



DEPARTAMENTO DE INFORMÁTICA
UNIVERSIDAD CARLOS III DE MADRID

Ingeniería en Informática

Paradigmas de Programación

Marzo 2008

Hoja de Ejercicios 2: Problemas de recursividad en Lisp

Comentarios generales sobre los ejercicios

- Es una buena idea pensar y razonar las soluciones antes de intentar programarlas
- Para la resolución de los problemas, sólo pueden emplearse las funciones de LISP vistas en teoría
- **Todos** los ejercicios de esta hoja deben resolverse usando funciones recursivas
- Describir las soluciones a los ejercicios de la manera más formal posible

1. Escribir una función que calcule el factorial de un número natural

```
(defun factorial (n)
  (if (zerop n)
      1
      (* n (factorial (1- n)))))
```

2. Definir la función de fibonacci recordando que:

- fibonacci(0) = fibonacci(1) = 1
- fibonacci(n) = fibonacci(n - 1) + fibonacci(n - 2)

Tener en cuenta su eficiencia computacional a la hora de escribirla.

```
; Versión aplicando la definición directamente

; int -> int
(defun fibonacci (n)
  (if (< n 2)
      1
      (+ (fibonacci (1- n)) (fibonacci (- n 2)))))

; Versión "optimizada". El truco consiste en aplicar
; la definición "hacia delante".
```

```

; int -> int
(defun fibonacci-opt (n)
  (case n
    ((1 2) 1)
    (t (fibonacci-opt-aux 1 1 2 n))))

(defun fibonacci-opt-aux (fn1 fn2 i n)
  (if (= i n)
      (+ fn1 fn2)
      (fibonacci-opt-aux fn2 (+ fn1 fn2) (1+ i) n)))

```

3. Escribir una función que calcule el valor de la función a^b recursivamente (esto es, no se permite usar la función `expt`), siendo a y b números enteros **cualesquiera**
-

```

; int * int -> int
(defun eleva (a b)
  (cond ((zerop b) 1)
        ((< b 0) (/ 1 (eleva a (- 0 b))))
        (t (* a (eleva a (1- b))))))

```

4. Escribir funciones que calculen el valor de los operadores booleanos `and`, `or`, `xor` (binario) y `not` (unario), para un número cualquiera de operandos
-

```

; bool * bool * bool * ... -> bool
(defun and-op (&rest l)
  (and-op-aux l))

(defun and-op-aux (l)
  (if l
      (if (car l)
          (and-op-aux (cdr l))
          nil)
      t))

; bool * bool * bool * ... -> bool
(defun or-op (&rest l)
  (or-op-aux l))

(defun or-op-aux (l)
  (if l
      (if (car l)
          t
          (or-op-aux (cdr l)))
      nil))

; bool -> bool
(defun not-op (a)
  (if a
      nil
      t))

; bool * bool * bool * ... -> bool

```



```
(t (if (evenp (car lista))
      1
      0))))
```

7. Definir una función que calcule el número de términos que siguen a un átomo dado dentro de una lista
-

```
(defun cuenta-terminos (elemento lista)
  (length (mi-member elemento lista)))
```

8. Escribir la función `get-posicion` que determina la posición de la primera aparición de un elemento en una lista

```
> (get-posicion 'c '(a b c d))
> 3
```

```
(defun get-posicion (e lista)
  (get-posicion-aux e lista 0))

(defun get-posicion-aux (e lista resul)
  (cond ((null lista) nil)
        ((eq e (car lista)) (+ resul 1))
        (t (get-posicion-aux e (cdr lista) (+ resul 1)))))

;;otra forma
(defun get-posicion2 (elem lista)
  (+ 1 (- (length lista)
          (length (member elem lista :test #'equal)))))
```

