

Introducción

En el capítulo anterior se utilizó el lenguaje de programación para construir fórmulas matemáticas. Los *números* fueron los principales elementos que se utilizaron como variables de entrada y de salida para las funciones construidas.

En el presente capítulo se muestran otros elementos además de los números, estos nuevos elementos serán llamados *símbolos*. Los números son un tipo particular de símbolos. Los símbolos se agrupan en conjuntos que se denominan *listas*. Se presentarán las funciones básicas para la manipulación de las listas, se construirán funciones adicionales para su descripción y operación, finalmente se utilizarán en la solución de problemas computacionales.

Descripción de una lista

Un símbolo es cualquier expresión válida. Una lista es un conjunto de símbolos relacionados entre sí. Una lista posee una estructura que inicia con un paréntesis izquierdo, termina con un paréntesis derecho y dentro de ellos se encuentran los símbolos separados por espacios en blanco. Formalmente una lista se puede describir por medio de la siguiente estructura:

() que representa la lista vacía o lista nula

($s_1 \ s_2 \ \dots \ s_N$) que representa una lista que puede tener de 1 a N símbolos

La lista vacía, representada como () se suele llamar lista nula o lista nil. La lista vacía es un caso particular de la descripción general de una lista.

Las funciones CAR y CDR

La función car regresa como valor el primer elemento de una lista. Su sintaxis se puede describir de la siguiente forma:

```
> (car '( ))  
error  
> (car '(s1 s2 ... sN))  
s1
```

A continuación se presentan algunos ejemplos:

```
> (car '( ))  
error  
> (car '(1 2 3))  
1  
> (car '(a b c))  
a  
> (car '(+ (* 2 3) (* 4 5)))  
+  
> (car '((1 2) (2 3) (3 4)))  
(1 2)
```

La función cdr realiza una función complementaria. Regresa una lista que contiene todos los elementos excepto el primero. Su sintaxis se puede describir de la siguiente forma:

```
> (cdr '( ))  
error  
> (cdr '(s1 s2 ... sN))  
(s2 ... sN)
```

A continuación se presentan algunos ejemplos:

```
> (cdr '( ))  
error
```

```

> (cdr '(1))
'( )
> (cdr '(1 2 3))
(2 3)
> (cdr '(a b c))
(b c)
> (cdr '(+ (* 2 3) (* 4 5)))
((2 3) (4 5))
> (cdr '((1 2) (2 3) (3 4)))
((2 3) (3 4))}
```

Se pueden combinar las funciones para obtener un valor deseado.

```

> (car (car '((1 2) (2 3) (3 4))))
1
> (car (cdr '((1 2) (2 3) (3 4))))
(2 3)
> (car (cdr (cdr ((1 2) (2 3) (3 4)))))
(3 4)
```

Como las combinaciones anteriores son muy usuales existe una forma abreviada que inicia con la letra *c* y termina con la letra *r* y puede tener hasta cuatro valores intermedios de la forma: *c----**r*. En los valores centrales se puede sustituir por una letra *a* para realizar la operación de *car*, o bien por una letra *d* para realizar la operación de *cdr*. Utilizando esta forma resumida se pueden reescribir los ejemplos anteriores como:

```

> (caar '((1 2) (2 3) (3 4)))
1
> (cadr '((1 2) (2 3) (3 4)))
(2 3)
> (caddr '((1 2) (2 3) (3 4)))
(3 4)
```

Las funciones CONS y LIST

A diferencia de las funciones de car y cdr que sirven para destruir listas en partes, la función cons sirve para construir listas. Esta función toma un símbolo cualquiera y lo inserta como el primer elemento de la lista. Su sintaxis es:

```
> (cons s0 '(s1 s2 ... sN))
(s0 s1 s2 ... sN)
```

A continuación se presentan algunos ejemplos:

```
> (cons 1 '(2 3 4))
(1 2 3 4)

> (cons 'a '(b c d))
(a b c d)

> (cons '(1 2) '(a b c))
((1 2) a b c)

> (cons '(1 2) '((3 4) (5 6)))
((1 2) (3 4) (5 6))
```

La función list también se utiliza para construir listas. Recibe un número variable de parámetros y produce como resultado una lista con todos los parámetros. Su sintaxis está dada por:

```
> (list s0 s1 s2 ... sN)
(s0 s1 s2 ... sN)
```

Otra forma de describir la función que realiza list es:

```
> (list s0 s1 s2 ... sN)
```

es equivalente a

```
> (cons s0
        (cons s1
              (cons s2
                    ...
                    (cons sN ( )))))
```

A continuación se presentan algunos ejemplos.

```
> (list)
()
```

```
> (list 1)
(1)

> (list 1 2 3 4)
(1 2 3 4)

> (list '1 '2 '3 '4)
(1 2 3 4)

> (list 'a 'b 'c 'd)
(a b c d)
```

Las funciones LIST?, EQUAL?, NULL?

La función list? recibe un argumento y devuelve el valor de #t en caso que dicho argumento sea una lista y #f en cualquier otro caso. Su sintaxis es:

```
> (list? s1)
;; Devuelve #t si s1 es una lista
;; Devuelve #f si s1 no es una lista
```

A continuación se presentan algunos ejemplos.

```
> (list? '())
#t

> (list? 3)
#f

> (list? '(1 2 3))
#t

> (list? (- 8 7))
#f
```

La función equal? se utiliza para comparar elementos. Se puede utilizar con argumentos numéricos o con argumentos más complejos. Recibe dos argumentos y devolverá el valor de #t en caso que sus dos argumentos sean iguales y devolverá el valor de #f en cualquier otro caso. Su sintaxis es:

```
> (equal? arg1 arg2)
;; Devuelve #t si arg1 es igual a arg2
;; Devuelve #f si no son iguales
```

Se utiliza de la siguiente forma:

```
> (equal? 0 0)
#t

> (equal? 0 (- 7 7))
#t

> (equal? '(1 1) '(1 1))
#t
```

La función null? recibe un argumento y devuelve el valor de #t en caso que dicho argumento sea la lista nula y #f en cualquier otro caso. Su sintaxis es:

```
> (null? arg1)
;; Devuelve #t si arg1 se evalua a la lista nula
;; Devuelve #f en cualquier otro caso
```

A continuación se presentan algunos ejemplos.

```
> (null? '())
#t

> (null? '(1))
#f

> (null? (cdr '(1)))
#t

> (null? (cdr '(1 2)))
#f
```

Aritmética para números racionales

En este ejemplo se presenta una primera abstracción de datos. Se construirán funciones para crear números racionales, con sus partes de numerador y denominador. Se construirán además funciones para simplificar números racionales y para determinar si dos números racionales son iguales. Por último se implementarán las funciones de sumar, restar, multiplicar y dividir. Se espera que cualquier operación produzca como resultado un nuevo número racional, que se debe expresar en su forma numérica más sencilla.

