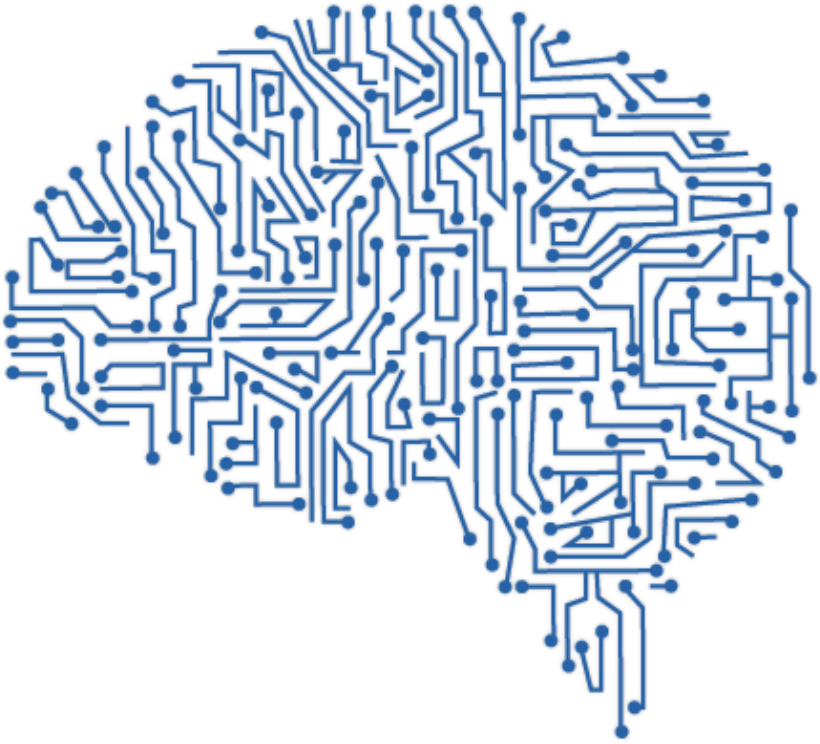
Práctica 1

Inteligencia Artificial



Alfonso Bonilla Trueba – David García Fernández
PAREJA 4 | GRUPO 2313 | EPS-UAM

Ejercicio 1 – Vector más cercano a un vector dado:

1.1 Implementa una función que calcule la norma Lp de un vector x de dos formas

Entrada: x (Vector representado como lista)
p (Orden de la norma no negativo, mayor o igual a 1)
Salida: lp ($\vec{x}$)

PSEUDOCODIGO

Si p es menor que 1
evalua null
en caso contrario
sumatorio-noma(x, p) elevado a 1/9

---Sumatorio-norma
Si x es null
evalua a 0
en caso contrario
evalua (|primero de x| ^ p) + sumatorio-norma(siguente de x, p)

CODIGO RECURSION

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; lp-rec (x p)
;;; Calcula la norma Lp de un vector de forma recursiva
;;;
;;; INPUT: x: vector, representado como una lista
;;; p: orden de la norma que se quiere calcular
;;;
;;; OUTPUT: norma Lp de x
;;;
(defun lp-rec (x p)
  (unless (< p 1) ;;; Comprobacion p >= 1
    (expt
      (sumatorio-norma x p)
      (float (/ 1 p)))))


;;; Funcion auxilar calcula el sumatorio
(defun sumatorio-norma (x p)
  (if (null x)
    0 ;;; Comprobar si la lista esta vacia
    (+
      (expt (abs (first x) ) p)
      (sumatorio-norma (rest x) p) )))

;;; Ejemplo
;;; (setf milista '(6 2))
;;; (lp-rec milista 2)
;;; (lp-rec milista 0) ;;; Caso especial
```

COMENTARIO

Hemos dividido la implantación creando una función que simplemente calcula el sumatorio haciendo uso de recursión.

$\sum_{d=1}^{D}|x_d|^p = |x_1|^p + \sum_{d=2}^{D}|x_d|^p$ [Recursión]

$\sum_{d=1}^{D}|null|^p = 0$ [Base]

Después de que nuestra función ha calculado el sumatorio ya en la función que se nos pedía en el enunciado hemos realizado el ultimo calculo, que es elevar el sumatorio a 1/p.

En esta función tenemos que comprobar que p sea mayor o igual a 1, es decir $1 \leq p \leq \infty$

CODIGO CON MAPCAR

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; lp-mapcar (x p)
;;; Calcula la norma Lp de un vector usando mapcar
;;;
;;; INPUT: x: vector, representado como una lista
;;; p: orden de la norma que se quiere calcular
;;;
;;; OUTPUT: norma Lp de x
;;;
(defun lp-mapcar (x p)
  (unless (< p 1) ;;; Comprobacion p >= 1
    (expt
      (reduce '+ (mapcar #'(lambda (z) (expt (abs z) p)) x))
      (float (/ 1 p)))))

;;; Ejemplo
;;; (setf milista '(6 2))
;;; (lp-mapcar milista 2)
;;; (lp-mapcar milista 0) ;;; Caso especial
```

