

; PREDICADOS SCHEME - lenguaje R5RS - v3 - 05/2005
 ; por RB-JJC-MG-CEP-

; FUNCIONES LOGICAS Y MATEMATICAS

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

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

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

(define nonnull? (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 9 1 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 '(potencia) 'h) => (ponencia)

```
(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 3 2 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 (car L)))))))

; (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 6 5 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 tamaño.
; 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 tamaño.
; 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 descendientemente
; 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) (eliminalra L (car SL) ))
            (eliminalra 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) (eliminalra C2 (car C1)))))))

```

```
(define eliminalra
  (lambda (L e)
    (if (null? L) '()
        (if (equal? e (car L)) (cdr L)
            (cons (car L) (eliminalra (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 45)
```

```
(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 caddr)

(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 árbol 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 comisiones 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 (cadr 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))
                      (caddr 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"))))))

```