Para lograr expresar los números racionales en su forma más simple es necesario construir una función auxiliar que se llamará máximo común divisor o simplemente mcd. El mcd recibirá dos números representados por las letras a y b. Obtendrá como resultado el mayor entero que divide con residuo de cero a ambos números.

Por ejemplo:

> (mcd 1 6)

1

> (mcd 7 15)

1

> (mcd 2 6)

2

> (mcd 6 9)

3

> (mcd 4 12)

4

> (mcd 12 16)

4

> (mcd 15 20)

5

> (mcd 12 18)

6

> (mcd 24 36)

12

> (mcd 168 216)

24

Existen varios métodos para implementar la función de mcd. A continuación se presenta un método muy eficiente llamado el algoritmo de Euclides. Este método utiliza residuos sucesivos entre los números. En general se tiene que:

$$(mcd \ b \ a) = (mcd \ a \ r)$$

donde r es el residuo de la división entera entre b y a.

Es posible demostrar que si se inicia con dos números enteros y se repite este proceso, eventualmente se obtendrá que el segundo parámetro se convierte en 0 y el mcd será el primer parámetro. Es decir ($\text{mcd } a \ 0 = a$).

Así si se desea calcular ($\text{mcd } 12 \ 16$) se obtienen los siguientes pasos:

Se desea calcular ($\text{mcd } 12 \ 16$)

- = ($\text{mcd } 16 \ 12$) pues el ($\text{remainder } 12 \ 16$) es 12
- = ($\text{mcd } 12 \ 4$) pues el ($\text{remainder } 16 \ 12$) es 4
- = ($\text{mcd } 4 \ 0$) pues el ($\text{remainder } 12 \ 4$) es 0
- = 4

Así si se desea calcular ($\text{mcd } 168 \ 216$) se obtienen los siguientes pasos:

Se desea calcular ($\text{mcd } 168 \ 216$)

- = ($\text{mcd } 216 \ 168$) pues el ($\text{remainder } 168 \ 216$) es 168
- = ($\text{mcd } 168 \ 48$) pues el ($\text{remainder } 216 \ 168$) es 48
- = ($\text{mcd } 48 \ 24$) pues el ($\text{remainder } 168 \ 48$) es 24
- = ($\text{mcd } 24 \ 0$) pues el ($\text{remainder } 48 \ 24$) es 0
- = 24

De esta forma se puede construir la siguiente función:

```
;; Función que encuentra el
;; máximo común divisor entre a y b
;; Precondiciones:
;; a,b: números enteros
;;
(define (mcd a b)
  (cond ((zero? b)
         a)
        (else
         (mcd b (remainder a b)))))
```

La siguiente función que se desea construir se llamará (hacer-rac num den), que significa hacer un racional. Esta función recibe como argumentos dos números enteros que representan el numerador y el denominador. Simplifica la expresión y la almacena en una lista cuyo primer elemento representa el numerador y cuyo segundo elemento representa el denominador. Debe producir los siguientes resultados.

> (hacer-rac 12 6)

(2 1)

> (hacer-rac 18 6)

(3 1)

> (hacer-rac 18 12)

(3 2)

> (hacer-rac 15 12)

(5 4)

A continuación se presenta el programa que realiza dicha función.

```
;; Construye un número racional
;; con base en en numerador, num
;; y el denominador den
;; Precondiciones
;; num, den: números enteros
;;
(define (hacer-rac n d)
  (let ( (maxcd (mcd n d))
         )
    (list (/ n maxcd)
          (/ d maxcd))))
```

Además de la función que construye los números racionales se necesitan dos funciones auxiliares que extraen información. Se construyen dos, la función de nombre num y otra de nombre den, para obtener el numerador y el denominador de la estructura creada por hacer-rac.

```
;; Devuelve la parte del numerador
;; de un número racional
;; Precondiciones
;; q: es un número racional
;;
(define (num q)
  (car q))
;;
;; Devuelve la parte del denominador
;; de un número racional
```

```
;; Precondiciones
;; q: es un número racional
;;
(define (den q)
  (cadr q))
```

Para implementar las operaciones de los números racionales, se utilizan las reglas aritméticas usuales:

$$\frac{n1}{d1} = \frac{n2}{d2} \text{ si } (n1 * d2) = (d1 * n2)$$

$$\frac{n1}{d1} + \frac{n2}{d2} = \frac{(n1 * d2) + (d1 * n2)}{(d1 * d2)}$$

$$\frac{n1}{d1} - \frac{n2}{d2} = \frac{(n1 * d2) - (d1 * n2)}{(d1 * d2)}$$

$$\frac{n1}{d1} * \frac{n2}{d2} = \frac{(n1 * n2)}{(d1 * d2)}$$

$$\frac{n1}{d1} / \frac{n2}{d2} = \frac{(n1 * d2)}{(d1 * n2)}$$

Se desea construir un conjunto de funciones que realicen los siguientes procesos:

```
> (define a (hacer-rac 12 6))
> (define b (hacer-rac 4 2))
> (equal-rac? a b)
#t
> (define a (hacer-rac 3 2))
> (define b (hacer-rac 4 5))
> (sum-rac a b)
(23 10)
```

> (sub-rac a b)

(7 10)

> (mul-rac a b)

(6 5)

> (div-rac a b)

(15 8)

A continuación se implementan los programas para cada una de las operaciones descritas.

;; Determina si dos números

;; racionales, q1 y q2 son iguales

;;

(define (equal-rac? q1 q2)

 (equal? (* (num q1) (den q2))
 (* (den q1) (num q2))))

;; Calcula la suma de dos racionales

;;

(define (sum-rac q1 q2)

 (hacer-rac (+ (* (num q1) (den q2))
 (* (den q1) (num q2)))
 (* (den q1) (den q2))))

;; Calcula la resta de dos racionales

;;

(define (sub-rac q1 q2)

 (hacer-rac (- (* (num q1) (den q2))
 (* (den q1) (num q2)))
 (* (den q1) (den q2))))

;; Calcula la multiplicación de dos racionales

;;

(define (mul-rac q1 q2)

 (hacer-rac (* (num q1) (num q2))
 (* (den q1) (den q2))))

;; Calcula la división de dos racionales

;;



```
(define(div-rac q1 q2)
  (hacer-rac (* (num q1) (den q2))
             (* (den q2) (num q1))))
```

Si se desea hacer un cambio a la representación de los números racionales es suficiente cambiar las funciones hacer-rac, den y num.

