PRÁCTICA 1: LISP

FECHAS DE ENTREGA:

- Entrega parcial (ejercicios 1, 2 y 3):
  - Todos los grupos: martes, 12 de febrero de 2019 (23:55)
- Entrega final (todos los ejercicios, incluyendo los ejercicios 1,2 y 3 corregidos):
  - Grupos de jueves: miércoles, 27 de febrero de 2019 (23:55)
  - Grupos de viernes: jueves, 28 de febrero de 2019 (23:55)

GENERAL (ver en Moodle):

- Normativa de prácticas.
- Criterios de evaluación.
- Lista de errores comunes.

EVALUACIÓN P1:

- Correcto funcionamiento: 40 %
- Estilo de programación: 30 %
- Memoria: 30 %

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 no debe dar errores en SBCL (recuerda reiniciar el intérprete antes de hacer esta comprobación).

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`

1 Distancia Coseno. (1.5 puntos)

Una de las principales aplicaciones de la inteligencia artificial al procesamiento de lenguaje natural es determinar cómo de similares son varios textos y clasificarlos en categorías. Por ejemplo, se pueden asignar a las páginas web etiquetas de acuerdo a la temática que abordan: deportes, noticias, videojuegos, etc...

Un método para cuantificar la similitud entre textos consiste en representar las categorías mediante vectores. Para ello eligimos una serie de términos lingüísticos que puedan ser utilizados para discriminarlas. Por ejemplo, para la categoría de deportes podemos elegir los siguientes términos: {Jugador, fútbol, partido, árbitro, equipo, baloncesto}, para la categoría de noticias: {Suceso, ha pasado, acontecimiento, noticia, declarado, entrevista}... Podremos elegir tantos términos como queramos. Cuando hayamos terminado, concatenamos todos estos términos en un único vector V = {term_1_categoría_1,...,term_n_categoría_1, ... , term_1_categoría_N, ..., term_m_categoría_N}.

Los vectores que reprensentan las páginas webs estarán formados por la frecuencia con la que aparece en ella cada uno de los términos. Por ejemplo, si en una página web dada ha aparecido 3 veces la palabra fútbol, 2 veces partido y 1 vez equipo, el vector V de esa página web contendrá el número de veces que aparecen en el texto dichos términos en la posición correspondiente y 0 en el resto.

Para categorizar una página web calculamos la distancia entre el vector que representa dicha página y vectores tipo para cada una de las cateogorías. Finalmente se seleccionará la categoría que se corresponda con la distancia mínima.

Una vez hecha la representación, queda definir cómo calcular la distancia entre estos vectores. Una medida clásica que se ha utilizado para esta tarea es la distancia coseno, sobre la que versa este ejercicio. Dados dos vectores $\mathbf{x} = \{x_1, ..., x_n\}$ e $\mathbf{y} = \{y_1, ..., y_n\}$ en $\mathbb{R}^n$, la distancia coseno $\cos(\theta)$, viene dada por la siguiente expresión:

$$\text{cosine-distance}(\mathbf{x},\mathbf{y}) = 1 - \cos(\theta) = 1 - \frac{\mathbf{x}\cdot\mathbf{y}}{\|\mathbf{x}\|_2\|\mathbf{y}\|_2} = 1 - \frac{\sum_{i=1}^{n} x_i y_i}{\sqrt{\sum_{i=1}^{n} x_i^2}\sqrt{\sum_{i=1}^{n} y_i^2}}$$

El resultado está acotado entre 0 (máxima semejanza) y 2 (máxima diferencia). Se nos ha pedido implementar esta medida en LISP.