9. Escribir la función `invertir` que invierte el contenido de una lista

```
> (invertir '(1 2 3 4))
> (4 3 2 1)
```

```
(defun invertir (lista)
  (if lista
      (append (last lista) (invertir (butlast lista)))))

;;otra forma
(defun invertir2 (lista)
  (invertir-recursivo lista 0))

(defun invertir-recursivo (lista indice)
  (cond ((< indice (floor (/ (length lista) 2)))
        (append (list (nth (- (length lista) 1 indice) lista))
                (invertir-recursivo lista (+ 1 indice))
                (list (nth indice lista))))
        ((= indice (floor (/ (length lista) 2)))
        (list (nth indice lista)))))
```

-
10. Escribir la función `mi-butlast` que tome una lista y un número natural `n` y retorne la lista original sin los últimos `n` elementos. **No está permitido usar la función `reverse` de Lisp**
-

```
(defun mi-butlast (L n)
  (cond ((<= (length L) n) nil)
        (t (cons (car L) (mi-butlast (cdr L) n)))))
```

11. Definir la función `palindromo` que crea el palíndromo de una lista dada como argumento. Un palíndromo es una estructura capicúa que, por lo tanto, se lee igual de izquierda a derecha que de derecha a izquierda. **No está permitido usar la función `reverse` de Lisp**
-

```
(defun palindromo (lista)
  (append lista (mi-reverse lista)))
```

12. Escribir la función `get-numeros` que extrae todos los números que aparecen en una lista

```
> (get-numeros '((1 (2)) a ((5 c 7)) 4))
> (1 2 5 7 4)
```

```
(defun get-numeros (lista)
  (get-numeros-aux lista nil))

(defun get-numeros-aux (lista resul)
  (cond ((null lista) resul)
        ((numberp lista) (append resul (list lista)))
        ((not (listp lista)) resul)
        (t (append resul
                    (get-numeros-aux (car lista) resul)
                    (get-numeros-aux (cdr lista) resul)))))
```

13. Definir de forma **recursiva** la función `primero-y-ultimo` que toma como argumento una lista y que devuelva otra lista con el primer y último elemento de la lista
-

```
(defun primero-ultimo (lista)
  (cons (car lista)
        (ultimo (cdr lista))))

(defun ultimo (lista)
  (if (cdr lista)
      (ultimo (cdr lista))
      lista))
```

14. Definir de forma **recursiva** la función `mi-reverse` que toma como entrada una lista y devuelve la misma lista en orden inverso

```
> (mi-reverse '(es una lista))
> (LISTA UNA ES)
```

```
(defun mi-reverse (lista)
  (if lista
      (append (mi-reverse (cdr lista))
              (cons (car lista) nil))))
```

15. Definir de forma **recursiva** la función `mi-reverse-arbol` que toma como entrada una lista y devuelve la misma lista en orden inverso en cada subnivel de la lista

```
> (mi-reverse-arbol '(es una lista))
> (LISTA UNA ES)
> (mi-reverse-arbol '(es una (esto es una sublista)))
> ((SUBLISTA UNA ES ESTO) UNA ES)
```

```
(defun mi-reverse-arbol-1 (lista)
  (if lista
      (if (consp (car lista))
          (append (mi-reverse-arbol-1 (cdr lista))
                  (cons (mi-reverse-arbol-1 (car lista)) nil))
              (append (mi-reverse-arbol-1 (cdr lista))
                      (cons (car lista) nil))))))
```

;; puesta de otra manera

```
(defun mi-reverse-arbol-2 (lista)
  (if lista
      (append (mi-reverse-arbol-1 (cdr lista))
              (cons (if (consp (car lista))
                      (mi-reverse-arbol-1 (car lista))
                      (car lista))
                    nil))))
```