COMENTARIO

Usando mapcar no necesitamos hacer recursión, pues esta actúa sobre todos los elementos de la lista según la función lambda. Esta función lambada realiza a cada elemento $|x_d|^p$, mapcar devuelve una lista con esta operación realizada en cada elemento, sumamos todos los elementos de la lista con reduce y la función +. El resultado de esta operación lo elevamos a 1/p teniendo en cuenta que $1 \leq p \leq \infty$ que comprobamos al principio con (unless (< p 1))

1.2 A partir de las dos implementaciones de arriba de lp, define funciones l2-rec, l2-mapcar, l1-rec y l1-mapcar

```
(defun l2-rec (x)          ;;; Calcula la norma L2 de un vector de forma recursiva
 (lp-rec x 2))

(defun l2-mapcar (x)     ;;; Calcula la norma L2 de un vector usando mapcar
 (lp-mapcar x 2))

(defun l1-rec (x)        ;;; Calcula la norma L1 de un vector de forma recursiva
 (lp-rec x 1))

(defun l1-mapcar (x)     ;;; Calcula la norma L1 de un vector usando mapcar
 (lp-mapcar x 1))
```

```
;;; Ejemplos
;;; (setf milista '(6 2))
;;; (l2-rec milista)
;;; (l2-mapcar milista)
;;; (l1-rec milista)
;;; (l1-mapcar milista)
```

COMENTARIO

La implementación de estas 4 funciones se basa simplemente en llamadas a las dos anteriormente descritas. Lo único que según la norma que calcule se pasa el valor de p, y estas nuevas funciones solo reciben la lista de la que se quiere calcular la norma.

Norma L_∞:

Para la norma infinito propondría como ya se dijo en clase probar con la norma 50, ya que si probamos valores mas grandes obtenemos el siguiente error:

```
Error: This integer is too large to be converted to a double-float
```

```
;;; Probamos con 50
(setf milista '(6 2))
(lp-rec milista 50)

;;; 5.9999995
```

Y haciendo pruebas con otros valores entre 50 y 100 nunca superamos 6.0
Por lo tanto, ese parece ser el techo de la norma.

1.3 Codifica una función nearest que devuelva, a partir de una lista de vectores, cuál es el más cercano a uno dado

Entrada:

- `lst-vectors: lista de vectores para los que calcular la distancia.`
- `vector: vector referencia, representado como una lista.`
- `fn-dist: referencia a función para medir distancias.`

Salida: el vector cuya distancia e la menor al dado

PSEUDOCODIGO

Se recorre la lista recursivamente.
Si el siguiente elemento de la lista de vectores no es nil:

1. Comprobamos si la norma fn-dist de la resta entre los vectores es menor para el vector actual, o para el menor de los siguientes de la lista. Si es menor para el elemento actual:
   1.1. Se devuelve el elemento actual.
   1.2. Se devuelve el resultado de la llamada recursiva para el resto de la lista.
2. Si el siguiente elemento de la lista es nil, se devuelve el vector actual.

CODIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; nearest (lst-vectors vector fn-dist)
;;; Calcula de una lista de vectores el vector más cercano a uno dado,
;;; usando la función de distancia especificada
```

```
;;;
;;; INPUT: lst-vectors: lista de vectores para los que calcular la distancia
;;; vector: vector referencia, representado como una lista
;;; fn-dist: referencia a función para medir distancias
;;;
;;; OUTPUT: vector de entre los de lst-vectors más cercano al de referencia
;;;
(defun nearest (lst-vectors vector fn-dist)
  (if (rest lst-vectors)
    (let ((vector-actual (first lst-vectors))
          (vector-recursivo (nearest (rest lst-vectors) vector fn-dist)))
    (if (<
          (funcall fn-dist (mapcar #'- vector vector-actual))
          (funcall fn-dist (mapcar #'- vector vector-recursivo)))
      vector-actual ;Si secumple la condicion
      vector-recursivo)) ;Caso recursion
    (first lst-vectors)))

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

COMENTARIO

En cuanto al estilo del código, podemos ver que, para evitar repeticiones de código, hemos empleado la función let. Dicha función para pre-procesa el código asignado a la macro elegida, por lo que también nos ahorra el tiempo de una doble ejecución de dicho código. Por otro lado, en cuanto a la funcionalidad, se observa que hemos utilizado una estrategia recursiva que nos permite comparar los vectores uno a uno desde el final de la lista hasta el principio, conservando en todo momento el más cercano.

1.4 Verifica y compara los resultados y tiempos de ejecución

```
(setf vectors '((0.1 0.1 0.1 0.1 0.1 0.1) (0.2 -0.1 -0.1 0.1 0.1 0.1)))

(time (nearest vectors '(1.0 -2.0 3.0 0.1 0.1 0.1) #'l2-rec))
; cpu time (non-gc) 0 msec user, 0 msec system
; cpu time (gc)     0 msec user, 0 msec system
; cpu time (total)  0 msec user, 0 msec system
; real time  0 msec
; space allocation:
;  696 cons cells, 24,184 other bytes, 0 static bytes
(0.1 0.1 0.1 0.1 0.1 0.1)

(time (nearest vectors '(1.0 -2.0 3.0 0.1 0.1 0.1) #'l2-mapcar))
; cpu time (non-gc) 0 msec user, 0 msec system
; cpu time (gc)     0 msec user, 0 msec system
; cpu time (total)  0 msec user, 0 msec system
; real time  1 msec
; space allocation:
;  434 cons cells, 11,144 other bytes, 0 static bytes
0.1 0.1 0.1 0.1 0.1)

(time (nearest vectors '(1.0 -2.0 3.0 0.1 0.1 0.1) #'l1-rec))
; cpu time (non-gc) 0 msec user, 0 msec system
; cpu time (gc)     0 msec user, 0 msec system
; cpu time (total)  0 msec user, 0 msec system
; real time  0 msec
```

```
; space allocation:
;  696 cons cells, 23,800 other bytes, 0 static bytes
-0.1 -0.1 0.1 0.1 0.1)

(time (nearest vectors '(1.0 -2.0 3.0 0.1 0.1 0.1) #'l1-mapcar))
; cpu time (non-gc) 0 msec user, 0 msec system
; cpu time (gc)     0 msec user, 0 msec system
; cpu time (total)  0 msec user, 0 msec system
; real time  0 msec
; space allocation:
;  434 cons cells, 10,760 other bytes, 0 static bytes
(0.2 -0.1 -0.1 0.1 0.1 0.1)
```

Vemos que la versión l2-mapcar, tarda más (muy poco mas, inapreciable para el vector usado)
También vemos que las versiones de mapcar necesitan menos espacio de memoria.

Ejercicio 2 – Ceros de una función:

2.1 Implementa una función secante que realice el proceso descrito arriba

PSEUDOCODIGO

Calculamos la función, comprobamos que no nos pasemos del máximo de iteraciones, si aún no hemos alcanzado el máximo, llamamos recursivamente cambiando las semillas, la que antes era x_n ahora pasa a ser x_{n-1} y el nuevo valor calculado siguiendo la formula lo pasamos como x_n

CODIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; secante (f tol-abs max-iter par-semillas)
;;; Estima el cero de una función mediante el método de la secante
;;;
;;; INPUT: f: función cuyo cero se desea encontrar
;;; tol-abs: tolerancia para convergencia
;;; max-iter: máximo número de iteraciones
;;; par-semillas: estimaciones iniciales del cero (x0 x1)
;;;
;;; OUTPUT: estimación del cero de f, o NIL si no converge
;;;
(defun secante (f tol-abs max-iter par-semillas)
  (let* ((xn (second par-semillas))     ;;;Macro para xn
         (xnm (first par-semillas))     ;;;Macro para xn-1
         (fxn (funcall f xn))           ;;;Macro para f(xn)
         (red-iter (- max-iter 1))

