PRÁCTICA 1. PROGRAMACIÓN EN LISP

Fecha de publicación: 2014/02/22
Fecha límite de entrega: 2014/02/18, 23:55
Planificación: semana 1 Ejercicios 1,2,3
semana 2 Ejercicios 4,5 Versión: 2014/01/22 19:00

Forma de entrega: Según lo dispuesto en las normas de entrega generales, publicadas en Moodle

- El código debe estar en un único fichero.
- 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 comentari oel 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, MAPCAN`
  - 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 `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 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 ..))
```

Consideraciones adicionales

- Las implementaciones de las funciones no deben ser destructivas.
- En LISP existen funciones para tratar las listas como conjuntos: ***member, member-if, subsetp, adjoin, union, intersection, set-difference, remove-duplicates*.**
- Utilice **recursión, *mapcar y mapcan***.
- En el caso de que las **listas sean consideradas como conjuntos**, el **orden** en el que aparecen los elementos del conjunto **no es importante** (ej. el orden de los literales en una cláusula, el orden de las cláusulas en una FNC).
- **No** se puede utilizar *setf*.
- **No** se puede utilizar bucles explícitos (*loop*, *dolist*, etc.)
- Se debe utilizar **let** para evitar evaluar la misma expresión varias veces.
- En la memoria se deben incluir
  - Breve descripción de la implementación elegida.
  - Ejemplos que se han utilizado para evaluar la corrección del código.
  - Comentarios adicionales sobre la implementación, uso, excepciones, etc.
  - Respuestas con explicación a las cuestiones concretas planteadas en cada apartado (en caso de que las hubiera) y al final de la práctica.

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 elt-1
              (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>)`

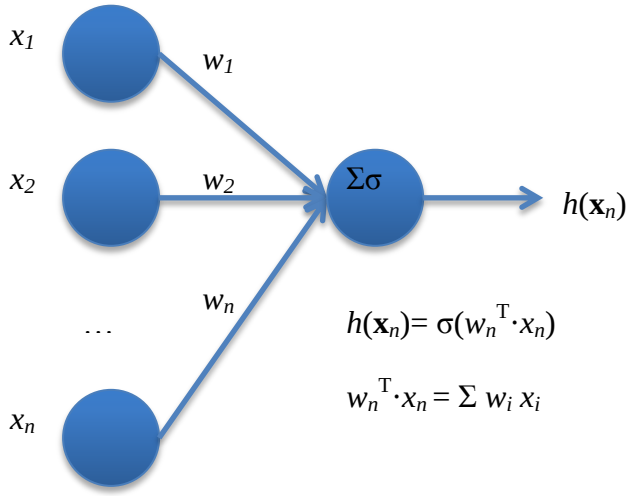
LISP I, EJERCICIOS

Evaluación: Para evaluar el trabajo se tendrá en cuenta el correcto funcionamiento del código entregado, el estilo de programación, el contenido de la memoria y un examen.

1. No se puede utilizar `setf` o similares dentro de las funciones.
2. Ninguna función debe ser destructiva.
3. No se puede utilizar iteración directa (`dolist, do, dotimes`…).
4. Utiliza `mapcar, mapcan` o recursión.
5. No utilices `mapcar` en caso de que no sea estrictamente necesario recorrer toda la lista. En ese caso, utiliza recursión.
6. Se pueden definir funciones auxiliares en caso de que estas fueran necesarias.
7. Reutiliza código.
8. Utiliza el comando LISP `trace` para depurar el código.

1. Activación neuronal [2 punto]

Implementa (1) de forma recursiva y (2) con `mapcar` una función que calcule la salida del siguiente perceptrón lineal:



Es necesario definir varias funciones de activación (**σ**):

- Tangente hiperbólica: $tanhip(z) = (e^z - e^{-z}) / (e^z + e^{-z})$
- Sigmoidal logística: $logit(z) = 1 / (1 + e^{-z})$

```
Prototipos:  (defun tanhip (z) ...)
             (defun logit (z) ...)
             (defun h-recursive (x v sigma) ...)
             (defun h-mapcar (x w sigma) ...)

Ejemplo:
>> (h-mapcar '(0.1 -0.5 0.7) '(-0.1 0.2 0.3) #'tanhip)
0.099668
```

2. Combinación de listas [2 puntos]

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

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

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

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

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

Ejemplo:
>> (combine-lst-lst '(a b c) '(1 2))
((A 1) (A 2) (B 1) (B 2) (C 1) (C 2))
```

2.3 Producto de lista de listas: Diseñar una función que calcule todas las posibles disposiciones de elementos pertenecientes a n listas de forma que en cada disposición aparezca únicamente un elemento de cada lista.