16. Definir de forma **recursiva** la función `mi-member-p` que toma como entrada una expresión LISP y una lista y devuelve T si la expresión pertenece a la lista del primer nivel y NIL en otro caso

```
> (mi-member-p 'es '(es una lista))
> T
> (mi-member-p '(es sublista) '(es una (es sublista)))
> T
> (mi-member-p 'es '((es) una lista))
> NIL
```

```
(defun mi-member-p (elemento lista)
  (or (equal elemento (car lista))
      (and lista
            (mi-member-p elemento (cdr lista)))))
```

```

(defun recorrer-post-orden (arbol)
  (if (endp arbol)
      nil
      (append (recorre-post-orden (cadr arbol))
              (recorre-post-orden (caddr arbol))
              (list (car arbol)))))

(defun recorrer-in-orden (arbol)
  (if (endp arbol)
      nil
      (append (recorre-in-orden (cadr arbol)) (list (car arbol))
              (recorre-in-orden (caddr arbol)))))

; Definición de un pequeño árbol con tres niveles
(defun nodo11 () '(nodo11 nil nil))
(defun nodo12 () '(nodo12 nil nil))
(defun nodo1 () (cons 'nodo1 (list (nodo11) (nodo12))))
(defun nodo21 () '(nodo21 nil nil))
(defun nodo22 () '(nodo22 nil nil))
(defun nodo2 () (cons 'nodo2 (list (nodo21) (nodo22))))
(defun nodo-raiz () (cons 'raiz (list (nodo1) (nodo2))))
(recorre-pre-orden (nodo-raiz))
(recorre-in-orden (nodo-raiz))
(recorre-post-orden (nodo-raiz))

```

21. Dada una lista de números enteros positivos, escribir algoritmos recursivos que computen:

- a) El máximo de un elemento de la lista, o -1 si la lista está vacía
- b) El mínimo elemento de la lista, o -1 si la lista está vacía
- c) La suma de los elementos de la lista, o -1 si la lista está vacía
- d) El producto de los elementos de la lista, o -1 si la lista está vacía
- e) La media de los elementos de la lista, o -1 si la lista está vacía

```

(defun max-lista (L)
  (if (null L)
      -1
      (max (car L)
           (max-lista (cdr L)))))

(defun min-lista (L)
  (cond ((null L)
        -1)
        ((= (length L) 1)
         (car L))
        (t
         (min (car L)
              (min-lista (cdr L))))))

(defun suma-lista (L)
  (cond ((null L)
        -1)
        ((= (length L) 1)
         (car L))

```

```

(t
 (+ (car L)
    (suma-lista (cdr L))))))

(defun prod-lista (L)
  (cond ((null L)
        -1)
        ((= (length L) 1)
         (car L))
        (t
         (* (car L)
            (prod-lista (cdr L))))))

```

22. Implementar una función recursiva que calcule la longitud de la mayor secuencia de 1s en una lista que contiene únicamente 0s y 1s

```

> (max-ones-fwd '(1 1 0 0 1 0 0 0 1 1 1 1 0 0 1 1 1))
4

```

```

(defun max-ones (L &key (long 0) (long-max 0))
  (cond ((null L)
        (max long long-max))
        ((plusp (car L))
         (max-ones (cdr L)
                   :long (1+ long)
                   :long-max long-max))
        (t
         (max-ones (cdr L)
                   :long 0
                   :long-max (max long long-max)))))

```

23. Programar la función (sublist L inicio fin) de modo recursivo que devuelva la sublista de L con los elementos en el intervalo [inicio,fin) donde la primera posición tiene el índice 0

```

> (sublist '(1 2 3 4 5) 2 4)
(3 4)

```

```

(defun sublist (L inicio fin)
  (cond ((null L) ; Caso base #1: la lista está vacía
        nil) ; -> no se devuelve nada
        ((zerop fin) ; Caso base #2: ya hemos llegado al final
         nil) ; -> no se procesan más elementos
        ((zerop inicio) ; Caso general #1: hemos llegado al inicio
         (cons (car L) ; -> se acumulan los valores a extraer
               (sublist (cdr L)
                       inicio
                       (1- fin))))
        (t ; Caso general #2: no hemos llegado al inicio
         (sublist (cdr L) ; -> progresamos sin acumular elementos hasta
                  (1- inicio) ; alcanzar el inicio
                  (1- fin)))))

```

-
24. El método de ordenación de intercambio directo se basa en la comparación sucesiva e intercambio de elementos adyacentes hasta que, por fin, esté ordenada la lista. En cada paso se van comparando, empezando por el final, cada elemento con el anterior y, en caso de estar desordenados, se intercambian. De esta forma, al final del primer paso quedará el menor elemento en la posición más a la izquierda, en el segundo el menor del resto, quedará situado en la segunda posición, etc. Al final, la lista quedará ordenada en $(n-1)$ pasos donde n es el número de elementos de la lista L .