Por ejemplo si para representar el número racional de 1/2 en lugar de utizar la estructura de (1 2) se utilizara la estructura (1 / 2) se debe modificar el código de estas funciones, pero NO es necesario modificar el código de las otras. Por lo tanto si se desea reprentar números de esta forma:

```
> (hacer-rac 1 2)
(1 / 2)

> (hacer-rac 18 12)
(3 / 2)

> (hacer-rac 15 12)
(5 / 4)
```

Será necesario realizar las modificaciones:

```
;; Construye un número racional
;; con base en en numerador, num
;; y el denominador den
;; Precondiciones
;; num, den: números enteros
;;
(define (hacer-rac n d)
  (let ( (maxcd (mcd n d))
        )
    (list (/ n maxcd)
          '/'
          (/ d maxcd)))))

;; Devuelve la parte del numerador
;; de un número racional
;; Precondiciones
;; q: es un número racional
;;
```

```
(define (num q)
  (car q))

;; Devuelve la parte del denominador
;; de un número racional
;; Precondiciones
;; q: es un número racional
;;
(define (den q)
  (caddr q))
```

Esta forma de programar se conoce como programación en capas. Para ello se deben utilizar abstracciones sucesivas de manera tal que que se programa cada abstracción dentro de una capa o nivel.

capa 2: sum-rac / res-rac / mul-rac / div-rac

capa 1: hacer-rac / den / num

capa 0: estructura: (d / n)

Miembro de una lista

Algunas de las funciones que se presentan a continuación se encuentran dentro del ambiente de programación del lenguaje. Sin embargo resulta útil construirlas para comprender el algoritmo en el cual se basan.

La función *miembro?* es una función que recibe dos argumentos. Un elemento y una lista. En caso que el elemento se encuentre dentro de la lista devuelve un valor de #t, en caso contrario devuelve el valor de #f. Existe una función que realiza un proceso similar y se denomina *MEMBER*. La función *miembro?* debe producir las siguientes respuestas:

```
> (miembro? 1 '( ))
#f

> (miembro? 1 '(2 3 4 5))
#f
```

```

> (miembro? 1 '(1 2 3 4))
#t
> (miembro? 1 '(2 3 1 5 6))
#t
> (miembro? 'a '())
#f
> (miembro? 'a '(b c d e))
#f
> (miembro? 'a '(a b c d))
#t
> (miembro? 'a '(b c a d e))
#t
> (miembro? '(1 2) '((1 1) (1 2) (1 3)))
#t

```

Para construir esta función se debe utilizar el siguiente algoritmo. Si el elemento deseado ele es igual al primer elemento de la lista, se debe devolver un valor de #t. En caso contrario se debe seguir el proceso de buscar ele en el resto de la lista. Cuando la lista se acaba, se devuelve el valor de #f.

```

> (miembro? 'x '(a b c x d))
  (miembro? 'x '(b c x d))
  (miembro? 'x '(c x d))
  (miembro? 'x '(x d))
#t
> (miembro? 'x '(a b c d))
  (miembro? 'x '(b c d))
  (miembro? 'x '(c d))
  (miembro? 'x '(d))
  (miembro? 'x '())
#f

```

El código de dicha función se puede escribir de la siguiente manera:

;; Encuentra si un elemento se encuentra en una lista
 ;; Precondiciones:

```
;; ele: un símbolo cualquiera
;; lista: una lista lineal
;;
(define (miembro? ele lista)
  (cond ((null? lista)
         #f)
        ((equal? ele (car lista))
         #t)
        (else
         (miembro? ele (cdr lista)))))
```

Largo de una lista

La función *largo* recibe como parámetro una lista y debe contar el número de elementos que tiene esa lista. Existe una función que realiza este proceso y se denomina *LENGTH*. Se espera que produzca los siguientes resultados:

```
> (largo '())
0
> (largo '(a))
1
> (largo '(a b))
2
> (largo '(a b c d))
4
> (largo '(a b c d e))
5
```

Para construir esta función se utiliza un método muy sencillo. Se debe iniciar con un contador en 0 y sumar un 1 por cada elemento que posea la lista. Para llevar a cabo este procedimiento se hará de la siguiente manera, sumo 1 por el primer elemento de la lista y luegouento el largo del resto de la lista, se continua este proceso hasta que la lista quede vacía en cuyo caso sumo un 0.

```
> (largo '(a b c d))
(+ 1 (largo '(b c d)))
```

```
(+ 1 (+ 1 (largo '(c d))))
(+ 1 (+ 1 (+ 1 (largo '(d))))))
(+ 1 (+ 1 (+ 1 (+ 1 (largo '( ))))))
(+ 1 (+ 1 (+ 1 (+ 1 0)))))
(+ 1 (+ 1 (+ 1 1)))
(+ 1 (+ 1 2))
(+ 1 3)
4
```

Código de la función largo:

```
;; Encuentra el largo de una lista
;; lista: lista de elementos
;;
(define (largo lista)
  (cond ((null? lista)
         0)
        (else
         (+ 1 (largo (cdr lista))))))
```

Extraer el n-ésimo elemento

Se presentará la función elemento que recibe dos parámetros, un índice y una lista. Dicha función debe extraer el elemento indicado por el índice de la lista. En caso que la lista sea de menor tamaño que el índice debe devolver el valor de #f. Existe una función que realiza un proceso similar de nombre *LIST-REF*. De manera tal que:

```
> (elemento 1 '(a b c d))
a
> (elemento 2 '(a b c d))
b
> (elemento 3 '(a b c d))
c
> (elemento 2 '((a a) (b b) (c c)))
(b b)
```

Para construir esta función se utiliza el siguiente razonamiento. Extraer el ~~n~~º elemento es igual que quitarle el primer elemento a la lista y solicitar que ~~se~~ extraiga el elemento $n-1$ del resto de la lista. Este proceso continua hasta que ~~se~~ solicite extraer el 1er elemento de la lista. En caso que la lista se agotará antes de alcanzar el valor de 1 se debe devolver el valor de #f. De igual forma número ~~del~~ elemento que se indica para ser extraído debe ser un número positivo mayor que cero, en caso contrario se debe devolver el valor de #f.

- > (elemento 3 '(a b c d e))
 (elemento 2 '(b c d e))
 (elemento 1 '(c d e))
 c

A continuación se presenta el código.

```
;; Extrae el elemento situado
;; en la posición index de la lista
;; la lista inicia con el índice 1
;; index: número entero positivo, index>0
;; lista: una lista lineal
;;
(define (elemento index lista)
  (cond ((null? lista)
          #f)
        ((equal? index 1)
         (car lista))
        (else
         (elemento (- index 1) (cdr lista)))))
```

Extraer el último elemento

La función *último*, como su nombre lo indica, debe devolver el último elemento de una lista cualquiera. Recibe un único argumento y en caso que se trate de la lista vacía debe devolver el valor de #f. Su comportamiento debe ser:

- > (último '())
 #f

```
> (último '(a b c))
c
> (último '(a b c d))
d
```

Una forma de resolver este problema consistiría en contar el número de elementos de la lista y posteriormente extraer el elemento que corresponde a dicho número. Es decir si una lista tiene n elementos, se debe extraer elemento n -ésimo. Este solución se puede implementar de la siguiente forma:

```
;; Una 1era versión para
;; extraer el último elemento de una lista
;; lista: una lista lineal
;;
(define (último lista)
  (elemento (largo lista) lista))
```

A pesar que la función anterior es muy sencilla tiene un inconveniente. Cuando se cuentan los elementos de la lista se debe recorrer la misma una vez. Posteriormente, cuando se extrae el último elemento se debe volver a recorrer la lista.

