CÓDIGOS SCHEME – MIGUEL SCHPEIR

UNIVERSIDAD NACIONAL DEL LITORAL

2011

```
; FUNCIONES LOGICAS Y MATEMATICAS

; Distinto

; Ej: (!= 3 3) => #f       (!= 3 6) => #t


(define != (lambda (N1 N2) (not (= N1 N2))))


; Ej: (notnull? 4) => #t    (notnull? '()) => #f


(define notnull? (lambda (E) (not (null? E))))


; Incrementa 1

; Ej: (++ 1) => 2


(define ++ (lambda (N) (+ N 1)))


; Decrementa 1

; Ej: (-- 1) => 0


(define -- (lambda (N) (- N 1)))


; Par (se asume que el 0 es numero par)

; Ej: (par? 4) => #t         (par? 7) => #f
```

```
(define par? (lambda (N)
        (if (< N 0) (parAux (* N -1)) (parAux N))))

(define parAux (lambda (N)
        (if (= N 0) #t (impar? (- N 1)))))

; Impar
; Ej: (impar? 5) => #t

(define impar? (lambda (N)
        (if (< N 0) (imparAux (* N -1)) (imparAux N))))

(define imparAux (lambda (N)
        (if (= N 0) #f (par? (- N 1)))))

; Potencia
; Ej: (potencia 2 8) => 256

(define potencia (lambda (B E)
        (cond ((= E 0) 1)
            ((= E 1) B)
            (else (* B (potencia B (- E 1)))))))

; Calcular factorial (2 versiones)
; Ej: (factorial 6) => 720

(define factorial (lambda (N) (if (= N 0) 1 (* N (fact1 (- N 1) )))))
```

; TRABAJOS CON LISTA - GENERALIDADES

; Devuelve el numero de elementos que compone una lista.

; Ej: (long '(1 2 3 4)) => 4 (long '(1 2 (3 4))) => 3

```
(define long (lambda (L)
        (if (null? L) 0 (+ 1 (long (cdr L))))))
```

```
; Devuelve el numero de elementos (atomos) que compone una lista. Multinivel.
; Ej: (longM '(1 2 3 4)) => 4            (longM '(1 2 (3 4))) => 4

(define longM (lambda (L)
                (long (listaAtomos L))))



; Devuelve el maximo elemento de una lista.
; Ej: (maxL '(1 2 3 4 5 6)) => 6

(define maxL (lambda (L)
            (if (null? L) '()
            (if (null? (cdr L)) (car L)
                        (if (>= (car L) (maxl(cdr L))) (car L)
                                                (maxl(cdr L)))))))

; Devuelve el maximo elemento de una lista. Multinivel.
; Ej: (maxM '(1 2 3 (4 5 6) (7 8) 4 (10 2))) => 10

(define maxM (lambda (L)
          (if (null? L) '()
             (if (null? (cdr L))
                (if (list? (car L)) (maxM (car L)) (car L))
                (if (> (if (list? (car L)) (maxM (car L)) (car L)) (maxM (cdr L)))
                   (if (list? (car L)) (maxM (car L)) (car L))
                   (maxM (cdr L)))))))

; Devuelve el menor elemento de una lista.
```

```
; Ej: (minL '(1 2 3 4 5 6)) => 1

(define minL ( lambda (L)
            (if (null? L) '()
            (if (null? (cdr L)) (car L)
            (if (<= (car L) (minl(cdr L))) (car L) (minl (cdr L)))))))

; Devuelve el menor elemento de una lista. Multinivel.
; Ej: (minM '(1 2 3 (4 5 6) (7 8) 4 (10 2 -2))) => -2

(define minM (lambda (L)
          (if (null? L) '()
             (if (null? (cdr L))
                (if (list? (car L)) (minM (car L)) (car L))
                (if (< (if (list? (car L)) (minM (car L)) (car L)) (minM (cdr L)))
                   (if (list? (car L)) (minM (car L)) (car L))
                   (minM (cdr L)))))))

; Determina si una lista es estrictamente creciente.
; Ej: (crece '(1 2 3 4 5 6 7)) => #t

(define crece (lambda (L)
     (if (null? (cdr L)) #t
        (if (< (car L) (cadr L)) (crece (cdr L)) #f))))

; Determina si una lista es estrictamente decreciente.
; Ej: (decrece '(7 6 5 4 3)) => #t
```

```
(define decrece (lambda (L)
    (if (null? (cdr L)) #t
        (if (> (car L) (cadr L)) (decrece (cdr L)) #f))))

; Verifica si una lista es capicua
; Ej: (capicua '(1 3 5 (2 4 6 (8) 6 4 2) 5 3 1)) => #t
;     (capicua '(1 3 5 (2 4 6 (8) 7 4 2) 5 3 1)) => #f

(define capicua (lambda (L)
    (if (null? L) #t
        (if (equal? L (invertirM L)) #t #f))))

; L1 es prefijo de L2?, no pueden estar ambas vacias
; Ej: (prefijo '(1 2) '(1 2 3 4 5)) => #t

(define prefijo (lambda (L1 L2)
(if (null? L2) #f (if (or (null? L1) (equal? L1 L2)) #t (prefijo L1 (qu L2))))))

; Version 2

(define prefijo2 (lambda(L1 L2)
(if (equal? L1 L2) #t (if (< (long L2) (long L1)) #f (prefijo2 L1 (qu L2))))))


; L1 es posfijo de L2?
; Ej: (posfijo '(7 6) '(1 2 6 5 7 6)) => #t

(define posfijo (lambda (L1 L2)
```

```
            (if (null? L2) #f
                (if (equal? L1 L2) #t (posfijo L1 (cdr L2))))))

; Version 2

(define posfijo2 (lambda(L1 L2)
            (if (or (null? L1) (equal? L1 L2)) #t
                (if (< (long L2) (long L1)) #f (posfijo2 L1 (cdr L2))))))

; sublista!! L1 de L2, no pueden estar ambas vacias
; Ej: (subLista '(1 2 3) '(3 5 1 2 3 7 8)) => #t

(define subLista (lambda (L1 L2)
    (if (null? L2) #f
        (if (prefijo L1 L2) #t
            (sublista L1 (cdr L2))))))

; subListaM!! L1 es sublista de L2? Multinivel
; Ej: (subListaM '(3 5 7) '(1 2 3 (7 6 (3 5 7 1) (2 3)) 2)) => #t

(define subListaM (lambda (L1 L2)
  (if (not (list? L2)) #f
  (if (equal? L1 L2) #t
  (if (null? L2) #f
  (or (subListaM L1 (cdr L2)) (subListaM L1 (qu L2)) (subListaM L1 (car L2))) )))))

; Determina si los valores X e Y son consecutivo en L
; Ej: (consecutivo '(1 2 3 4 5 6 7 8) 6 7) => #t
```

```
(define consecutivo (lambda (L X Y)
                    (sublista (list X Y) L)))

; Consecutivos Multinivel.
; Ej: (consecutivoM '(1 2 3 (4 (5 6)) 7 8) 5 6) => #t

(define consecutivoM (lambda (L X Y)
                    (subListaM (list X Y) L)))

;  BUSQUEDA DE ELEMENTOS
```

```
; Obtener el ultimo elemento de una lista.
; Ej: (ultimoElem '(1 2 3 4 5 6)) => 6

(define ultimoElem( lambda (L)
      (if (null? L) '() (if (equal? (cdr L) '()) (car L) (ultimoElem (cdr L))))))

; Devuelve el enesimo elemento de una lista.
; Ej: (enesimoElem 10 '(7 8 9 10)) => 0            (enesimoElem 2 '(7 8 9 10)) => 8

(define enesimoElem(lambda(N L)
           (if (< (long L) N) 0
           (if (equal? N 1) (car L)
              (enesimoElem (- N 1) (cdr L))))))

; Devuelve el numero de posicion de la primera ocurrencia de X.
; Ej: (xPosicion 2 '(2 3 4 5)) => 1

(define xPosicion (lambda (X L)
      (if (or (null? L) (not (pertenece X L))) 0
         (xBusca X L))))

(define xBusca (lambda (X L)
      (if (null? L) 0
         (if (equal? (car L) X) 1
            (+ 1 (xBusca X (cdr L)))))))

; Dado un valor X, busca en una lista L (simple) para determinar si esta o no en ella.
; Ej: (pertenece 3 '(1 2 3 4 5 6)) => #t
```

```
(define pertenece (lambda (X L)
(if (null? L) #f (if (equal? (car L) X) #t (pertenece X (cdr L))))))

; Dado un valor X, busca en una lista L para determinar si esta o no en ella. Multinivel.
; Ej: (perteneceM 3 '(1 (2 3) 4 (5 6))) => #t

(define perteneceM (lambda (X L)
        (if (null? L) #f
            (if (list? (car L)) (perteneceM X (car L))
                (if (equal? X (car L)) #t
                    (perteneceM X (cdr L)))))))
```

```
; ELIMINACION O REEMPLAZOS DE LOS ELEMENTOS EN UNA LISTA

; No elimina el n-esimo elemento, si sus predecesores.
; Ej: (sacaNpri 3 '(1 1 1 3 4 5)) => (3 4 5)

(define sacaNpri (lambda (N L)
        (if (= 0 N) L
           (sacaNpri (- N 1) (cdr L)))))

; Elimina el elemento X de la lista en el 1er nivel.
; Ej: (eliminaX 3 '(3 4 5 (3 7) 3)) => (4 5 (3 7))

(define eliminaX(lambda(X L)
         (if(null? L) '()
         (if (equal? X (car L))
            (eliminaX X (cdr L))
            (cons (car L) (eliminaX X (cdr L)))))))

; Elimina el elemento X de la lista en todos los niveles
; Ej: (eliminaMx '((1 2 3) 2 7 (1 3 (6 2 8) 0 2)) 2) => ((1 3) 7 (1 3 (6 8) 0))

(define eliminaMx (lambda (L X)
      (if (null? L) '()
         (if (equal? (car L) X) (eliminaMx (cdr L) X)
            (if (list? (car L)) (cons (eliminaMx (car L) X) (eliminaMx (cdr L) X))
               (cons (car L) (eliminaMx (cdr L) X)))))))
```

```
; Elimina todos los elementos de la lista 2 que estan en la 1
; Ej: (elim12 '(2 1) '((1 2 3) 2 7 (1 3 (6 2 8) 0 2))) => ((3) 7 (3 (6 8) 0))

(define elim12 (lambda (L1 L2)
        (if (null? L1) L2
           (elim12 (cdr L1) (eliminaMx L2 (car L1)) ))))

; Devuelve la lista sin elementos repetidos. Elimina las primeras ocurrencias.
; (eliminaR '(3 5 3 1 9 2 3 8 9 3)) => (5 1 2 8 9 3)

(define eliminaR (lambda (L)
                (if (null? L) '()
                   (if (not (pertenece (car L) (cdr L)))
                      (cons (car L) (eliminaR (cdr L)))
                      (eliminaR (cdr L))))))

; Devuelve la lista sin elementos repetidos. Deja las primeras ocurrencias.
; (eliminaR2 '(3 5 3 1 9 2 3 8 9 3)) => (3 5 1 9 2 8)

(define eliminaR2 (lambda (L)
                (if (null? L) '() (invertir (eliminaR (invertir L))))))


; Elimina el primer elemento X que aparece en la lista.
; Ej: (eliminarPri 2 '(4 2 1 2 1)) => (4 1 2 1)

(define eliminarPri (lambda (X L)
```

```
    (if (null? L) '()
       (if (equal? (car L) X) (cdr L)
          (cons (car L) (eliminarPri X (cdr L)))))))

; Elimina el elemento que se encuentra en la enesima posicion.
; Ej: (xElimina '(4 8 91 2 7 5) 3) => (4 8 2 7 5)

(define xElimina (lambda (L N)
                (if (or (null? L) (> N (long L) )) #f
                (if (= N 1)  (cdr L)
                   (cons (car L) (xElimina (cdr L) (- N 1)))))))

; Reemplazar X por Y en L
; Ej: (xReemplazar '(p o t e n c i a) 't 'n)=> (p o n e n c i a)

(define xReemplazar (lambda (L X Y)
(if (null? L) '()
          (if (equal? (car L) X) (cons Y (xReemplazar (cdr L) X Y))
          (cons (car L) (xReemplazar (cdr L) X Y))))))

; Reemplazar X por Y en L (multinivel)
; Ej: (xReemplazarM 2 3 '( 2 1 ( 1 2 (2 2 6 2 2) 3 1 3) 2 5)) => (3 1 (1 3 (3 3 6 3 3) 3 1 3) 3 5)

(define xReemplazarM (lambda (X Y L)
                (if (null? L) '()
                (if (list? (car L)) (cons (xReemplazarM X Y (car L)) (xReemplazarM X Y (cdr L)))
                   (if (equal? X (car L))
                      (cons Y (xReemplazarM X Y (cdr L)))
```

```
                    (cons (car L) (xReemplazarM X Y (cdr L))))))))

;EJERCICIO 29: Sustituir en L P1 por P2 (P1 y P2 son listas, patrones).
; Ej: (sust '(1 2 3 4) '(1 2) '(4)) => (4 3 4)

(define sust (lambda(L P1 P2)
      (if (null? L) '()
      (if (prefijo P1 L)
         (concatenar P2 (sust (aPartirN (long P1) L) P1 P2))
         (cons (car L) (sust (cdr L) P1 P2))))))

; Quita el ultimoElem elemento de una L, si es vacia retorna '()
; Ej: (qu '(1 2 3 4 5)) => (1 2 3 4)

(define qu (lambda  (L)
(if (null? L) '() (if (equal? '() (cdr L)) '() (cons (car L) (qu (cdr L)))))))

; Construye una lista con el primer y ultimoElem elemento de una lista.
; Ej: (leex '(1 2 3 4 5)) => (1 5)

(define leex (lambda (L) (list  (car L) (ultimoElem L) )))

; Quita los extremos de una lista.
; Ej: (quitaex '(1 2 3 4 5)) => (2 3 4)

(define quitaex (lambda (L)
            (if (> (long L) 1) (qu (cdr L)) '())))
```

```
; Reemplaza el elemento en la posicion N por X.
; Ej: (insertarXenN '(1 32 7 4) 2 5) => (1 5 7 4)

(define insertarXenN(lambda(L N X)
        (if (or (null? L) (= N 0)) L
        (if (= N 1) (cons X (cdr L))
           (cons (car L) (insertarXenN (cdr L) (- N 1) X))))))
```

```
; CREACION DE LISTAS

; Dada dos listas, L1 y L2, las concatena.
; Ej: (concatenar '(1 2) '(3 4 5)) => (1 2 3 4 5)

(define concatenar (lambda(L1 L2)
            (if(null? L1) L2  (cons (car L1) (concatenar (cdr L1)L2)))))

; Concatenar un elemento a una lista.
; Ej: (concatenarElem '(1 2 3) 4) => (1 2 3 4)
;     (concatenarElem '(1 2 3) '(4)) => (1 2 3 (4))

(define concatenarElem (lambda (L X)
      (if (null? L) (list X) (invertir (cons X (invertir L) )))))

; Version ampliada de concatenar en la que los elementos no deben ser necesariamente listas.
; Ej: (concatenar2 '(2 3) 9) => (2 3 9)
;     (concatenar2 '(1 2 3) '(4)) => (1 2 3 4)

(define concatenar2 (lambda(E1 E2)
      (if (and (list? E1) (list? E2)) (concatenar E1 E2)
      (if (and (not (list? E1)) (not (list? E2))) (concatenar (list E1) (list E2))
      (if (list? E1) (concatenar E1 (list E2))
            (concatenar (list E1) E2))))))
```

```
; Devuelve todas las sublistas posibles de una lista dada.

; Ej: (sublistas '(1 (2 3) 4 5)) => ((1) (1 (2 3)) (1 (2 3) 4) (1 (2 3) 4 5) ((2 3)) ((2 3) 4) ((2 3) 4 5) (4) (4 5) (5))

;     (sublistas '(1 2 3))       => ((1) (1 2) (1 2 3) (2) (2 3) (3))


(define sublistas (lambda (L)

                (if (null? L) '()

                   (concatenar (sublistasAux L 1) (sublistas (cdr L))))))


(define sublistasAux (lambda (L N)

                  (if (> N (long L)) '()

                     (cons  (nPrimeros N L) (sublistasAux L (+ N 1))))))


; Devuelve una lista con los elementos atomicos.

; Ej: (listaAtomos '(3 4 (6 8) (2 (10 11 ) 19 ) -1 -2 (-3 ))) => (3 4 6 8 2 10 11 19 -1 -2 -3)


(define listaAtomos (lambda (L)

                (if (null? L) '()

                (if (nodo? (car L)) (cons (car L) (listaAtomos (cdr L)))

                   (concatenar (listaAtomos (car L)) (listaAtomos (cdr L)))))))


; Devuelve los N primeros elementos de L

; Ej: (nPrimeros 4 '(6 5 1 2 4 7)) => (6 5 1 2)


(define nPrimeros(lambda (N L)

          (if (= N 0) '()

            (cons (car L) (nPrimeros (- N 1) (cdr L))))))
```

```
; Devuelve todos los resultados posibles de prefijos.
; Ej: (nPrimerosT '(1 2 3 4 5)) => ((1) (1 2) (1 2 3) (1 2 3 4) (1 2 3 4 5))

(define nPrimerosT (lambda (L)
      (if (null? L) '()
         (nPrimerosTaux L 1))))

(define nPrimerosTAux (lambda (L N)
      (if (= N (long L)) (list L)
         (cons (nPrimeros N L) (nPrimerosTAux L (+ N 1))))))

; Devuelve los N ultimos elementos de L
; Ej: (nUltimos '(71 82 23 14 25 6) 3) => (14 25 6)

(define nUltimos (lambda (L N)
               (if (or (null? L) (> N (long L) )) #f
               (cond ((= N 1) (cdr L))
                   ((= N 0) L)
                   (else (nUltimos (cdr L) (- N 1)))))))

;Devuelve a partir de n (sin tomar el elemento de la posición n)
; Ej: (aPartirN 3 '(7 6 5 4 3 2)) => (4 3 2)

(define aPartirN (lambda(N L)
      (if (= N 0) L
         (aPartirN (- N 1) (cdr L)))))
```

```
; Menores que N
; Ej: (xMenores '(1 8 3 4 5 2 7) 4) => (1 2 3)

(define xMenores (lambda (L N) (xMenoresAux (ordenar<= L) N)))

(define xMenoresAux (lambda (L N)
          (if (null? L) '()
             (if (< (ultimoElem L) N) L
                (xMenoresAux (qu L) N)))))

; Mayores que N
; Ej: (xMayores '(1 8 3 4 5 2 7) 4) => (5 6 7)

(define xMayores (lambda (L N) (xMayoresAux (ordenar<= L) N)))

(define xMayoresAux (lambda (L N)
                (if (null? L) '()
                   (if (> (car L) N) L
                      (xMayoresAux (cdr L) N)))))

; Devuelve una lista con los anteriores elementos a X
; Ej: (ant 5 '(1 2 3 4 5 6 7 8 9)) => (1 2 3 4)

(define ant (lambda (X L)
   (if (null? L) '()
      (if (equal? (car L) X) '()
         (cons (car L) (ant X (cdr L)))))))
```

```
; Devuelve una lista con los siguientes elementos a X

; Ej: (sig 5 '(1 2 3 4 5 6 7 8 9)) => (6 7 8 9)


(define sig (lambda (X L)

   (if (null? L) '()

      (if (equal? (car L) X) (cdr L)

         (sig X (cdr L))))))


; Devuelve una lista conteniendo dos listas, una con los elementos anteriores a X y otra con los siguientes a X.

; Ej: (izq_der 5 '(1 2 3 4 5 6 7 8 9)) => ((1 2 3 4) (6 7 8 9))


(define izq_der (lambda (X L)

   (if (null? L) '()

      (cons (ant X L) (list (sig X L))))))


; Devuelve la lista L invertida.

; Ej: (invertir '(1 2 3 4 5 6)) => (6 5 4 3 2 1)


; sin primitivas

(define invertir (lambda (L)

      (if (null? L) '() (cons (enesimoElem (long L) L) (invertir (qu L))))))


; con primitivas

(define invertir (lambda (L)

     (if (null? L) '()  (append (invertir (cdr L)) (list (car L))))))


; Invierte una lista en todos sus niveles.
```

```
; Ej: (invertirM '(1 2 (3 4 (5 6) 7) 8 9 (10 11 (12 13)))) => (((13 12) 11 10) 9 8 (7 (6 5) 4 3) 2 1)


; sin primitivas
(define invertirM (lambda (L)
(if (null? L) '()
        (if (list? (enesimoElem (long L) L))
           (cons (invertirM (enesimoElem (long L) L)) (invertirM (qu L)))
           (cons (enesimoElem (long L) L) (invertirM (qu L)))))))


; con primitivas
(define invertirM (lambda (L)
     (if (null? L) '()
        (if (list? (car L)) (append (invertirM (cdr L)) (list (invertirM (car L))))
                        (append (invertirM (cdr L)) (list (carL)))))))


;  (invertirM '(1 2 3 (4 5 6) 9)) => (9 (6 5 4) 3 2 1)
;  (invertirM '(a(b(c(d(e)))))) => (((((e) d) c) b) a)


; Devuelve una lista conteniendo dos sublistas, una con elementos menores que N y otra con los mayores a N
; Ej: (xMayMen '(1 8 3 4 5 2 7) 4) => ((1 2 3) (5 6 7))


(define xMayMen (lambda (L N) (list (xMenores L N) (xMayores L N))))


; Arma una lista con todas las posiciones de X en L
; Ej: (ocurrencias '(1 3 2 4 65 7 3) 3) => (2 7)


(define ocurrencias (lambda (L X) (ocurrenciasAux L X 1)))
```

```
(define ocurrenciasAux (lambda (L X A)
(if (null? L) '()
        (if (= (car L) X)  (cons A  (ocurrenciasAux (cdr L) X (+ A 1)))
                    (ocurrenciasAux (cdr L) X (+ A 1))))))

; Mover atras N elementos de L
; Ej: (moverAtras '(1 2 3 4 5 6) 3) => (4 5 6 1 2 3)

(define moverAtras(lambda(L N)
            (if(or(equal? N 0) (null? L)) L
              (moverAtras(concatenar(cdr L)(list(car L))) (- N 1)))))

; Mover adelante N elementos de L
; Ej: (moverAdelante '(1 2 3 4 5 6) 3) => (4 5 6 1 2 3)

(define moverAdelante(lambda(L N)
(if (or (equal? N 0) (null? L)) L
        (moverAdelante (concatenar (list (ultimoElem L)) (qu L)) (- N 1)))))
```

; MATEMATICAS CON LISTAS

; Suma los elementos de una lista.

