PRÁCTICA 1: LISP

FECHAS DE ENTREGA:

- Entrega parcial (ejercicios 1, 2 y 3):
  - Grupos de los martes: 8 de febrero (23:55)
  - Grupos de los jueves: 10 de febrero (23:55)
  - Grupos de los viernes: 11 de febrero (23:55)
- Entrega final (todos los ejercicios, incluyendo los ejercicios 1,2 y 3 corregidos):
  - Grupos de los martes: 22 de febrero (23:55)
  - Grupos de los jueves: 24 de febrero (23:55)
  - Grupos de los viernes: 25 de febrero (23:55)

EVALUACIÓN:

- Normativa de prácticas.
- Criterios de evaluación.
- Pesos en la evaluación:
  - Correcto funcionamiento: 40 %
  - Estilo de programación: 30 %
  - Contenido de la memoria: 30 %
- Errores más frecuentes (a evitar).

1. Vector más cercano a un vector dado [1,5 puntos]

Dados dos vectores x e y, la *distancia de Minkowski* (también llamada habitualmente *norma p*, l_p o L_p) se define mediante la siguiente fórmula, donde p es el número que determina la norma calculada y D es la dimensión de los vectores:

$$lp\ (\vec{x}, \vec{y}) = \left(\sum_{d=1}^{D} |x_d - y_d|^p\right)^{1/p}$$

1.1) Implementa una función que calcule la norma Lp de dos vectores x, y de dos formas:

- **Recursiva**
- **Usando mapcar**

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; lp-rec (x y p)
;;;   Calcula la norma Lp de la diferencia de dos vectores de
;;;   forma recursiva
;;;
;;;   INPUT:   x: vector 1
;;;            y: vector 2
;;;            p: norma que se quiere calcular
;;;
;;;   OUTPUT:  norma Lp de x-y
;;;
(defun lp-rec (x y p) ...)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; lp-mapcar (x y p)
;;;   Calcula la norma Lp de la diferencia de dos vectores
;;;   usando mapcar
;;;
;;;   INPUT:   x: vector 1
;;;            y: vector 2
;;;            p: norma que se quiere calcular
;;;
;;;   OUTPUT:  norma Lp de x-y
;;;
(defun lp-mapcar (x y p) ...)
```

No utilices bucles ni llames a la función *length* para realizar ningún cálculo. Esto es aplicable no sólo a este ejercicio, sino a toda la práctica.

Puedes definir funciones auxiliares además de las solicitadas, si ves que eso te facilita la tarea. Esto también es aplicable a toda la práctica.

La función nativa *expt* puede serte de utilidad para calcular potencias.

Algunos casos particulares de la norma $L_p$muy populares son los siguientes:

- Distancia euclidiana o L_2: $\|\vec{x} - \vec{y}\|_2 = \sqrt{\sum_{d=1}^{D}(x_d - y_d)^2}$
- Distancia de Manhattan o L_1: $\|\vec{x} - \vec{y}\|_1 = \sum_{d=1}^{D}|x_d - y_d|$
- Distancia de Chebyshev o L_∞: $\|\vec{x} - \vec{y}\|_\infty = \max_{d=1}^{D}|x_d - y_d|$

1.2) A partir de las dos implementaciones de arriba de *lp*, define funciones *l2-rec*, *l2-mapcar*, *l1-rec* y *l1-mapcar*. Haz pruebas con distintos vectores para comprobar la correcta implementación. ¿Qué sugerirías para probar la norma L_∞?

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; l2-rec (x y)
;;;   Calcula la norma L2 de la diferencia de dos vectores de
;;;   forma recursiva
;;;
;;;   INPUT:   x: vector 1
;;;            y: vector 2
;;;
;;;   OUTPUT:  norma L2 de x-y
;;;
(defun l2-rec (x y) ...)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; l2-mapcar (x y)
;;;   Calcula la norma L2 de la diferencia de dos vectores
;;;   usando mapcar
;;;
;;;   INPUT:   x: vector 1
;;;            y: vector 2
;;;
;;;   OUTPUT:  norma L2 de x-y
;;;
(defun l2-mapcar (x y) ...)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; l1-rec (x y)
;;;   Calcula la norma L1 de la diferencia de dos vectores de
;;;   forma recursiva
;;;
;;;   INPUT:   x: vector 1
;;;            y: vector 2
;;;
;;;   OUTPUT:  norma L1 de x-y
;;;
(defun l1-rec (x y) ...)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; l1-mapcar (x y)
;;;   Calcula la norma L1 de la diferencia de dos vectores
;;;   usando mapcar
;;;
;;;   INPUT:   x: vector 1
;;;            y: vector 2
;;;
;;;   OUTPUT:  norma L1 de x-y
;;;
(defun l1-mapcar (x y) ...)
```