Es posible construir un algoritmo que realice un único recorrido sobre la lista. Se debe revisar si después de extraer el elemento actual la lista queda vacía, de ser así se trata del último elemento. En caso contrario se extrae el primer elemento y se repite la secuencia anterior. Este proceso se puede escribir como:

La función `último` realizará lo siguiente:

```
> (último '(a b c d e f))
(último '(b c d e f))
(último '(c d e f))
(último '(d e f))
(último '(e f))
(último '(f))
f
```

El código de la función `último` es el siguiente:

```
;; Una versión mejorada
;; para extraer el último elemento de una lista
```



```
;; lista: una lista lineal
;;
(define (último lista)
  (cond ((null? lista)
         #f)
        ((null? (cdr lista))
         (car lista))
        (else
         (último (cdr lista)))))
```

Sacar los números pares

Pares es una función que recibe una lista de números enteros y devuelve como resultado únicamente los números pares que se encuentran en la lista.

```
> (pares '(1 2 3 4))
(2 4)
> (pares '(1 2 3 4 5 6))
(2 4 6)
> (pares '(6 3 8 4 2 1 8))
(6 8 4 2 8)
```

Para ello utilizamos el siguiente algoritmo. Se toma el primer elemento de la lista, si es par se agrega a una lista en construcción mediante la instrucción de cons, si no es par se desecha. Luego se continua este proceso con el resto de la lista hasta que la lista se acabe.

```
> (pares '(1 2 3 4 5 6))
(pares '(2 3 4 5 6))
(cons 2 (pares '(3 4 5 6)))
(cons 2 (pares '(4 5 6)))
(cons 2 (cons 4 (pares '(5 6))))
(cons 2 (cons 4 (pares '(6))))
(cons 2 (cons 4 (cons 6 (pares '()))))
(cons 2 (cons 4 (cons 6 '())))
(cons 2 (cons 4 '(6)))
(cons 2 '(4 6))
(2 4 6)
```

Este es el código:

```
;; Función que devuelve los números pares
;; Precondiciones:
;; lista: lista lineal de números enteros
;;
(define (pares lista)
  (cond ((null? lista)
         '())
        ((even? (car lista))
         (cons (car lista)
               (pares (cdr lista))))
        (else
         (pares (cdr lista)))))
```

Pegar dos listas

La función *pegar* recibe como argumentos dos listas y debe devolver una única lista. La lista que produce mantiene el orden indicado en las listas originales. En Scheme existe una función que realiza un proceso similar y se denomina APPEND. La diferencia fundamental entre APPEND y Pegar es que APPEND puede realizar el proceso para más de dos listas, en cambio la función aquí presentada trabaja únicamente con dos parámetros. Pegar debe realizar el siguiente procedimiento:

```
> (pegar '() '())
()
> (pegar '(a) '())
(a)
> (pegar '() '(a))
(a)
> (pegar '(a b c) '(d e))
(a b c d e)
> (pegar '(a b c) '(d e f))
(a b c d e f)
```

Para resolver este problema se utilizará el siguiente algoritmo. Pegar una lista vacía con otra cualquiera es muy sencillo, sólo se devuelve la otra lista. Si las dos listas son no vacías, se toma el primer elemento de la primera lista y se inicia la construcción de una nueva lista, este proceso continúa hasta que se agaben los elementos de la primera lista. Cuando esto ocurre, se tiene el caso inicial donde una lista es vacía y la otra no.

```
> (pegar '(a b c d) '(e f g))
(cons a (pegar '(b c d) '(e f g)))
(cons a (cons b (pegar '(c d) '(e f g))))
(cons a (cons b (cons c (pegar '(d) '(e f g))))))
(cons a (cons b (cons c (cons d (pegar '() '(e f g))))))
(cons a (cons b (cons c (cons d '(e f g))))))
(cons a (cons b (cons c '(d e f g))))
(cons a (cons b '(c d e f g)))
(cons a '(b c d e f))
(a b c d e f)
```

A continuación se presenta el código.

```
;; Pegar une dos listas manteniendo el orden
;; lista1, lista2: listas lineales
;;
(define (pegar lista1 lista2)
  (cond ((null? lista1)
         lista2)
        ((null? lista2)
         lista1)
        (else
         (cons (car lista1)
               (pegar (cdr lista1)
                     lista2)))))
```

Invertir una lista

La función *invertir* recibe una lista y produce una lista en la cual los elementos van ordenados de forma inversa a la lista original. Existe una función en Scheme

que realiza esta función y se denomina REVERSE. Invertir debe producir los siguientes resultados.

> (invertir ' ())
' ()
> (invertir ' (a))
(a)
> (invertir ' (a b c))
(c b a)
> (invertir ' (a b c d e))
(e d c b a)

Para construir esta función, se hará uso de otras funciones definidas con anterioridad. Invertir una lista nula es sencillo, pues de debe producir ese mismo valor. Invertir una lista no nula es equivalente a tomar el primer elemento, invertir el resto de la lista y volver a poner ese primer elemento de último.

(append invertir ‘(b c d))	(define (invertir ‘(c d))
‘(a))	‘(b))
(append (append (append (invertir ‘(d))	‘(c))
‘(b))	‘(a))
(append (append (append (append (invertir ‘())	‘(d))
‘(a))	‘(c))
(append (append (append (append (append (invertir ‘()	‘(d))
‘(a))	‘(b))
‘())	‘())

Aplicar una función

Aplicar-función
cuyo segundo argumento es
los elementos de la lista. Es
una labor similar a la función
funciones de los lenguajes
Aplicar-función para aplicar las
funciones de los lenguajes
que implementan la

```

        ‘(b))
‘(a))
(append (append (append ‘(d)
                           ‘(c))
                           ‘(b))
                           ‘(a))
(append (append ‘(d c))
                           ‘(b))
                           ‘(a))
(append ‘(d c b))
                           (a))
(d c b a)

```

A continuación se presenta el código de esta función.

```

;; Invertir una lista
;; lista: lista lineal
;;
(define (invertir lista)
  (cond ((null? lista)
         ‘( ))
        (else
          (append (invertir (cdr lista))
                  (list (car lista)))))))