; Ej: (sumaElem '(1 2 3 4 5 6 7)) => 28

```
(define sumaElem (lambda (L)
        (if (null? L) 0 (+ (car L) (sumaElem (cdr L))))))
```

; Suma los elementos de una lista. Multinivel.

; Ej: (sumaElemM '(1 2 (3 4) 5 (6 7) 0)) => 28

```
(define sumaElemM (lambda (L)
        (if (null? L) 0 (sumaElem (listaAtomos L)))))
```

; Suma los elementos respectivos de dos listas, generando otra con los resultados. Ambas listas de igual tamano.

; Ej: (sumaListas '(1 7 4) '(9 3 6)) => (10 10 10)

```
(define sumaListas (lambda (L1 L2)
    (if (null? L1) '()
       (cons (+ (car L1) (car L2)) (sumaListas (cdr L1) (cdr L2))))))

; Resta los elementos respectivos de dos listas, generando otra con los resultados. Ambas listas de igual tamano.
; Ej: (restaListas '(1 7 4) '(9 3 6)) => (-8 4 -2)

(define restaListas (lambda (L1 L2)
    (if (null? L1) '()
       (cons (- (car L1) (car L2)) (restaListas (cdr L1) (cdr L2))))))

; Multiplicacion de una lista por un escalar
; Ej: (multiplicaEscalar 2 '(1 2 3 4)) => (2 4 6 8)

(define multiplicaEscalar (lambda (N L)
     (if (null? L) '()
        (cons (* N (car L)) (multiplicaEscalar N (cdr L))))))

; Producto cartesiano de dos listas (ambas de igual longitud)
; Ej: (productoCA '(1 2 3) '(2 3 4)) => 20

(define productoCA (lambda (L1 L2)
       (if (null? L1) 0
          (+ (* (car L1) (car L2)) (productoCA (cdr L1) (cdr L2))))))
```

```
; Determina cuantas veces se repite X en una lista.
; Ej: (ocurre 4 '(1 3 2 4 8 9 4)) => 2

(define ocurre (lambda (X L)
      (if (null? L) 0
         (if (equal? (car L) X) (+ 1 (ocurre X (cdr L)))
            (ocurre X (cdr L))))))

; Determina cuantas veces se repite X en una lista. (Multinivel)
; Ej: (ocurreM 3 '(1 2 (4 6 3) (2 4) 8 (1 (5 4 0)) 3)) => 2

(define ocurreM (lambda (X L)
      (if (null? L) 0
         (if (list? (car L)) (+ (ocurreM X (car L)) (ocurreM X (cdr L)))
            (if (equal? (car L) X) (+ 1 (ocurreM X (cdr L)))
               (ocurreM X (cdr L)))))))

; Calcula la cantidad de elementos iguales en la misma posicion en dos listas.
; Ej: (elemIguales '(1 2 3 4 5) '(7 2 8 6 5)) => 2  (por el 2 y 5)

(define elemIguales (lambda (L1 L2)
       (if (or (null? L1) (null? L2)) 0
          (if (equal? (car L1) (car L2)) (+ 1 (elemIguales (cdr L1) (cdr L2)))
             (elemIguales (cdr L1) (cdr L2))))))
```

```
; METODOS DE ORDENAMIENTO


; Ordena acendentemente
; Ej: (ordenar<= '(8 4 2 1 9 2 0 7)) => (0 1 2 2 4 7 8 9)


(define ordenar<= (lambda (L)
(if (null? L) '()
   (if (<= (car L) (minl L)) (concatenar (list (car L)) (ordenar<= (cdr L)))
            (ordenar<= (concatenar (cdr L) (list (car L))))))))


; Ordena descendentemente
```

```
; Ej: (ordenar<= '(8 4 2 1 9 2 0 7)) => (9 8 7 4 2 2 1 0)

(define ordenar>= (lambda (L) (invertir (ordenar<= L))))


; ARBOLES BINARIOS

; Ejemplos:
; (raiz '(a (b () ()) (c () ()))) => a
; (izq '(a (b () ()) (c () ())))  => (b () ())
; (der '(a (b () ()) (c () ())))  => (c () ())

(define raiz (lambda (A) (car A)))

(define izq (lambda (A) (cadr A)))

(define der (lambda (A) (caddr A)))

(define hoja? (lambda (A) (if (and (null? (izq A)) (null? (der A))) #t #f)))

; los nodos de un arbol pueden ser numeros o simbolos
; Ej: (nodo? 'A) => #t

(define nodo? (lambda (C) (or (number? C) (symbol? C))))

; arbol?
; Ej: (arbol? '(a (b () ()) (c () ()))) => #t
```

```
(define arbol? (lambda (A)
            (cond  ((null? A) #t)
                ((nodo? A) #f)
                ((!= (long A) 3) #f)
                (else (and (nodo? (car A))
                        (arbol? (cadr A))
                        (arbol? (caddr A)))))))

; Ej: (arbolito? '(a (b () ()) (c () ())) => #t

(define arbolito? (lambda (L)
            (if(equal? L '()) #t
               (if(and (not (nodo? L)) (=(long L)3) (nodo?(car L)))
                 (and (arbolito? (cadr L)) (arbolito? (caddr L)) )
                  #f))))

; Ej: (arbusto? '(a (b () ()) (c () ())) => #t

(define arbusto? (lambda (A)
            (if (null? A) #t
              (if (= (long A) 3)
                 (and (arbusto? (cadr A)) (arbusto? (caddr A)))
                 #f))))

; Peso de un Arbol: sumatoria (cada hojas por su nivel)
; Ej: (peso '(0 (6 () () ) (7 (3 () ()) () )) 0) => 12   (6*1 + 3*2)

(define peso (lambda (A N)
```

```
            (if (null? A) 0
               (if (hoja? A) (* N (raiz A))
                  (+ (peso (izq A) (+ 1 N)) (peso (der A) (+ 1 N)))))))

; Inorden
; Ej: (inorden '(a (b () ()) (c () ()))) => (b a c)
;     (inorden '(a (b (d () ()) (e () ())) (c (f () ()) (g () ())))) => (d b e a f c g)

(define inorden(lambda(A)
(if (not (arbol? A))#f
   (if (null? A)'()
          (if (and (nodo? A) (null? (cadr A)) (null? (caddr A)))
         (car A)
            (concatenar (concatenar (inorden(cadr A))
         (list(car A))) (inorden(caddr A)))))))

; Version 2
(define inorden2 (lambda (A) (if (null? A) '()
                          (append (inorden2 (izq A))
                                (list (raiz A))
                                (inorden2 (der A))  ))))

; Postorden
; Ej: (postorden '(a (b () ()) (c () ()))) => (b c a)
;     (postorden '(a (b (d () ()) (e () ())) (c (f () ()) (g () ())))) => (d e b f g c a)

(define postorden(lambda(A)
(if (not (arbol? A))#f
```

```
   (if (null? A)'()
  (if (and (nodo? A) (null? (cadr A) ) (null? (caddr A)))
(car A)
                (concatenar (concatenar (postorden (cadr A))
      (postorden(caddr A))) (list(car A))  ))))))

; Version 2
(define postorden2 (lambda (A) (if (null? A) '()
(append (postorden2 (izq A)) (postorden2 (der A)) (list (raiz A)) ))))

; Preorden
; Ej: (preorden '(a (b () ()) (c () ()))) => (a b c)
;     (preorden '(a (b (d () ()) (e () ())) (c (f () ()) (g () ())))) => (a b d e c f g)

(define preorden(lambda(A)
(if (not (arbol? A))#f
   (if (null? A)'()
  (if (and (nodo? A) (null? (cadr A)) (null? (caddr A)))
(car A)
                                (concatenar (concatenar (list (car A)) (preorden(cadr A)))
(preorden(caddr A))  ))))))
; Version 2
(define preorden2 (lambda (A) (if (null? A) '()
                           (append (list (raiz A))
                                 (preorden2 (izq A))
                                 (preorden2 (der A))  ))))

; Recorrido Horizontal (ARBOL BINARIO)
```

```
; Ej: (RH '(a (b (d () ()) (e () ())) (c (f () ()) (g () ())))) => (a b c d e f g)

(define RH (lambda (A) (RHaux (list A))))

(define RHaux (lambda (A)
  (if (null? A) '()
     (if (null? (car A)) (RHaux (cdr A))
     (cons (raiz (car A))
          (RHaux (append (cdr A) (list (izq (car A)) (der (car A))))))))))

; Pertenece el elemento N al arbol A?
; Ej: (aPertenece 'd '(a (b () (f () ())) (c (e () ()) (d () ())))) => #t

(define aPertenece(lambda(N A)
                (if(not(arbol? A))#f
                (if(null? A)#f
                (if(equal? N (car A))#t
                  (or (aPertenece N (cadr A)) (aPertenece N (caddr A))))))))

; Arbol completo de nivel n y funciones auxiliares
; Ejemplo de arbol completo: '(a (b (d () ()) (e () ())) (c (f () ()) (g () ())))

; Arbol de nivel 0?
; Ej: (adn0? '()) => #t

(define adn0? (lambda (A) (cond ((null? A) #t)
                          ((and (null? (izq A))
                               (null? (der A))) #t)
```

```
                        (else #f))))

; Devuelve el numero de nivel de un arbol
; Ej: (nivel '(a (b (d () ()) (e () ())) (c (f () ()) (g () ())))) => 2

(define nivel (lambda (A)
(if (adn0? A) 0 (+ 1 (max (nivel (izq A)) (nivel (der A)))))))

; #t si un Arbol de nivel N es completo.
; Ej: (completo? '(a (b (d () ()) (e () ())) (c (f () ()) (g () ())))) => # t
;     (completo? '(a (b (d () ()) (e () ())) (c (f () ()) ()))) => #f

(define completo? (lambda (A) (completoAux A (nivel A))))

(define completoAux (lambda (A N)
  (cond ((null? A) #f)
     ((= N 0) #t)
     ((= N 1) (and (notnull? (izq A)) (notnull? (der A))))
     (else (and (completoAux (izq A) (- N 1)) (completoAux (der A) (- N 1)))))))
```

```
; GRAFOS

; Representación ((a b) (b a) (a c))  c<--a<-->b

; adyacente no dirigido
; no usa la representación de arriba, sino ((a b) (a c)) c<-->a<-->b

(define adyacente (lambda (N1 N2 G)
 (if (or (pertenece (list N1 N2) G) (pertenece (list N2 N1) G)) #t #f)))

;adyacente dirigido
(define adyacenteD (lambda (N1 N2 G)
                (if (pertenece (list N1 N2) G) #t #f)))
```

```
; Devuelve una lista de nodos
; (nodos '((a b) (b d) (d a) (a c) (c b))) => (d a c b)

(define nodos (lambda(G)
      (eliminaR (listaAtomos G))))

; Existe un camino entre los nodos E1 y E2 en el grafo G?
; Ej: (camino 'a 'd '((a b) (b c) (c d) (e e) (b d) (e a) (d e))) => #t

(define camino (lambda (E1 E2 G)
         (caminoAux E1 E2 (nodos G) G)))

(define caminoAux(lambda(E1 E2 N G)
       (if (or (null? N) (sumidero? E1 G)) #f
       (if (adyacenteD E1 E2 G) #t
       (if (adyacenteD E1 (car N) G)
          (caminoAux (car N) E2 (eliminaX E1 N) G)
          (caminoAux E1 E2 (moverAtras N 1) G))))))

; Existe un sumidero en el grafo G?
; Ej: (sumidero? 'e '((a b) (b c) (c d) (e e) (b d) (e a) (d e))) => #f
;     (sumidero? 'e '((a b) (b c) (c d) (b d) (d e))) => #t

(define sumidero? (lambda(N G)
        (if (null? G) #f
          (sumidero?Aux N G))))

(define sumidero?Aux (lambda(N G)
```

```
        (if (null? G) #t
        (if (equal? (caar G) N) #f
           (sumidero?Aux N (cdr G))))))

; Es conexo el grafo G?
; Usamos un algoritmo distinto  al que usamos en prolog, en lugar de ver si para cada nodo
; hay un camino a todos los demás, vemos si en una lista de todos los nodos se puede llegar
; del 1º al 2º, del 2º al 3º...
; (conexo? '((a b) (b c) (c a) (a e))) => #t
; (conexo? '((a b) (b c) (c a) (a e) (t r))) => #f

(define conexo? (lambda (G)
              (conexoAux (nodos G) G)))

(define conexoAux (lambda (LN G)
                 (if (null? (cdr LN)) #t
                    (if (camino (car LN) (cadr LN) G) (conexoAux (cdr LN) G)
                       #f))))
```

```
; COMBINATORIA

; PERMUTACION CON REPETICIONES TOMADAS DE A N
; Ej: (perConRep '(1 2 3) 2) => ((1 1) (1 2) (1 3) (2 1) (2 2) (2 3) (3 1) (3 2) (3 3))

(define perConRep (lambda (L long)
                  (if (= 1 long) (mapS list L)
                     (distribuirPCR L (perConRep L (- long 1))))))



(define distribuirPCR (lambda (L1 L2)
                   (if (null? L1) '()
                           (concatenar (mapS (lambda (X) (cons (car L1) X)) L2)
                                 (distribuirPCR (cdr L1) L2)))))

; PERMUTACIONES SIN REPETICIONES TOMADAS DE A N
; Ej: (perSinRep '(3 3 1) 2) => ((3 3) (3 1) (3 3) (3 1) (1 3) (1 3))
;      (perSinRep '(1 2 3) 2) => ((1 2) (1 3) (2 1) (2 3) (3 1) (3 2))
```

```
(define perSinRep (lambda (L N)
   (permutaAux L L N)))

   (define permutaAux (lambda (L M N)
      (if (or (< N 1) (> N (long L))) ()
      (if (= N 1) (listaElem L)
      (if (> N 2)
         (permutaAux (map1 L M) M (- N 1)) (map1 L M))))))

   (define map1
    (lambda (L M)
      (if (null? L) L
       (append (distribuye (car L) (borraelemSLdeL (car L) M))  (map1 (cdr L) M)))))



   ;Borra los elementos de SL en la lista L.
   (define borraelemSLdeL
    (lambda (SL L)
      (if (null? SL) L
         (if (list? SL)
      (borraelemSLdeL (cdr SL) (elimina1ra L (car SL) ))
         (elimina1ra L SL)))))

   ;genera una lista del tipo ((X L1) (X L2) ... (X Ln))
   (define distribuye (lambda (X L)
         (if (null? L) ()
           (if (list? X)
           (append (list(append X (list(car L)))) (distribuye X (cdr L)))
```

```
                (append (list (list X (car L))) (distribuye X (cdr L)))))))


   (define listaElem (lambda (L)
         (if (null? L) () (append (list (list (car L))) (listaElem (cdr L))))))

; COMBINACIONES CON REPETICIONES
; Ej: (comConRep '(1 2 3) 2) => ((1 1) (2 1) (2 2) (3 1) (3 2) (3 3))

(define comConRep (lambda (L long)
                (elimComRep (perConRep L long))))

; COMBINACIONES SIN REPETICIONES
; Ej: (comSinRep '(1 2 3) 2) => ((2 1) (3 1) (3 2))

(define comSinRep (lambda (L long)
                (elimComRep (perSinRep L long))))

; Eliminar Combinaciones Repetidas

(define elimComRep (lambda (L)
                (if (null? L) '()
                   (if (perteneceCom (car L) (cdr L)) (elimComRep (cdr L))
                      (cons (car L) (elimComRep (cdr L)))))))

(define perteneceCom (lambda (X L)
                  (if (null? L) #f
                     (if (mismaCom X (car L)) #t
                        (perteneceCom X (cdr L))))))
```

```
(define mismaCom (lambda (C1 C2)
                (if (null? C1) #t
                   (if (not (pertenece (car C1) C2)) #f
                      (mismaCom (cdr C1) (elimina1ra C2 (car C1)))))))

(define elimina1ra
  (lambda(L e)
     (if (null? L) '()
        (if (equal? e (car L)) (cdr L)
           (cons (car L) (elimina1ra (cdr L) e))))))



; FUNCIONES DE ORDEN SUPERIOR

; ((op +) 2 3) => 5  (la operacion es binaria -> +)



(define op (lambda (O) (lambda (X Y) (O X Y))))

; ((op +) 2 3) => 5  (la operacion es binaria -> +)

(define op2 (lambda (O) O))

(define dos (lambda (F) (lambda (X)
                     (F(F X)))))
; ((dos ++) 1) => 3  (la operacion es unaria -> ++)
```

```
; Repetir  (F(F(F(F..F(X)))))

(define repetir (lambda (F N) (lambda (X)
      (if (= N 1) (F X) (F ((repetir F (- N 1)) X))))))

; Realiza una accion 'F' sobre todos los elementos de una lista (Ej ++, -- ...)
; Similar al do: de SmallTalk
; ((map ++) '(1 2 3 4)) => (23 4 5)

(define map (lambda (F) (lambda (L)
                  (if (null? L) '()
                     (cons (F (car L)) ((map F) (cdr L)))))))

; Igual que map pero con un solo lambda

(define mapS (lambda (F L)
           (if (null? L) '()
              (cons (F (car L)) (mapS F (cdr L))))))


; Ejemplo de uso de map. A cada elemento de la lista se le suma N
; (usoMap 1 2 '(1 1 1)) => (4 4 4)

(define usoMap (lambda  (N1 N2 L)
             ((map (lambda (X) (+ N1 N2 X))) L)))

(define usoMap2 (lambda  (N1 N2 L)
             (mapS (lambda (X) (+ N1 N2 X)) L)))
```

```
(define filterT (lambda (F L)
            (if (null? L) '()
               (if (F (car L)) (cons (car L) (filterT F (cdr L)))
                  (filterT F (cdr L))))))

(define filterF (lambda (F L) (filterT (lambda (X) (not (F X))) L)))



; EJERCICIOS RESUELTOS

; Serie de Fibonacci: 1 1 2 3 5 8 13 ...
; Devuelve el elemento enesimo de la serie de Fibonacci.
;
; Ej: (Fibonacci 2) => 1        (Fibonacci 6) => 8

(define Fibonacci (lambda (N)
              (cond ((= N 1) 1)
                  ((= N 2) 1)
                  (else (+ (Fibonacci (- N 1)) (Fibonacci (- N 2)))))))

; Fibonacci iterativo
; Ej: (Fibonacci 2) => 1        (Fibonacci 6) => 8

(define fib
  (lambda (n)
```

```
   (fib-iter 1 0 n)))

(define fib-iter
  (lambda (a b count)
    (if (= count 0)
        b
        (fib-iter (+ a b) a (- count 1)))))

; Tartaglia
; Devuelve la fila del triangulo de Tartaglia correspondiente a N
;
; Ej: (Tartaglia 7) => (1 6 15 20 15 6 1)

(define Tartaglia (lambda (N)
 (cond ((= N 1) '(1))
 ((= N 2) '(1 1))
 (else (append (append '(1) (sumaDeADos (Tartaglia (- N 1)))) '(1)) ))))

(define sumaDeADos (lambda (L)            ; L=(1 2 3 4 5)  => (3 5 7 9)
 (if (> (long L) 1) (cons (+ (car L) (cadr L)) (sumaDeADos (cdr L))) '() )))

; Aceptador de Estados Finitos (maquina de estados finitos).
;(AEF '(1 2 1) '(A B A ((A B 1) (A C 2) (B A 2) (C B 3)))) => t

(define AEF (lambda (S M)
     (if (null? S)
        (if (equal? (F M) (A M) )#t #f)
     (AEF (cdr S) (trans (car S) M M)))))
```

```
(define trans (lambda (Tr M1 M2)
    (if (and (equal? (N1 M1) (A M1)) (equal? (T M1) Tr) )
        (list (I M1) (F M1) (N2 M1) (G M2))
        (trans  Tr (list (I M1) (F M1) (A M1) (cdr (G M1))) M2))))

(define I car) (define F cadr) (define A caddr) (define G cadddr)

(define N1 (lambda (M)
        (caar (G M))))

(define N2 (lambda (M)
        (cadar (G M))))

(define T (lambda (M)
        (caddar (G M))))

; EJERCICIO 34
; Representa más eficientemente una matriz rala. Devuelve una lista conteniendo ternas (X Y V) V<>0
; Ej: (convierte '((10 0 0 20 0) (0 1 5 0 0) (2 0 0 0 0)))=> ((1 1 10) (1 4 20) (2 2 1) (2 3 5) (3 1 2))

(define convierte2 (lambda (M)
            (if (null? M) '()
                (convierte2Aux 1 M))))

(define convierte2Aux (lambda (F M)
```

```
 (if (null? M) '()
    (concatenar (fila F 1 (car M)) (convierte2Aux (++ F) (cdr M))))))


(define fila (lambda (F C L)
 (if (null? L) '()
            (if (equal? (car L) 0) (fila F (++ C) (cdr L))
               (cons (list F C (car L)) (fila F (++ C) (cdr L)))))))


; EJERCICIO 37
; Dado un arbol n-ario con nodos 1 o 0, determina si una palabra puede leerse recorriendo alguna rama
; Ej: (check '(1 0 0 0) '(1 (1 (1 (1)) (1)) (0 (0)) (1 (0)))) => #f
;     (check '(1 0 0)  '(1 (1 (1 (1)) (1)) (0 (0)) (1 (0)))) => #t


(define check (lambda (P A)
      (if (null? P) #t
      (if (and (equal? (car A) (car P)) (enHijo (cdr A) (cdr P))) #t #f))))


(define enHijo (lambda (H P)
      (if (and (null? H) (null? P)) #t
      (if (and (null? H) (not (null? P))) #f
        (not (null? (filterT (lambda (X) (check P X)) H)))))))


; EJERCICIO 38
; Dada una lista de funciones (de un argumento) y una de elementos, aplica cada una de las funciones
; a los elementos de la lista.
; (mapFun '(++ -- par?) '(1 2 3 5)) => ((2 3 4 6) (0 1 2 4) (#f #t #f #f))
```

```
(define mapFun (lambda (F L)
            (if (null? F) '()
               (append (list (mapS (eval (car F)) L))
                     (mapFun (cdr F) L)))))

; Ejercicio 36: Funcion impar
; F es una funcion y L una lista de valores para el argumento x de F
;  (fi '(+ 1 X) '(1 2 3)) => #f
;  (fi '(* X X) '(1 2 3)) => #t
;  (fi '(* (+ X 2) (+ X 2)) '(1 2 3)) => #f
;  (fi '(* (+ X 2) (- X 2)) '(1 2 3)) => #t

 (define fi (lambda (F L)
    (if (equal? (fiAux F 1 L) (fiAux F -1 L)) #t #f)))

 (define fiAux (lambda (F V L)
       (if (null? L) L
        (cons (evaluar (xReemplazarM 'X (* V (car L)) F) ) (fiAux F V (cdr L))))))

 (define evaluar (lambda (F)
        (cond [(and (list? (cadr F)) (list? (caddr F)))
                ((eval (car F)) (evaluar (cadr F) ) (evaluar(caddr F) ))]
            [(list? (cadr F)) ((eval (car F)) (evaluar (cadr F) ) (caddr F))]
            [(list? (caddr F)) ((eval (car F)) (cadr F) (evaluar (caddr F)))]
            [else ((eval (car F)) (cadr F) (caddr F))] )))

; EJERCICIO 39-b
; Dada una lista de árboles, devuelve el de menor peso.
```

```
;Ej: (mapeoArboles '((0 (6 () () ) (7 () () )) (0 (0 (2 () ()) (3 () () )) (5 () () )))) => (0 (6 () ()) (7 () ()))

(define mapeoArboles (lambda (LA)
     (if (null? LA) '0
        (enesimoElem (car (ocurrencias (armalistaSumas LA)
    (minL (armalistaSumas LA)))) LA))))

(define armalistaSumas (lambda (LA)
     (if (null? LA) '()
        (cons (peso (car LA) 0) (armalistaSumas (cdr LA))))))



; EJERCICIO 40
; Cambia de una forma de representación de grafos a otra.
; Ej: (convierte '((a b c d e) ((a b) (b c) (b d) (c e) (d a) (d e) (e a))))
;     ((a (b)) (b (c d)) (c (e)) (d (a e)) (e (a)))

(define convierte(lambda (L)
    (if (null? L) '()
       (convierteAux (car L) (cadr L)))))

(define convierteAux(lambda (L G)
    (if (null? L) '()
       (cons (list (car L) (extrae (car L) G)) (convierteAux (cdr L) G)))))

(define extrae (lambda (N G)
    (if (null? G) '()
       (if (equal? N (caar G)) (cons (cadar G) (extrae N (cdr G)))
```

```
          (extrae N (cdr G))))))


; EJERCICIO 41
; Se quiere calcular las comiciones de un viajante.
; Ej: (comisiones '((Pepe (100 A) (50 B)) (Ana (50 A) (100 B) (20 C))) '((A 0.5) (B 0.2) (C 0.1)))
;     ((pepe (100 a) (50 b) 60.0) (ana (50 a) (100 b) (20 c) 47.0))


(define comisiones (lambda (Viajantes P)
(if (null? Viajantes) '()
   (append (list (append (car Viajantes)
         (list (sumaElem (comicionV (cdar Viajantes) P)))))
         (comisiones (cdr Viajantes) P)))))


(define comicionV (lambda (LV P)
               (mapS (lambda (X) ( comicionP P X)) LV)))


(define comicionP (lambda (P V)
  (if (equal? (caar P) (cadr V))
     (* (cadar P) (car V))
     (comicionP (cdr P) V))))
```