         (xnmas (- xn
                   (*
                    (/ (- xn xnm)
                       (- fxn (funcall f xnm)))
                    fxn)
                   )))
    (if (tolerancia xnmas xn tol-abs)
        xnmas
        (unless (eq red-iter 0)
          (secante f tol-abs red-iter (list xn xnmas))))))
```

```
;;; Auxiliar Tolerancia
(defun tolerancia(a b tol-abs)
  (<
    (abs (- a b))
    tol-abs))

;;; Casos de Prueba
;;; (setf tol 1e-6)
;;; (setf iters 50)
;;; (setf funcion (lambda (x) (+ (* x 3) (sin x) (- (exp x)))))
;;; (setf semillas1 '(0 1))
;;; (setf semillas2 '(3 5))
;;; (secante funcion tol iters semillas1)
;;; 0.3604217
;;; (secante funcion tol iters semillas2)
;;; 1.8900298
```

COMENTARIO

Hemos realizado una descomposición funcional, de manera que la tolerancia la calculamos en una función auxiliar a parte, y así simplificamos el código de la función principal y facilitamos su lectura.

2.2 Codifica una función un-cero-secante que llame a secante con distintos pares de semillas hasta encontrar un cero de la función

PSEUDOCODIGO

Vamos probando a llamar a la función secante con cada par de semillas

- Si encontramos un cero, devolvemos y acabamos
- Si no lo hemos encontrado con ese par, hacemos una llamada recursiva con el siguiente par de semillas

CODIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; un-cero-secante (f tol-abs max-iter pares-semillas)
;;; Prueba con distintos pares de semillas iniciales hasta que
;;; la secante converge
;;;
;;; INPUT: f: función de la que se desea encontrar un cero
;;; tol-abs: tolerancia para convergencia
;;; max-iter: máximo número de iteraciones
;;; pares-semillas: pares de semillas con las que invocar a secante
;;;
;;; OUTPUT: el primer cero de f que se encuentre, o NIL si se diverge
;;; para todos los pares de semillas
;;;
(defun un-cero-secante (f tol-abs max-iter pares-semillas)
  (unless (null pares-semillas)
    (let (( sec (secante f tol-abs max-iter (first pares-semillas)))) ;;;Macro para la secante
      (if (null sec)
        (un-cero-secante f tol-abs max-iter (last pares-semillas))
        sec))))

;;; Casos de Prueba
;;; (un-cero-secante funcion tol iters (list semillas1 semillas2))
;;; 0.3604217
;;; (un-cero-secante funcion tol iters (list semillas2 semillas1))
;;; 1.8900298
```

2.3 Codifica una función todos-ceros-secante que llame a secante con distintos pares de semillas y devuelva todas las raíces encontradas

PSEUDOCODIGO

Recursivamente vamos haciendo llamadas con el siguiente par de semillas, y haciendo un append con la posible semilla del par actual.

CODIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; todos-ceros-secante (f tol-abs max-iter pares-semillas)
;;; Prueba con distintas pares de semillas iniciales y devuelve
;;; las raíces encontradas por la secante para dichos pares
;;;
;;; INPUT: f: función de la que se desea encontrar un cero
;;; tol-abs: tolerancia para convergencia
;;; max-iter: máximo número de iteraciones
;;; pares-semillas: pares de semillas con las que invocar a secante
;;;
;;; OUTPUT: todas las raíces que se encuentren, o NIL si se diverge
;;; para todos los pares de semillas
;;;
(defun todos-ceros-secante (f tol-abs max-iter pares-semillas)
  (unless (null pares-semillas)
    (append
      (list (secante f tol-abs max-iter (first pares-semillas)))
      (todos-ceros-secante f tol-abs max-iter (rest pares-semillas)))))

;;; Casos de Prueba
;;; (todos-ceros-secante funcion tol iters (list semillas1 semillas2))
;;; (0.3604217 1.8900298)
;;; (todos-ceros-secante funcion tol iters (list semillas2 semillas1))
;;; (1.8900298 0.3604217)
```

Ejercicio 3 – Combinación de listas:

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

PSEUDOCODIGO

Por cada elemento de la lista original devolver ese elemento de la lista combinado con el elemento pasado por argumento.

CODIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; combine-elt-lst (elt lst)
;;; Combina un elemento dado con todos los elementos de una lista
;;;
;;; INPUT: elt: elemento a combinar
;;; lst: lista en la que combinar
;;;
;;; OUTPUT: lista resultado de combinar con elt
;;;
(defun combine-elt-lst (elt lst)
  (mapcar #'(lambda(z) (list elt z)) lst))

;;; Casos de Prueba
;;; (combine-elt-lst 'a nil)
```

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

COMENTARIO

Lo hemos implementado con mapcar para evitar tener que realizar nosotros la recursión. La función lambda crea una lista con cada elemento de mapcar y el elemento pasado por argumento.

