INTELIGENCIA ARTIFICIAL

Programación en LISP

-Indicaciones de estilo y errores comunes-

Última actualización: 2016/01/28

Indicaciones generales

1. En el caso de que el argumento sea una lista, contemplad siempre el caso NIL como posible argumento.
2. Las funciones que codifiquéis pueden fallar (por ejemplo, si los argumentos recibidos no son del tipo esperado), 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.

Formato

- El código debe estar en un único fichero.
- La evaluación del código en el fichero no debe dar errores (recuerda CTRL+A CTRL+E debe evaluar todo el código sin errores).
- Comentad el código con profusión.
- No se deben cerrar paréntesis en líneas aparte.

  En lugar de escribir:

  ```
  (defun foo ()
    (dotimes (i 10)
      (format t "~d. hello~%" i)
    )
  ```

```
)
```

Es preferible escribir:

```
(defun foo ()
  (dotimes (i 10)
    (format t "~d. hello~%" i)))
```

- Utilizad exactamente los nombres de funciones (prototipos) indicados.

Control de errores

- El control de errores debe hacerse únicamente en las funciones más generales. Para el resto de funciones, suponed que los argumentos son correctos
- En la mayoría de las funciones que resuelven tareas concretas se puede hacer un control de errores básico (¿he recibido un átomo?, ¿he recibido una lista?, ¿he recibido la lista vacía?), **asumiendo que los valores recibidos son del tipo correcto**.
- Si algún **argumento es del tipo lista**, manejad siempre el caso de la **lista vacía**.

Descomposición funcional

- Realizad las descomposiciones funcionales adecuada.
- Se prefieren las **funciones cortas** que realizan una única **tarea bien definida**.
- Para resolver un problema determinado a menudo es útil utilizar funciones auxiliares adicionales a las que se indican en el enunciado de la práctica.
- Identificad las tareas elementales que se requieren para resolver una tarea compleja e implementadlas en funciones aparte.
- Evitad un exceso de anidaciones de `mapcars`, o `mapcar` y recursión en la misma función (código difícil de entender).

Uso de let

- Usad cuando sea posible let para evitar repetir código o para hacer más clara la expresión de una función.
- `Let` debe tener el menor ámbito de visibilidad posible (es decir, el ámbito definido por `let` debe ser lo más local posible). Evitad utilizar `lets` cuyo ámbito sea todo el código de la función.
- Tened cuidado con la malformación de let. Un ejemplo de código como `(let (retorno)) (setf retorno 3)` crea la variable global retorno, ya que el ámbito del let se cierra antes de tiempo.

Errores concretos

- ERROR: Utilizar `(eql lst NIL)` o `(eql (length lst) 0)` para determinar si una lista es vacía.
  - Lo correcto es (null lst).

- ERROR: Utilizar `setf` para modificar el valor de variables que no son locales a la función, es decir, que no se reciben como parámetro ni se definen con un `let`.
  - Estas variables se crean como globales y siguen existiendo una vez finalizada la evaluación de la función.
  - Por lo general, las funciones LISP deberían recibir parámetros y, devolver un valor dado sin realizar ninguna otra modificación en el entorno.

- ERROR: Utilizar `(remove nil (mapcar ...))` para realizar selecciones en listas.
  - En este caso obligatoriamente hacemos dos recorridos en la lista: uno para el mapcar y otro para eliminar los nil.
  - Es más elegante un tratamiento recursivo de la lista, con lo cual se hace sólo una pasada.
  - También se aconseja el uso de mapcan, como en el ejemplo que sigue:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; IMPLEMENTACIÓN CON MAPCAR (desaconsejada)
;;;
(defun elimina-pares (lst)
  (remove nil
      (mapcar #'(lambda (x)
                  (when (oddp x) x)) lst)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; IMPLEMENTACIÓN CON MAPCAN (preferible)
;;;
(defun elimina-pares (lst)
  (mapcan #'(lambda (x)
              (when (oddp x)
              (list x)))
      lst))
```

  - Aunque en este caso concreto la mejor implementación sería con remove-if (función ya existente):

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; IMPLEMENTACIÓN CON remove-if (óptima)
;;;
(defun elimina-pares (lst)
  (remove-if #'evenp lst))
```