; EJERCICIO DE EXAMEN: 05/08/2002

; Crear una función de orden superior para calcular el seno de A (en radianes)

; con una serie de N elementos

; NO esta resultando igual a la funcion seno, no debe ser correcta la formula que nos dieron.-

(define Serie (lambda (N) (lambda (A)

```
                    (if (= N -1) 0
                       (+ (/ (* (potencia -1 N) (potencia A (+ N 1)))
                            (fact (+ N 1)))
                          ((Serie (- N 1)) A))))))

(define SerieS ( lambda (N A)
                    (if (= N -1) 0
                       (+ (/ (* (potencia -1 N) (potencia A (+ N 1)))
                            (fact (+ N 1)))
                          (SerieS (- N 1) A)))))

; EJERCICIO DE EXAMEN: Sumador de números binarios
; Ej: (SumaBin '(1 0 0) '(1)) => (1 0 1)
;     (SumaBin '(1 1 1 0 1) '(1 0 0 1)) => (1 0 0 1 1 0)

(define SumaBin (lambda (N1 N2)
   (cond [(null? N2) N1]
   [(null? N1) N2]
   [else (if (= 2 (+ (ultimoElem N1) (ultimoElem N2)))
      (concatenar2 (SumaBin (SumaBin '(1)  (qu N2)) (qu N1)) 0)
            (concatenar2 (SumaBin (qu N2) (qu N1))
   (+ (ultimoElem N2) (ultimoElem N1))))])))

; EJERCICIO DE EXAMEN: Composicion de funciones (f g h) => (f(g(h)))
; (componer '((+ (ln (^ x 2)) x) (+ x 1) (sin x))) => (+ (ln (^ (+ (sin x) 1) 2)) (+ (sin x) 1))
```

```
(define componer (lambda (L)
            (if (= (long L) 1) (car L)
               (componer (cons (xReemplazarM 'x (cadr L) (car L))
                        (cddr L))))))

; Orden( Criterio Lista )
; Dada una lista de números y un criterio de ordenación (<, >, >=, ...) devuelve las subsecuencias
; (de longitud mayor a 1) completas que verifican este criterio.
; Ej: (orden < '(1 2 3 1 7 15 24 6 67 78 9)) => ((1 2 3) (1 7 15 24) (6 67 78))

(define orden(lambda(criterio L)
      (if (null? L) L
      (if (null? (ordenAux criterio L)) (orden criterio (cdr L))
         (cons (cons (car L) (ordenAux criterio L))
         (orden criterio (sacaNpri(+ 1 (long (ordenAux criterio L))) L )))))))

(define ordenAux (lambda(criterio L)
      (if (= (long L) 1) '()
      (if (criterio (car L) (cadr L))
         (cons (cadr L) (ordenAux criterio (cdr L)))
         '()))))

; EJERCICIO FOR (no tiene mucha logica..)
; Ej: (for 1 2000 1 '((lambda () #t) () ())) => ()

(define for  (lambda (i f c s)
            (if (<= i f)
               (and (evaluarFor s)
```

```
                (for (+ i c) f c s)
            () ))))

(define evaluarFor (lambda (s)
        (if (null? s) s
           (if ((eval (car s))) (evaluarFor (cadr s))
                   (evaluarFor (caddr s))))))

; ARBOL DE DIRECTORIO
; Ej: (buscar ar1 '(d0 d1 a1)) => "archivo"
; Ej: ((eval (lambda () "archivo"))) => "archivo"

(define buscar (lambda (D A)
 (if (null? A) ((eval (car D)))
    (buscar (cdar (filterT (lambda (X) (equal? (car A) (car X))) D)) (cdr A)))))

(define ar1 '((d0 (d1 (a1 (lambda() "archivo"))))))
```

```
-----RECIBE 3 NUMEROS, Y DEVUELVE LA SUMA DE LOS CUADRADOS DE LOS DOS MAYORES-----
(define suma2cuadradosmaximos
  (lambda (x y z)
    (cond
      ((and (> y z) (> x z))(+ (* x x)(* y y)))
      ((and (> y x) (> z x))(+ (* y y)(* z z)))
      (else (+(* x x)(* z z))))))
```

```
--------------------SUMA EL VALOR DE A CON EL VALOR ABSOLUTO DE B--------------------
(define a-suma-abs-b
  (lambda (a b)
    (if (< b 0) (- a b) (+ a b))))


------------------------------------------------------------FACTORIAL------------------------------------------------------------
(define factorial
  (lambda (x)
    (if (= x 0) 1
        (* x (factorial (- x 1))))))


------------------------------------------------------------FIBONACCI------------------------------------------------------------
(define fibs
  (lambda (x)
    (if (= x 1) 1
        (if (= x 2) 1
           (+ (fibs (- x 1))(fibs(- x 2)))))))


--------------------RECIBE DOS ARGUMENTOS, LA BASE Y EL EXPONENTE--------------------
(define my-expt
  (lambda (x y)
    (if (= y 0) 1
        (* x (my-expt x (- y 1))))))


--------------------MAXIMO COMUN DIVISOR ENTRE DOS NUMEROS--------------------
(define mcd
  (lambda (x y)
    (if (= y 0) x
```

```
      (mcd y (modulo x y)))))

----------OBTENER TODOS LOS DIVISORES----------

(define primo
  (lambda (x)
    (letrec ((obtenerdivisores
           (lambda (a b)
             (if (= b 1) (list 1)
                 (if (= (modulo a b) 0) (append (list b) (obtenerdivisores a (- b 1)))
                     (obtenerdivisores a (- b 1)))))))
          (obtenerdivisores x x))))


----------TRUE SI ES UN NUMERO PRIMO----------

(define mcd
  (lambda (x y)
    (if (= y 0) x
      (mcd y (modulo x y)))))

(define concatenar
  (lambda (l1 l2)
    (if (null? l1) l2
      (cons (car l1) (concatenar (cdr l1) l2)))))

(define obtenerdivisores
  (lambda (x y)
    (if (= y 1) (list 1)
      (if (= (modulo x y) 0) (concatenar (list y) (obtenerdivisores x (- y 1)))
```

```
          (obtenerdivisores x (- y 1))))))


(define primo
  (lambda (x)
    (if (> (length (obtenerdivisores x x)) 2) #f #t)))


----------SUMA EL CUADRADO DE LOS PRIMEROS K NATURALES----------
(define sumakcuadradosnaturales
  (lambda (k)
    (if (= k 0) 0
        (+ (* k k) (sumakcuadradosnaturales (- k 1))))))


----------SUMA EL COSENO DE LOS PRIMEROS K NATURALES----------
(define sumakcosenosnaturales
  (lambda (k)
    (if (= k 0) 1
        (+ (cos k) (sumakcosenosnaturales (- k 1))))))


----------DADA UNA LISTA, DIVIDE CADA ELEMENTO POR LA MITAD----------
(define list-halve
  (lambda (l)
    (map (lambda (x) (/ x 2)) l)))


----------DADA UNA LISTA, RETORNA LA RESTA SUCESIVA DE SUS ELEMENTOS----------
(define list-sub
  (lambda (l)
    (letrec ((suma (lambda (x)
                     (if (null? x) 0
```

```
                    (+ (car x)(suma (cdr x)))))))
    (- (* 2 (car l))(suma l)))))

----------RETORNA EL K-ESIMO ELEMENTO DE UNA LISTA----------

(define my-list-ref
  (lambda (k l)
    (if (or (< k 1)(> k (length l))) #f
        (if (= k 1) (car l)
            (my-list-ref (- k 1) (cdr l))))))

----------RETORNA LA LISTA INVERTIDA----------

(define my-reverse
  (lambda (l)
    (if (null? l) '()
        (concatenar (my-reverse (cdr l)) (list (car l))))))

----------RETORNA LOS N PRIMEROS ELEMENTOS----------

(define my-take
  (lambda (n l)
    (if (> n (length l)) l
        (if (= n 0) '()
            (concatenar (list (car l)) (my-take (- n 1) (cdr l)))))))

----------RETORNA LOS N ULTIMOS ELEMENTOS----------

(define my-drop
  (lambda (n l)
    (if (> n (length l)) '()
        (if (= n 0) l
```

```
        (my-drop (- n 1) (cdr l))))))

----------AGREGAR SI----------

(define concatenar (lambda (l1 l2) (if (null? l1) l2 (cons (car l1) (concatenar (cdr l1) l2)))))

(define pertenece (lambda (x L) (if (null? L)
                                    #f
                                    (if (eqv? (car L) x)
                                       #t
                                       (pertenece x (cdr L))))))

;agrega el elemento x al final de la lista en caso de no existir sino no lo agrega
(define addif (lambda (x L) (if (pertenece x L)
                                L
                                (concatenar L (list x)))))


;Prueba
(define lista1 '(a b c d e f g h 3 i j k l m n o))
(define lista2 (list 1 2 3 4))
(concatenar lista1 lista2)
(pertenece 3 lista1)
(addif 'z lista2)

----------VALIDAR FECHA----------

(define valdia
  (lambda (x)
    (if (or (< x 1)(> x 31)) #f #t)))
```

```
(define valmes
  (lambda (x)
    (if (or (< x 1)(> x 12)) #f #t)))
(define valdiamesaño
  (lambda (x y z)
    (if (and (> x 29)(= 2 y))#f
        (if (and (> x 30)(or (= 4 y)(= 6 y)(= 9 y)(= 11 y)))#f
            (if (and (= x 29)(= 2 y)(not (integer? z/4))) #f #t)))))
(define valfecha
  (lambda (x y z)
    (if (and (valdia x)(valmes y)(valdiamesaño x y z)) #t #f)))

----------CONTAR CANTIDAD A LA IZQUIERDA DE UN ELEMENTO----------
(define cant-izq
  (lambda (x l)
    (if (null? l) 0
        (if (eqv? x (car l)) 0 (+ 1 (cant-izq x (cdr l)))))))

----------VALIDAR HORA----------
(define sumahora
  (lambda (listaH1 listaH2)
    (if (or (null? listaH1)(null? listaH2))
        (display "error")
        (let ((hh (+ (car listaH1) (car listaH2)))
              (mm (+ (cadr listaH1) (cadr listaH2)))
              (ss (+ (caddr listaH1) (caddr listaH2)))
              )
          (if (and (> ss 59) (> mm 59))
```

```
            (list (+ 1 hh) (- (+ mm 1) 60) (- ss 60))
            (if(> ss 59)
              (list  hh (+ mm 1) (- ss 60))
              (if  (> mm 59)
                  (list  (+ 1 hh) (- mm 60) ss)
                  (list hh mm ss))))
          ;(cons hh (cons mm (cons ss '())))
          )
    )
  )
  )

---------------------------------CREA UNA LISTA CON UN ARCHIVO---------------------------------
;Esta funcion crea una lista con el archivo
(define mostrar
  (lambda (x) (define i (open-input-file x));abro el archivo
  (define m (lambda (line)
    (cond
      ((equal? line eof) '())
      (else (cons line (m (read-line i))))
  )
  )) (m (read-line i))
  )
)

---------------------------------DISTANCIA ENTRE 2 PUNTOS---------------------------------
(define distance2d(lambda(x y)(sqrt(+(*(- (car y) (car x))(- (car y) (car x)))(*(- (cdr y) (cdr x))(-
(cdr y) (cdr x)))))))
```

```
----------------------------------------------APLANAR----------------------------------------------
;Concatena dos listas
(define concatenar (lambda (L1 L2) (if (null? L1)
                                       L2
                                       (cons (car L1) (concatenar (cdr L1) L2)))))

;Crea una sola lista con los elementos de las listas que componen a L
(define aplanar (lambda (L) (if (null? L)
                                (list)
                                (if (real? (car L))
                                   (concatenar (list (car L)) (aplanar (cdr L)))
                                   (concatenar (aplanar (car L)) (aplanar (cdr L)))))))

(define lista '((1 2 3)(9 (2 3 4))((((3 4 (7)))))))
lista
(aplanar lista)

-------------------------CUENTA ELEMENTOS A LA IZQUIERDA DE "X"-------------------------
;Cuenta la cantidad de elementos que se encuantran a la izquierda de x en la lista L
(define cant-izq (lambda (x L) (if (eqv? (car L) x)
                                   0
                                   (+ 1 (cant-izq x (cdr L))))))

-----------------------------------------CONCATENAR 2 LISTAS-----------------------------------------
;Concatena dos listas
(define concatenar (lambda (L1 L2) (if (null? L1)
                                       L2
```

```
                        (cons (car L1) (concatenar (cdr L1) L2)))))

----------CONCATENAR 2 LISTAS INTERCALANDO ELEMENTOS----------

;Concatena dos listas
(define concatenar (lambda (L1 L2) (if (null? L1)
                        L2
                        (cons (car L1) (concatenar (cdr L1) L2)))))

;Concatena dos listas elemento a elemento
(define conc1-1 (lambda (L1 L2) (if (null? L1)
                        L2
                        (if (null? L2)
                           L1
                           (concatenar (concatenar (list(car L1)) (list(car L2))) (conc1-1 (cdr L1) (cdr
L2)))))))


;Prueva
(define lista1 '(a b c d e f g h i j k l m n o))
(define lista2 (list 1 2 3 4))
(conc1-1 lista1 lista2)

----------CUENTA LOS ELEMENTOS A LA DERECHA DE "X"----------
(define contar(lambda (L)
            (if (null? L)
               0
               (+ 1 (contar (cdr L)))
               )
```

```
            )
 )

(define cont-der(lambda (L x)
              (if (null? L)
                 0
                 (if (eqv? (car L) x)
                    (contar (cdr L))
                    (cont-der (cdr L) x)
                      ))))

(cont-der (list 1 2 3 4) 2)

-------------------------------------MIEMBRO-------------------------------------
(define miembro?
          (lambda (a ls)
                    (cond
                              ((null? ls) #f)
                              ((equal? a (car ls)) #t)
                              (else (miembro? a (cdr ls))))))


(define pert (lambda (X L) (if (null? L)
                        #f
                        (if (eqv? (car L) X)
                           #t
                           (pert X (cdr L))))))
```

```
----------FILTRAR----------
(define filtrar
  (lambda (ls)
    (cond
      ((null? ls) `())
      ((miembro? (car ls) (cdr ls)) (filtrar (cdr ls)))
      (else (cons (car ls) (filtrar (cdr ls)))))))

----------LONGITUD----------
(define longitud (lambda (L) (if (null? L)
                                 0
                                 (+ 1 (longitud (cdr L))))))

(define lista (list 1 2 3 4 5 6 7 8 9))
(longitud lista)

----------INVERTIR----------
(define invertir (lambda (L) (if (null? L)
                                 (list )
                                 (append (invertir (cdr L)) (list (car L))))))

----------OCURRENCIA----------
(define ocurrencia
  (lambda (val ls)
    (let ocur ((v val) (l ls) (cont 0))
      (cond
        ((null? l) cont)
        ((equal? (car l) v) (ocur v (cdr l) (+ 1 cont)))
```

```
      (else (ocur v (cdr l) cont))))))

----------------------------------SUMA "X " A CADA ELEMENTO DE LA LISTA----------------------------------

;Concatena dos listas

(define concatenar (lambda (L1 L2) (if (null? L1)

                                       L2

                                       (cons (car L1) (concatenar (cdr L1) L2)))))

;Suma el valor x a cada elemento de L

(define sum_a_list (lambda (x L) (if (null? L)

                                     ()

                                     (concatenar (list(+ x (car L))) (sum_a_list x (cdr L))))))

---------------------------------------MAXIMO---------------------------------------

(define maximo

  (lambda (L)

    (if (null? (cdr L)) (car L)

        (max (car L) (maximo (cdr L))))))

(define maximo

  (lambda(l)

    (if(= (longitud l) 1) (car l)

       (if(> (car l) (maximo (cdr l))) (car l)

          (maximo (cdr l))

          )

       )

    )

  )
```

```
----------MINIMO----------
(define minimo
 (lambda(l)
  (if(= (longitud l) 1) (car l)
    (if(< (car l) (minimo (cdr l))) (car l)
      (minimo (cdr l))
      )
    )
  )
 )

----------ELIMINA----------
(define elimina
 (lambda (L x)
  (if (eqv? (car L) x) (cdr L)
    (concatenar (list (car L)) (elimina (cdr L) x)))))

----------ROTAR----------
;Concatena dos listas
(define concatenar (lambda (L1 L2) (if (null? L1)
                                       L2
                                       (cons (car L1) (concatenar (cdr L1) L2)))))
;Invierte los elementos de una lista
(define invertir (lambda (L) (if (null? L)
                                 (list )
                                 (concatenar (invertir (cdr L)) (list (car L))))))
```

```
(define ls (list ))
(define rotar (lambda (dir L) (if (eqv? dir 'izquierda)
                          (if (null? L)
                             (list )
                             (concatenar (cdr L) (list (car L))))
                          (if (null? (cdr L))
                             (list )
                             (concatenar (list (car (invertir L))) (invertir (cdr (invertir L))))))))
(define lista (list 1 2 3 4 5 6 7 8 9 0))
(rotar 'izquierda lista)
(rotar 'derecha lista)

-------------------------------------SUMA SI ES PAR (EVEN) O IMPAR (ODD)-------------------------------------
;Concatena dos listas
(define concatenar (lambda (L1 L2) (if (null? L1)
                              L2
                              (cons (car L1) (concatenar (cdr L1) L2)))))

;suma x a los elementos impares de L
(define sum_if_odd (lambda (x L) (if (null? L)
                          '()
                          (if (odd? (car L))
                             (concatenar (list (+ x (car L))) (sum_if_odd x (cdr L)))
                             (concatenar (list (car L)) (sum_if_odd x (cdr L)))))))

;suma x a los elementos impares de L
(define sum_if_even (lambda (x L) (if (null? L)
                          '()
```

```
                    (if (even? (car L))
                        (concatenar (list (+ x (car L))) (sum_if_even x (cdr L)))
                        (concatenar (list (car L)) (sum_if_even x (cdr L)))))))
```

