Criterios de evaluación de las prácticas de IA **Actualizado 2017/02/02**

1. Las **etapas del proceso de desarrollo** han sido completadas en el **orden correcto**:
   (i) Elaborar la **batería de ejemplos** (esta batería se enriquece con nuevos ejemplos a medida que se va descubriendo que son relevantes).
   (ii) Realizar el **diseño** del código.
   (iii) Esbozar el **pseudocódigo**.
   (iv) Escribir el **código** [en Allegro CL 6.2 ANSI with IDE].
   (v) **Depurar el código** y **comprobar**, con la ayuda de la batería de ejemplos, que puede ser **evaluado sin errores**, **resuelve el problema** planteado y es **eficiente**.
   (vi) **Iterar los pasos anteriores** para mejorar el código (alcanzar mayor eficiencia, depurar el estilo de programación, completar los comentarios, etc.) tantas veces como sea necesario.

2. El estilo de programación es **funcional:**
   - Se **usa recursión, mapcar, mapcan**, ... evitando el uso excesivo de anidaciones.
   - **No se han utilizado bucles**.
   - **No se usa setf** dentro de las funciones [**error grave**].
   - **No hay efectos indirectos** (no se utilizan funciones destructivas)**.**
   - **Las funciones implementadas son cortas y realizan una tarea bien definida.**

3. Se usan correctamente las **estructuras condicionales:**
   - **when** o **unless** para decisiones con una única opción (la otra es NIL).
   - **if** para decisiones con dos opciones.
   - **cond** para decisiones con varias opciones.

4. Para implementar **predicados** (funciones booleanas) o **expresiones booleanas:**
   - Se usan funciones LISP booleanas (**and, or, not, null, atomp, listp**,...).
   - **No se utilizan estructuras condicionales para implementar predicados.**

5. Se usan correctamente los **cierres léxicos:**
   - Se usa **let** para evitar la repetición de código.
   - Solo se usa **let*** (en lugar de let) sólo en los casos en los que sea necesario.

6. Se utilizan correctamente las **funciones de orden superior:**
   - Se usan correctamente **funcall**, **apply**, **mapcar**...
   - No se usa **eval.**

7. Se usan de manera apropiada los **recorridos de lista:**
   - Se utiliza **recursión** cuando **no es necesario recorrer toda la lista** o cuando **un paso en el procesamiento que realiza la recursión depende de los anteriores (y por lo tanto, influye en los posteriores).**
     - Contempla **todos los casos base relevantes** y **únicamente esos.**
     - Incluye los casos base estándar (podría haber excepciones):
       - **0** para argumentos enteros.
       - La lista vacía - NIL - para argumentos que son listas.

    - **No entra en recursiones infinitas**.
- Se utiliza **mapcar** cuando es necesario **procesar todos los elementos de la lista de manera independiente**.
- **No se utiliza length**.

8. El **programa es correcto:**
    - La batería de ejemplos es **completa** y **compacta**.
    - **Se explica**, mediante comentarios, **la razón** por la que cada **ejemplo** ha sido **incluido en la batería**.
    - **Los ejemplos incluyen,** al menos**, los casos típicos y casos singulares (ej. casos base)**.
    - Los **nombres de las variables y las funciones son informativos**.
    - El código está comentado y **los comentarios son claros, concisos y correctos** (CCC).
    - **Se adoptan los usos** de programación (indentación, nombres de funciones, etc.) **de la comunidad LISP.**
    - Se **utilizan correctamente las funciones, macros y estructuras de programación en LISP** (ej. uso correcto de cons para hacer crecer las listas agregando un elemento al principio).
    - **No se han definido funciones que ya existen en LISP.**
    - El código es correcto:
        - **Resuelve el problema** planteado.
        - Contempla **casos típicos** y **casos especiales.**
        - Tiene una **descomposición funcional correcta.**
        - Es **eficiente.**
    - **El código está un único fichero.**

9. La **memoria es clara, concisa y completa** (CCC).
    - Da **respuesta a las preguntas realizadas**.
    - **Ilustra las explicaciones con ejemplos** concretos de **evaluación del código**, incluyendo de manera explícita el **resultado de dichas evaluaciones** de código.
    - **Expone** las distintas **opciones de diseño** que han sido **consideradas** y las **razones por las que se ha preferido la opción implementada**.

10. La entrega
    - Sigue las **especificaciones de formato** incluidas en la normas de entrega.
    - Se **realiza y entrega dentro del plazo establecido**

EN RESUMEN:

(1) La memoria debe ser **clara, concisa y completa**.
(2) Se debe **elaborar en el laboratorio** y completar posteriormente.
(3) Para cada función, se deben incluir
    (3.1) **Batería de ejemplos** de prueba (lo más pequeña y completa posible).
    (3.2) **Pseudo-código.**
    (3.3) **Código con comentarios.**
    (3.4) **Comentarios sobre la implementación.**

EJEMPLO: Memoria para el ejercicio consistente en implementar la **función factorial en LISP**

PSEUDOCÓDIGO

```
Entrada: n  (entero no negativo)
Salida:  n! (factorial de n)

Procesamiento:
      Si n es 0,
      evalúa a 1
      en caso contrario
      evalúa a n*(n-1)!
```

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; factorial(n)
;;;   Calcula el factorial de un entero
;;;
;;;   INPUT:   n:  Entero no negativo
;;;   OUTPUT:  n!: Factorial de n
;;;
(defun factorial (n)
  (if (= n 0)
      1                              ; caso base: 0! = 1
      (* n (factorial (- n 1))))) ; recursión: n! = n (n-1)!
;;;
;;;   EJEMPLOS:
;;;   (factorial 0) ;-> 1      ; caso especial
;;;   (factorial 5) ;-> 120  ; caso típico
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

COMENTARIOS:

- Se ha realizado una implementación basada en la definición recursiva

  ```
  n! = n (n-1)!      [recursión]
  0! = 1             [caso base]
  ```

- La implementación basada en la fórmula

  ```
  n! = \prod_{i=1}^n i = 1*2*...*(n-1)*n
  ```

  requiere el uso de una variable que almacene los productos parciales y bucles, por lo que no sería funcional.

  ```
  factorial = 1.0;
  for i := 1 To n
     factorial = factorial*i;
  end
  ```

- Se ha implementado el factorial en aritmética de enteros. Puede que sea más razonable hacerlo en aritmética de reales (doble precisión) para evitar desbordamientos numéricos.
- La función Gamma(x) generaliza el concepto de factorial para reales. Se podría también realizar una implementación basada en la evaluación numérica de esta función utilizando la igualdad

  ```
  n! = Gamma(n+1)
  ```