```

Aplicar una función a los elementos de una lista

Aplicar-f es una función que recibe como primer argumento una función y cuyo segundo argumento es una lista. Debe aplicar la función que recibe a todos los elementos de la lista. En Scheme existe una función llamada MAP que realiza una labor similar, la función que se presenta aquí puede recibir únicamente funciones de una variable, MAP puede recibir funciones de múltiples variables. *Aplicar-f* debe producir los siguientes resultados:

```

> (aplicar-f suc ‘( ))
‘( )

```

```
> (aplicar-f suc '(1 2 3))
(2 3 4)

> (aplicar-f cuadrado '( ))
()

> (aplicar-f cuadrado '(1 2 3))
(1 4 9)
```

Para construir esta función se utiliza el siguiente algoritmo. Se toma la función y se aplica al primer elemento de la lista y se continúa así hasta que se acaben todos los elementos de la lista.

```
> (aplicar-f cuadrado '(1 2 3))
  (cons 1 (aplicar-f cuadrado '(2 3)))
  (cons 1 (cons 4 (aplicar-f cuadrado '(3))))
  (cons 1 (cons 4 (cons 9 (aplicar-cuadrado '( )))))
  (cons 1 (cons 4 (cons 9 '( ))))
  (cons 1 (cons 4 '(9)))
  (cons 1 '(4 9))
(1 4 9)
```

A continuación el programa:

```
;; Aplica una función
;; a todos los elementos de una lista
;; fun: función de un argumento de scheme
;; lista: lista lineal
;;
(define (aplicar-f fun lista)
  (cond ( (null? lista)
           '())
        ( else
          (cons (fun (car lista))
                (aplicar-f fun (cdr lista))))))
```

Encontrar el mayor de una lista

Se construirá una función llamada *mayor* que recibe como argumento una lista. En esta lista debe existir alguna relación de orden entre sus elementos. El

objetivo de la función es indicar cuál es el elemento mayor que existe dentro de esa lista. El comportamiento de esta función es:

```
> (mayor '( ))  
( )  
> (mayor '(1))  
1  
> (mayor '(1 2 3))  
3  
> (mayor '(1 2 3 4 3))  
4  
> (mayor '(1 2 5 3 2))  
5
```

Para construir esta función se utiliza el siguiente algoritmo. Se invocará a una función auxiliar que tiene como argumentos un elemento mayor provisional y la lista que debe evaluarse. Se selecciona temporalmente como el mayor elemento el primer elemento de la lista. Se continua con los demás elementos de la siguiente forma, si el siguiente elemento es mayor que el elemento escogido se debe tomar como el nuevo elemento mayor, en caso contrario se continua este proceso con los demás elementos hasta que se acaba la lista. Finalmente cuando la lista se termina se debe devolver el elemento escogido que haya quedado en la posición del mayor.

```
> (mayor '(1 2 5 3 2))  
(mayor-aux 1 '(2 5 3 2))  
(mayor-aux 1 '(5 3 2))  
(mayor-aux 5 '(3 2))  
(mayor-aux 5 '(2))  
(mayor-aux 5 '( ))  
5
```

El programa resultante es el siguiente:

```
;; Encuentra el mayor elemento de una lista  
;; lista: lista lineal  
;;  
(define (mayor lista)
```

```
(cond ( (null? lista)
        #f)
     ( else
       (mayor-aux (car lista) (cdr lista)))))

(define (mayor-aux ele lista)
  (cond ( (null? lista)
           ele)
        ( (< ele (car lista))
           (mayor-aux (car lista) (cdr lista)))
        ( else
           (mayor-aux ele (cdr lista)))))
```

Encontrar el menor de una lista

Se construirá una función llamada *menor* que recibe como argumento una lista. En esta lista debe existir alguna relación de orden entre sus elementos. El objetivo de la función es indicar cuál es el elemento menor que existe dentro de esa lista. El comportamiento de esta función es:

```
> (menor '( ))
()
> (menor '(1))
1
> (menor '(1 2 3))
1
> (menor '(3 2 1 2 3 4 ))
1
> (menor '(2 5 3 2))
2
```

Para construir esta función se utiliza el mismo algoritmo que para encontrar el mayor elemento con la única excepción que el elemento temporalmente seleccionado se compara por la relación de menor estricto.

```
> (menor '(4 2 5 3 2))
(menor-aux 4 '(2 5 3 2))
```

```
(menor-aux 2 '(5 3 2))
(menor-aux 2 '(3 2))
(menor-aux 2 '(2))
(menor-aux 2 '())
2
```

El algoritmo se puede implementar de la siguiente forma:

```
:: Encuentra el mayor elemento de una lista
:: lista: lista lineal
::
(define (menor lista)
  (cond ( (null? lista)
          #f)
        ( else
          (menor-aux (car lista) (cdr lista)))))

(define (menor-aux ele lista)
  (cond ( (null? lista)
          ele)
        ( (< (car lista) ele)
          (menor-aux (car lista) (cdr lista)))
        ( else
          (menor-aux ele (cdr lista)))))
```

Eliminar un elemento de una lista

La función eliminar-prm recibe como argumentos un elemento y una lista. Su función consiste en crear una nueva lista donde se haya eliminado la primera aparición del elemento indicado.

```
> (eliminar-prm 1 '(1))
'( )
> (eliminar-prm 1 '(1 2 3))
(2 3)
> (eliminar-prm 1 '(1 1 2 2 3 3))
(1 2 2 3 3)
```

```
> (eliminar-prm 1 '(3 2 1 3 2 1))
(3 2 3 2 1)
```

Para construir esta función se toma el primer elemento de la lista, si es el elemento buscado se devuelve el resto de la lista, en caso contrario se pone este elemento dentro de la nueva lista y se repite el paso anterior hasta que no encuentre el elemento o se acabe la lista.

```
> (eliminar-prm 1 '(3 2 4 1 3 2 1))
(cons 3 (eliminar-prm 1 '(2 4 1 3 2 1)))
(cons 3 (cons 2 (eliminar-prm 1 '(4 1 3 2 1))))
(cons 3 (cons 2 (cons 4 (eliminar-prm 1 '(1 3 2 1)))))
(cons 3 (cons 2 (cons 4 '(3 2 1))))
(cons 3 (cons 2 '(4 3 2 1)))
(cons 3 ('2 4 3 2 1))
(3 2 4 3 2 1)
```

Se obtiene entonces el siguiente programa.

```
;; Se elimina la 1era aparición
;; de ele en una lista
;;
;; Precondiciones:
;; ele: un símbolo
;; lista: una lista lineal
;;
(define (eliminar-prm ele lista)
  (cond ((null? lista)
         '())
        ((equal? ele (car lista))
         (cons (cdr lista)
               (else
                 (cons (car lista)
                       (eliminar-prm ele (cdr lista)))))))
        (else
          (cons (car lista)
                (eliminar-prm ele (cdr lista)))))))
```