Se pide implementar una función recursiva que ordene una lista de números L por el método de intercambio directo de menor a mayor

```
(defun intercambio-directo-aux (L)
  (cond ((null L)
        L)
        ((= (length L) 1)
         L)
        (t
         (append (list (min (car L) (car (intercambio-directo-aux (cdr L))))
                    (max (car L) (car (intercambio-directo-aux (cdr L))))
                  (cdr (intercambio-directo-aux (cdr L)))))))

(defun intercambio-directo (L)
  (cond ((null L)
        L)
        ((= (length L) 1)
         L)
        (t
         (cons (car (intercambio-directo-aux L))
               (intercambio-directo (cdr (intercambio-directo-aux L)))))))
```

25. Programar una función de ordenación de una lista por la técnica de Divide y Vencerás con Merge Sort:

Merge Sort:

- a) Sea k el elemento medio del vector
 - b) Ordenar los elementos A_1, A_k
 - c) Ordenar los elementos A_{k+1}, A_n
 - d) Mezclar ambos
-

; mergelist devuelve una única lista ordenada con los elementos de las
; listas L1 y L2, dado que estas tengan sus elementos ordenados

```
(defun mergesorted (L1 L2)
  (cond ((null L1)
        L2)
        ((null L2)
         L1)
        ((< (car L1)
            (car L2))
         (cons (car L1)
               (mergesorted (cdr L1)
                           L2)))
        (t
         (cons (car L2)
               (mergesorted L1
                           (cdr L2))))))
```

```
(mergesorted L1
  (cdr L2))))))
```

; La siguiente función resuelve la ordenación por merge-sort de los
; elementos de una lista de números L

```
(defun merge-sort (L)
  (if (or (null L) ; Caso base: si la lista está vacía o
        (= (length L) 1)) ; tiene un solo elemento
      L ; -> ¡entonces ya está ordenada!
      (mergesorted (merge-sort (sublist L 0 (floor (/ (length L) 2))))
                    (merge-sort (sublist L (floor (/ (length L) 2)) (1+ (length L)))))))
```

26. Una forma de calcular el máximo común divisor (mcd) de dos números es la siguiente:

$MCD(x, y) = y$	si $y \leq x, x \bmod y = 0$
$MCD(x, y) = MCD(y, x)$	si $x < y$
$MCD(x, y) = MCD(y, x \bmod y)$	en otro caso

Escribir una función para calcularlo

```
(defun mcd (x y)
  (cond ((and (<= y x)
              (zerop (mod x y)))
         y)
        ((< x y)
         (mcd y x))
        (t
         (mcd y (mod x y)))))
```

27. Escribir una función `prof-elementos` que indique cuál es la profundidad de cada elemento de una lista (es decir, el nivel de la lista anidada en la que aparece). Para ello devolverá una lista en la que cada elemento es una lista de exactamente dos elementos: el elemento en cuestión y su profundidad. Se considera que los elementos del primer nivel tienen profundidad 1

```
> (prof-elementos '( 3 (1 (2)) ((( 5 7))) 4))
> ((3 1) (1 2) (2 3) (5 4) (7 4) (4 1))
```

```
(defun prof-elementos (lista)
  (cond ((null lista) nil)
        ((listp (car lista)) (append (incrementa-uno (prof-elementos (car lista)))
                                       (prof-elementos (cdr lista))))
        (t (cons (list (car lista) 1) (prof-elementos (cdr lista)))))
```