----------N-ESIMO ELEMENTO DE UNA LISTA----------

```
(define n-esimo
 (lambda(l n)
  (if(= n 1) (car l)
     (n-esimo (cdr l) (- n 1))
     )
  )
 )
```

------RETORNA EL PRIMER ELEMENTO DE LA LISTA QUE ES UN NUMERO, SINO RETORNA FALSO------

```
(define primer_numero
 (lambda(l1)
  (if(null? l1) null?
     (if(number? (car l1)) (car l1)
        (primer_numero (cdr l1))
        )
     )
  )
 )
```

------CREA UNA LISTA CON ELEMENTOS QUE ESTAN EN LA PRIMERA PERO NO EN LA SGUNDA------

```
(define resta-listas
 (lambda(l1 l2)
  (if(null? l1) l1
```

```
      (if(pertenece (car l1) l2) (resta-listas (cdr l1) l2)
        (cons (car l1) (resta-listas (cdr l1) l2))
        )
      )
   )
 )
```

----------ULTIMO ELEMENTO DE UNA LISTA----------

```
(define ultimo
  (lambda(l)
      (if(null? (cdr l)) (car l)
        (ultimo (cdr l))
        )
   )
 )
```

----------PRIMER SUBLISTA DE UNA LISTA----------

```
(define primer_subl
  (lambda(x l)
    (if(null? l) l
      (if(not(equal? x (car l))) (cons (car l) (primer_subl x (cdr l)))
        (list)
        )
      )
   )
 )
```

----------LONGITUDES DE LAS SUBLISTAS ELEMENTOS DE UNA LISTA----------

```
(define longitudes
  (lambda (L)
    (letrec ((a (lambda (L)
                  (if (null? L) 0
                      (+ 1 (a (cdr L)))))))
      (if (null? L) '()
          (cons (a (car L)) (longitudes (cdr L)))
          )
      )
    )
  )

----------INVIERTE LOS PARES SI EL PRIMERO ES MAYOR AL SEGUNDO ELEMENTO----------
(define dar-vuelta
  (lambda (l)
    (if (< (cdr l) (car l))
        (cons (cdr l) (car l))
        l
    )
  )
)

(define order-pair
  (lambda (l)
    (if (null? l)
        '()
        (if (pair? (car l))
            (cons (dar-vuelta (car l))(order-pair (cdr l)))
```

```
          (order-pair (car l))
          )
     )
  )
)

;(order-pair '((1 . 2) (3 . 1) (4 . 5) (5 . 1))) ;--> ((1 . 2) (1 . 3) (4 . 5) (1 . 5))

------------------------------INVIERTE LISTA TODOS LOS NIVELES (FULLREVERSE------------------------------
(define fullreverse
  (lambda (x)
    (if (null? x)
        x
        (if (not (list? (car x)))
           (append (fullreverse (cdr x))(list(car x)))
           (append (fullreverse (cdr x))(list (fullreverse (car x))))
           )
        )
    )
)

;(fullreverse '(1 (2 3 4 (4 5) (3 (5 6)) 4) 5 6)) ;-->; ((4 ((6 5) 3) (5 4) 4 3 2) 1)

----------------------------------------SUMARUSA----------------------------------------
(define concatenar
  (lambda (l1 l2)
    (if (null? l1) l2
        (cons (car l1)(concatenar (cdr l1) l2)))))
(define dividea
```

```
 (lambda (l)
   (if (null? l) 0
      (if (= l 1) (list 1)
         (concatenar (list l)(dividea (quotient l 2)))))))
(define multiplicab
 (lambda (b l)
   (if (null? b) 0
      (if (= 0 (length l)) '()
         (concatenar (list b)(multiplicab (* 2 b)(cdr l)))))))
(define sumalistas
 (lambda (la lb)
   (if (null? la) 0
      (if (odd? (car la)) (+ (car lb) (sumalistas (cdr la)(cdr lb)))
         (sumalistas (cdr la) (cdr lb))))))
(define sumarusa
 (lambda (a b)
   (sumalistas (dividea a)(multiplicab b (dividea a)))))

; En UN SOLO define:
(define sumarusa
 (lambda (a b)
   (letrec
      ((
      dividea
      (lambda (l)
        (if (null? l) 0
           (if (= l 1) (list 1)
              (append (list l)(dividea (quotient l 2))))))
```

```
      ))
     (letrec
       ((multiplicab
         (lambda (b l)
          (if (null? b) 0
              (if (= 0 (length l)) '()
                  (append (list b)(multiplicab (* 2 b)(cdr l))))))
         ))
        (letrec
          ((sumalistas
            (lambda (la lb)
             (if (null? la) 0
                 (if (odd? (car la)) (+ (car lb) (sumalistas (cdr la)(cdr lb)))
                     (sumalistas (cdr la) (cdr lb)))))
            ))
   (sumalistas (dividea a)(multiplicab b (dividea a))))))))


;(sumarusa 2 3)
----------------------------------------CONVERTIR FORMATO---------------------------------------
(define validarboolean
  (lambda (l)
    (map (lambda (x)
          (if (equal? x "V") 1 0))
         l)))
(define validartext
  (lambda (l)
    (map (lambda (x)
          (string->number x))
```

```
        l)))
(define validardecimal
  (lambda (l)
    (map (lambda (x)
          (if (> x 0) x (- 0 x)))
        l)))
(define convertir
  (lambda (x)
    (if (null? x) '()
       (map (lambda (l)
                 (if (equal? (car l) "D") (validardecimal (car (cdr l)))
                    (if (equal? (car l) "T") (validartext (car (cdr l)))
                       (validarboolean (car (cdr l))))))
            x))))

----------BLACKJACK----------
(define stop-at
  (lambda (x)
    (pedir x (random 13))))

(define pedir
   (lambda (x r)
     (if (> r x) r
        (pedir x (+ r (random 13))))))

(define test-strategy
  (lambda (p1 p2 veces)
    (if (= 0 veces) 0
```

```
      (if (and (> (stop-at p1) (stop-at p2)) (< (stop-at p1) 22))(+ 1 (test-strategy p1 p2 (- veces 1)))
         (test-strategy p1 p2 (- veces 1))))))
```

----------PROFUNDIDAD----------

```
(define get-profundidad
  (lambda (x)
    (if (null? x)
        0
        (if (list? (car x))
           (max (+ 1 (get-profundidad (car x))) (get-profundidad (cdr x)))
           (get-profundidad (cdr x))
           )
        )
    )
  )
```

----------SUMA LISTAS----------

```
(define suma-lista
  (lambda (x y)
    (if (null? y)
        x
        (if (encuentra (car y) x)
           (suma-lista x (cdr y))
           (suma-lista (concat x (list (car y))) (cdr y))
           )
        )
    )
  )
```

```
----------RESTA LISTAS----------

(define resta-lista
  (lambda (x y)
    (if (null? x)
        '()
        (if (encuentra (car x) y)
            (resta-lista (cdr x) y)
            (concat (list (car x)) (resta-lista (cdr x) y))
            )
        )
    )
  )

----------ROTAR----------

(define rotar-izq
  (lambda (l)
    (concatenar (cdr l) (list (car l)))))
(define rotar-der
  (lambda (l)
    (concatenar (list (car (invierte l)))(invierte (cdr (invierte l))))))
(define rotar
  (lambda (x y)
    (if (equal? 'izquierda x)
        (rotar-izq y)
        (rotar-der y)
        )
```

```
   )
  )
```

```
----------PUZZLE----------
(define-struct intermedia (u d l r))
(define-struct final (l r))
(define i1 (make-intermedia 0 1 2 0))
(define i2 (make-intermedia 0 1 2 2))
(define i3 (make-intermedia 0 1 2 0))
(define f1 (make-final 0 2))
(define f2 (make-final 2 0))

----------SUMA HORAS----------

;; Contrato :sumahora: lista1 lista 2 con formato de hora: (0 30 0) > listahora
;; Proposito: sumar dos listas que representan horas
;; Definicion:

; caddr = (car (cdr (cdr)))

(define sumahora
  (lambda (listaH1 listaH2)
    (if (or (null? listaH1) (null? listaH2))
        "Error"
        (let ((hh (+ (car listaH1) (car listaH2)))
              (mm (+ (cadr listaH1) (cadr listaH2)))
              (ss (+ (caddr listaH1) (caddr listaH2)))
```

```
            )
          (list (+ hh (if (>
                          (+ mm (if (> ss 59) 1 0))
                          59) 1 0)
                    )
                (if (> (+ mm (if (> ss 59) 1 0)) 59)
                   (- (+ mm (if (> ss 59) 1 0)) 60)
                     (+ mm (if (> ss 59) 1 0))
                     )
                (if (> ss 59)
                   (- ss 60)
                   ss)
                )
          )
        )
      )
    )


(define demora
  (lambda (lista hora)
    (if (null? lista)
        '()
        (cons
         (list (caar lista) (sumahora hora (cadr lista)))
         (demora (cdr lista) (sumahora hora (cadr lista)))
         )
        )
```

```
   )
 )

(sumahora '(0 50 10) '(1 30 25))
----RECIBE UNA LISTA DE PARES Y LOS CAMBIA USANDOLOS COMO PARAMETRO USANDO MAP----
(define ListSust
 (lambda (ls lob)
   (if (null? ls)
      lob
      (ListSust (cdr ls) (map
                  (lambda (x)
                    (if (eqv? x (caar ls))
                       (cdr (car ls))
                       x)
                    )
                  lob)
            )
      )
   )
 )

(ListSust '((a . b) (b . c) (c . d)) '(a a a))
(ListSust '((a . b) (c . d)) '(a b c d))

-----------------------------------RECORRER ARBOLES BINARIOS EN INORDEN----------------------------------
; Ejemplo de USO
; (inorden '(Papi(HijoIzquierdo (A (E () ()) (F () ()))(B (g () ()) (h () ()))) (HijoDerecho (C () ()) (D () ()))))
```

```
; O para definir solamente el arbol.
; arbol? '(Papi(HijoIzquierdo () ()) (HijoDerecho () ()))
; Para tener en cuenta -> DEBE DE MANTERSE LA RELACIÓN DE LISTAS.
;Útiles

;Define el == y el !=

(define != (lambda (N1 N2) (not (= N1 N2))))
(define == (lambda (N1 N2) (= N1 N2)))

; Define si no es nulo
(define notnull? (lambda (E) (not (null? E))))

;Long -> Cantidad de elementos de una lista

(define long (lambda (L)
               (if (null? L) 0 (+ 1 (long (cdr L))))))
;Concatena Dos Listas
(define concatenar (lambda(L1 L2)
             (if(null? L1) L2  (cons (car L1) (concatenar (cdr L1)L2)))))

;Definición de los atributos del árbol

(define raiz (lambda (A) (car A)))

(define izq (lambda (A) (cadr A)))
```

```
(define der (lambda (A) (caddr A)))

(define hoja? (lambda (A) (if (and (null? (izq A)) (null? (der A))) #t #f)))

;Definición de Nodo

(define nodo? (lambda (C) (or (number? C) (symbol? C))))

;Definición del árbol

(define arbol? (lambda (A)
            (cond  ((null? A) #t)

----------BLACK JACK COMPLETO----------

;; The cards
 (define (deal) (+ 1 (random 10)))

;; The hands
 (define (make-hand up-card total)
  (cons up-card total))

(define (hand-up-card hand)
  (car hand))

(define (hand-total hand)
  (cdr hand))

(define (make-new-hand first-card)
```

```
 (make-hand first-card first-card))


(define (hand-add-card hand new-card)
 (make-hand (hand-up-card hand)
          (+ new-card (hand-total hand))))
;; Play hand loop
 (define (play-hand strategy my-hand opponent-up-card)
  (cond ((> (hand-total my-hand) 21) my-hand) ; I lose... give up
       ((strategy my-hand opponent-up-card) ; hit?
        (play-hand strategy
                (hand-add-card my-hand (deal))
                opponent-up-card))
       (else my-hand)))              ; stay
;; The Engine
 (define (twenty-one player-1-strategy player-2-strategy)
  (let ((player-2-initial-hand (make-new-hand (deal))))
    (let ((player-1-hand
          (play-hand player-1-strategy
                  (make-new-hand (deal))
                      (hand-up-card player-2-initial-hand))))
       (if (> (hand-total player-1-hand) 21)
          2
          (let ((player-2-hand
                  (play-hand player-2-strategy
                          player-2-initial-hand
                          (hand-up-card player-1-hand))))
            (cond ((> (hand-total player-2-hand) 21)
                   1)
```

```
                ((> (hand-total player-1-hand) (hand-total player-2-hand))
                  1)
                (else 2)))))))
;; hit?
;;
(define (hit? your-hand opponent-up-card)
  (newline)
  (display "Opponent up card ")
  (display opponent-up-card)
  (newline)
  (display "Your Total: ")
  (display (hand-total your-hand))
  (newline)
  (display "Hit? ")
  (eq? (read) 'y))
;;-----------------------Q1----------------------
(define stop-at
  (lambda (x)
          (lambda (hand op-up-card)
            (< (hand-total hand) x))))
;;-----------------------Q2----------------------
(define test-strategy
  (lambda (strategy-1 strategy-2 iterations)
    (if (= iterations 0)
        0
        (+ (if (eq? 1 (twenty-one strategy-1 strategy-2)) 1 0)
           (test-strategy strategy-1 strategy-2 (- iterations 1))))))
;;-----------------------Q3----------------------
```

```
(define watch-player
  (lambda (strategy)
    (lambda (hand op-up-card)
      (newline)
      (display "Opponent up card ")
      (display op-up-card)
      (newline)
      (display "Your Total: ")
      (display (hand-total hand))
      (newline)
      (display "Decision: ")
      (display (strategy hand op-up-card))
      (strategy hand op-up-card))))
;;-----------------------Q4-----------------------
(define tisona-strategy
      (lambda (my-hand opponent-up-card)
         (cond ((case-a my-hand opponent-up-card) #t)
               ((case-b my-hand opponent-up-card) #f)
               ((case-c my-hand opponent-up-card) (case-c-action opponent-up-card))
               ((case-d my-hand opponent-up-card) (case-d-action opponent-up-card))
               ((case-e my-hand opponent-up-card) (case-d-action opponent-up-card)))))

(define case-a
  (lambda (my-hand opponent-up-card)
    (< (cdr my-hand) 12)))

(define case-b
  (lambda (my-hand opponent-up-card)
```

```
    (> (cdr my-hand) 16)))

(define case-c
  (lambda (my-hand opponent-up-card)
    (= (cdr my-hand) 12)))

(define case-d
  (lambda (my-hand opponent-up-card)
    (= (cdr my-hand) 16)))

(define case-e
  (lambda (my-hand opponent-up-card)
    #t))

(define case-c-action
  (lambda (opponent-up-card)
    (< opponent-up-card 4)))

(define case-d-action
  (lambda (opponent-up-card)
    (not (= opponent-up-card 10))))

(define case-e-action
  (lambda (opponent-up-card)
    (> opponent-up-card 6)))
;;------------------------Q5-----------------------
(define majority
  (lambda (strategy-list)
```

```
   (lambda (my-hand opponent-up-card)
     (if (null? strategy-list)
         #f
         (majority-counter 0 0 strategy-list my-hand opponent-up-card)))))



(define majority-counter
  (lambda (hits stays strategy-list my-hand opponent-up-card)
    (if (null? strategy-list)
        (or (> hits stays) (= hits stays))
        (if ((car strategy-list) my-hand opponent-up-card)
            (majority-counter (+ hits 1) stays (cdr strategy-list) my-hand opponent-up-card)
            (majority-counter hits (+ stays 1) (cdr strategy-list) my-hand opponent-up-card)))))
```

```
;Concatena dos listas y las devuelve en una conjunta, append en scheme
(define concatenar (lambda(L1 L2)
                     (if(null? L1)
                       L2
                       (cons (car L1) (concatenar (cdr L1) L2))
                       )
                     )
  )
;Invierte la lista L, utiliza la funcion concatenar, pero tambien puede usar append
(define invertir (lambda (L)
                   (if (null? L)
                     '()
                     (concatenar (invertir (cdr L)) (list (car L)))
```

```
                )
            )
 )

;Predicado para saber si un elemento x está en la lista l
(define pertenece (lambda (x l)
            (if (null? l)
                #f
                (if (= (car l) x)
                    #t
                    (pertenece x (cdr l))
                    )
                )
            )
 )

;VALIDAR NUMERO
(define validar_numero(lambda (x rangomin rangomax)
                (if (not (number? x))
                    #f
                    (if (< x rangomin)
                        #f
                        (if (> x rangomax)
                            #f
                            #t
                            )
                        )
                    )
```

```
                    )
  )
;Ejemplos: hora => (validar_numero x 0 23)

;-----------------------------------------------------AGRUPAR-----------------------------------------------------
;Definir una función recursiva AGRUPAR que reciba dos argumentos, compruebe
;cuál de ellos es un átomo y cuál una lista, y a continuación introduzca el átomo
;junto a los átomos iguales que hubiera en la lista o al final de la misma, en el caso
;de no encontrar semejantes.
;(AGRUPAR '(A A A B B B C C C) 'B) ? (A A A B B B B C C C )

(define agrupar-aux(lambda (x y)
                (if (null? x)
                   (if (null? y)
                      '()
                      (list y)
                      )
                   (if (eqv? (car x) y)
                      (cons (car x) (agrupar-aux x null))
                      (cons (car x) (agrupar-aux (cdr x) y))
                   )
                )
  )
  )

(define AGRUPAR(lambda (x y)
              (if (list? x)
                 (if (list? y)
```

```
                "Error, x e y son listas!"
                (agrupar-aux x y)
                )
            (agrupar-aux y x)
            )
        )
 )
;-----------------------------------------------------------------------------------------------------

;Dada una lista, da su profundidad

(define prof (lambda (ls)
        (if (null? ls) ;Si es lista vacia, prof 0
            0
            (if (or (symbol? ls) (number? ls)) ;Si tiene 1 elemento y es numero o simbolo => es un
nodo
                1
                (if  (list? (car ls))
                    (+ 1(prof (car ls))) ;Si es una lista, sumamos 1 por su nivel + el nivel de lo que sigue
                    1 ;Si no es lista es porque no hay q recusrear mas
                    )
                )
        )
 )
 )
(define maxprof (lambda (ls)
            (if (> (prof (car ls)) (maxprof (cdr ls)))
                (prof (car ls))
```

```
                (maxprof (cdr ls)))
            )
  )

;-------------------------------------------------------------------------------------------------------

;Devuelve los elementos de una lista o lista de listas en una lista
;Similar al aplanar
(define recorrer (lambda (L)
            (if (null? L) '()
            (if (nodo? (car L)) (cons (car L) (recorrer (cdr L)))
               (append (recorrer (car L)) (recorrer (cdr L)))))))


;-------------------------------------------------------------------------------------------------------
;Full reverse list, invierte los elementos de una lista MULTINIVEL
(define fullreverse (lambda (L)
    (if (null? L) '()
       (if (list? (car L)) (append (fullreverse (cdr L)) (list (fullreverse (car L))))
                   (append (fullreverse (cdr L)) (list (car L)))))))

;-------------------------------------------------------------------------------------------------------



(define mod
  (lambda (x y)
    (if (not(= 0 y))
```

```
      (let ((z (/ x y))) (- x (* (floor z) y)))
      )
   )
 )

(define primo-aux( lambda (x n)
                (if (< (sqrt x) n)
                   #t
                   (if (= (mod x n) 0)
                      #f
                      (primo-aux x (+ 1 n))
                      )
                   )
                )
 )

(define primo? (lambda (x)
              (if (= x 1)
                 #t
                 (primo-aux x 2)
                 )
              )
 )
;-----------------------------------------------------------------------------------------------------------
;Contar caracteres y contar palabras... modificacion en el de caracteres para no contar espacios
(define o (open-input-file "test.txt"))

(define contar-palabras (lambda (o )
```

```
                    (let ((palabra (read o)))
                     (if (eof-object? palabra)
                        0
                        (+ 1 (contar-palabras o))
                        )
                     )
                    )
 )

(define contar-caracteres (lambda (o )
                    (let ((car (read-char o)))
                     (if (eof-object? car)
                        0
                        (if (not (or (eqv? car #\newline) (eqv? car #\space)))
                         (+ 1 (contar-caracteres o))
                         (contar-caracteres o)
                        )
                     )
                    )
 )
 )
;-------------------------------------------------------------------------------------------------------
(define distance2d
  (lambda (x y)
    (let ((d1 (- (car y) (car x)))
          (d2 (- (cdr y) (cdr x)))
        )
      (sqrt(+ (* d1 d1) (* d2 d2)))
```

```
    )
   )
  )

;-------------------------------------------------------------------------------------------------------

(define menorlista
  (lambda (l)
    (if (null? l)
        '()
        (if (= 1 (length l))
            (car l)
            (if (< (car l) (menorlista (cdr l)))
                (car l)
                (menorlista (cdr l))
                )
            )
        )
    )
  )
;-------------------------------------------------------------------------------------------------------
;Defina un procedimiento en Scheme llamado attach-at-end que reciba como
;parámetro un valor y una lista y retorne la lista con los mismos valores excepto el
;que se pasó por parámetro que se agregará al final.
;(attach-at-end 2 '(2 3 4 5 2 6)) ? (3 4 5 6 2)

(define delnumber(lambda (x l)
                   (if (null? l)
```

```
                '()
               (if (not(= x (car l)))
                  (cons (car l) (delnumber x (cdr l)))
                  (delnumber x (cdr l))
                  )
               )
            )
 )
(define attach-at-end(lambda (x l)
               (concatenar (delnumber x l) (list x))
               )
 )
;-----------------------------------------------------------------------------------------------------
;Defina un procedimiento en Scheme que reciba como parámetros los puntos de
;dos rectas y determine el punto de intersección de las mismas.
;Ecuación de la recta que pasa por dos puntos:
;(y – y1) / (y2 – y1) = (x – x1) / (x2 - x1)

(define intersection
 (lambda (p1 p2 p3 p4)
  ;p1 (x1,y1) ,p2 (x2,y2) p3 (x3,y3)  p4 (x4, y4)
  ;y=((y2-y1)/(x2-x1))*(x-x1)+y1
  ;y=((y4-y3)/(x4-x3))*(x-x3)+y3
  (if (or (= (- (car p1) (car p2)) 0) (= (- (car p3) (car p4)) 0))
     (display "Las rectas no se intersectan")
     (let* ((pen1 (/ (- (cdr p2) (cdr p1)) (- (car p2) (car p1)))) (pen2 (/ (- (cdr p4) (cdr p3)) (- (car p4) (car p3)))) (x (/ (- (+ (- (* pen1 (car p1)) (* pen2 (car p3))) (cdr p3)) (cdr p1)) (- pen1 pen2))) (y (+ (- (* pen1 x) (* pen1 (car p1))) (cdr p1)))) (cons x y))
     )
```

```
   )
 )
;-------------------------------------------------------------------------------------------------
;Funcion que devuelve pares de valores con el primero y la lista
;Se utiliza para generar la combinatoria 2 de las posibles combinaciones de la lista

(define pares(lambda (x l)
            (if (null? l)
                '()
                (if (= (length l) 1)
                    (cons (cons x (car l)) '())
                    (cons (cons x (car l)) (pares x (cdr l)))
                    )
                )
            )
 )

(define combinatoria2( lambda (l )
                  (if (< (length l) 2)
                      "No se puede hacer la combinatoria entre 2 elementos si tenes menos de 2"
                      (if (= (length l) 2)
                          (pares (car l) (cdr l))
                          (append (pares (car l) (cdr l)) (combinatoria2 (cdr l)))
                          )
                      )

                  )
 )
```

```
;(combinatoria2 '(1 2 3 4 5 6))

;-------------------------------------------------------------------------------------------------

; Invertir una lista sin usar append ni concatenar

(define (inverso L)
  (if (= (length L) 1)
      (car L)
     (cons (inverso (cdr L))  (car L) )
      )
  )

;(inverso '(1 2 3 4))
```