IMPORTANTE: **Puede que sea necesario crear funciones distintas a la de los apartados anteriores.**

```
Prototipo: (defun combine-list-of-lsts (lstolsts)...)

Ejemplo:
>> (combine-list-of-lsts '((a b c) (+ -) (1 2 3 4)))
   ((A + 1) (A + 2) (A + 3) (A + 4) (A - 1) (A - 2) (A - 3) (A - 4)
    (B + 1) (B + 2) (B + 3) (B + 4) (B - 1) (B - 2) (B - 3) (B - 4)
    (C + 1) (C + 2) (C + 3) (C + 4) (C - 1) (C - 2) (C - 3) (C - 4))
```

3. Tablas de verdad [4 puntos]

Una **base de conocimiento** Δ consiste en una colección de FBFs que reflejan conocimiento sobre un dominio determinado

$$\Delta = \{w_1, w_2, \ldots, w_n\}.$$

Implícitamente se supone que hay un conector $\wedge$ entre las FBFs de una base de conocimiento

$$w_1 \wedge w_2 \wedge \ldots \wedge w_n$$

Representaremos una base de conocimiento en LISP como una lista de FBFs en formato prefijo

Ejemplo: $\Delta = \{ p <=> \neg q,\ r => (q \wedge p),\ a <=> p)))$

EN LISP '(^ (<=> P (¬ Q)) (=> R (^ Q P)) (<=> A P)))

El objetivo de este ejercicio es utilizar las tablas de verdad para determinar si una base de conocimiento es satisfactible (¿es Δ SAT?) o no. Es decir buscaremos **interpretaciones** que sean **modelos** de Δ.

Valor	Representación LISP
Verdadero	T

Falso	NIL

Una **interpretación** es un conjunto de asociaciones de átomos a proposiciones sobre el entorno (mundo real). En una interpretación, la proposición asociada a un átomo es llamada la denotación del átomo. Dentro de una interpretación dada, cada uno los átomos tiene un valor de verdad (Verdadero o Falso) determinado por la verdad o falsedad de la correspondiente proposición sobre el mundo.

Las interpretaciones se representarán como listas LISP de parejas (representadas como listas LISP de dos elementos) que asocian átomo simbólico y valor de verdad (primero el átomo y luego el valor).

Ejemplo:

Átomos:

Átomo	Denotación
P	La pila está cargada
I	Interruptor en posición “Encendido”
L	La linterna está encendida

Interpretación

Hechos	Interpretación
La pila está descargada	(P Falso)
El interruptor está en posición “Encendido”	(I Verdadero)
La linterna está apagada	(L Falso)

EN LISP: ((P NIL) (I T) (L T))

En LISP, la lista de las 8 interpretaciones posibles (2 elevado al número de átomos):

```
(((P T)   (I T)   (L T)) ((P T)   (I T)   (L NIL))
 ((P T)   (I NIL) (L T)) ((P T)   (I NIL) (L NIL))
 ((P NIL) (I T)   (L T)) ((P NIL) (I T)   (L NIL))
 ((P NIL) (I NIL) (L T)) ((P NIL) (I NIL) (L NIL)))
```

Supongamos que tenemos la siguiente BASE DE CONOCIMIENTO:

$\Delta = \{P \wedge I \Rightarrow L,\ \neg P \Rightarrow \neg L\ ,\ \neg P\ ,\ L\}$

Pregunta: ¿Es Δ SAT?
Es decir ¿Existe alguna interpretación que sea un modelo para nuestra base de conocimiento?

Respuesta: Δ es UNSAT porque no existe ninguna interpretación tal que todas las FBFs de Δ tengan valor de verdad True.
Vamos a analizar (razonamiento informal) si las conclusiones son conformes a nuestro conocimiento sobre el mundo.

Nuestra base de conocimiento Δ está compuesta por hechos (¬P, L) y por reglas (¬P ⇒ ¬L, P∧I ⇒ L).

Reglas: Lo que sabemos sobre cómo funciona el mundo de linternas a pilas accionadas por un interruptor.

1. ¬P ⇒ ¬L "Si la pila está descargada la linterna no se enciende"

2. P∧I ⇒ L "Si la pila está cargada y el interruptor en posición encendido la linterna se enciende"

Hechos:
"La pila está descargada" ¬P
"La linterna está encendida" L

Como podemos ver, los hechos ¬P, L y la regla ¬P ⇒ ¬L son incompatibles: Cuando la pila está descargada la linterna no puede estar encendida.

Luego, la base de conocimiento inicial es insatisfactible (UNSAT).