Observa la siguiente función que calcula el vector más cercano a uno dado, en términos de una función de distancia que se pasa como argumento:

```
(defun nearest (vector vectors distance)
(unless (null vectors)
    (reduce
    #'(lambda (x y)
         (if (< (funcall distance vector x)
                (funcall distance vector y))
           x
           y)) vectors)))
```

1.3) Haz pruebas llamando a *nearest* con las distintas variantes de normas. Verifica y compara los resultados y tiempos de ejecución. Documenta en la memoria las conclusiones a las que llegues.

Un ejemplo sencillo de llamadas que podrían hacerse sería el siguiente:

```
>> (setf vectors '((0.1 0.1 0.1) (0.2 -0.1 -0.1)))
>> (nearest '(1.0 -2.0 3.0) vectors #'l2-mapcar)
(0.1 0.1 0.1)
>> (nearest '(1.0 -2.0 3.0) vectors #'l1-rec)
(0.2 -0.1 -0.1)
```

2. Ceros de una función [1,5 puntos]

Dada una función $f(x)$ en el dominio de los reales, que asumiremos continua, queremos encontrar un *cero* de f; es decir, algún valor x^* tal que $f(x^*) = 0$.

Para resolver esta tarea, uno de los algoritmos más conocidos es el *método de Newton* (también llamado a veces *Newton-Raphson*). Partiendo de la semilla x_0, que es una estimación inicial del valor del cero de la función proporcionada por el usuario, este algoritmo va iterando sucesivamente en busca del cero de f mediante la fórmula:

$$x_0 \text{ (semilla)}$$
$$x_{n+1} = x_n - \frac{f(x_n)}{g(x_n)} \quad n = 0,1,2 \ldots$$

donde g es la derivada de f.

Se espera que los valores x_n proporcionen estimaciones progresivamente más precisas del cero x^* a medida que se incrementa n. El algoritmo termina cuando se cumple:

1. La estimación x_{n+1} está suficientemente cerca de x^*. Esto normalmente se determina de manera aproximada cuando el cambio en el valor estimado para x^* en iteraciones sucesivas es suficientemente pequeño. Para ello, se considera un valor para la tolerancia `tol-abs`, de forma que el algoritmo para cuando se cumple la condición $|x_{n+1} - x_n| <$ `tol-abs`. En este caso se considera que el algoritmo *converge* y proporciona la estimación x_{n+1} para el cero de la función.
2. Se ha alcanzado un número máximo de iteraciones sin haber terminado por la condición anterior. En este caso se considera que el algoritmo *diverge*.