- ERROR: Usar `mapcar` como `dolist`.
  - La función mapcar devuelve una lista en la que cada elemento proviene de

aplicar una función sobre el elemento correspondiente en la lista que recibe.
    - Si se utiliza el `mapcar` para hacer dentro `setf` de una variable auxiliar, y no se utiliza para nada la lista que `mapcar` retorna, entonces lo programado es equivalente utilizar un `dolist`, con el inconveniente añadido del trabajo extra en crear la lista de retorno del `mapcar` que no se usa.

- ERROR: Usar (listp expr) para comprobar si expr es una lista.
    - Esta expresión únicamente comprueba si expr es NIL o un cons, pero no compueba si se trata de una lista bien formada (es decir, aquella para la que (rest expr) es una lista bien formada).
    - Por ejemplo, `(listp (cons 'a 'b))` evalúa a T.

- ERROR: Abusar de append.
    - Cuando se quiere añadir un elemento al principio de una lista es mejor usar cons.
    - Por ejemplo, `(append (list elt) cjto)` debería ser `(cons elt cjto)`.

- ERROR: Iterar listas con `loop` y `dotimes`, accediendo al elemento en la posición n mediante `nth`.
    - Recuerda a la programación procedural típica de C o Java.
    - Si usamos *arrays* no sería ningún problema acceder al enésimo elemento (existen funciones específicas para ello), pero en el caso de listas cada vez que accedemos al elemento n hay que recorrer la lista desde el principio hasta llegar a él.

- ERROR: Usar `defun` dentro de `defun`.
    - LISP lo permite, pero el código no es claro cuando se definen unas funciones dentro de otras.

- ERROR: Usar `equal` en lugar de = para expresiones numéricas.
    - Para comparaciones entre valores numéricos deben usarse las funciones =, <, <=, >, >=, /=.

- ERROR: Usar condicionales para obtener valores booleanos.
    - Por ejemplo, código como `(if (and (= x 3) (< y 4)) t nil)` debe sustituirse por `(and (= x 3) (< y 4))`.

- ERROR: Anidar varios `mapcar`, o `mapcar` y recursión en una misma sentencia.
    - Ese código es difícil de entender.

- ERROR: Utilizar `mapcar` cuando no corresponde.
    - Se debe usar cuando la operación es paralelizable y si es necesario hacer la operación para todos los elementos de la lista.
    - En caso contrario es mejor utilizar recursión. Por ejemplo, para saber si todos los elementos de una lista son impares:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; IMPLEMENTACIÓN CON MAPCAR  (desaconsejada)
```

```
;;;
(defun oddp-lst (lst)
  (eval (cons 'and (mapcar #'oddp lst))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; IMPLEMENTACIÓN RECURSIVA (preferible)
;;;
(defun oddp-lst (lst)
 (or (null lst)
   (and (oddp (first lst))
   (oddp-lst (rest lst)))))
```

- ERROR: Usar `if` y `cond` para menos ramas de lo que están diseñados.
  - Condicional con una rama / `nil`: `when`, `unless`.
  - Condicional con 2 ramas: `if`.
  - Condicional con 3 o más ramas: `cond`.

- ERROR: Volver a implementar funciones que ya existen.
  - Hay que utilizar las funciones nativas siempre que sea posible.
  - No sólo en su uso normal, sino también con las opciones adicionales que ofrecen.
  - Por ejemplo, para ordenar pares por su segundo elemento:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; IMPLEMENTACIÓN INCORRECTA
;;;
(sort (copy-list lst) #'(lambda(x y) (> (second x) (second y))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; IMPLEMENTACIÓN CORRECTA
;;;
(sort (copy-list lst) #'> :key #'second)
```