En LISP, la base de conocimiento Δ

Δ = {P∧I ⇒ L, ¬P ⇒ ¬L , ¬P , L}

será representada como una lista de FBFs en formato prefijo, es decir

```
(setf Delta '((=> (^ P I) L) (=> (¬ P) (¬ L)) (v P) (L))
```

4.1 Utilizando las definiciones

```
(defconstant +bicond+ '<=>)
(defconstant +cond+   '=>)
(defconstant +and+    '^)
(defconstant +or+     'v)
(defconstant +not+    '¬)
```

y los predicados

```
(defun truth-value-p (x)
  (or (eql x T) (eql x NIL)))

(defun unary-connector-p (x)
  (eql x +not+))

(defun binary-connector-p (x)
  (or (eql x +bicond+)
      (eql x +cond+)))

(defun n-ary-connector-p (x)
  (or (eql x +and+)
      (eql x +or+)))

(defun connector-p (x)
  (or (unary-connector-p  x)
      (binary-connector-p x)
      (n-ary-connector-p   x)))
```

diseñar una función LISP para extraer todos los símbolos atómicos a partir de una FBF en cualquier tipo de formato.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO: extrae-simbolos
;;
;; RECIBE : fórmula bien formada en cualquier formato (expr)
;; EVALÚA A : Lista de símbolos atómicos (sin repeticiones)
;; utilizados en la fórmula bien formada. El orden en la lista no es
;; relevante.
;;
(defun extrae-simbolos (expr) ...)
```

3.2 A partir de una lista con N símbolos, escriba una función LISP que genere una lista con las 2^N posibles interpretaciones

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO: genera-lista-interpretaciones
;;
;;
;; RECIBE   : Lista de N símbolos atómicos
;; EVALÚA A : Lista de 2^N posibles interpretaciones
;;              (lista de pares (<símbolo> <valor de verdad>))
;;
;;
;;

(defun genera-lista-interpretaciones (lst) ...)

EJEMPLOS:

(genera-lista-interpretaciones '(P I L))

     ;;;        (((P T)   (I T)   (L T)) ((P T)   (I T)   (L NIL))
     ;;;         ((P T)   (I NIL) (L T)) ((P T)   (I NIL) (L NIL))
     ;;;         ((P NIL) (I T)   (L T)) ((P NIL) (I T)   (L NIL))
     ;;;         ((P NIL) (I NIL) (L T)) ((P NIL) (I NIL) (L NIL)))
```

3.3 Escriba un predicado LISP que permita saber si una interpretación es modelo de una base de conocimiento

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO interpretacion-modelo-p
;;
;; RECIBE   : base de conocimiento (KB) e interpretación
;; EVALÚA A : T en caso de que la interpretación sea un modelo de KB
;;            NIL en caso contrario
;;
(defun interpretacion-modelo-p (kb interpretacion) ...)

EJEMPLOS:

(interpretacion-modelo-p  '((<=> A (¬ H)) (<=> P (^ A  H)) (<=> H P))
                          '((A NIL) (P NIL) (H T)))
;;; NIL

(interpretacion-modelo-p '((<=> A (¬ H)) (<=> P (^ A  H)) (<=> H P))
                          '((A T) (P NIL) (H NIL)))
```

```
;;; T
```

3.4 Escriba una función que permita encontrar todas las interpretaciones que son modelo de una base de conocimiento

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO: encuentra-modelos
;;
;; RECIBE   : base de conocimiento (KB)
;; EVALÚA A : lista de interpretaciones que son modelo para KB
;;
;;
(defun encuentra-modelos (kb) ...)

EJEMPLOS:
(encuentra-modelos '((=> A (¬ H)) (<=> P (^ A  H)) (=> H P)))
;;; (((A T) (P NIL) (H NIL)) ((A NIL) (P NIL) (H NIL)))

(encuentra-modelos '((=> (^ P I)  L)  (=> (¬ P) (¬ L)) (¬ P) L))
;;; NIL
```

3.5 Utilizando las funciones anteriores, escriba una función que determine si una base de conocimiento es SAT o no

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO SAT-p
;;
;; RECIBE   : base de conocimiento (KB)
;; EVALÚA A : T si existe al menos un modelo para kb
;;            NIL en caso contrario
;;
;;
EJEMPLOS:

(SAT-p '((<=> A (¬ H)) (<=> P (^ A  H)) (<=> H P)))         ;;; T
(SAT-p '((=> (^ P I)  L)  (=> (¬ P) (¬ L))   (¬ P)  L))     ;;; NIL
```

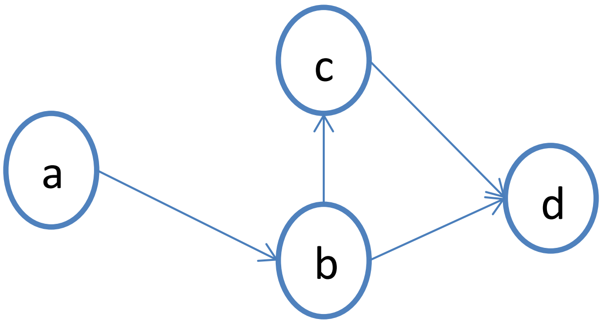
4. Búsqueda en anchura [1 punto]

BÚSQUEDA EN ANCHURA

Este es la implementación del algoritmo de búsqueda en anchura incluida en
"ANSI Common Lisp", Paul Graham (Fig. 3.12, pg. 52) [http://www.paulgraham.com/acl.html]

Grafo: Lista cuyos elementos son de la forma (<nodo> <nodos-adjacentes>)
Los nodos se identifican por el símbolo correspondiente a su etiqueta.

Ejemplo: ((a c) (b d) (c b d) (d))




```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;; Breadth-first-search in graphs
(defun bfs (end queue net)
  (if (null queue)
      nil
    (let ((path (car queue)))
      (let ((node (car path)))
        (if (eql node end)
            (reverse path)
          (bfs end
               (append (cdr queue)
                       (new-paths path node net))
               net))))))

(defun new-paths (path node net)
  (mapcar #'(lambda(n)
              (cons n path))
    (cdr (assoc node net))))
;;;
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

(i) Describe el algoritmo de búsqueda en anchura e ilústralo resolviendo a mano algunos ejemplos ilustrativos de búsqueda en grafos.
   a. Casos especiales.
   b. Caso típico: grafo dirigido ejemplo.
   c. Caso típico distinto del grafo ejemplo anterior.

(ii) Explica por qué esta función LISP resuelve el problema de encontrar el camino más corto entre dos nodos del grafo.

```
(defun shortest-path (start end net)
   (bfs end (list (list start)) net))
```

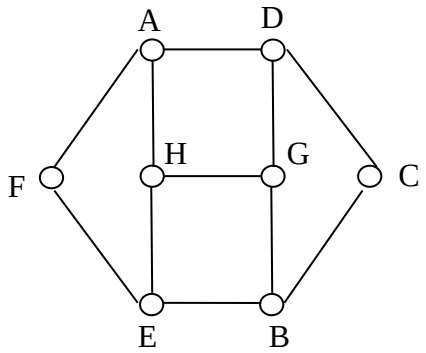
(iii) Escribe pseudocódigo para implementar el algoritmo de búsqueda en anchura.

(iv) Por comentarios en el código propuesto por Graham de forma que se ilustre cómo se ha traducido el pseudocódigo propuesto en (iii).

(v) Ilustra el funcionamiento del código con los ejemplos que has resuelto a mano en (i). En concreto, especifica la secuencia de llamadas a las que da lugar la evaluación

```
(shortest-path 'a 'd '((a b) (b c d) (c d) (d)))
```

(vi) Utilizando el código anterior,
   a. ¿cuál es la evaluación para determina el camino más corto entre f y c en el grafo no dirigido?
   b. ¿cuál es el resultado de dicha evaluación?



(vii) El código anterior falla (entra en una recursión infinita) cuando hay ciclos en el grafo y el problema de búsqueda no tiene solución.
Modifica el código de forma que sea corregido este problema.

```
(defun bfs-improved (end queue net) ...)
(defun shortest-path-improved (end queue net) ...)
```

PLANTILLA PARA LA MEMORIA: Máximo 1 página [= 2 caras] por función

1. Nombre y sintaxis de la función

```
;; <fn-name>:  <one-line description of the function>
;; SYNTAX: (<fn-name>  <arg-1> <arg-2> ... <arg-n>)
```

2. Batería de ejemplos de prueba

```
<test 1>  ; <explanation of test 1>
<test 2>  ; <explanation of test 2>
<test 3>  ; <explanation of test 3>
   ...
```

3. Comentarios sobre la implementación

```
c1. <comment 1>
c2. <comment 2>
c3. <comment 3>
...
```

4. Respuestas a las preguntas

```
q1. <question 1>: <answer to question 1>
q2. <question 2>: <answer to question 2>
q3. <question 3>: <answer to question 3>
...
```

5. Otros comentarios

6. Pseudocódigo

```
<pseudocode>
```

7. Código

```
;; <fn-name>:  <one-line description of the function>
;; SYNTAX: (<fn-name>  <arg-1> <arg-2> ... <arg-n>)
;;
;; INPUT:   <arg-1>:  <description of arg-1>
;;              <arg-2>:  <description of arg-2>
;;          ...
;;         <arg-n>:  <description of arg-n>
;;
;; OUTPUT:  <description of output>


<code>
```