
>> (our-member-2 NIL '(NIL))
```

con

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

¿Por qué es incorrecta la implementación our-member-equal-2?

EJEMPLO 13: Invierte 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 (rest lst)
                      (cons (first 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. ¿Cuál sería el coste de estas funciones teniendo en cuenta que `nth` tiene coste $O(n)$?

[]