Es fácil construir una función similar que elimine todas las apariciones de un elemento en una lista, el comportamiento de esta función debe ser:

```
> (eliminar 1 '(1))
()
> (eliminar 1 '(1 2 3))
```

```
(2 3)
> (eliminar 1 '(1 1 2 2 3 3))
(2 2 3 3)
> (eliminar 1 '(3 2 1 3 2 1))
(3 2 3 2)
```

Para eliminar todas las apariciones del elemento en la lista una vez que se haya encontrado la primera aparición se debe invocar recursivamente la función para eliminar las demás apariciones del mismo.

```
> (eliminar 1 '(3 2 4 1 3 2 1))
  (cons 3 (eliminar 1 '(2 4 1 3 2 1)))
  (cons 3 (cons 2 (eliminar 1 '(4 1 3 2 1))))
  (cons 3 (cons 2 (cons 4 (eliminar 1 '(1 3 2 1)))))
  (cons 3 (cons 2 (cons 4 (eliminar 1 '(3 2 1)))))
  (cons 3 (cons 2 (cons 4 (cons 3 (eliminar 1 '(2 1))))))
  (cons 3 (cons 2 (cons 4 (cons 3 (cons 2 (eliminar 1 '(1)))))))
  (cons 3 (cons 2 (cons 4 (cons 3 (cons 2 (eliminar 1 '( )))))))
  (cons 3 (cons 2 (cons 4 (cons 3 (cons 2 ('( )))))))
  (cons 3 (cons 2 (cons 4 (cons 3 (cons 2 ('( )))))))
  (cons 3 (cons 2 (cons 4 (cons 3 ('(2))))))
  (cons 3 (cons 2 (cons 4 ('(3 2)))))
  (cons 3 (cons 2 ('(4 3 2))))
  (cons 3 ('(2 4 3 2)))
  (3 2 4 3 2)
```

Esto generaría el siguiente código:

```
;; Se elimina todas las apariciones
;; de ele en una lista
;; Precondiciones:
;; ele: un símbolo
;; lista: una lista lineal
;;
(define (eliminar ele lista)
  (cond ( (null? lista)
          '())
        ( (equal? ele (car lista))
          (eliminar ele (cdr lista))))
```

```
( else
  (cons (car lista)
    (eliminar ele (cdr lista)))))
```

Ordenar una lista

La función *ordenar* recibe como parámetro una lista. Los elementos de esta lista deben estar asociados mediante alguna relación de orden. Es decir, tiene que existir un mecanismo que permita establecer si un elemento es menor, igual o mayor que otro. En los ejemplos que se presentan se utilizarán números y se utilizará la relación de orden menor o igual.

Ordenar produce una nueva lista donde sus elementos se encuentran enumerados del menor al mayor. Algunos ejemplos de su funcionamiento son:

```
> (ordenar '( ))
()
> (ordenar '(1 2 3))
(1 2 3)
> (ordenar '(3 2 1))
(1 2 3)
> (ordenar '(2 3 4 1 1 2 5))
(1 1 2 2 3 4 5)
```

Una forma intuitiva y relativamente sencilla de construir esta función es utilizar las funciones de menor y eliminar, donde la función eliminar elimina únicamente la primera aparición de un elemento dentro de una lista. Para construir la lista ordenada se selecciona el menor elemento y se elimina de la lista, se continua este proceso hasta que la lista original quede vacía. A continuación se presenta un ejemplo de como trabaja.

```
> (ordenar '(2 2 1 3 4))
(cons 1
  (ordenar '(2 2 3 4)))
(cons 1
  (cons 2
    (ordenar '(2 3 4))))
```

```
(cons 1
      (cons 2
              (cons 2
                      (cons 2
                              (ordenar '(3 4)))))

(cons 1
      (cons 2
              (cons 2
                      (cons 2
                              (cons 3
                                      (ordenar '(4))))))

(cons 1
      (cons 2
              (cons 2
                      (cons 2
                              (cons 3
                                      (cons 4
                                              (ordenar '(()))))))

(cons 1
      (cons 2
              (cons 2
                      (cons 2
                              (cons 3
                                      (cons 4
                                              '(()))))))

(cons 1
      (cons 2
              (cons 2
                      (cons 2
                              '(3 4)))))

(cons 1
      (cons 2
              '(2 3 4)))
```

```
(cons 1
  '(2 2 3 4))
(1 2 2 3 4)
```

Aquí está la representación del algoritmo anterior:

```
;; Se ordenan de menor a mayor
;; los elementos de una lista lineal de números
;; Precondiciones:
;; lista: lista lineal de números
;;
(define (ordenar lista)
  (cond ((null? lista)
         '())
        (else
         (cons (menor lista)
               (ordenar (eliminar-prm
                          (menor lista)
                          lista)))))))
```

El programa anterior puede mejorarse utilizando la función de let. De esta forma es suficiente recorrer la lista únicamente una vez para encontrar el elemento menor.

```
;; Se ordenan de menor a mayor
;; los elementos de una lista lineal de números
;; Precondiciones:
;; lista: lista lineal de números
;;
(define (ordenar lista)
  (cond ((null? lista)
         '())
        (else
         (let ((minn (menor lista)))
           (cons minn
                 (ordenar (eliminar-prm minn lista)))))))
```

Ordenar una lista por inserción

Se presenta ahora un método de ordenamiento de listas denominado *insert-sort*. Este procedimiento mantiene una lista ordenada y cada vez que se agrega un nuevo elemento se mantiene el orden de la lista.

Inicialmente se construirá una función que inserta elementos en una lista manteniendo el orden. Posteriormente se construirá un mecanismo que utiliza esta función para ordenar una lista cualquiera.

La primera función se denomina *insertar-lista-ordenada* y debe tener el siguiente comportamiento:

```
> (insertar-lista-ordenada 1 '( ))  
(1)  
> (insertar-lista-ordenada 1 '(1 2 3))  
(1 1 2 3)  
> (insertar-lista-ordenada 4 '(1 2 3 5 6))  
(1 2 3 4 5 6)  
> (insertar-lista-ordenada 4 '(1 2 3))  
(1 2 3 4)
```

Para construir esta función se utiliza el siguiente algoritmo. Insertar un elemento en una lista vacía produce una lista con dicho elemento. Si el elemento que se desea insertar es mayor que el primer elemento de la lista se debe insertar en el resto de la lista, en caso contrario se debe insertar como primer elemento de la lista.

```
> (insertar-lista-ordenada 4 '(1 2 3 5 6))  
(cons 1 (insertar-lista-ordenada 4 '(2 3 5 6)))  
(cons 1 (cons 2 (insertar-lista-ordenada 4 '(3 5 6))))  
(cons 1 (cons 2 (cons 3 (insertar-lista-ordenada 4 '(5 6))))))  
(cons 1 (cons 2 (cons 3 '(4 5 6))))  
(cons 1 (cons 2 '(3 4 5 6)))  
(cons 1 '(2 3 4 5 6))  
(1 2 3 4 5 6)
```