1.1 Implementa una función que calcule la distancia coseno entre dos vectores, x e y, de dos formas:

Recursiva:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; cosine-distance-rec
;;; Calcula la distancia coseno de un vector de forma recursiva
;;; Se asume que los dos vectores de entrada tienen la misma longitud.
;;;
;;; INPUT: x: vector, representado como una lista
;;;        y: vector, representado como una lista
;;; OUTPUT: distancia coseno entre x e y
;;;
(defun cosine-distance-rec (x y) ...)
```

Usando mapcar:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; cosine-distance-mapcar
;;; Calcula la distancia coseno de un vector usando mapcar
;;; Se asume que los dos vectores de entrada tienen la misma longitud.
```

```
;;;
;;; INPUT: x: vector, representado como una lista
;;;        y: vector, representado como una lista
;;; OUTPUT: distancia coseno entre x e y
;;;
(defun cosine-distance-mapcar (x y) ...)
```

- Es necesario realizar una descomposición funcional
- No debería haber en el programa código repetido. Si lo hubiere, la descomposición funcional debe ser mejorada.

Responde en la memoria con el resultado de evaluar los siguientes casos:

1. (cosine-distance '(1 2) '(1 2 3))
2. (cosine-distance nil '(1 2 3))
3. (cosine-distance '() '())
4. (cosine-distance '(0 0) '(0 0))

1.2 Codifica una función que reciba un vector que representa a una categoría, un conjunto de vectores y un nivel de confianza que esté entre 0 y 1. La función deberá retornar el conjunto de vectores ordenado según más se parezcan a la categoría dada si su semejanza es superior al nivel de confianza.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; order-vectors-cosine-distance
;;; Devuelve ordenados aquellos vectores similares a una categoria
;;; por encima de un nivel de confianza (0 por defecto)
;;; INPUT:      vector: vector que representa a una categoria,
;;;                 representado como una lista
;;;             lst-of-vectors: vector de vectores
;;;             confidence-level: Nivel de confianza (parametro opcional)
;;; OUTPUT: Vectores cuya semejanza con respecto a la
;;;         categoria es superior al nivel de confianza,
;;;         ordenados
;;;
(defun order-vectors-cosine-distance (vector lst-of-vectors
                                      &optional (confidence-level 0)) ...)
```

```
>> (order-vectors-cosine-distance '(1 2 3) '((32 454 123) (133 12 1) (4 2 2)) 0.5)
;; ---> ((4 2 2) (32 454 123))
>> (order-vectors-cosine-distance '(1 2 3) '((32 454 123) (133 12 1) (4 2 2)) 0.3)
;; ---> ((4 2 2) (32 454 123) (133 12 1))
>> (order-vectors-cosine-distance '(1 2 3) '((32 454 123) (133 12 1) (4 2 2)) 0.99)
;; ---> NIL
```

Responde en la memoria con el resultado de evaluar los siguientes casos:

1. (order-vectors-cosine-distance '(1 2 3) '())
2. (order-vectors-cosine-distance '() '((4 3 2) (1 2 3)))

1.3 Clasificador por distancia coseno. Codifica una función que reciba un conjunto de categorías, cuyo primer elemento es su identificador, un conjunto de vectores que representan textos, cuyo primer elemento es su identificador, y devuelva una lista que contenga pares definidos por un identificador de la categoría que minimiza la distancia coseno con respecto a ese texto y el resultado de la distancia coseno. El tercer argumento es la función usada para evaluar la distancia coseno (mapcar o recursiva).

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; get-vectors-category (categories vectors distance-measure)
;;; Clasifica a los textos en categorias.
;;;
;;; INPUT:  categories: vector de vectores, representado como
;;;                     una lista de listas. Cada uno de ellos
;;;                     caracteriza una categoria
;;;         texts:      vector de vectores, representado como
;;;                     una lista de listas.
;;;                     Cada uno caracteriza un texto.
;;;         distance-measure: funcion de distancia
;;; OUTPUT: Pares formados por el vector que identifica la categoria
;;;         de menor distancia, junto con el valor de dicha distancia
;;;
(defun get-vectors-category (categories texts distance-measure) ...)
```

Los identificadores se incrementarán de manera consecutiva para las categorías y los textos según el orden por el que sean suministrados a la función. Un ejemplo de categorías y textos sería:

```
>>(setf categories '((1 43 23 12) (2 33 54 24)))
>>(setf texts '((1 3 22 134) (2 43 26 58)))}
```

1.4 Haz pruebas llamando a get-vectors-category con las distintas variantes de la distancia coseno y para varias dimensiones de los vectores de entrada. Verifica y compara tiempos de ejecución. Documenta en la memoria las conclusiones a las que llegues.