2.1) Implementa una función *newton* que realice el proceso descrito arriba. Prueba con distintas funciones, semillas y valores para asegurar el correcto funcionamiento:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; newton (f g tol-abs max-iter x0)
;;;   Estima el cero de una función mediante Newton_Raphson
;;;
;;;   INPUT:        f:  función cuyo cero se desea encontrar
;;;                 g:  derivada de f
;;;          tol-abs:  tolerancia para convergencia
;;;         max-iter:  máximo número de iteraciones
;;;               x0:  estimación inicial del cero (semilla)
;;;
;;;   OUTPUT:  estimación del cero de f, o NIL si no converge
;;;
(defun newton (f g tol-abs max-iter x0) ...)
```

2.2) Codifica una función *un-cero-newton* que llame a *newton* con distintas semillas hasta encontrar un cero de la función. Prueba a variar los parámetros de entrada y discute los resultados obtenidos:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; un-cero-newton (f g tol-abs max-iter semillas)
;;;   Prueba con distintas semillas iniciales hasta que Newton
;;;   converge
;;;
;;;   INPUT:        f:  función de la que se desea encontrar un cero
;;;                 g:  derivada de f
;;;          tol-abs:  tolerancia para convergencia
;;;         max-iter:  máximo número de iteraciones
```

```
;;;          semillas:  semillas con las que invocar a Newton
;;;
;;;   OUTPUT:  el primer cero de f que se encuentre, o NIL si se diverge
;;;       para todas las semillas
;;;
(defun un-cero-newton (f g tol-abs max-iter semillas) ...)
```

2.3) Codifica una función *todos-ceros-newton* que llame a *newton* con distintas semillas y devuelva todas las raíces encontradas (<u>no sólo la primera como en el apartado anterior, sino todas</u>). Prueba a variar los parámetros de entrada y discute los resultados obtenidos:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; todos-ceros-newton (f g tol-abs max-iter semillas)
;;;   Prueba con distintas semillas iniciales y devuelve las raíces
;;;   encontradas por Newton para dichas semillas
;;;
;;;   INPUT:        f:  función de la que se desea encontrar un cero
;;;                 g:  derivada de f
;;;          tol-abs:  tolerancia para convergencia
;;;         max-iter:  máximo número de iteraciones
;;;          semillas:  semillas con las que invocar a Newton
;;;
;;;   OUTPUT: todas las raíces que se encuentren, o NIL si se diverge
;;;       para todas las semillas
;;;
(defun todos-ceros-newton (f g tol-abs max-iter semillas) ...)
```

Algunos ejemplos sencillos de llamadas:

```
>> (setf tol 1e-6)
>> (setf iters 50)
>> (setf funcion (lambda (x) (* (- x 4) (- x 1) (+ x 3))))  ; raíces: 4, 1, -3
>> (setf derivada (lambda (x) (- (* x (- (* x 3) 4)) 11)))  ; 3x^2 - 4x – 11
>> (setf semilla1 2.35287527)
>> (setf semilla2 2.35284172)
>> (setf semilla3 2.352836323)
>> (setf lst-semillas1 (list semilla1 semilla2 semilla3))
>> (setf lst-semillas2 (list semilla2 semilla3 semilla1))
>> (setf lst-semillas3 (list semilla3 semilla1 semilla2))
>> (newton funcion derivada tol iterssemilla1)
4.0
>> (newton funcion derivada tol iterssemilla2)
-3.0
>> (newton funcion derivada tol iterssemilla3)
1.0
>> (un-cero-newton funcion derivada tol iters lst-semillas1)
4.0
>> (un-cero-newton funcion derivada tol iters lst-semillas2)
-3.0
>> (un-cero-newton funcion derivada tol iters lst-semillas3)
1.0
>> (todos-ceros-newton funcion derivada tol iters lst-semillas1)
(4.0 -3.0 1.0)
>> (todos-ceros-newton funcion derivada tol iters lst-semillas2)
(-3.0 1.0 4.0)
>> (todos-ceros-newton funcion derivada tol iters lst-semillas3)
(1.0 4.0 -3.0)
```

3. Combinación de listas [1,5 puntos]

3.1) Define una función que combine un elemento dado con todos los elementos de una lista:

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

>> (combine-elt-lst 'a nil)
NIL
>> (combine-elt-lst 'a '(1 2 3))
((A 1) (A 2) (A 3))
```

3.2) Diseña una función que calcule el producto cartesiano de dos listas:

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

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

3.3) Diseña 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:

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

>> (combine-list-of-lsts '(() (+ -) (1 2 3 4)))
NIL
>> (combine-list-of-lsts '((a b c) () (1 2 3 4)))
NIL
>> (combine-list-of-lsts '((a b c) (1 2 3 4) ()))
NIL
>> (combine-list-of-lsts '((1 2 3 4)))
((1) (2) (3) (4))
>> (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))
```

4. Problema SAT [4 puntos]

El problema de la *satisfactibilidad booleana*, abreviado comúnmente *SAT*, consiste en averiguar si en una expresión booleana dada existe al menos una asignación booleana a sus variables (es decir, se pueden poner a true o false de alguna manera) tal que haga que el valor de la expresión sea true.

Por ejemplo, usando notación C, la expresión (A && (!A)) no es satisfactible, ya que el valor de la expresión es falso, independientemente de que A sea true o false. En cambio, (A || (!A)) es trivialmente satisfactible, ya que la expresión es siempre verdadera, ya se ponga A a true o a false.

En nuestro caso, partiremos de una *base de conocimiento* (que denotaremos con la letra griega Δ) que modela lo que sabemos sobre un determinado dominio. Esta base estará formada por una serie de proposiciones lógicas:

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

Donde las proposiciones w_i representan hechos y reglas que conocemos sobre dicho dominio. Por tanto, implícitamente se asume que hay un conector AND (denotado por el símbolo ^) entre las proposiciones de una base de conocimiento, de forma que para saber si nuestra base de conocimiento se satisface basta con averiguar el valor de la expresión

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

Ya en notación LISP, **una base Δ se representará mediante una lista de proposiciones lógicas. Una proposición lógica a su vez podrá ser 1) atómica (un literal como P o Q), 2) una serie (lista) de otras proposiciones conectadas mediante un conector. Los conectores permitidos son los siguientes:**

- **not: denotado mediante el símbolo ¬ (conector unario: sólo admite una proposición).**
- **condicional: denotado mediante el símbolo => (binario: dos proposiciones).**
- **bicondicional: denotado mediante el símbolo <=> (binario: dos proposiciones).**
- **and: denotado mediante el símbolo ^ (n-nario: n proposiciones, con n >= 2).**
- **or: denotado mediante la letra v (n-ario: n proposiciones, con n >= 2).**

Donde se usará notación prefija para las conexiones, al estilo del resto de operadores de LISP. Por ejemplo, la traducción de la base Δ = {p<=> ¬q, r => (q v p v a), (a <=> p) ^ q} a LISP sería la lista:

```
((<=> P (¬ Q)) (=> R (v Q P A)) (^ (<=> A P) Q))
```

Donde las 3 proposiciones se entiende que estarían conectadas mediante un *and* implícito. La primera proposición es una bicondicional de 2 proposiciones: una atómica con el literal P, y otra que es el *not* de otra proposición atómica Q. La segunda proposición es una condicional de la proposición atómica R y la proposición n-aria (v Q P A), la cual es un *or* de las correspondientes 3 proposiciones atómicas. Por último, la tercera proposición es un and de una primera proposición bicondicional y una segunda atómica.

Por concretar más, supongamos que tenemos la siguiente base de conocimiento:

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

Donde la interpretación de las variables es la siguiente:

- P: la pila está cargada.
- I: el interruptor está en posición de encendido.
- L: la linterna luce.

Es decir, que lo que nos está diciendo la base es:

- Si la pila está cargada y el interruptor está en encendido, entonces la linterna luce.
- Si la pila está descargada entonces la linterna no luce.
- La pila está descargada.
- La linterna luce.

Con lo cual resolver el problema SAT sobre esta base es ver si esta conjunción de hechos es o no posible. En este caso vemos inmediatamente que la respuesta es que no, porque las tres últimas expresiones son incompatibles entre sí.

Para comprobarlo formalmente habría que ver que ninguna de las asignaciones posibles de valores de verdad para las variables da como resultado que la base se satisfaga.**Dichas asignaciones (también llamadas *interpretaciones*) las representaremos en LISP en forma de pares (variable valor)**: por ejemplo, (P T) querría decir que la pila está cargada, mientras que (P NIL) sería que la pila no está cargada.

Dado que para este ejemplo se puede actuar sobre 3 variables, habrá $2^3 = 8$ interpretaciones posibles a comprobar, determinadas por la siguiente lista:

```
(((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)))
```

Podemos concluir que dicha base no es satisfactible dado que, si se prueba con cada una de estas 8 interpretaciones, el valor de la lista que representa a la base resulta ser siempre false (es decir, *nil*). En otras palabras, decimos que ninguna de las interpretaciones resulta ser un *modelo* de la base de conocimiento

```
((=> (^ P I) L) (=> (¬ P) (¬ L)) (¬ P) L)
```

Para conseguir implementar lo descrito arriba, partiremos de las siguientes definiciones para los conectores:

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

Así como de 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-px)
      (binary-connector-p x)
      (n-ary-connector-p x)))
```

4.1) Diseña un predicado que determine si una proposición lógica en formato prefijo está correctamente formada o no. A partir del anterior, codifica otro predicado que determine si una base de conocimiento está correctamente formada.

```
(defun proposicion-p (expr) ...)

>> (proposicion-p 'A)
T
>> (proposicion-p '(H <=> (¬ H)))
NIL
>> (proposicion-p '(<=> H (¬ H)))
T

(defun base-p (expr) ...)

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

Las proposiciones lógicas no atómicas se componen de otras proposiciones, lo que sugiere recursión.

4.2) Diseña una función que extraiga una lista con todos los símbolos atómicos (sin repeticiones) de una base de conocimiento. <u>El orden en que se obtengan dichos símbolos no es relevante</u>.

```
(defun extrae-simbolos (kb) ...)

>> (extrae-simbolos '(A))
(A)
>> (extrae-simbolos '((v (¬ A) A B (¬ B))))
(A B)
>> (extrae-simbolos '((=> A (¬ H)) (<=> P (^ A (¬ H))) (=> H P)))
(A P H)
```

4.3) A partir de una lista con N símbolos (correspondientes a átomos simbólicos), escribe una función que genere una lista con las 2^N posibles interpretaciones. <u>El orden de las interpretaciones no es relevante</u>.

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

>> (genera-lista-interpretaciones nil)
NIL
>> (genera-lista-interpretaciones '(P))
(((P T)) ((P NIL)))
>> (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)))
```

Intenta reutilizar el código de combinación de listas que implementaste para el apartado 3.

4.4) Escribe un predicado que permita saber si una interpretación es modelo de una base de conocimiento:

```
(defun interpretacion-modelo-p (kb interp) ...)

>> (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
```

4.5) Escribe una función que permita encontrar todas las interpretaciones que son modelo de una base de conocimiento:

```
(defun encuentra-modelos (kb) ...)

>> (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
```

4.6) Utilizando las funciones anteriores, diseña un predicado que determine si una base de conocimiento es satisfactible o no.

```
(defun SAT-p (kb) ...)

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

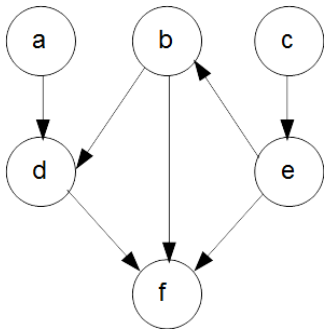
5. Búsqueda en anchura [1,5 puntos]

Los *grafos* son un tipo de datos de los más utilizados en informática para modelar distintos problemas. Matemáticamente, un *grafo* viene determinado por un conjunto de *nodos* o *vértices* (abreviado normalmente mediante V), y un conjunto de *aristas* entre pares de vértices (denotado comúnmente E).

Dado que los elementos de E son pares de vértices en V, se suelen usar dos representaciones de grafos:

- *Matriz de adyacencia*: una matriz cuadrada del tamaño de V en la que la casilla (i, j) es 1 si existe una arista del vértice i al vértice j, y 0 si no existe tal arista.
- *Listas de adyacencia*: una lista por cada vértice en la que se especifican sus *vecinos* o *vértices adyacentes* (es decir, con qué nodos está conectado mediante alguna arista).

Dada que LISP trabaja nativamente con listas usaremos una representación basada en listas de adyacencia. En concreto, un grafo viene dado por una lista de listas de adyacencia, donde el primer elemento de cada lista de adyacencia es el vértice origen y el resto son sus vecinos. Por ejemplo, dado el grafo



Su representación será la lista