PROPIOS

```
(define concatenar
  (lambda (l1 l2)
    (if (null? l1)
        l2
        (cons (car l1) (concatenar (cdr l1) l2))
        )
    )
  )

(define invert
```

```
 (lambda (ls)
   (if (null? ls)
       '()
       (append (invert (cdr ls)) (list(car ls)))
       )
   )
 )

(invert '(1 2 3 4 5))
(concatenar '(1 2 3) '(4 5 6))

;
*************************************************************************
*******

; Orden ascendente:

(define borrar
 (lambda(X L)
   (if(null? L) '()
      (if (equal? X (car L))
          (borrar X (cdr L))
          (cons (car L) (borrar X (cdr L)))
          )
      )
   )
 )

(define minl ( lambda (L)
```

```
            (if (null? L) '()
            (if (null? (cdr L)) (car L)
            (if (<= (car L) (minl(cdr L))) (car L) (minl (cdr L)))))))

(define ordasc
 (lambda (ls)
   (if (null? ls)
       '()
       (if (= (car ls) (minl ls))
          (cons (car ls) (ordasc (cdr ls)))
          (cons (minl ls) (ordasc (borrar (minl ls) ls)))
          )
       )
   )
 )

(ordasc '(8 3 2 1 4 0))



;
*************************************************************************
*******
; ARBOLES

(define raiz  (lambda (A) (car A)))
(define izq  (lambda (A) (cadr A)))
(define dcho (lambda (A)(caddr A)))

(define nodo? (lambda (A) (or (number? A) (symbol? A))))
```

```
(define largo
  (lambda (ls)
    (if (null? ls)
        0
        (+ 1 (largo (cdr ls))))))

(define arbol?
  (lambda (A)
    (cond ((null? A) #t)
          ((not (= (largo A) 3)) #f)
          ((and (nodo? (raiz A)) (arbol? (izq A)) (arbol? (dcho A))))
          )
    )
  )

(arbol? '(a (b () ()) (c () ())))

;
**************************************************************************
*******

; Borra los elementos de la segunda lista que estan en la primera

(define pertenece
  (lambda (X L)
    (if (null? L)
        #f
        (if (equal? (car L) X)
```

```
          #t
          (pertenece X (cdr L))))))



(define borra2en1
  (lambda (L1 L2)
    (if (null? L2)
        '()
        (if (pertenece (car L2) L1)
            (borra2en1 L1 (cdr L2))
            (cons (car L2) (borra2en1 L1 (cdr L2)))
            )
        )
    )
  )

(borra2en1 '(1 2 3 4 5) '(8 7 6 5 4))


;
*************************************************************************
*******


; Pertenece Mutinivel


(define perteneceMN
  (lambda (x ls)
    (if (null? ls)
        #f
        (if (list? (car ls))
```

```
        (or (perteneceMN x (car ls)) (perteneceMN x (cdr ls)))
        (if (eqv? x (car ls))
           #t
           (perteneceMN x (cdr ls))
           )
        )

      )
   )
 )



(perteneceMN 1 '(85 (3) 2 (j 1) (4 a 1) 5 8 (5)))


;
*************************************************************************
*******

; Lista los atomos de un arbol o grafo (aplana una lista)

(define aplanar
  (lambda (ls)
    (if (null? ls)
       '()
       (if (list? (car ls))
          (concatenar (car ls) (aplanar (cdr ls)))
          (concatenar (list (car ls)) (aplanar (cdr ls)))
          )
       )
```

```
   )
 )

(aplanar '(-3 (1 2 0) -4 (5) 6))

;
*************************************************************************
*******

; Devuelve una lista con los nodos de un grafo dirigido

(define aplanar
  (lambda (ls)
    (if (null? ls)
        '()
        (if (list? (car ls))
            (append (car ls) (aplanar (cdr ls)))
            (append (list (car ls)) (aplanar (cdr ls)))
            )       )    )  )

(define pertenece
  (lambda (X L)
    (if (null? L)
        #f
        (if (equal? (car L) X)
            #t
            (pertenece X (cdr L))))))

(define eliminaR
```

```
 (lambda (L)
   (if (null? L)
       '()
       (if (not (pertenece (car L) (cdr L)))
          (cons (car L) (eliminaR (cdr L)))
          (eliminaR (cdr L))))))

(define nodos
 (lambda (G)
   (if (null? G)
       '()
       (eliminaR (aplanar G))
       )
   )
 )

(nodos '((a b) (b d) (d a) (a c) (c b)))

;
*************************************************************************
*******

; Determina si un arbol es completo

(define raiz  (lambda (A) (car A)))
(define izq  (lambda (A) (cadr A)))
(define dcho (lambda (A)(caddr A)))

(define nodo? (lambda (A) (or (number? A) (symbol? A))))
```

```
(define largo
  (lambda (ls)
    (if (null? ls)
        0
        (+ 1 (largo (cdr ls))))))


(define adn0?
  (lambda (A)
    (cond ((null? A) #t)
          ((and (null? (izq A))(null? (dcho A))) #t)
          (else #f))))

(define nivel
  (lambda (A)
    (if (adn0? A)
        0
        (+ 1 (max (nivel (izq A)) (nivel (dcho A)))))))

(define completo?
  (lambda (A)
    (cond ((null? A) #t)
          ((not (= (largo A) 3)) #f)
          ((not (= (nivel (izq A)) (nivel (dcho A)))) #f)
          (else (and (nodo? (raiz A)) (completo? (izq A)) (completo? (dcho A))))
          )
    )
```

```
 )


(completo? '(a (b (d () ()) ()) (c () ())))


;
*************************************************************************
*******
; Determina si x es consecutivo a y en ls, funciona en orden descendente y ascendente de la lista.


(define consec
 (lambda (x y ls)
  (if (null? ls)
      #f
      (cond ((and (eqv? x (car ls)) (not (null? (cdr ls))))
            (if (eqv? y (cadr ls))
               #t
               (consec x y (cdr ls))))
           ((eqv? y (car ls))
            (if (eqv? x (cadr ls))
               #t
               (consec x y (cdr ls))))
           (else (consec x y (cdr ls)))
           )
      )
  )
 )


(consec 1 2 '(5 4 1 2 0 1))
```

```
;
*************************************************************************
*******

; Dada una lista de listas, devuelve una lista con las longitudes de c/u de los elementos

; NO usar length, ej: (longitudes '((1 2 3) (1 2) (1))) -> (3 2 1)


(define largo
  (lambda (ls)
    (if (null? ls)
        0
        (+ 1 (largo (cdr ls))))))


(define longitudes
  (lambda (ls)
    (if (null? ls)
        '()
        (cons (largo (car ls)) (longitudes (cdr ls)))
        )
    )
  )



(longitudes '((1 2 3) (1 0 2) (1)))


;
*************************************************************************
*******

; Dada una lista de pares, los devuelve ordenados de menor a mayor
```

```
(define comp_inv
 (lambda (par)
  (if (null? par)
      '()
      (if (< (car par) (cdr par))
          par
          (cons (cdr par) (car par))
          )
      )
  )
 )

(define order-pair
 (lambda (ls)
  (if (null? ls)
      '()
      (cons (comp_inv (car ls)) (order-pair (cdr ls)))
      )
  )
 )

(order-pair '((1 . 2)(3 . 1)(4 . 5)(5 . 1)))

;
*************************************************************************
*******
; Recibe una lista y devuelve otra con un par para cada elemento con la cantidad de apariciones
; Ej. (itemcount '(a b a b c c c d)) -> ((a . 2)(b . 2)(c . 3)(d . 1))
```

```
(define contar
  (lambda (x ls)
    (if (null? ls)
        0
        (if (eqv? x (car ls))
            (+ 1 (contar x (cdr ls)))
            (contar x (cdr ls))
            )
        )
    )
  )

(define borrar
  (lambda(X L)
    (if(null? L)
       '()
       (if (equal? X (car L))
           (borrar X (cdr L))
           (cons (car L) (borrar X (cdr L)))
           )
       )
    )
  )

(define itemcount
  (lambda (ls)
    (if (null? ls)
        '()
```

```
      (append (list(cons (car ls) (contar (car ls) ls))) (itemcount (borrar (car ls) ls)))
      )
   )
 )

(itemcount '(a b a b c c c d))

;
*************************************************************************
*******

(define sublista
 (lambda (sl L)
   (if (and (null? sl)(null? L))
      #t
      (if (null? L)
         #f
         (if (not (list? sl))
            #f
            (if (null? sl)
               #t
               (if (eqv? (car sl) (car L))
                  (sublista (cdr sl) (cdr L))
                  (sublista sl (cdr L))
                  )
               )
            )
         )
      )
```

```
   )
 )

(sublista '(1 2 3) '(1 5 9 5 1 2 3 5))

;
*************************************************************************
*******
; Largo multinivel

(define largoMN
 (lambda (ls)
   (if (null? ls)
       0
       (if (list? (car ls))
          (+ (largoMN(car ls)) (largoMN(cdr ls)))
          (+ (largoMN(cdr ls)) 1)
          )
       )
   )
 )

(largoMN '(1 2 (3 4) 5 (6)))

;
*************************************************************************
*******
; Borra el elemento x multinivel
```

```
(define borraxMN
 (lambda (x ls)
  (if (null? ls)
      '()
      (if (list? (car ls))
         (cons (borraxMN x (car ls))(borraxMN x (cdr ls)))
         (if (eqv? x (car ls))
            (borraxMN x (cdr ls))
            (cons (car ls) (borraxMN x (cdr ls)))
            )
         )
      )
  )
 )

(borraxMN 5 '((4 5) (2 3 4 5) (5) (6) 0))

;
*************************************************************************
*******
;Definir una función recursiva AGRUPAR que reciba dos argumentos, compruebe
;cuál de ellos es un átomo y cuál una lista, y a continuación introduzca el átomo
;junto a los átomos iguales que hubiera en la lista o al final de la misma, en el caso
;de no encontrar semejantes.
;(AGRUPAR '(A A A B B B C C C) 'B) ? (A A A B B B B C C C )

(define agrupar
 (lambda (x1 x2)
  (if (and (list? x1) (not (list? x2)))
```

```
      (agruparaux x1 x2)
      (if (and (list? x2) (not (list? x1)))
         (agruparaux x2 x1)
         "Alguno de los dos parametros debe ser una lista"
         )
      )
   )
 )

(define agruparaux
 (lambda (L x)
   (if (null? L)
      (list x)
      (if (eqv? (car L) x)
         (cons x L)
         (cons (car L) (agruparaux (cdr L) x))
         )
      )
   )
 )

(agrupar '(A A A B B C D D) B)

;
*************************************************************************
*******

; Sublista (multinivel, diferente a la respuesta dada en el parcial V2.0)
```

```
(define subl
  (lambda (sl L)
    (if (and (null? L)(not (null? sl)))
        #f
        (if (and (null? sl) (null? L))
            #t
            (if (list? (car L))
                (sublMN sl L)
                (if (equal? (car sl) (car L))
                    (subl (cdr sl) (cdr L))
                    (subl sl (cdr L))
                    )
                )
            )
        )
    )
  )

(define sublMN
  (lambda (sl L)
    (if (null? L)
        #f
        (if (list? (car L))
            (if (equal? sl (car L))
                #t
                (or (sublMN sl (car L)) (sublMN sl (cdr L)))
                )
            (sublMN sl (cdr L))
```

```
        )
      )
   )
 )

(subl '(1 2 3) '(2 4 5 1 2 3)) ;--> #T
(subl '(1 2 3) '(2 4 5 (1 2 3) 7 8 9)); --> #T
(subl '(1 2 3) '(2 4 5 (1 2) 3 4)) ;--> #F
(subl '(1 2 3) '(2 4 5 ( 1 5 (1 2 3)) 8)) ;--> #T

;
*************************************************************************
*******

; Profundidad (multinivel, no funciona correctamente, suma uno de menos)

(define nodo? (lambda (C) (or (number? C) (symbol? C))))

(define prof
  (lambda (ls)
    (if (null? ls)
        0
        (if (nodo? ls)
            0
            (if (list? (car ls))
                (+ 1 (prof (car ls)) (prof (cdr ls)))
                (prof (cdr ls))
                )
            )
```

```
      )
   )
  )

(define maxprof
  (lambda (ls)
    (if (null? ls)
        0
        (if (> (prof (car ls)) (maxprof (cdr ls)))
            (prof (car ls))
            (maxprof (cdr ls))
            )
        )
    )
  )

(maxprof '((1 (2)) (((5 7))) 4))
```

```
; FUNCIONES LOGICAS Y MATEMATICAS

; Distinto
; Ej: (!= 3 3) => #f        (!= 3 6) => #t

(define != (lambda (N1 N2) (not (= N1 N2))))

; Ej: (notnull? 4) => #t    (notnull? '()) => #f
```

```
(define notnull? (lambda (E) (not (null? E))))

; Incrementa 1
; Ej: (++ 1) => 2

(define ++ (lambda (N) (+ N 1)))

; Decrementa 1
; Ej: (-- 1) => 0

(define -- (lambda (N) (- N 1)))

; Par (se asume que el 0 es numero par)
; Ej: (par? 4) => #t          (par? 7) => #f

(define par? (lambda (N)
            (if (< N 0) (parAux (* N -1)) (parAux N))))

(define parAux (lambda (N)
              (if (= N 0) #t (impar? (- N 1)))))

; Impar
; Ej: (impar? 5) => #t

(define impar? (lambda (N)
              (if (< N 0) (imparAux (* N -1)) (imparAux N))))

(define imparAux (lambda (N)
```

```
            (if (= N 0) #f (par? (- N 1)))))

; Potencia
; Ej: (potencia 2 8) => 256

(define potencia (lambda (B E)
            (cond ((= E 0) 1)
                ((= E 1) B)
                (else (* B (potencia B (- E 1)))))))

; Calcular factorial (2 versiones)
; Ej: (factorial 6) => 720

(define factorial (lambda (N) (if (= N 0) 1 (* N (fact1 (- N 1) )))))


; TRABAJOS CON LISTA - GENERALIDADES

; Devuelve el numero de elementos que compone una lista.
; Ej: (long '(1 2 3 4)) => 4          (long '(1 2 (3 4))) => 3

(define long (lambda (L)
            (if (null? L) 0 (+ 1 (long (cdr L))))))

; Devuelve el numero de elementos (atomos) que compone una lista. Multinivel.
; Ej: (longM '(1 2 3 4)) => 4          (longM '(1 2 (3 4))) => 4

(define longM (lambda (L)
```

```
            (long (listaAtomos L))))


; Devuelve el maximo elemento de una lista.
; Ej: (maxL '(1 2 3 4 5 6)) => 6

(define maxL (lambda (L)
          (if (null? L) '()
          (if (null? (cdr L)) (car L)
                    (if (>= (car L) (maxl(cdr L))) (car L)
                                  (maxl(cdr L)))))))

 ; Devuelve el maximo elemento de una lista. Multinivel.
; Ej: (maxM '(1 2 3 (4 5 6) (7 8) 4 (10 2))) => 10

(define maxM (lambda (L)
        (if (null? L) '()
           (if (null? (cdr L))
              (if (list? (car L)) (maxM (car L)) (car L))
              (if (> (if (list? (car L)) (maxM (car L)) (car L)) (maxM (cdr L)))
                 (if (list? (car L)) (maxM (car L)) (car L))
                 (maxM (cdr L)))))))

; Devuelve el menor elemento de una lista.
; Ej: (minL '(1 2 3 4 5 6)) => 1

(define minL ( lambda (L)
          (if (null? L) '()
```

```
            (if (null? (cdr L)) (car L)
            (if (<= (car L) (minl(cdr L))) (car L) (minl (cdr L)))))))

; Devuelve el menor elemento de una lista. Multinivel.
; Ej: (minM '(1 2 3 (4 5 6) (7 8) 4 (10 2 -2))) => -2

(define minM (lambda (L)
          (if (null? L) '()
             (if (null? (cdr L))
                (if (list? (car L)) (minM (car L)) (car L))
                (if (< (if (list? (car L)) (minM (car L)) (car L)) (minM (cdr L)))
                   (if (list? (car L)) (minM (car L)) (car L))
                   (minM (cdr L)))))))

; Determina si una lista es estrictamente creciente.
; Ej: (crece '(1 2 3 4 5 6 7)) => #t

(define crece (lambda (L)
     (if (null? (cdr L)) #t
        (if (< (car L) (cadr L)) (crece (cdr L)) #f))))

; Determina si una lista es estrictamente decreciente.
; Ej: (decrece '(7 6 5 4 3)) => #t

(define decrece (lambda (L)
     (if (null? (cdr L)) #t
        (if (> (car L) (cadr L)) (decrece (cdr L)) #f))))
```

```
; Verifica si una lista es capicua
; Ej: (capicua '(1 3 5 (2 4 6 (8) 6 4 2) 5 3 1)) => #t
;     (capicua '(1 3 5 (2 4 6 (8) 7 4 2) 5 3 1)) => #f

(define capicua (lambda (L)
    (if (null? L) #t
       (if (equal? L (invertirM L)) #t #f))))

; L1 es prefijo de L2?, no pueden estar ambas vacias
; Ej: (prefijo '(1 2) '(1 2 3 4 5)) => #t

(define prefijo (lambda (L1 L2)
(if (null? L2) #f (if (or (null? L1) (equal? L1 L2)) #t (prefijo L1 (qu L2))))))

; Version 2

(define prefijo2 (lambda(L1 L2)
(if (equal? L1 L2) #t (if (< (long L2) (long L1)) #f (prefijo2 L1 (qu L2))))))



; L1 es posfijo de L2?
; Ej: (posfijo '(7 6) '(1 2 6 5 7 6)) => #t

(define posfijo (lambda (L1 L2)
            (if (null? L2) #f
               (if (equal? L1 L2) #t (posfijo L1 (cdr L2))))))

; Version 2
```

```
(define posfijo2 (lambda(L1 L2)
                (if (or (null? L1) (equal? L1 L2)) #t
                   (if (< (long L2) (long L1)) #f (posfijo2 L1 (cdr L2))))))

; sublista!! L1 de L2, no pueden estar ambas vacias
; Ej: (subLista '(1 2 3) '(3 5 1 2 3 7 8)) => #t

(define subLista (lambda (L1 L2)
    (if (null? L2) #f
       (if (prefijo L1 L2) #t
          (sublista L1 (cdr L2))))))

; subListaM!! L1 es sublista de L2? Multinivel
; Ej: (subListaM '(3 5 7) '(1 2 3 (7 6 (3 5 7 1) (2 3)) 2)) => #t

(define subListaM (lambda (L1 L2)
  (if (not (list? L2)) #f
  (if (equal? L1 L2) #t
  (if (null? L2) #f
  (or (subListaM L1 (cdr L2)) (subListaM L1 (qu L2)) (subListaM L1 (car L2))) )))))

; Determina si los valores X e Y son consecutivo en L
; Ej: (consecutivo '(1 2 3 4 5 6 7 8) 6 7) => #t

(define consecutivo (lambda (L X Y)
                   (sublista (list X Y) L)))
```

```
; Consecutivos Multinivel.
; Ej: (consecutivoM '(1 2 3 (4 (5 6)) 7 8) 5 6) => #t

(define consecutivoM (lambda (L X Y)
                    (subListaM (list X Y) L)))

;  BUSQUEDA DE ELEMENTOS

; Obtener el ultimo elemento de una lista.
; Ej: (ultimoElem '(1 2 3 4 5 6)) => 6

(define ultimoElem( lambda (L)
      (if (null? L) '() (if (equal? (cdr L) '()) (car L) (ultimoElem (cdr L))))))

; Devuelve el enesimo elemento de una lista.
; Ej: (enesimoElem 10 '(7 8 9 10)) => 0            (enesimoElem 2 '(7 8 9 10)) => 8

(define enesimoElem(lambda(N L)
            (if (< (long L) N) 0
            (if (equal? N 1) (car L)
               (enesimoElem (- N 1) (cdr L))))))

; Devuelve el numero de posicion de la primera ocurrencia de X.
; Ej: (xPosicion 2 '(2 3 4 5)) => 1

(define xPosicion (lambda (X L)
       (if (or (null? L) (not (pertenece X L))) 0
          (xBusca X L))))
```

```
(define xBusca (lambda (X L)
        (if (null? L) 0
           (if (equal? (car L) X) 1
              (+ 1 (xBusca X (cdr L)))))))

; Dado un valor X, busca en una lista L (simple) para determinar si esta o no en ella.
; Ej: (pertenece 3 '(1 2 3 4 5 6)) => #t

(define pertenece (lambda (X L)
(if (null? L) #f (if (equal? (car L) X) #t (pertenece X (cdr L))))))

; Dado un valor X, busca en una lista L para determinar si esta o no en ella. Multinivel.
; Ej: (perteneceM 3 '(1 (2 3) 4 (5 6))) => #t

(define perteneceM (lambda (X L)
              (if (null? L) #f
                 (if (list? (car L)) (perteneceM X (car L))
                    (if (equal? X (car L)) #t
                       (perteneceM X (cdr L)))))))


; ELIMINACION O REEMPLAZOS DE LOS ELEMENTOS EN UNA LISTA

; No elimina el n-esimo elemento, si sus predecesores.
; Ej: (sacaNpri 3 '(1 1 1 3 4 5)) => (3 4 5)

(define sacaNpri (lambda (N L)
```

```
        (if (= 0 N) L
           (sacaNpri (- N 1) (cdr L)))))

; Elimina el elemento X de la lista en el 1er nivel.
; Ej: (eliminaX 3 '(3 4 5 (3 7) 3)) => (4 5 (3 7))

(define eliminaX(lambda(X L)
         (if(null? L) '()
         (if (equal? X (car L))
            (eliminaX X (cdr L))
            (cons (car L) (eliminaX X (cdr L)))))))

; Elimina el elemento X de la lista en todos los niveles
; Ej: (eliminaMx '((1 2 3) 2 7 (1 3 (6 2 8) 0 2)) 2) => ((1 3) 7 (1 3 (6 8) 0))

(define eliminaMx (lambda (L X)
      (if (null? L) '()
         (if (equal? (car L) X) (eliminaMx (cdr L) X)
            (if (list? (car L)) (cons (eliminaMx (car L) X) (eliminaMx (cdr L) X))
               (cons (car L) (eliminaMx (cdr L) X)))))))

; Elimina todos los elementos de la lista 2 que estan en la 1
; Ej: (elim12 '(2 1) '((1 2 3) 2 7 (1 3 (6 2 8) 0 2))) => ((3) 7 (3 (6 8) 0))

(define elim12 (lambda (L1 L2)
       (if (null? L1) L2
          (elim12 (cdr L1) (eliminaMx L2 (car L1)) ))))
```

```
; Devuelve la lista sin elementos repetidos. Elimina las primeras ocurrencias.
; (eliminaR '(3 5 3 1 9 2 3 8 9 3)) => (5 1 2 8 9 3)

(define eliminaR (lambda (L)
                (if (null? L) '()
                   (if (not (pertenece (car L) (cdr L)))
                      (cons (car L) (eliminaR (cdr L)))
                      (eliminaR (cdr L))))))

; Devuelve la lista sin elementos repetidos. Deja las primeras ocurrencias.
; (eliminaR2 '(3 5 3 1 9 2 3 8 9 3)) => (3 5 1 9 2 8)

(define eliminaR2 (lambda (L)
                (if (null? L) '() (invertir (eliminaR (invertir L))))))


; Elimina el primer elemento X que aparece en la lista.
; Ej: (eliminarPri 2 '(4 2 1 2 1)) => (4 1 2 1)

(define eliminarPri (lambda (X L)
     (if (null? L) '()
        (if (equal? (car L) X) (cdr L)
           (cons (car L) (eliminarPri X (cdr L)))))))

; Elimina el elemento que se encuentra en la enesima posicion.
; Ej: (xElimina '(4 8 91 2 7 5) 3) => (4 8 2 7 5)

(define xElimina (lambda (L N)
```

```
            (if (or (null? L) (> N (long L) )) #f
            (if (= N 1)  (cdr L)
               (cons (car L) (xElimina (cdr L) (- N 1)))))))

; Reemplazar X por Y en L
; Ej: (xReemplazar '(p o t e n c i a) 't 'n)=> (p o n e n c i a)

(define xReemplazar (lambda (L X Y)
(if (null? L) '()
        (if (equal? (car L) X) (cons Y (xReemplazar (cdr L) X Y))
        (cons (car L) (xReemplazar (cdr L) X Y))))))

; Reemplazar X por Y en L (multinivel)
; Ej: (xReemplazarM 2 3 '( 2 1 ( 1 2 (2 2 6 2 2) 3 1 3) 2 5)) => (3 1 (1 3 (3 3 6 3 3) 3 1 3) 3 5)

(define xReemplazarM (lambda (X Y L)
            (if (null? L) '()
            (if (list? (car L)) (cons (xReemplazarM X Y (car L)) (xReemplazarM X Y (cdr L)))
               (if (equal? X (car L))
                  (cons Y (xReemplazarM X Y (cdr L)))
                  (cons (car L) (xReemplazarM X Y (cdr L))))))))

;EJERCICIO 29: Sustituir en L P1 por P2 (P1 y P2 son listas, patrones).
; Ej: (sust '(1 2 3 4) '(1 2) '(4)) => (4 3 4)

(define sust (lambda(L P1 P2)
      (if (null? L) '()
      (if (prefijo P1 L)
```

```
        (concatenar P2 (sust (aPartirN (long P1) L) P1 P2))
        (cons (car L) (sust (cdr L) P1 P2))))))

; Quita el ultimoElem elemento de una L, si es vacia retorna '()
; Ej: (qu '(1 2 3 4 5)) => (1 2 3 4)

(define qu (lambda  (L)
(if (null? L) '() (if (equal? '() (cdr L)) '() (cons (car L) (qu (cdr L)))))))

; Construye una lista con el primer y ultimoElem elemento de una lista.
; Ej: (leex '(1 2 3 4 5)) => (1 5)

(define leex (lambda (L) (list  (car L) (ultimoElem L) )))

; Quita los extremos de una lista.
; Ej: (quitaex '(1 2 3 4 5)) => (2 3 4)

(define quitaex (lambda (L)
            (if (> (long L) 1) (qu (cdr L)) '())))

; Reemplaza el elemento en la posicion N por X.
; Ej: (insertarXenN '(1 32 7 4) 2 5) => (1 5 7 4)

(define insertarXenN(lambda(L N X)
      (if (or (null? L) (= N 0)) L
      (if (= N 1) (cons X (cdr L))
         (cons (car L) (insertarXenN (cdr L) (- N 1) X))))))
```

```
; CREACION DE LISTAS

; Dada dos listas, L1 y L2, las concatena.
; Ej: (concatenar '(1 2) '(3 4 5)) => (1 2 3 4 5)

(define concatenar (lambda(L1 L2)
            (if(null? L1) L2  (cons (car L1) (concatenar (cdr L1)L2)))))

; Concatenar un elemento a una lista.
; Ej: (concatenarElem '(1 2 3) 4) => (1 2 3 4)
;     (concatenarElem '(1 2 3) '(4)) => (1 2 3 (4))

(define concatenarElem (lambda (L X)
      (if (null? L) (list X) (invertir (cons X (invertir L) )))))

; Version ampliada de concatenar en la que los elementos no deben ser necesariamente listas.
; Ej: (concatenar2 '(2 3) 9) => (2 3 9)
;     (concatenar2 '(1 2 3) '(4)) => (1 2 3 4)

(define concatenar2 (lambda(E1 E2)
      (if (and (list? E1) (list? E2)) (concatenar E1 E2)
      (if (and (not (list? E1)) (not (list? E2))) (concatenar (list E1) (list E2))
      (if (list? E1) (concatenar E1 (list E2))
            (concatenar (list E1) E2))))))

; Devuelve todas las sublistas posibles de una lista dada.
```

```
; Ej: (sublistas '(1 (2 3) 4 5)) => ((1) (1 (2 3)) (1 (2 3) 4) (1 (2 3) 4 5) ((2 3)) ((2 3) 4) ((2 3) 4 5) (4) (4 5) (5))

;    (sublistas '(1 2 3))      => ((1) (1 2) (1 2 3) (2) (2 3) (3))


(define sublistas (lambda (L)
                  (if (null? L) '()
                     (concatenar (sublistasAux L 1) (sublistas (cdr L))))))


(define sublistasAux (lambda (L N)
                     (if (> N (long L)) '()
                        (cons  (nPrimeros N L) (sublistasAux L (+ N 1))))))


; Devuelve una lista con los elementos atomicos.

; Ej: (listaAtomos '(3 4 (6 8) (2 (10 11 ) 19 ) -1 -2 (-3 ))) => (3 4 6 8 2 10 11 19 -1 -2 -3)


(define listaAtomos (lambda (L)
                    (if (null? L) '()
                    (if (nodo? (car L)) (cons (car L) (listaAtomos (cdr L)))
                       (concatenar (listaAtomos (car L)) (listaAtomos (cdr L)))))))


; Devuelve los N primeros elementos de L

; Ej: (nPrimeros 4 '(6 5 1 2 4 7)) => (6 5 1 2)


(define nPrimeros(lambda (N L)
           (if (= N 0) '()
              (cons (car L) (nPrimeros (- N 1) (cdr L))))))


; Devuelve todos los resultados posibles de prefijos.
```

```
; Ej: (nPrimerosT '(1 2 3 4 5)) => ((1) (1 2) (1 2 3) (1 2 3 4) (1 2 3 4 5))

(define nPrimerosT (lambda (L)
      (if (null? L) '()
          (nPrimerosTaux L 1))))

(define nPrimerosTAux (lambda (L N)
      (if (= N (long L)) (list L)
         (cons (nPrimeros N L) (nPrimerosTAux L (+ N 1))))))

; Devuelve los N ultimos elementos de L
; Ej: (nUltimos '(71 82 23 14 25 6) 3) => (14 25 6)

(define nUltimos (lambda (L N)
               (if (or (null? L) (> N (long L) )) #f
               (cond ((= N 1) (cdr L))
                   ((= N 0) L)
                   (else (nUltimos (cdr L) (- N 1)))))))

;Devuelve a partir de n (sin tomar el elemento de la posición n)
; Ej: (aPartirN 3 '(7 6 5 4 3 2)) => (4 3 2)

(define aPartirN (lambda(N L)
      (if (= N 0) L
         (aPartirN (- N 1) (cdr L)))))

; Menores que N
; Ej: (xMenores '(1 8 3 4 5 2 7) 4) => (1 2 3)
```

```
(define xMenores (lambda (L N) (xMenoresAux (ordenar<= L) N)))

(define xMenoresAux (lambda (L N)
            (if (null? L) '()
               (if (< (ultimoElem L) N) L
                  (xMenoresAux (qu L) N)))))

; Mayores que N
; Ej: (xMayores '(1 8 3 4 5 2 7) 4) => (5 6 7)

(define xMayores (lambda (L N) (xMayoresAux (ordenar<= L) N)))

(define xMayoresAux (lambda (L N)
                  (if (null? L) '()
                     (if (> (car L) N) L
                        (xMayoresAux (cdr L) N)))))

; Devuelve una lista con los anteriores elementos a X
; Ej: (ant 5 '(1 2 3 4 5 6 7 8 9)) => (1 2 3 4)

(define ant (lambda (X L)
   (if (null? L) '()
      (if (equal? (car L) X) '()
         (cons (car L) (ant X (cdr L)))))))

; Devuelve una lista con los siguientes elementos a X
; Ej: (sig 5 '(1 2 3 4 5 6 7 8 9)) => (6 7 8 9)
```

```
(define sig (lambda (X L)
   (if (null? L) '()
      (if (equal? (car L) X) (cdr L)
         (sig X (cdr L))))))
```