Este proceso se puede describir mediante el siguiente código.

```
;; Inserta un elemento ele  
;; en una lista ordenada de nombre lista
```

```

;; ele: número
;; lista: lista lineal de números
;;
(define (insertar-lista-ordenada ele lista)
  (cond ( (null? lista)
           (list ele))
        ( (> ele (car lista))
           (cons (car lista)
                 (insertar-lista-ordenada ele
                                           (cdr lista))))
        ( else
           (cons ele lista)))))


```

Si se tiene una lista desordenada se puede utilizar la función anterior para ordenarla. Para hacer esto se van tomando los elementos de la lista desordenada y se van insertando uno a uno en una lista vacía, al final de este proceso se tendrá una lista ordenada. A continuación se presenta un función llamada *insert-sort* que realiza este proceso. *Insert-sort* funciona igual que la función anterior de ordenamiento, pero utiliza su propio algoritmo.

```

> (insert-sort '())
()
> (insert-sort '(1 2 3))
(1 2 3)
> (insert-sort '(3 2 1))
(1 2 3)
> (insert-sort '(2 3 4 1 1 2 5))
(1 1 2 2 3 4 5)


```

A continuación se presenta un ejemplo de como trabaja el *insert-sort*.

```

> (insert-sort '(2 2 1 3 4))
(insert-sort-aux '(2 2 1 3 4) '())
(insert-sort-aux '(2 1 3 4) '(2))
(insert-sort-aux '(1 3 4) '(2 2))
(insert-sort-aux '(3 4) '(1 2 2))
(insert-sort-aux '(4) '(1 2 2 3))


```

```
(insert-sort-aux '( ) '(1 2 2 3 4))  
(1 2 2 3 4)
```

La función Scheme resultante es:

```
;; Ordena una lista poniendo los elementos  
;; en una lista que ya tiene orden  
;; lista: lista lineal de números  
;;  
(define (insert-sort lista)  
  (cond ( (null? lista)  
          '( ))  
        ( else  
          (insert-sort-aux lista '( )))))  
  
(define (insert-sort-aux lista resultado)  
  (cond ( (null? lista)  
          resultado)  
        ( else  
          (insert-sort-aux  
            (cdr lista)  
            (insertar-lista-ordenada (car lista)  
                                      resultado))))))
```

Ordenar una lista por particiones sucesivas

Se presenta ahora un método de ordenamiento de listas denominado *quick-sort*. Este procedimiento toma el primer elemento de una lista y lo utiliza como un punto de pivote para partir la lista en dos grupos. Un primer grupo de elementos menores y otro grupo de elementos mayores con respecto al pivote. Ordena cada uno de estos grupos y produce una lista donde primero van los elementos menores, después el elemento de pivote y por último los elementos mayores. El algoritmo se vuelve a aplicar a cada nueva partición hasta que se obtiene una lista vacía. Así ordenar la lista vacía da como resultado lista vacía.

Para implementar esta función se construirá una función auxiliar denominada *pivotd*. Esta función recibe como argumento una lista y produce una lista con dos componentes. El primer componente es una lista de los elementos menores

que el pivote, el segundo componente es otra lista con los elementos mayores que el pivote. Pivot funciona de la siguiente manera:

```
> (pivot '( ))
#f

> (pivot '(1 2 3 4 5))
(( ) (5 4 3 2))

> (pivot '(5 4 3 1 2))
((2 1 3 4) ( ))

> (pivot '(3 1 4 5 2))
((2 1) (5 4))

> (pivot '(3 1 4 5 6 2))
((2 1) (6 5 4))
```

En el ejemplo que se presenta a continuación pivot devuelve las listas invertidas. Si se desea se puede usar reverse para corregir esto, sin embargo es innecesario pues ambas listas están desordenadas. No tiene caso invertirlas ya que se gastará tiempo computacional y no se puede garantizar que siempre salgan ordenadas.

```
> (pivot '(3 1 4 6 5 2))
(pivot-aux 3 '(1 4 6 5 2) '( ) '( ))
(pivot-aux 3 '(4 6 5 2) '(1) '( ))
(pivot-aux 3 '(6 5 2) '(1) '(4))
(pivot-aux 3 '(5 2) '(1) '(6 4))
(pivot-aux 3 '(2) '(1) '(5 6 4))
(pivot-aux 3 '( ) '(2 1) '(5 6 4))
((2 1) (5 6 4))
```

La función pivot se puede implementar de la siguiente forma:

```
;; La función de pivot
;; Toma el primer elemento de la lista
;; y separa a los menores y mayores
;; lista: lista lineal de números
;;
(define (pivot lista)
  (cond ((null? lista)
```

```

#f)
( else
  (pivot-aux (car lista) (cdr lista) '( ) '( ) ))))

(define (pivot-aux punto lista menores mayores)
  (cond ( (null? lista)
           (list menores mayores))
        ( (<= (car lista) punto)
           (pivot-aux punto
                      (cdr lista)
                      (cons (car lista) menores)
                      mayores))
        ( else
          (pivot-aux punto
                      (cdr lista)
                      menores
                      (cons (car lista) mayores))))))

```

La función de quick-sort proporciona la misma salida que las otras funciones de ordenamiento, pero utiliza su propio algoritmo. Su comportamiento está dado por:

```

> (quick-sort '( ))
()
> (quick-sort '(1 2 3))
(1 2 3)
> (quick-sort '(3 2 1))
(1 2 3)
> (quick-sort '(2 3 4 1 1 2 5))
(1 1 2 2 3 4 5)

```

A continuación se presenta un ejemplo de como trabaja el quick-sort. Inicialmente se explica el proceso que realiza quick-sort cuando recibe como argumento uno o dos números, posteriormente con una lista de mayor tamaño.

```

> (quick-sort '(1))
(append (quick-sort '( ))
        '(1)
        (quick-sort '( )))