```
((a d) (b d f) (c e) (d f) (e b f) (f))
```

La *búsqueda en anchura* (Breadth-First Search, BFS) es probablemente el algoritmo más intuitivo para recorrer un grafo. El nombre proviene del hecho de que la búsqueda se realiza "a lo ancho" a partir de un nodo raíz. Primero se exploran los vecinos de dicho nodo raíz (vecinos de primer nivel). A continuación, para cada uno de estos vecinos se exploran sus respectivos vecinos (vecinos de segundo nivel). El proceso se repite de la misma forma para los distintos niveles, hasta completar el grafo. A diferencia de la *búsqueda en profundidad* (Depth-First Search o DFS), no se empiezan a procesar los vértices de un nivel hasta que no se hayan procesado todos los vértices del nivel anterior.

5.1) Ilustra el funcionamiento del algoritmo resolviendo a mano algunos ejemplos ilustrativos:

- Grafos especiales.
- Caso típico (grafo dirigido ejemplo).
- Caso típico distinto del grafo ejemplo anterior.

5.2) Escribe el pseudocódigo correspondiente al algoritmoBFS.

5.3) Estudia con detalle la siguiente implementación del algoritmo BFS, tomada del libro "*ANSI CommonLisp*" de Paul Graham [http://www.paulgraham.com/acl.html]. Asegúrate de que comprendes su funcionamiento con algún grafo sencillo.

La función *assoc* devuelve la sublista dentro de una lista cuyo *car* sea el elemento que se le pase. Por ejemplo, si llamamos a la lista de arriba *grafo*, (assoc 'b grafo) devolvería la sublista (b d f).

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;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))))
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

5.4) Pon comentarios en el código anterior, de forma que se ilustre cómo se ha implementado el pseudocódigo propuesto en 4.2).

5.5) Explica por qué esta función resuelve el problema de encontrar el camino más corto entre dos nodos del grafo:

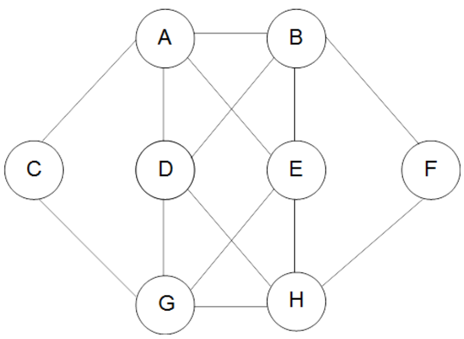
```
(defun shortest-path (start end net)
   (bfs end (list (list start)) net))
```

5.6) Ilustra el funcionamiento del código especificando la secuencia de llamadas a las que da lugar la evaluación:

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

La macro *trace* es especialmente útil para este tipo de tareas. Con *untrace* se desactivan las trazas.

5.7) Utiliza el código anterior para encontrar el camino más corto entre los nodos F y C en el siguiente grafo no dirigido. ¿Cuál es la llamada concreta que tienes que hacer? ¿Qué resultado obtienes con dicha llamada?



5.8) El código anterior falla (entra en una recursión infinita) cuando hay ciclos en el grafo y el problema de búsqueda no tiene solución. Ilustra con un ejemplo este caso problemático y modifica el código para corregir este problema:

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