; Devuelve una lista conteniendo dos listas, una con los elementos anteriores a X y otra con los siguientes a X.

; Ej: (izq_der 5 '(1 2 3 4 5 6 7 8 9)) => ((1 2 3 4) (6 7 8 9))

```
(define izq_der (lambda (X L)
   (if (null? L) '()
      (cons (ant X L) (list (sig X L))))))
```

; Devuelve la lista L invertida.

; Ej: (invertir '(1 2 3 4 5 6)) => (6 5 4 3 2 1)

; sin primitivas

```
(define invertir (lambda (L)
      (if (null? L) '() (cons (enesimoElem (long L) L) (invertir (qu L))))))
```

; con primitivas

```
(define invertir (lambda (L)
     (if (null? L) '()  (append (invertir (cdr L)) (list (car L))))))
```

; Invierte una lista en todos sus niveles.

; Ej: (invertirM '(1 2 (3 4 (5 6) 7) 8 9 (10 11 (12 13)))) => (((13 12) 11 10) 9 8 (7 (6 5) 4 3) 2 1)

```
; sin primitivas

(define invertirM (lambda (L)

 (if (null? L) '()

        (if (list? (enesimoElem (long L) L))

           (cons (invertirM (enesimoElem (long L) L)) (invertirM (qu L)))

           (cons (enesimoElem (long L) L) (invertirM (qu L)))))))


; con primitivas

(define invertirM (lambda (L)

     (if (null? L) '()

        (if (list? (car L)) (append (invertirM (cdr L)) (list (invertirM (car L))))

                        (append (invertirM (cdr L)) (list (carL)))))))


;  (invertirM '(1 2 3 (4 5 6) 9)) => (9 (6 5 4) 3 2 1)

;  (invertirM '(a(b(c(d(e)))))) => (((((e) d) c) b) a)


; Devuelve una lista conteniendo dos sublistas, una con elementos menores que N y otra con los mayores a N

; Ej: (xMayMen '(1 8 3 4 5 2 7) 4) => ((1 2 3) (5 6 7))


(define xMayMen (lambda (L N) (list (xMenores L N) (xMayores L N))))


; Arma una lista con todas las posiciones de X en L

; Ej: (ocurrencias '(1 3 2 4 65 7 3) 3) => (2 7)


(define ocurrencias (lambda (L X) (ocurrenciasAux L X 1)))


(define ocurrenciasAux (lambda (L X A)
```

```
(if (null? L) '()
        (if (= (car L) X)  (cons A  (ocurrenciasAux (cdr L) X (+ A 1)))
                    (ocurrenciasAux (cdr L) X (+ A 1))))))

; Mover atras N elementos de L
; Ej: (moverAtras '(1 2 3 4 5 6) 3) => (4 5 6 1 2 3)

(define moverAtras(lambda(L N)
            (if(or(equal? N 0) (null? L)) L
              (moverAtras(concatenar(cdr L)(list(car L))) (- N 1)))))

; Mover adelante N elementos de L
; Ej: (moverAdelante '(1 2 3 4 5 6) 3) => (4 5 6 1 2 3)

(define moverAdelante(lambda(L N)
(if (or (equal? N 0) (null? L)) L
        (moverAdelante (concatenar (list (ultimoElem L)) (qu L)) (- N 1)))))


; MATEMATICAS CON LISTAS

; Suma los elementos de una lista.
; Ej: (sumaElem '(1 2 3 4 5 6 7)) => 28

(define sumaElem (lambda (L)
            (if (null? L) 0 (+ (car L) (sumaElem (cdr L))))))

; Suma los elementos de una lista. Multinivel.
```

```
; Ej: (sumaElemM '(1 2 (3 4) 5 (6 7) 0)) => 28


(define sumaElemM (lambda (L)
            (if (null? L) 0 (sumaElem (listaAtomos L)))))


 ; Suma los elementos respectivos de dos listas, generando otra con los resultados. Ambas listas de igual tamano.
; Ej: (sumaListas '(1 7 4) '(9 3 6)) => (10 10 10)


(define sumaListas (lambda (L1 L2)
    (if (null? L1) '()
      (cons (+ (car L1) (car L2)) (sumaListas (cdr L1) (cdr L2))))))


; Resta los elementos respectivos de dos listas, generando otra con los resultados. Ambas listas de igual tamano.
; Ej: (restaListas '(1 7 4) '(9 3 6)) => (-8 4 -2)


(define restaListas (lambda (L1 L2)
   (if (null? L1) '()
      (cons (- (car L1) (car L2)) (restaListas (cdr L1) (cdr L2))))))



; Multiplicacion de una lista por un escalar
; Ej: (multiplicaEscalar 2 '(1 2 3 4)) => (2 4 6 8)


(define multiplicaEscalar (lambda (N L)
    (if (null? L) '()
       (cons (* N (car L)) (multiplicaEscalar N (cdr L))))))
```

```
; Producto cartesiano de dos listas (ambas de igual longitud)
; Ej: (productoCA '(1 2 3) '(2 3 4)) => 20

(define productoCA (lambda (L1 L2)
      (if (null? L1) 0
         (+ (* (car L1) (car L2)) (productoCA (cdr L1) (cdr L2))))))

; Determina cuantas veces se repite X en una lista.
; Ej: (ocurre 4 '(1 3 2 4 8 9 4)) => 2

(define ocurre (lambda (X L)
     (if (null? L) 0
        (if (equal? (car L) X) (+ 1 (ocurre X (cdr L)))
           (ocurre X (cdr L))))))

; Determina cuantas veces se repite X en una lista. (Multinivel)
; Ej: (ocurreM 3 '(1 2 (4 6 3) (2 4) 8 (1 (5 4 0)) 3)) => 2

(define ocurreM (lambda (X L)
      (if (null? L) 0
        (if (list? (car L)) (+ (ocurreM X (car L)) (ocurreM X (cdr L)))
           (if (equal? (car L) X) (+ 1 (ocurreM X (cdr L)))
              (ocurreM X (cdr L)))))))

 ; Calcula la cantidad de elementos iguales en la misma posicion en dos listas.
; Ej: (elemIguales '(1 2 3 4 5) '(7 2 8 6 5)) => 2  (por el 2 y 5)

(define elemIguales (lambda (L1 L2)
```

```
      (if (or (null? L1) (null? L2)) 0
         (if (equal? (car L1) (car L2)) (+ 1 (elemIguales (cdr L1) (cdr L2)))
            (elemIguales (cdr L1) (cdr L2))))))


; METODOS DE ORDENAMIENTO

; Ordena acendentemente
; Ej: (ordenar<= '(8 4 2 1 9 2 0 7)) => (0 1 2 2 4 7 8 9)

(define ordenar<= (lambda (L)
(if (null? L) '()
   (if (<= (car L) (minl L)) (concatenar (list (car L)) (ordenar<= (cdr L)))
            (ordenar<= (concatenar (cdr L) (list (car L))))))))

; Ordena descendentemente
; Ej: (ordenar<= '(8 4 2 1 9 2 0 7)) => (9 8 7 4 2 2 1 0)

(define ordenar>= (lambda (L) (invertir (ordenar<= L))))


; ARBOLES BINARIOS

; Ejemplos:
; (raiz '(a (b () ()) (c () ()))) => a
; (izq '(a (b () ()) (c () ()))  => (b () ())
; (der '(a (b () ()) (c () ()))  => (c () ())
```

```
(define raiz (lambda (A) (car A)))

(define izq (lambda (A) (cadr A)))

(define der (lambda (A) (caddr A)))

(define hoja? (lambda (A) (if (and (null? (izq A)) (null? (der A))) #t #f)))

; los nodos de un arbol pueden ser numeros o simbolos
; Ej: (nodo? 'A) => #t

(define nodo? (lambda (C) (or (number? C) (symbol? C))))

; arbol?
; Ej: (arbol? '(a (b () ()) (c () ()))) => #t

(define arbol? (lambda (A)
        (cond  ((null? A) #t)
            ((nodo? A) #f)
            ((!= (long A) 3) #f)
            (else (and (nodo? (car A))
                    (arbol? (cadr A))
                    (arbol? (caddr A)))))))

; Ej: (arbolito? '(a (b () ()) (c () ()))) => #t

(define arbolito? (lambda (L)
        (if(equal? L '()) #t
```

```
                (if(and (not (nodo? L)) (=(long L)3) (nodo?(car L)))
                  (and (arbolito? (cadr L)) (arbolito? (caddr L)) )
                   #f))))

; Ej: (arbusto? '(a (b () ()) (c () ()))) => #t

(define arbusto? (lambda (A)
            (if (null? A) #t
               (if (= (long A) 3)
                  (and (arbusto? (cadr A)) (arbusto? (caddr A)))
                  #f))))

; Peso de un Arbol: sumatoria (cada hojas por su nivel)
; Ej: (peso '(0 (6 () () ) (7 (3 () ()) () )) 0) => 12   (6*1 + 3*2)

(define peso (lambda (A N)
           (if (null? A) 0
              (if (hoja? A) (* N (raiz A))
                 (+ (peso (izq A) (+ 1 N)) (peso (der A) (+ 1 N)))))))

; Inorden
; Ej: (inorden '(a (b () ()) (c () ()))) => (b a c)
;     (inorden '(a (b (d () ()) (e () ())) (c (f () ()) (g () ())))) => (d b e a f c g)

(define inorden(lambda(A)
(if (not (arbol? A))#f
   (if (null? A)'()
           (if (and (nodo? A) (null? (cadr A)) (null? (caddr A)))
```

```
        (car A)
            (concatenar (concatenar (inorden(cadr A))
        (list(car A))) (inorden(caddr A))))))))

; Version 2
(define inorden2 (lambda (A) (if (null? A) '()
                        (append (inorden2 (izq A))
                                (list (raiz A))
                                (inorden2 (der A))  ))))

; Postorden
; Ej: (postorden '(a (b () ()) (c () ()))) => (b c a)
;    (postorden '(a (b (d () ()) (e () ())) (c (f () ()) (g () ())))) => (d e b f g c a)

(define postorden(lambda(A)
(if (not (arbol? A))#f
   (if (null? A)'()
  (if (and (nodo? A) (null? (cadr A) ) (null? (caddr A)))
(car A)
            (concatenar (concatenar (postorden (cadr A))
     (postorden(caddr A))) (list(car A))  ))))))

; Version 2
(define postorden2 (lambda (A) (if (null? A) '()
(append (postorden2 (izq A)) (postorden2 (der A)) (list (raiz A)) ))))

; Preorden
; Ej: (preorden '(a (b () ()) (c () ()))) => (a b c)
```

```
;     (preorden '(a (b (d () ()) (e () ())) (c (f () ()) (g () ()))))) => (a b d e c f g)


(define preorden(lambda(A)
(if (not (arbol? A))#f
   (if (null? A)'()
  (if (and (nodo? A) (null? (cadr A)) (null? (caddr A)))
(car A)
                                (concatenar (concatenar (list (car A)) (preorden(cadr A)))
(preorden(caddr A))  ))))))
; Version 2
(define preorden2 (lambda (A) (if (null? A) '()
                           (append (list (raiz A))
                                 (preorden2 (izq A))
                                 (preorden2 (der A))  ))))


; Recorrido Horizontal (ARBOL BINARIO)
; Ej: (RH '(a (b (d () ()) (e () ())) (c (f () ()) (g () ()))))) => (a b c d e f g)


(define RH (lambda (A) (RHaux (list A))))


(define RHaux (lambda (A)
  (if (null? A) '()
     (if (null? (car A)) (RHaux (cdr A))
     (cons (raiz (car A))
          (RHaux (append (cdr A) (list (izq (car A)) (der (car A))))))))))


; Pertenece el elemento N al arbol A?
; Ej: (aPertenece 'd '(a (b () (f () ())) (c (e () ()) (d () ()))))) => #t
```

```
(define aPertenece(lambda(N A)
        (if(not(arbol? A))#f
        (if(null? A)#f
        (if(equal? N (car A))#t
          (or (aPertenece N (cadr A)) (aPertenece N (caddr A))))))))

; Arbol completo de nivel n y funciones auxiliares
; Ejemplo de arbol completo: '(a (b (d () ()) (e () ())) (c (f () ()) (g () ())))

; Arbol de nivel 0?
; Ej: (adn0? '()) => #t

(define adn0? (lambda (A) (cond ((null? A) #t)
                ((and (null? (izq A))
                    (null? (der A))) #t)
                (else #f))))

; Devuelve el numero de nivel de un arbol
; Ej: (nivel '(a (b (d () ()) (e () ())) (c (f () ()) (g () ())))) => 2

(define nivel (lambda (A)
(if (adn0? A) 0 (+ 1 (max (nivel (izq A)) (nivel (der A)))))))

; #t si un Arbol de nivel N es completo.
; Ej: (completo? '(a (b (d () ()) (e () ())) (c (f () ()) (g () ())))) => # t
;     (completo? '(a (b (d () ()) (e () ())) (c (f () ()) ()))) => #f
```

```
(define completo? (lambda (A) (completoAux A (nivel A))))

(define completoAux (lambda (A N)
  (cond ((null? A) #f)
     ((= N 0) #t)
     ((= N 1) (and (notnull? (izq A)) (notnull? (der A))))
     (else (and (completoAux (izq A) (- N 1)) (completoAux (der A) (- N 1)))))))



; GRAFOS

; Representación ((a b) (b a) (a c))  c<--a<-->b

; adyacente no dirigido
; no usa la representación de arriba, sino ((a b) (a c)) c<-->a<-->b

(define adyacente (lambda (N1 N2 G)
 (if (or (pertenece (list N1 N2) G) (pertenece (list N2 N1) G)) #t #f)))

;adyacente dirigido
(define adyacenteD (lambda (N1 N2 G)
                    (if (pertenece (list N1 N2) G) #t #f)))

; Devuelve una lista de nodos
; (nodos '((a b) (b d) (d a) (a c) (c b))) => (d a c b)

(define nodos (lambda(G)
      (eliminaR (listaAtomos G))))
```

```
; Existe un camino entre los nodos E1 y E2 en el grafo G?
; Ej: (camino 'a 'd '((a b) (b c) (c d) (e e) (b d) (e a) (d e))) => #t

(define camino (lambda (E1 E2 G)
        (caminoAux E1 E2 (nodos G) G)))

(define caminoAux(lambda(E1 E2 N G)
       (if (or (null? N) (sumidero? E1 G)) #f
       (if (adyacenteD E1 E2 G) #t
        (if (adyacenteD E1 (car N) G)
           (caminoAux (car N) E2 (eliminaX E1 N) G)
           (caminoAux E1 E2 (moverAtras N 1) G))))))

; Existe un sumidero en el grafo G?
; Ej: (sumidero? 'e '((a b) (b c) (c d) (e e) (b d) (e a) (d e))) => #f
;     (sumidero? 'e '((a b) (b c) (c d) (b d) (d e))) => #t

(define sumidero? (lambda(N G)
       (if (null? G) #f
          (sumidero?Aux N G))))

(define sumidero?Aux (lambda(N G)
        (if (null? G) #t
        (if (equal? (caar G) N) #f
           (sumidero?Aux N (cdr G))))))

; Es conexo el grafo G?
```

```
; Usamos un algoritmo distinto  al que usamos en prolog, en lugar de ver si para cada nodo
; hay un camino a todos los demás, vemos si en una lista de todos los nodos se puede llegar
; del 1º al 2º, del 2º al 3º...
; (conexo? '((a b) (b c) (c a) (a e))) => #t
; (conexo? '((a b) (b c) (c a) (a e) (t r))) => #f

(define conexo? (lambda (G)
            (conexoAux (nodos G) G)))

(define conexoAux (lambda (LN G)
             (if (null? (cdr LN)) #t
                (if (camino (car LN) (cadr LN) G) (conexoAux (cdr LN) G)
                   #f))))

; COMBINATORIA

; PERMUTACION CON REPETICIONES TOMADAS DE A N
; Ej: (perConRep '(1 2 3) 2) => ((1 1) (1 2) (1 3) (2 1) (2 2) (2 3) (3 1) (3 2) (3 3))

(define perConRep (lambda (L long)
            (if (= 1 long) (mapS list L)
               (distribuirPCR L (perConRep L (- long 1))))))


(define distribuirPCR (lambda (L1 L2)
             (if (null? L1) '()