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

PSEUDOCODIGO

Mediante recursión, mientras que no sea null la lista 1, vamos juntando el resultado de unir el primer elemento de lst1 con toda la lista 2 y lo que nos devuelva la llamada recursiva de lo que queda de la lista1, de esta manera combinamos ambas listas.

CODIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; combine-lst-lst (lst1 lst2)
;;; Producto cartesiano de dos listas
;;;
;;; INPUT: lst1: lista 1 para el producto
;;; lst2: lista 2 para el producto
;;;
;;; OUTPUT: lista resultado del prodcuto cartesiano
;;;
(defun combine-lst-lst (lst1 lst2)
  (unless (null lst1)
    (append
      (combine-elt-lst (first lst1) lst2)
      (combine-lst-lst (rest lst1) lst2))))

;;; Casos de Prueba
;;; (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:

CODIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; combine-list-of-lsts (lstolsts)
;;; Calcula todas las posibles disposiciones de elementos pertenecientes a N
;;;
;;; INPUT: lstolsts lista de listas que se quieren comnbinar
;;;
;;; OUTPUT: todas las combinaciones posibles
;;;
(defun combine-list-of-lsts (lstolsts)
  (if (null (rest lstolsts))
    (mapcar #'list (first lstolsts))
```

```
    (mapcar #'(lambda (x) (cons (first x) (first (rest x))))
      (combine-lst-lst
        (first lstolsts)
        (combine-list-of-lsts (rest lstolsts))))))

;;; Casos de Prueba
;;; (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))
```

Ejercicio 4 – Problema SAT:

4.1 Diseña un predicado que determine si una proposición lógica en formato prefijo está correctamente formada o no. Y comprobar que una base de conocimiento está bien formada.

PSEUDOCODIGO PROPOSICION-P

Comprobar que al principio de cada lista hay un operador, a no ser que se aun átomo. Dependiendo del tipo de operador habrá que comprobar que el número de operadores es el adecuado.

- Si es un operador unitario hay que comprobar que solo tenga un operador, y mediante recursión comprobar también que este en prefijo este operador
- Si es un operador binario hay que comprobar que tenga dos operadores y no más. Y evaluar mediante recursión estos dos operadores
- Si es un operador n-ario comprobar que al menos tiene dos operadores, y mediante recursión evaluar también estos dos operadores.

CODIGO PROPOSICION-P

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; proposicion-p (expr)
;;; Comprueba si una expresion esta en formato prefijo
;;;
;;; INPUT: expr Expresion a comporbar
;;;
;;; OUTPUT: todas las combinaciones posibles
;;;
(defun proposicion-p (expr)
  (unless (null expr) ;;; Comprobamos que la expresion no sea null
    (cond
      ((atom expr) ;;; Caso atomico
          (not (connector-p expr)))
      ((binary-connector-p (first expr)) ;;; Operadores Binarios
          (and
            (not (null (second expr)))
            (not (null (third expr)))
            (null (fourth expr))))
      ((n-ary-connector-p (first expr)) ;;; Operadores N arios
        (unless
          (and (null (rest expr))       ;;; Al menos dos operandos
```

```
          (and (null rest(rest expr))))
            (comprueba-prop-aux (rest expr))))
      ((unary-connector-p (first expr)) ;;; Operadores unitarios
          (and
            (not (null (rest expr)))    ;;; Un operador, y despues algun conector
            (not (connector-p (first (rest expr))))
            (comprueba-prop-aux (rest expr))))
      (t nil)))) ;;; Caso por defecto

;;; Funcion auxiliar
(defun comprueba-prop-aux (expr)
  (if (null expr)
    t
    (if (atom (first expr))
      (unless (connector-p (first expr))
        (comprueba-prop-aux (rest expr)))
      (when (proposicion-p (first expr))
        (comprueba-prop-aux (rest expr))))))


;;; Casos de Prueba
;;; (proposicion-p 'A)  ;;; T
;;; (proposicion-p '(H <=> (¬ H))) ;;; NIL
;;; (proposicion-p '(<=> H (¬ H))) ;;; T
```

COMENTARIOS PROPOSICION-P

Hemos declarado una función auxiliar que se encarga de evaluar los operadores, por si estos a su vez tienen dentro operaciones que también hay que comprobar que estén en formato prefijo.