Para medir el tiempo puedes usar la macro `time`, cuyos resultados puedes comentar.

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

```
>> (setf categories '((1 43 23 12) (2 33 54 24)))
>> (setf texts '((1 3 22 134) (2 43 26 58)))
>> (get-vectors-category categories texts #'cosine-distance-rec)
;; ---> ((2 0.510181) (1 0.184449))
>> (get-vectors-category categories texts #'cosine-distance-mapcar)
;; ---> ((2 0.510181) (1 0.184449))}
```

Responde en la memoria con el resultado de evaluar los siguientes casos:

1. (get-vectors-categories '(()) '(()) #'cosine-distance)
2. (get-vectors-categories '((1 4 2) (2 1 2)) '((1 1 2 3)) #'cosine-distance)
3. (get-vectors-categories '(()) '((1 1 2 3) (2 4 5 6)) #'cosine-distance)

2 Raíces de una función. (1.5 puntos)

Dada una función $f(x) : \mathbb{R} \to \mathbb{R}$ y un intervalo $[a, b] \subseteq \mathbb{R}$, queremos encontrar la raíz o raíces de f, es decir, el valor o valores $x \in [a, b]$ tales que $f(x) = 0$.

Un método muy potente para encontrar las raíces de una función numéricamente es el método de Newton-Raphson. Se trata de un método iterativo en el que, dada la función $f(x)$ de la que se quiere encontrar una raíz (r) se parte de una estimación de r o semilla (x_0) y se generan estimaciones más cercanas a r a partir de ella. Si consideramos que $r = x_0 + h$, h medirá la diferencia entre nuestra estimación y el valor de r. Si suponemos que h es una cantidad pequeña, podemos realizar la siguiente aproximación:

$$f(r) = f(x_0 + h) \approx f(x_0) + hf'(x_0) \implies h \approx -\frac{f(x)}{f'(x)}.$$

Por tanto,

$$r = x_0 + h \approx r = x_0 - \frac{f(x)}{f'(x)}.$$

Teniendo esto en cuenta, las nuevas estimaciones x_{n+1} de r se generan a partir del valor anterior x_n:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}, n = 0, 1, 2...$$

Una vez que sabemos cómo generar las nuevas estimaciones de la raíz, podemos llamar recursivamente a nuestra función hasta que encuentre una estimación x_n cuya diferencia con el valor verdadero de r sea menor que una determinada tolerancia. Si el método es capaz de encontrar una estimación que cumpla dicha condición, devolverá el valor x_n como raíz de $f(x)$. De no encontrarlo en un determinado número de iteraciones, el método no convergerá y no devolverá ningún resultado.

2.1 Implementar una función *newton* que aplique el método de Newton-Raphson para encontrar *una* raíz de una función

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; newton
;;; Estima el cero de una funcion mediante Newton-Raphson
;;;
;;; INPUT: f: funcion cuyo cero se desea encontrar
;;;         df: derivada de f
;;;         max-iter: maximo numero de iteraciones
;;;         x0: estimacion inicial del cero (semilla)
;;;         tol: tolerancia para convergencia (parametro opcional)
;;; OUTPUT: estimacion del cero de f, o NIL si no converge
;;;
(defun newton (f df max-iter x0 &optional (tol 0.001)) ...)
```

Ejemplos:

```
>> (newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3)))
#'(lambda (x) (- (* x (- (* x 3) 4)) 11)) 20 3.0)  ;;---> 4.0
>> (newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3)))
#'(lambda (x) (- (* x (- (* x 3) 4)) 11)) 20 0.6)  ;;---> 1.0
>> (newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3)))
#'(lambda (x) (- (* x (- (* x 3) 4)) 11)) 30 -2.5)  ;;---> -3.0
>> (newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3)))
#'(lambda (x) (- (* x (- (* x 3) 4)) 11)) 10 100.0)  ;;---> NIL
```

2.2 Implementar una función *one-root-newton* que recibe una función, su derivada, una lista de semillas y una tolerancia (opcional). Si el método de Newton encuentra una raíz a partir de alguna de las semillas, la función devuelve el valor de la primera raíz encontrada. Si no converge para ninguna semilla, devuelve *nil*.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; one-root-newton
;;; Prueba con distintas semillas iniciales hasta que Newton
;;; converge
;;;
;;; INPUT: f: funcion de la que se desea encontrar un cero
;;;        df: derivada de f
;;;        max-iter: maximo numero de iteraciones
;;;        semillas: semillas con las que invocar a Newton
;;;        tol: tolerancia para convergencia (parametro opcional)
;;;
;;; OUTPUT: el primer cero de f que se encuentre, o NIL si se diverge
;;; para todas las semillas
;;;
(defun one-root-newton (f df max-iter semillas &optional (tol 0.001)) ...)
```

Ejemplos:

```
>> (one-root-newton  #'(lambda(x) (* (- x 4) (- x 1) (+ x 3)))
#'(lambda (x) (- (* x (- (* x 3) 4)) 11)) 20 '(0.6 3.0 -2.5)) ;;---> 1.0
>> (one-root-newton  #'(lambda(x) (* (- x 4) (- x 1) (+ x 3)))
#'(lambda (x) (- (* x (- (* x 3) 4)) 11)) 20 '(3.0 -2.5)) ;;---> 4.0
>> (one-root-newton  #'(lambda(x) (* (- x 4) (- x 1) (+ x 3)))
#'(lambda (x) (- (* x (- (* x 3) 4)) 11)) 1 '(3.0 -2.5)) ;;---> NIL
```

2.3 Implementar una función *all-roots-newton* que recibe una función, su derivada, una lista de semillas y la tolerancia (opcional). La función devuelve una lista con las raíces encontradas o nil, a partir de las semillas.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; all-roots-newton
;;; Prueba con distintas semillas iniciales y devuelve las raices
;;; encontradas por Newton para dichas semillas
;;;
;;; INPUT: f: funcion de la que se desea encontrar un cero
;;;        df: derivada de f
;;;        max-iter: maximo numero de iteraciones
;;;        semillas: semillas con las que invocar a Newton
;;;        tol: tolerancia para convergencia (parametro opcional)
;;;
;;; OUTPUT: las raices que se encuentren para cada semilla o nil
;;;          si para esa semilla el metodo no converge
;;;
(defun all-roots-newton (f df tol-abs max-iter semillas
                         &optional (tol 0.001)) ...)
```

Ejemplos:

```
>> (all-roots-newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3)))
#'(lambda (x) (- (* x (- (* x 3) 4)) 11)) 20 '(0.6 3.0 -2.5)) ;;---> (1.0 4.0 -3.0)
>> (all-roots-newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3)))
#'(lambda (x) (- (* x (- (* x 3) 4)) 11)) 20 '(0.6 3.0 10000.0)) ;;---> (1.0 4.0 nil)
```

```
>> (all-roots-newton  #'(lambda(x) (* (- x 4) (- x 1) (+ x 3)))
#'(lambda (x) (- (* x (- (* x 3) 4)) 11)) 1 '(0.6 3.0 -2.5))  ;;---> (nil nil nil)
```

2.3.1 Implementa una función haciendo uso de mapcan que pase la salida de all-roots-newton a una lista de semillas (sin nil).

```
>> (list-not-nil-roots-newton  #'(lambda(x) (* (- x 4) (- x 1) (+ x 3)))
#'(lambda (x) (- (* x (- (* x 3) 4)) 11)) 20 '(0.6 3.0 10000.0)) ;;---> (1.0 4.0)
```

3 Combinación de listas (1 punto)

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

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

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

Responde en la memoria con el resultado de evaluar los siguientes casos:

1. (combine-elt-lst 'a nil)
2. (combine-elt-lst nil nil)
3. (combine-elt-lst nil '(a b))

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

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

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

Responde en la memoria con el resultado de evaluar los siguientes casos:

1. (combine-lst-lst nil nil)
2. (combine-lst-lst '(a b c) nil)
3. (combine-lst-lst nil '(a b c))

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

Antes de empezar a programar, lista los casos base de la función e inclúyelos en la memoria de la práctica.

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