                    (concatenar (mapS (lambda (X) (cons (car L1) X)) L2)
                          (distribuirPCR (cdr L1) L2)))))
```

```
; PERMUTACIONES SIN REPETICIONES TOMADAS DE A N
; Ej: (perSinRep '(3 3 1) 2) => ((3 3) (3 1) (3 3) (3 1) (1 3) (1 3))
;     (perSinRep '(1 2 3) 2) => ((1 2) (1 3) (2 1) (2 3) (3 1) (3 2))

(define perSinRep (lambda (L N)
   (permutaAux L L N)))

   (define permutaAux (lambda (L M N)
      (if (or (< N 1) (> N (long L))) ()
      (if (= N 1) (listaElem L)
      (if (> N 2)
         (permutaAux (map1 L M) M (- N 1)) (map1 L M))))))

   (define map1
    (lambda (L M)
      (if (null? L) L
       (append (distribuye (car L) (borraelemSLdeL (car L) M))  (map1 (cdr L) M)))))



   ;Borra los elementos de SL en la lista L.
   (define borraelemSLdeL
    (lambda (SL L)
      (if (null? SL) L
         (if (list? SL)
      (borraelemSLdeL (cdr SL) (elimina1ra L (car SL) ))
         (elimina1ra L SL)))))
```

```
   ;genera una lista del tipo ((X L1) (X L2) ... (X Ln))
   (define distribuye (lambda (X L)
         (if (null? L) ()
           (if (list? X)
           (append (list(append X (list(car L)))) (distribuye X (cdr L)))
                 (append (list (list X (car L))) (distribuye X (cdr L)))))))

   (define listaElem (lambda (L)
         (if (null? L) () (append (list (list (car L))) (listaElem (cdr L))))))

 ; COMBINACIONES CON REPETICIONES
; Ej: (comConRep '(1 2 3) 2) => ((1 1) (2 1) (2 2) (3 1) (3 2) (3 3))

(define comConRep (lambda (L long)
                (elimComRep (perConRep L long))))

; COMBINACIONES SIN REPETICIONES
; Ej: (comSinRep '(1 2 3) 2) => ((2 1) (3 1) (3 2))

(define comSinRep (lambda (L long)
                (elimComRep (perSinRep L long))))

; Eliminar Combinaciones Repetidas

(define elimComRep (lambda (L)
                (if (null? L) '()
                   (if (perteneceCom (car L) (cdr L)) (elimComRep (cdr L))
                      (cons (car L) (elimComRep (cdr L)))))))
```

```
(define perteneceCom (lambda (X L)
                  (if (null? L) #f
                     (if (mismaCom X (car L)) #t
                        (perteneceCom X (cdr L))))))

(define mismaCom (lambda (C1 C2)
                (if (null? C1) #t
                   (if (not (pertenece (car C1) C2)) #f
                      (mismaCom (cdr C1) (elimina1ra C2 (car C1)))))))

(define elimina1ra
  (lambda(L e)
     (if (null? L) '()
        (if (equal? e (car L)) (cdr L)
           (cons (car L) (elimina1ra (cdr L) e))))))


; FUNCIONES DE ORDEN SUPERIOR

; ((op +) 2 3) => 5  (la operacion es binaria -> +)


(define op (lambda (O) (lambda (X Y) (O X Y))))

; ((op +) 2 3) => 5  (la operacion es binaria -> +)

(define op2 (lambda (O) O))
```

```
(define dos (lambda (F) (lambda (X)
                    (F(F X)))))
; ((dos ++) 1) => 3  (la operacion es unaria -> ++)

; Repetir  (F(F(F(F..F(X)))))

(define repetir (lambda (F N) (lambda (X)
      (if (= N 1) (F X) (F ((repetir F (- N 1)) X))))))

; Realiza una accion 'F' sobre todos los elementos de una lista (Ej ++, -- ...)
; Similar al do: de SmallTalk
; ((map ++) '(1 2 3 4)) => (23 4 5)

(define map (lambda (F) (lambda (L)
                    (if (null? L) '()
                       (cons (F (car L)) ((map F) (cdr L)))))))

; Igual que map pero con un solo lambda

(define mapS (lambda (F L)
           (if (null? L) '()
              (cons (F (car L)) (mapS F (cdr L))))))


; Ejemplo de uso de map. A cada elemento de la lista se le suma N
; (usoMap 1 2 '(1 1 1)) => (4 4 4)
```

```
(define usoMap (lambda  (N1 N2 L)
            ((map (lambda (X) (+ N1 N2 X))) L)))

(define usoMap2 (lambda  (N1 N2 L)
            (mapS (lambda (X) (+ N1 N2 X)) L)))


(define filterT (lambda (F L)
             (if (null? L) '()
                (if (F (car L)) (cons (car L) (filterT F (cdr L)))
                   (filterT F (cdr L))))))

(define filterF (lambda (F L) (filterT (lambda (X) (not (F X))) L)))



; EJERCICIOS RESUELTOS

; Serie de Fibonacci: 1 1 2 3 5 8 13 ...
; Devuelve el elemento enesimo de la serie de Fibonacci.
 ;
; Ej: (Fibonacci 2) => 1        (Fibonacci 6) => 8

(define Fibonacci (lambda (N)
               (cond ((= N 1) 1)
                   ((= N 2) 1)
                   (else (+ (Fibonacci (- N 1)) (Fibonacci (- N 2)))))))
```

```
; Fibonacci iterativo
; Ej: (Fibonacci 2) => 1        (Fibonacci 6) => 8

(define fib
  (lambda (n)
    (fib-iter 1 0 n)))

(define fib-iter
  (lambda (a b count)
    (if (= count 0)
        b
        (fib-iter (+ a b) a (- count 1)))))

; Tartaglia
; Devuelve la fila del triangulo de Tartaglia correspondiente a N
;
; Ej: (Tartaglia 7) => (1 6 15 20 15 6 1)

(define Tartaglia (lambda (N)
 (cond ((= N 1) '(1))
 ((= N 2) '(1 1))
 (else (append (append '(1) (sumaDeADos (Tartaglia (- N 1)))) '(1)) ))))

(define sumaDeADos (lambda (L)            ; L=(1 2 3 4 5)  => (3 5 7 9)
 (if (> (long L) 1) (cons (+ (car L) (cadr L)) (sumaDeADos (cdr L))) '() )))

; Aceptador de Estados Finitos (maquina de estados finitos).
;(AEF '(1 2 1) '(A B A ((A B 1) (A C 2) (B A 2) (C B 3)))) => t
```

```
(define AEF (lambda (S M)
    (if (null? S)
       (if (equal? (F M) (A M) )#t #f)
    (AEF (cdr S) (trans (car S) M M)))))

(define trans (lambda (Tr M1 M2)
    (if (and (equal? (N1 M1) (A M1)) (equal? (T M1) Tr) )
          (list (I M1) (F M1) (N2 M1) (G M2))
          (trans  Tr (list (I M1) (F M1) (A M1) (cdr (G M1))) M2))))

(define I car) (define F cadr) (define A caddr) (define G cadddr)

(define N1 (lambda (M)
         (caar (G M))))

(define N2 (lambda (M)
         (cadar (G M))))

(define T (lambda (M)
        (caddar (G M))))

; EJERCICIO 34
; Representa más eficientemente una matriz rala. Devuelve una lista conteniendo ternas (X Y V) V<>0
; Ej: (convierte '((10 0 0 20 0) (0 1 5 0 0) (2 0 0 0 0)))=> ((1 1 10) (1 4 20) (2 2 1) (2 3 5) (3 1 2))

(define convierte2 (lambda (M)
```

```
            (if (null? M) '()
               (convierte2Aux 1 M))))


(define convierte2Aux (lambda (F M)
 (if (null? M) '()
    (concatenar (fila F 1 (car M)) (convierte2Aux (++ F) (cdr M))))))

(define fila (lambda (F C L)
 (if (null? L) '()
            (if (equal? (car L) 0) (fila F (++ C) (cdr L))
               (cons (list F C (car L)) (fila F (++ C) (cdr L)))))))

; EJERCICIO 37
; Dado un arbol n-ario con nodos 1 o 0, determina si una palabra puede leerse recorriendo alguna rama
; Ej: (check '(1 0 0 0) '(1 (1 (1 (1)) (1)) (0 (0)) (1 (0)))) => #f
;     (check '(1 0 0)  '(1 (1 (1 (1)) (1)) (0 (0)) (1 (0)))) => #t

(define check (lambda (P A)
      (if (null? P) #t
      (if (and (equal? (car A) (car P)) (enHijo (cdr A) (cdr P))) #t #f))))

(define enHijo (lambda (H P)
      (if (and (null? H) (null? P)) #t
      (if (and (null? H) (not (null? P))) #f
        (not (null? (filterT (lambda (X) (check P X)) H)))))))
```

```
; EJERCICIO 38

; Dada una lista de funciones (de un argumento) y una de elementos, aplica cada una de las funciones

; a los elementos de la lista.

; (mapFun '(++ -- par?) '(1 2 3 5)) => ((2 3 4 6) (0 1 2 4) (#f #t #f #f))


(define mapFun (lambda (F L)

            (if (null? F) '()

               (append (list (mapS (eval (car F)) L))

                     (mapFun (cdr F) L)))))


; Ejercicio 36: Funcion impar

; F es una funcion y L una lista de valores para el argumento x de F

;  (fi '(+ 1 X) '(1 2 3)) => #f

;  (fi '(* X X) '(1 2 3)) => #t

;  (fi '(* (+ X 2) (+ X 2)) '(1 2 3)) => #f

;  (fi '(* (+ X 2) (- X 2)) '(1 2 3)) => #t


 (define fi (lambda (F L)

    (if (equal? (fiAux F 1 L) (fiAux F -1 L)) #t #f)))


 (define fiAux (lambda (F V L)

       (if (null? L) L

        (cons (evaluar (xReemplazarM 'X (* V (car L)) F) ) (fiAux F V (cdr L))))))


 (define evaluar (lambda (F)

        (cond [(and (list? (cadr F)) (list? (caddr F)))

                ((eval (car F)) (evaluar (cadr F) ) (evaluar(caddr F) ))]
```

```
            [(list? (cadr F)) ((eval (car F)) (evaluar (cadr F) ) (caddr F))]
            [(list? (caddr F)) ((eval (car F)) (cadr F) (evaluar (caddr F)))]
            [else ((eval (car F)) (cadr F) (caddr F))] )))

; EJERCICIO 39-b
; Dada una lista de árboles, devuelve el de menor peso.
;Ej: (mapeoArboles '((0 (6 () () ) (7 () () )) (0 (0 (2 () ()) (3 () () )) (5 () () )))) => (0 (6 () ()) (7 () ()))

(define mapeoArboles (lambda (LA)
     (if (null? LA) '0
        (enesimoElem (car (ocurrencias (armalistaSumas LA)
    (minL (armalistaSumas LA)))) LA))))

(define armalistaSumas (lambda (LA)
     (if (null? LA) '()
        (cons (peso (car LA) 0) (armalistaSumas (cdr LA))))))


; EJERCICIO 40
; Cambia de una forma de representación de grafos a otra.
; Ej: (convierte '((a b c d e) ((a b) (b c) (b d) (c e) (d a) (d e) (e a))))
;     ((a (b)) (b (c d)) (c (e)) (d (a e)) (e (a)))

(define convierte(lambda (L)
    (if (null? L) '()
       (convierteAux (car L) (cadr L)))))

(define convierteAux(lambda (L G)
```

```
    (if (null? L) '()
       (cons (list (car L) (extrae (car L) G)) (convierteAux (cdr L) G)))))

(define extrae (lambda (N G)
    (if (null? G) '()
        (if (equal? N (caar G)) (cons (cadar G) (extrae N (cdr G)))
          (extrae N (cdr G))))))

; EJERCICIO 41
; Se quiere calcular las comiciones de un viajante.
; Ej: (comisiones '((Pepe (100 A) (50 B)) (Ana (50 A) (100 B) (20 C))) '((A  0.5) (B 0.2) (C 0.1)))
;     ((pepe (100 a) (50 b) 60.0) (ana (50 a) (100 b) (20 c) 47.0))

(define comisiones (lambda (Viajantes P)
(if (null? Viajantes) '()
   (append (list (append (car Viajantes)
          (list (sumaElem (comicionV (cdar Viajantes) P)))))
         (comisiones (cdr Viajantes) P)))))

(define comicionV (lambda (LV P)
              (mapS (lambda (X) ( comicionP P X)) LV)))

(define comicionP (lambda (P V)
  (if (equal? (caar P) (cadr V))
     (* (cadar P) (car V))
     (comicionP (cdr P) V))))

; EJERCICIO DE EXAMEN: 05/08/2002
```

```
; Crear una función de orden superior para calcular el seno de A (en radianes)
; con una serie de N elementos
; NO esta resultando igual a la funcion seno, no debe ser correcta la formula que nos dieron.-

(define Serie ( lambda (N) (lambda (A)
                    (if (= N -1) 0
                       (+ (/ (* (potencia -1 N) (potencia A (+ N 1)))
                            (fact (+ N 1)))
                          ((Serie (- N 1)) A))))))

(define SerieS ( lambda (N A)
                    (if (= N -1) 0
                       (+ (/ (* (potencia -1 N) (potencia A (+ N 1)))
                            (fact (+ N 1)))
                          (SerieS (- N 1) A)))))

; EJERCICIO DE EXAMEN: Sumador de números binarios
; Ej: (SumaBin '(1 0 0) '(1)) => (1 0 1)
;     (SumaBin '(1 1 1 0 1) '(1 0 0 1)) => (1 0 0 1 1 0)

(define SumaBin (lambda (N1 N2)
   (cond [(null? N2) N1]
   [(null? N1) N2]
   [else (if (= 2 (+ (ultimoElem N1) (ultimoElem N2)))
      (concatenar2 (SumaBin (SumaBin '(1)  (qu N2)) (qu N1)) 0)
            (concatenar2 (SumaBin (qu N2) (qu N1))
   (+ (ultimoElem N2) (ultimoElem N1))))])))
```

```
; EJERCICIO DE EXAMEN: Composicion de funciones (f g h) => (f(g(h)))
; (componer '((+ (ln (^ x 2)) x) (+ x 1) (sin x))) => (+ (ln (^ (+ (sin x) 1) 2)) (+ (sin x) 1))

(define componer (lambda (L)
            (if (= (long L) 1) (car L)
               (componer (cons (xReemplazarM 'x (cadr L) (car L))
                        (cddr L))))))

; Orden( Criterio Lista )
; Dada una lista de números y un criterio de ordenación (<, >, >=, ...) devuelve las subsecuencias
; (de longitud mayor a 1) completas que verifican este criterio.
; Ej: (orden < '(1 2 3 1 7 15 24 6 67 78 9)) => ((1 2 3) (1 7 15 24) (6 67 78))

(define orden(lambda(criterio L)
        (if (null? L) L
        (if (null? (ordenAux criterio L)) (orden criterio (cdr L))
           (cons (cons (car L) (ordenAux criterio L))
           (orden criterio (sacaNpri(+ 1 (long (ordenAux criterio L))) L )))))))

(define ordenAux (lambda(criterio L)
        (if (= (long L) 1) '()
        (if (criterio (car L) (cadr L))
           (cons (cadr L) (ordenAux criterio (cdr L)))
           '()))))

; EJERCICIO FOR (no tiene mucha logica..)
```

```
; Ej: (for 1 2000 1 '((lambda () #t) () ())) => ()

(define for  (lambda (i f c s)
            (if (<= i f)
               (and (evaluarFor s)
                    (for (+ i c) f c s)
                () ))))

(define evaluarFor (lambda (s)
         (if (null? s) s
            (if ((eval (car s))) (evaluarFor (cadr s))
                     (evaluarFor (caddr s))))))

; ARBOL DE DIRECTORIO
; Ej: (buscar ar1 '(d0 d1 a1)) => "archivo"
; Ej: ((eval (lambda () "archivo"))) => "archivo"

(define buscar (lambda (D A)
 (if (null? A) ((eval (car D)))
    (buscar (cdar (filterT (lambda (X) (equal? (car A) (car X))) D)) (cdr A)))))

(define ar1 '((d0 (d1 (a1 (lambda() "archivo"))))))
```