PSEUDOCODIGO BASE-P

Comprobar que la expresión es una lista de listas, y que cada una de estas listas está en formato prefijo usando la función anterior.

CODIGO BASE-P

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; base-p (expr)
;;; Comprueba si una expresion es una base de conocimiento
;;;
;;; INPUT: expr Expresion a comporbar
;;;
;;; OUTPUT: T si es base de conocimiento, nil en caso contrario
;;;
(defun base-p (expr)
  (unless (not (listp expr))
    (if (and
          (atom expr) ;;; Los atomos son NIL siempre
          (not(connector-p expr))) ; que no sean un conector
      T
      (and (proposicion-p (first expr))
            (base-p (rest expr))))))

;;; Casos de Prueba
;;; (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
```

4.2 Diseña una función que extraiga una lista con todos los símbolos atómicos (sin repeticiones) de una base de conocimiento.

PSEUDOCODIGO

Eliminamos los paréntesis, los valores de verdad (T, nil), de esta manera nos quedamos con solo los símbolos, y de esa lista eliminamos los duplicados, y así obtenemos los símbolos sin repeticiones.

CODIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; extrae-simbolos (kb)
;;; Extrae sin repeticiones los simbolos de una base de conocimiento
;;;
;;; INPUT: kb Base de conocimiento de la que extraer simbolos
;;;
;;; OUTPUT: simbolos sin repeticiones
;;;
(defun extrae-simbolos (kb)
  (remove-duplicates
    (remove-if #'connector-p
      (remove-if #'truth-value-p
        (elimina-parentesis kb)))))

;;; Funcion Auxiliar
(defun elimina-parentesis (expr)
  (unless (null expr)
    (if (atom (first expr))
      (cons
        (first expr)
        (elimina-parentesis (rest expr)))
      (append
        (elimina-parentesis (first expr))
        (elimina-parentesis (rest expr))))))

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

4.3 A partir de una lista con N símbolos (correspondientes a átomos simbólicos), escribe una función que genere una lista con las 2N posibles interpretaciones

CODIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; genera-lista-interpretaciones (lst-simbolos)
;;; Elabora todas las posibles combinaciones valores de verdad
;;;
;;; INPUT: lst-simbolos sobre los que crear las interpretaciones
;;;         (Todas los casos de una tabla de verdad)
;;;
```

```
;;; OUTPUT: lista con todas las posibles interpretaciones
;;;
(defun genera-lista-interpretaciones (lst-simbolos)
  (unless (null lst-simbolos)   ;;; Caso de lista vacia
    (combine-list-of-lsts     ;;; combina las listas reslultantes del mapcar
     ;;; Para cada valor de la listagenera una lista con 2 sublistas
     (mapcar #'(lambda (x) (list (list x 'T ) (list x 'nil))) lst-simbolos))))

;;; Casos de Prueba
;;; (genera-lista-interpretaciones nil)
;;; NIL
;;; (genera-lista-interpretaciones '()) ;;;caso especial
;;; 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)))
```

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

PSEUDOCODIGO

Reemplazamos las operaciones lógicas por las funciones internas de lisp como es and, or, not y eq. Y cambiamos los símbolos por su valor de verdad en función de la interpretación recibida. Hacemos esto de manera recursiva en toda la base de conocimiento. Y al final, una vez "traducido" todo a lógica entendible por el intérprete de lisp, evaluamos esta lista y devolvemos el valor.

CODIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; interpretacion-modelo-p (interp kb)
;;; Comprueba si una interpretacion es modelo de una base de conocimiento
;;;
;;; INPUT: interp interpretacion a comprobar
;;;        kb Base de Conocimiento sobre la que comprobar la interpretacion
;;;
;;; OUTPUT: T si es modelo, nil en caso contrario
;;;
(defun interpretacion-modelo-p (interp kb)
  (when (base-p kb) ;;Comprobar si la proposicion es correcta
      (eval
        (append
          '(and)
          (mapcar #'(lambda (x)
                      (if (listp x)
                        (cambia-valores interp (cambia-implicaciones-prop x))
                        (first (cambia-valores-verdad interp (list x)))))
          kb)))))

;;; Funcion auxiliar
(defun cambia-valores (interp x)
  (mapcar #'(lambda (y)
      (cond
        ((unary-connector-p y) 'not)          ;;; reemplazamos el operador ¬ por not
        ((eq +bicond+ y) 'eq)                 ;;; reemplazamos el operador <=> por eq
        ((eq +and+ y) 'and)                   ;;; reemplazamos el operador ^ por and
        ((eq +or+ y) 'or)                     ;;; reemplazamos el operador v por or
        ((listp y) (cambia-valores interp y)) ;;; Recursion
        (t y)))
```

```
    (cambia-valores-verdad interp x))) ;;; Actuamos sobre la lista ya cambiada segun la
interpretacion

;;; Funcion auxiliar
(defun cambia-valores-verdad (interp kb)
  (if (null interp)
    kb
    (cambia-valores-verdad
      (rest interp)
      (substitute
        (second (first interp))
        (first(first interp))
        kb))))

;;; Funcion auxiliar para tratar las implicaciones
(defun cambia-implicaciones-prop (prop)
  (if (atom prop)
    prop
    (unless (null prop)
      (if (eq +cond+ (first prop) )
        (list 'or (list 'not
                        (cambia-implicaciones-prop(second prop)))
                        (cambia-implicaciones-prop(third prop)))
        prop))))


;;; Casos de Prueba
;;; (interpretacion-modelo-p '((A nil) (P nil) (H t)) '((<=> A (¬ H)) (<=> P (^ A H)) (=> H P)))
;;; NIL
;;; (interpretacion-modelo-p '((A t) (P nil) (H nil)) '((<=> A (¬ H)) (<=> P (^ A H)) (=> H P)))
;;; T
```

COMENTARIOS

Para este problema hemos realizado una descomposición funcional creando 3 funciones auxiliares, que nos ayudan a un código más legible y eficiente.
Una función se encarga de cambiar los símbolos por su valor de verdad según la interpretación.
Otra, cambia los operadores en funciones lógicas de lisp, excepto la implicación, que es un caso especial.
La última es para el caso especial de la implicación, ya que no hay ninguna función interna de lisp que la resuelva. Nosotros la hemos resuelto con la equivalencia (A => P ::= ¬A v P)

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

PSEUDOCODIGO

Obtenemos todas las interpretaciones de la base de conocimiento después de comprobar que esta es correcta. Y con un mapcar evitamos la recursión, y si la interpretación ha sido modelo la retornamos, si no ha sido modelo seguimos. De esta manera obtenemos todas las interpretaciones que han sido modelo de la base de conocimiento.

CODIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; encuentra-modelos (kb)
;;; Devuelve todas las interpretaciones que son modelo de la base de conocimiento
;;;
;;; INPUT: kb Base de Conocimiento de la que se quieren los modelos
;;;
;;; OUTPUT: Lista con las interpretaciones que son modelo
```

```
;;;
(defun encuentra-modelos (kb)
  (when (base-p kb)
    (remove-if
        #'(lambda(x)
            (not (interpretacion-modelo-p x kb)))
        (genera-lista-interpretaciones (extrae-simbolos kb)))))

;;; Casos de Prueba
;;; (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 proposición es consecuencia lógica de una base de conocimiento o no

PSEUDOCODIGO

Para que sea consecuencia tenemos que probar que todos los modelos de la base de conocimiento son modelos de la proposición. Obtenemos todos los modelos, y los vamos comprobando recursivamente, si uno no se cumple terminamos con nil. Si llegamos al final de la lista de los modelos significa que todos son modelo de la proposición, por lo cual podemos concluir que prop es consecuencia lógica de la base de conocimiento y devolvemos T.

CODIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; consecuencia-p (prop kb)
;;; Comprueba si una proposion es consecuencia de una base de conocimiento
;;;
;;; INPUT: kb Base de Conocimiento
;;;        prop Proposicon a comprobar si es consecuencia
;;;
;;; OUTPUT: T si es consecuencia, nil en caso contrario
;;;
(defun consecuencia-p (prop kb)
  (when
    (and
      (base-p kb)               ;;; kb es una base de conocimiento correcta
      (not (null prop)))        ;;; la proposion no es una lista vacia
    (probar-consecucias (encuentra-modelos kb) (list prop)))) ;;; Probamos si algun modelo de kb
satisface prop

;;; Funcion Auxiliar
(defun probar-consecucias (modelos kprop)
  (if (null modelos) ;;; Comprobamos no haber terminado la lista
    T  ;;; Si ninguno ha fallado es que ha pasado todos los modelos
    (if
      (not (interpretacion-modelo-p ;;; Si el modelo se satisface
        (first modelos)             ;;; podemos concluir que es consecuencia
        kprop))
    nil                             ;;; Si falla el modelo, no es consecuencia
    (probar-consecucias(rest modelos) kprop)))) ;;; Recursion para seguir probando el resto de
modelos

;;; Casos de Prueba
;;; (consecuencia-p 'A '(A))
;;; T
;;; (consecuencia-p 'A '(¬ A))
```