```
>> (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))
```

Responde en la memoria con el resultado de evaluar los siguientes casos:

1. (combine-list-of-lsts '(() (+ -) (1 2 3 4)))
2. (combine-list-of-lsts '((a b c) () (1 2 3 4)))
3. (combine-list-of-lsts '((a b c) (1 2 3 4) ()))
4. (combine-list-of-lsts '((1 2 3 4)))
5. (combine-list-of-lsts '(nil))
6. (combine-list-of-lsts nil)

4 Árboles de verdad en lógica proposicional (5 puntos)

Una base de conocimiento D consiste en una colección de Fórmulas Bien Formadas (FBFs) que reflejan conocimiento sobre un dominio determinado:

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

Implícitamente se supone que hay una conjunción lógica (representada por el símbolo $\wedge$) entre las distintas FBFs de una base de conocimiento:

$$w_1 \wedge w_2 \wedge ... \wedge w_n$$

El objetivo de este ejercicio es utilizar árboles de verdad para determinar si una base de conocimiento es o no es satisfacible. Los árboles de verdad son un método sistemático que permite encontrar contradicciones a partir de las FBFs que forman una base de conocimiento. Para ello, se aplican las reglas de inferencia que permiten construir el árbol y explorar cada rama. Si todas las ramas del árbol incluyen contradicciones, Δ será no satisfacible (UNSAT) y si alguna de las ramas no incluye contradicciones, habremos encontrado un modelo de Δ y, por tanto, será satisfacible (SAT).

Para consultar las reglas de derivación que se deben aplicar para construir los árboles de verdad y algunos ejemplos de su uso, se puede consultar la información disponible en `http://home.sandiego.edu/~baber/logic/10.TruthTrees.pdf`. [1]

Representaremos una base de conocimiento en LISP como una lista de FBFs en formato prefijo, por ejemplo, la base de conocimiento $\Delta = \{p \iff \neg q, r \implies (q \wedge p), a \iff p\}$ quedaría expresada en LISP como $'(\wedge\ (\iff\ P\ (\neg Q))(\implies\ R\ (\wedge\ Q\ P))(\iff\ A\ P)))$.

A continuación mostramos algunos ejemplos de posibles entradas del programa con sus correspondientes salidas. La salida t indica que Δ es SAT, mientras la salida *nil* indica UNSAT:

```
>> (truth-tree '())  ->  nil
>> (truth-tree '(^ (v A B)))  ->  t
>> (truth-tree '(^ (! B) B))  ->  nil
>> (truth-tree '(<=> (=> (^ P Q) R) (=> P (v (! Q) R)))) -> t
```

También se muestra la salida de hacer *trace* de una llamada a la función truth-tree:

```
>> (truth-tree '(=> A (^ B (! A))))
 0: (EXPAND-TRUTH-TREE-AUX NIL (=> A (^ B (! A))))
    1: (EXPAND-TRUTH-TREE-AUX NIL (^ (V (! A) (^ B (! A)))))
      2: (EXPAND-TRUTH-TREE-AUX NIL (^ (! A)))
        3: (EXPAND-TRUTH-TREE-AUX ((! A)) (^))
        3: EXPAND-TRUTH-TREE-AUX returned ((! A))
      2: EXPAND-TRUTH-TREE-AUX returned ((! A))
      2: (EXPAND-TRUTH-TREE-AUX NIL (^ (^ B (! A))))
        3: (EXPAND-TRUTH-TREE-AUX NIL (^ B (! A)))
          4: (EXPAND-TRUTH-TREE-AUX (B) (^ (! A)))
            5: (EXPAND-TRUTH-TREE-AUX ((! A) B) (^))
            5: EXPAND-TRUTH-TREE-AUX returned ((! A) B)
          4: EXPAND-TRUTH-TREE-AUX returned ((! A) B)
        3: EXPAND-TRUTH-TREE-AUX returned ((! A) B)
      2: EXPAND-TRUTH-TREE-AUX returned ((! A) B)
    1: EXPAND-TRUTH-TREE-AUX returned ((! A))
  0: EXPAND-TRUTH-TREE-AUX returned ((! A))
t
```

Utilizando las siguientes definiciones y predicados:

[1] En la nomenclatura americana los $\implies$ y $\iff$, se representan con $\supset$ y $\equiv$, respectivamente.

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

(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 bicond-connector-p (x)
  (eql x +bicond+))

(defun cond-connector-p (x)
    (eql x +cond+))

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

(defun positive-literal-p (x)
  (and (atom x)
       (not (truth-value-p x))
       (not (connector-p x))))

(defun negative-literal-p (x)
  (and (listp x)
       (eql +not+ (first x))
       (null (rest (rest x)))
       (positive-literal-p (second x))))

(defun literal-p (x)
  (or (positive-literal-p x)
      (negative-literal-p x)))
```