VARIAS VERSIONES DE TORRES DE HANOI

```
; ********************************* Version 1 **********************************

(define (hanoi n)
```

```
  (if (zero? n)
      (display "Huh?\n")
      (transfer 'A 'B 'C n)))

(define (print-move from to)
  (format #t "Move disk from ~s to ~s~%" from to))

(define (transfer from to via n)
  (if (= n 1)
      (print-move from to)
      (transfer from via to (1- n))
      (print-move from to)
      (transfer via to from (1- n))))

(hanoi 3)


; ********************************* Version 2 **********************************

(define *start* 0)
(define *aux* 1)
(define *end* 2)

(define hanoi
        (lambda (n start aux end)
                (if (= n 1)
                        (list start end)
                        (append
```

```
                                (hanoi (- n 1) start aux end)
                                (list start aux)
                                (hanoi (- n 1) aux start end)))))


; ********************************* Version 3 **********************************

(define (mover origen destino)
   (display "Mueve un anillo")
   (display " desde ")
   (display origen)
   (display " hacia la varilla ")
   (display destino)
   (display "\n")
   1
 )

(define (hanoi n a c b)
  (if (> n 1)
     (+ (hanoi (- n 1) a b c)
        (mover a c)
        (hanoi (- n 1) b c a)
          )
     (mover a c)
  )
 )

; ********************************* Version 4 **********************************
```

```
% hanoi(Discos,Origen,Ayuda,Destino).

hanoi(0,_,_,_) :- !.

hanoi(N,I,F,A) :- N>0, N1 is N-1, hanoi(N1,I,A,F), write(I),write('->'),write(F), nl, hanoi(N1,A,F,I).

% Ejemplo con 3 discos en la primera columna: hanoi(3,pri,seg,ter).
```

TRABAJOS PRÁCTICOS

GUIA Nº 5

Ejercicio 1:

a. (+ (- (* 3 a) b) (+ (* 3 a) b))

(let ((a 3) (b 2))

(let ((x (* 3 a)))

(+ (- x b) (+ x b))))

b. (cons (car (list a b c)) (cdr (list a b c)))

(let ((x (list 'a 'b 'c)))

(cons (car x) (cdr x)))

Ejercicio 2:

(let ((x 9))

(* x

(let ((x (/ x 3)))

(+ x x))))

Para poder derivar dicho resultado, se debe analizar la expresión de la siguiente manera:

- "let" asigna el valor 9 a la variable x.

- El segundo "let" asigna a x la expresión x/3 y sabiendo que la variable x posee el valor 9, el

nuevo valor de dicha variable es 3. La expresión definida por este último "let" es la de x+x, por lo tanto de 6.

- Finalmente se calcula el valor de la expresión del primer "let", x*(resultado del segundo "let"). El valor final es 54.

Ejercicio 3:

a.(let ((x 'a) (y 'b))

(list (let ((z 'c)) (cons z y))

(let ((w 'd)) (cons x w))))

(let ((x 'a) (y 'b)) (list (let ((z 'c))(cons z y)) (let ((w 'd)) (cons x w))))

b. (let ((x '((a b) c)))

(cons (let ((y (cdr x)))

(car y))

(let ((z (car x)))

(cons (let ((s (cdr z)))

(car s))

(cons (let ((w (car z)))

w)

(cdr z))))))

(let ((x '((a b) c))) (cons (let((y (cdr x)))(car y)) (let((z (car x)))(cons (let ((q (cdr z)))(car q))(cons

(let ((w (car z)))w) (cdr z))))))

Ejercicio 4:

a. (let ((f (lambda (x) x)))

(f 'a))

Devuelve el valor "a" ya que "let" asigna a la variable f la funcion lambda que toma como

parámetro la variable x y solo la retorna.

b. (let ((f (lambda x x)))

(f 'a))

c. (let ((f (lambda (x . y) x)))

(f 'a))

Lambda recive como parámetro la lista impropia (x . y), devuelve el valor "x" y "let" asigna a la variable f la expresión declarada en el lambda. Resultado que da "a".

d. (let ((f (lambda (x . y) y)))

(f 'a))

Lambda recive como parámetro la lista impropia (x . y), devuelve el valor "y" y "let" asigna a la variable f la expresión declarada en el lambda. Resultado que da una lista vacía porque el parámetro

que se le pasa a la función f es el elemento "a" que es asignado a la "x" de la lista impropia, quedando "y" con el valor "()".

Ejercicio 5:

```
(define shorter
 (lambda (x y)
  (if (< (length x) (length y))
      x
      (if (= (length x) (length y))
          x
          y
          )
      )
  )
 )
```

Ejercicio 6:

```
(define area-circ
  (lambda (r)
    (* 3.14 (* r r))
    )
  )
```

Ejercicio 7:

Miguel Schpeir – Ingeniería en Informática

```
(define super-circ
  (lambda (r)
    (* 3.14 (* 2 r))
    )
  )
```

Ejercicio 8:

```
(define circlestuff
  (lambda (r)
    (list (area-circ r) (super-circ r))
    )
  )
```

Ejercicio 9:

(distance2d '(1 . 1) '(2 . 2)) → 1.41

D = ((x2 – x1) ^ 2 + (y2 – y1) ^ 2) ^ (1 / 2)

```
(define distance2d
```

```
 (lambda (x y)
   (let ((x1 (car x)) (x2 (car y)) (y1 (cdr x)) (y2 (cdr y)))
     (sqrt (+ (* (- x2 x1) (- x2 x1)) (* (- y2 y1) (- y2 y1))))
     )
   )
 )
```

Ejercicio 10:

```
(define prox
 (lambda (x y)
   (let ((pc '(999 . 999)))
     (if (< (distance2d x (car y)) (distance2d x pc))
         (let ((pc (car y)))
           (if (eqv? (cdr y) '())
               (distance2d x pc)
               (prox x (cdr y)))
           )
         (prox x (cdr y))
         )
     )
   )
 )
```

GUIA Nº 6

Miguel Schpeir - Ingeniería en Informática

Ejercicio 1:

```
 (define count-elem
           (lambda (x y)
```

```
   (if (null? y)
      0
     (if (eqv? x (car y))
                  (+ (count-elem x (cdr y)) 1)
                  (count-elem x (cdr y))
                    )
              )
)
)
 (count-elem 3 '(1 2 3 4 5 4 3 2 1 ))
```

Ejercicio 2:

```
 (define fact
  (lambda (n)
    (if (= n 0)
       1
           (* (fact (- n 1)) n)
    )
  )
)
(fact 9999)
```

Ejercicio 3:

```
 (define largo
  (lambda (x)
    (if (null? x)
      0
     (+ (largo (cdr x)) 1)
```

```
    )
)
)
 (largo '(1 2 3 4 5 6))
```

Ejercicio 4:

```
(define sumahora
  (lambda (listaH1 listaH2)
    (if (or (null? listaH1) (null? listaH2))
        "Error"
        (let ((hh (+ (car listaH1) (car listaH2)))
              (mm (+ (cadr listaH1) (cadr listaH2)))
              (ss (+ (caddr listaH1) (caddr listaH2)))
              )
          (list
           (+ hh (car (sexa (+ mm (car (sexa ss))))))
           (cadr (sexa (+ mm (car (sexa ss)))))
           (cadr (sexa ss))
          )
        )
    )
  )
  )
```

```
; Recibe la lista con las estaciones y sus correspondientes demoras asi como tambien el horario de salida
 (define demora
  (lambda (lista hora)
```

```
   (if (null? lista)
       '()
       (cons
        (list (caar lista) (sumahora hora (cadr lista)))
        (demora (cdr lista) (sumahora hora (cadr lista)))
        )
       )
   )
 )
 (estacion 5)
```

Ejercicio 5:

```
 (define concatenar
  (lambda(L1 L2)
    (if (null? L1)
        L2
        (cons (car L1) (concatenar (cdr L1) L2))
    )
  )
)
(concatenar '(1 2 3 4) '(5 6 7 8))
```

Ejercicio 6:

```
 (define firstnum
  (lambda (ls)
    (if (null? ls)
        "null"
        (if (number? (car ls))
```

```
          (display (car ls))
          (firstnum (cdr ls))
       )
   )
  )
)
 (firstnum '(j h k i 2 o 1))
```

Ejercicio 7:

```
 (define attach-at-end
  (lambda (x ls)
    (if (null? ls)
        (cons x ls)
        (if (eqv? x (car ls))
           (attach-at-end x (cdr ls))
           (cons (car ls) (attach-at-end x (cdr ls)))
         )
    )
  )
)
 (attach-at-end 2 '(2 3 4 5 2 6))
```

Ejercicio 8:

```
(define subst
  (lambda (uno dos ls)
    (if (null? ls)
        '()
        (if (eqv? uno (car ls))
```

```
          (cons dos (subst uno dos (cdr ls)))
          (cons (car ls) (subst uno dos (cdr ls)))
       )
   )
 )
)
(subst 'c 'k '( c o c o n u t))
```

Ejercicio 9:

```
(define range-sum
  (lambda (n1 n2)
    (if (eqv? n1 n2)
        n2
        (+ n1 (sum (+ n1 1) n2))
    )
   )
)
(range-sum 20 34)
```

Ejercicio 10:

```
(define pendiente
  (lambda (p1 p2)
    (/(- (cdr p2) (cdr p1)) (- (car p2) (car p1)))
  )
)

(define inter
```

```
 (lambda(p1 p2 p3 p4)
  (if (eqv? (/ (- (cdr p2) (cdr p1)) (- (car p2) (car p1))) (/ (- (cdr p4) (cdr p3)) (- (car p4) (car p3))))
     (display "Las rectas son paralelas")
     (let((x (/ (-(+ (- (* (pendiente p1 p2) (car p1)) (* (pendiente p3 p4) (car p3))) (cdr p3)) (cdr p1))
(- (pendiente p3 p4) (pendiente p1 p2)))))
    (cons x (+(-(* (pendiente p1 p2) x) (*(pendiente p1 p2) (car p1)))(cdr p1)))
  )
 )
)

(inter '(1 . 2) '(3 . 4) '(5 . 6) '(8 . 8))
```

GUIA Nº 7

Miguel Schpeir - Ingeniería en Informática

Ejercicio 1:

```
(define aux
 (lambda (x y)
  (let ((x1 (car x)) (x2 (car y)) (y1 (cdr x)) (y2 (cdr y)))
   (sqrt (+ (* (- x2 x1) (- x2 x1)) (* (- y2 y1) (- y2 y1))))
  )
 )
)

(define distance2d
 (lambda (p1 p2)
  (let ((a (cons (coord-x p1) (coord-y p1))) (b(cons (coord-x p2) (coord-y p2))))
```

```
   (aux a b)
   )
 )
)

(define-struct coord (x y))
(define punto1 (make-coord 1 2))
(define punto2 (make-coord 3 4))
(distance2d punto1 punto2)
```

Ejercicio 2:

```
 (define sumx
  (lambda (ls)
    (if (null? ls)
        0
        (+ (coord-x (car ls)) (sumx (cdr ls)))))
 )
(define sumy
  (lambda (ls)
    (if (null? ls)
        0
        (+ (coord-y (car ls)) (sumy (cdr ls)))))
 )
(define sumz
  (lambda (ls)
    (if (null? ls)
        0
        (+ (coord-z (car ls)) (sumz (cdr ls)))))
```

```
 )
(define gravity
  (lambda (l)
    (if (not (null? l))
        (let ((x (/ (sumx l) (length l)))
              (y (/ (sumy l) (length l)))
              (z (/ (sumz l) (length l))))
          (cons x (cons y (list z)))
         )
        )
 )
)

(define-struct coord (x y z))
(define punto1 (make-coord 1 2 3))
(define punto2 (make-coord 4 5 6))
(define punto3 (make-coord 7 8 9))
(let (( l (list punto1 punto2 punto3)))
  (gravity l)
)
```

Ejercicio 3:

```
 (define calc
  (lambda (a p1 p2)
    (let* ((x1 (/ (* (distance2d p1 p2) (auto-consumo a)) (auto-velocidad a)))
           (x2 (/ x1 (auto-consumo a))))
    (cons x1 x2)
    )
```

```
  )
)

(define-struct coord (x y))
(define punto1 (make-coord 1 2))
(define punto2 (make-coord 3 4))
(define-struct auto (consumo velocidad))
(define a1 (make-auto 10 100))
(define a2 (make-auto 13 90))
(calc a1 punto1 punto2)
```

Ejercicio 4:

```
 (define aux
  (lambda (x ls)
    (if (not (null? ls))
        (cons (distance2d (x (car ls))) (aux x (cdr ls)))
        0
    )
  )
)

(define comp
  (lambda (x ls)
    (if (not (null? ls))
        (if (> (car ls) x)
            (car ls)
            (comp x (cdr ls))
        )
```

```
    )
)
)

(define distmaxaux
  (lambda (ls max)
    (if (not (null? ls))
        (let ((a(comp max (aux (car ls) (cdr ls)))
        (distmaxaux (cdr ls) max)

(define-struct coord (x y))
(define punto1 (make-coord 1 2))
(define punto2 (make-coord 3 4))
(define-struct auto (consumo velocidad))
(define a1 (make-auto 10 100))
(define a2 (make-auto 13 90))
(calc a1 punto1 punto2)
```

Ejercicio 5:

```
 (define countchar
  (lambda (l)
    (if (eof-object? (read-char l))
        0
        (+ (countchar l) 1)
    )
  )
)
```

```
(define countword
  (lambda (l)
    (if (eof-object? (read l))
        0
        (+ (countword l) 1)
    )
  )
)

(define i (open-input-file "C:/migue.txt"))
(display "Cantidad de caracteres: ")
(countchar i)
(display "")
(display "Cantidad de palabras: ")
(define i (open-input-file "C:/migue.txt"))
(countword i)
```

Ejercicio 6:

```
 (define show
     (lambda (x)
           (define temp (read x))
                  (if (eof-object? temp)
                     (display #\ )
                     (begin
                       (display temp)
                       (display #\ )
                       (show x)
                       )
```

```
                    )
      )
 )
 (define i (open-input-file "C:/migue.txt"))
(show i)
```

Ejercicio 7:

```
 (define opera
  (lambda (path)
    (let ((b (read i)) (a (read i)) (c (read i)))
      (if (eqv? a '-)
          (- b c)
          (+ b c)
      )
   )
  )
)

(define i (open-input-file "C:/migue.txt"))
(opera i)
```

MIGUEL SCHPEIR – UNIVERSIDAD NACIONAL DEL LITORAL