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

COMENTARIOS

Hemos hecho una descomposición funcional, la función auxiliar prueba los modelos de kb, y la función principal, llama a esta y también comprueba que la base de conocimiento es correcta.

Ejercicio 5 – Búsqueda en anchura:

5.1. Ilustra el funcionamiento del algoritmo con ejemplos.

Para el grafo del ejemplo, si queremos ir de c a f:

Nodo	Cola
-	**{c}**
c	**{e}**
e	**{b,f}**
b	**{f,d}**
f	**FIN**

5.2. Escribe el pseudocodigo correspondiente al algoritmo de BFS.

PSEUDOCODIGO

Si el primer nodo de la cola es nil:

1. Se devuelve nil
2. Si no es nil, se comprueba si el nodo actual es el nodo objetivo
   2.1. Si lo es, se devuelve el nodo actual y su padre en orden inverso
   2.2. Si no lo es, se hace una llamada recursiva expandiendo el nodo y avanzando al siguiente elemento de la lista.

5.4. Estudia el algoritmo BFS y pon comentarios al código dado.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;Breadth-first-search in graphs
;;;
(defun bfs (end queue net)
  (if (null queue) nil                    ; caso de quedarse sin elementos en cola
    (let ((path (car queue)))
      (let ((node (car path)))
        (if (eql node end) (reverse path) ; Si se llega al nodo objetivo, return path-1
          (bfs end (append
                     (cdr queue)
                     (new-paths path node net))
                 net))))))                                ; Si no se llega al nodo objetivo
                                                          ; se exande el nuevo nodo
                                                          ; y se avanza al siguiente elemento
```

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

5.5. Explica por qué esta función resuelve el problema de encontrar el camino más corto entre dos nodos del gráfo.

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

Al ser búsqueda en anchura, siempre se llega a la solución en el nivel más cercano al nodo inicial. Es decir, que no se puede encontrar antes una solución que pase por niveles más profundos del grafo. De todas formas, existe la complicación mostrada en el apartado 1 que puede hacer que se llegue por caminos con un paso de más.

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)))
;;; Al usar trace obtenemos

CG-USER(8):
 0[1]: (SHORTEST-PATH A F ((A D) (B D F) (C E) (D F) (E B F) (F)))
 0[1]: returned (A D F)
(A D F)
```

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 que tienes que hacer? ¿Qué resultado se obtiene?

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

;;; El resultado obtenido es:

CG-USER(14):
 0[1]: (SHORTEST-PATH F C
                      ((A C D E B) (B A D E F) (C A G) (D A G B H) (E A G B H) (F B H)
                       (G C D E H) (H G D E F)))
 0[1]: returned (F B A C)
(F B A C)
```

5.8. El código anterior falla 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)
 (unless (null queue)  ;Caso base
   (let ((path (first queue))) ;;;Se vinculan los valores de path y de node
     (let ((node (first path)))
       (if (eql node end)    ;Caso base
         (reverse path)
         ;;;Recursion, se exploran los caminos con
         ;;;origen en el nodo con el que estamos trabajando
         (bfs-improved end
```

```
                (append
                  (rest queue)
                  (new-paths-improved path node net))
                net))))))


     (defun new-paths-improved (path node net)
       (unless (null (repetidos path))
         (mapcar #'(lambda(n)
                           (cons n path))
                 (rest (assoc node net)))))

     (defun repetidos (lst)
       (or (null lst)
           (and
             (not (member (first lst)
             (rest lst)))
           (unique-p (rest lst)))))

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