4.1 Implementar funciones que apliquen las reglas de derivación necesarias para construir el árbol de verdad.

4.2 Implementar las funciones que construyan el árbol de verdad a partir de una base de conocimiento.

```lisp
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; TRUTH-TREE
;; Recibe una expresion y construye su arbol de verdad para comprobar
;; si es SAT o UNSAT
;; PARAMS : exp - expresion a analizar.
;; RETURN : t - la expresion es SAT
;;          nil - la expresion es UNSAT
```

```
8 (defun truth-tree (exp) ...)
```

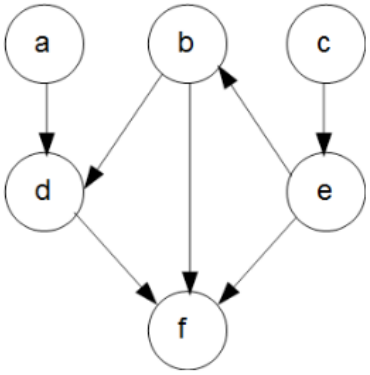
Pista 1: Partiendo de que para que la expresión $(\wedge\ A\ (\vee\ B\ C))$ se cumpla, es equivalente que se cumpla $(\wedge\ A\ B)$ o $(\wedge\ A\ C)$, podemos tener en cuenta que, para nuestro problema, si $(\wedge\ A\ B)$ es válido no es necesario comprobar si $(\wedge\ A\ C)$ lo es.
Pregunta 1: si en lugar de $(\wedge\ A\ (\vee\ B\ C))$ tuviésemos $(\wedge\ A\ (\neg\ A)\ (\vee\ B\ C))$, ¿qué sucedería?
Pregunta 2:¿Y en el caso de $(\wedge\ A\ (\vee\ B\ C)\ (\neg\ A))$?
Pregunta 3: estudia la salida del trace mostrada más arriba. ¿Qué devuelve la función expand-truth-tree?

5 Búsqueda en anchura (1 punto)

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).

Dado 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 algoritmo BFS.

5.3 Estudia con detalle la siguiente implementación del algoritmo BFS, tomada del libro "*ANSI Common Lisp*" 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) '()
    (let* ((path (first queue))
           (node (first path)))
      (if (eql node end)
          (reverse path)
        (bfs end
             (append (rest queue)
                     (new-paths path node net))
             net)))))
(defun new-paths (path node net)
  (mapcar #'(lambda(n)
              (cons n path))
          (rest (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:

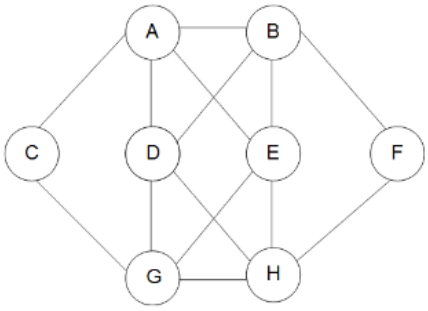
```
(defun shortest-path (start end net)
(bfs end (list (list start)) net))
```

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 B y G 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:**

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