```

```
(append '( )
        '(1)
        (quick-sort '( )))

(append '( )
        '(1)
        '( ))

(1)

> (quick-sort '(2 1))
(append (quick-sort '(1))
        '(2)
        (quick-sort '( )))

(append '(1)
        '(2)
        (quick-sort '( )))

(append '(1)
        '(2)
        '( ))

(1 2)

> (quick-sort '(2 2 1 3 4))

(append (quick-sort '(1 2))
        '(2)
        (quick-sort '(4 3)))

(append (append (quick-sort '( ))
                  '(1)
                  (quick-sort '(2)))
        '(2)
        (quick-sort '(4 3)))

(append (append '( )
                  '(1)
                  (quick-sort '(2)))
        '(2)
        (quick-sort '(4 3)))

(append (append '( )
                  '(1)
                  (quick-sort '(2)))
        '(2)
        (quick-sort '(4 3)))

(append (append '( )
                  '(1)
```

```

‘(2))
‘(2)
(quick-sort ‘(4 3)))
(append ‘(1 2)
‘(2)
(quick-sort ‘(4 3)) )
(append ‘(1 2)
‘(2)
(append (quick-sort ‘(3))
‘(4)
(quick-sort ‘( ))))
(append ‘(1 2)
‘(2)
(append ‘(3)
‘(4)
(quick-sort ‘( ))))
(append ‘(1 2)
‘(2)
(append ‘(3)
‘(4)
‘( )))
(append ‘(1 2)
‘(2)
‘(3 4))
(1 2 2 3 4)

```

Para implementar el quick-sort se utiliza la función pivot anterior y se realizan particiones sucesivas de la lista. Observe que para construir esta función se utilizan además las funciones de let* y append. Se obtiene entonces el siguiente código.

```

;; Realiza un Quick-Sort
;; de una lista de números
;; lista: lista lineal de números
;;

```

```
(define (quick-sort lista)
  (cond ( (null? lista)
           '())
        ( else
          (let*
            ( (punto             (car lista))
              (menores-mayores (pivot lista))
              (menores          (car menores-mayores))
              (mayores          (cadr menores-mayores))
            )
            (append (quick-sort menores)
                    (list punto)
                    (quick-sort mayores)))))))
```



Resumen

- *Car* y *cdr* se utilizan para destruir listas.
- *Cons* y *list* se utilizan para construir listas.
- *List?*, *equal?* y *null?* son predicados para listas.
- Muchas funciones realizan operaciones sobre las listas. Como *miembro?*, *largo*, *pegar*, *invertir*, etc. Algunas de ellas se encuentran definidas en el ambiente de programación.
- Existen varios algoritmos para ordenar listas. Se han presentado el ordenamiento por extracciones sucesivas del mínimo, *insert-sort* y *quick-sort*.

Ejercicios

Ejercicio N° 1

Investigue si (*hacer-rac num den*) representa los números racionales en una forma normalizada. Esto significa que los números racionales negativos deben llevar el signo negativo siempre en el numerador. Si la función lo hace bien



Explique por qué, en caso contrario haga las correcciones necesarias para que la función *ele* se muestre a continuación.

```
> (hacer-rac 21 14)
(3 2)
> (hacer-rac -21 14)
(-3 2)
> (hacer-rac 21 -14)
(-3 2)
> (hacer-rac -21 -14)
(3 2)
```

Ejercicio N° 2

Construya una función que se llame (*convertir ele lista*). Esta función debe convertir todos los elementos de la lista por el elemento *ele*. A continuación se presentan algunos ejemplos:

```
> (convertir 'a '(1))
(a)
> (convertir 'a '(1 2))
(a a)
> (convertir 'a '(1 2 3))
(a a a)
> (convertir 'a '(1 2 3 4 5 6))
(a a a a a)
```

lmos para ordenar listas. Se han presentado el

aciones sucesivas del mínimo, insert-sort y quick-

Ejercicio N° 3

Construya una función que se llame (*hacer-lista ele num*). Esta función debe construir una lista de tamaño *num*, con el elemento *ele*. Por ejemplo:

```
> (hacer-lista 'a 0)
()
> (hacer-lista 'a 1)
(a)
```

num den) representa los números racionales en una

gnifica que los números racionales negativos deben siempre en el numerador. Si la función lo hace bien

```
> (hacer-lista 'a 2)
(a a)
```

```
> (hacer-lista 'a 3)
(a a a)
```

```
> (hacer-lista 'a 4)
(a a a a)
```

Ejercicio N° 4

Existe una función de nombre (*number?* *arg*) que devuelve el valor de #t si *arg* es un número y #f en cualquier otro caso. Por ejemplo:

```
> (number? 7)
#t
> (number? 8.10)
#t
> (number? 'a)
#f
```

Utilice esta función para construir (*números?* *lista*), que recibe como argumento una lista. Si todos los elementos de la lista son números devuelve #t, en caso contrario devuelve #f. Por ejemplo:

```
> (números? '(1 2.7 4))
#t
> (números? '(1 a 3))
#f
```

Ejercicio N° 5

Existe una función de nombre (*integer?* *arg*) que devuelve el valor de #t si *arg* es un número entero y #f en cualquier otro caso. Por ejemplo:

```
> (integer? 7)
#t
> (integer? 7.0)
#f
```

```
> (integer? 7.11)  
#f
```

Utilice esta función para escribir una función de nombre (*pares lista*). Esta función recibe una lista de números reales y devuelve aquellos números que son pares. Por ejemplo:

```
> (pares '(1 2 3 4 5 6))  
(2 4 6)  
> (pares '(1 1.11 4 7.0 8.0))  
(4 8)  
> (pares '(8 6 3.12 3 12))  
(8 6 12)
```

Ejercicio N° 6

Existe una función de nombre (*sumar-números lista*) que recibe una lista lineal que tiene números y símbolos. La función debe producir como resultado la suma de los números.

```
> (sumar-números '())  
0  
> (sumar-números ' (1 a 2 b 3 c))  
6  
> (sumar-números ' (1 a 2 b 3 4))  
10
```

Ejercicio N° 7

Construya una función que determine si una lista de números está ordenada de menor a mayor. Por ejemplo:

```
> (ordenada? '())  
#t  
> (ordenada? '(1 2 3 4 5))  
#t  
> (ordenada? '(1 2 3 4 1))  
#f
```

Ejercicio N° 8

Construya una función que determine si una lista que contiene únicamente 1s y 0s está constituida de manera tal que los números se alternen. Por ejemplo:

```
> (alternada? '( ))
#t
> (alternada? '(1 0 1 0 1 0))
#t
> (alternada? '(0 1 0 1 0 ))
#t
> (alternada? '(1 1 1 0 1 0))
#f
```

Ejercicio N° 9

Construya una función que se llame (*duplo ele lista*). Esta función debe duplicar cualquier aparición del elemento ele en la lista lista. Por ejemplo:

```
> (duplo 'a '(b c d a e f a))
(b c d a a e f a a)
> (duplo '4 '(1 4 2 3 4 5 4))
(1 4 4 2 3 4 4 5 4 4)
> (duplo 'es '(hoy es jueves))
(hoy es es es jueves)
```

Ejercicio N° 10

Construya una función que se llame (*repetir num ele lista*). Esta función debe repetir num veces cualquier aparición del elemento ele en la lista lista. Por ejemplo:

```
> (repetir 2 'a '(b c d a e f))
(b c d a a e f)
> (repetir 3 'a '(b a c d a f))
(b a a a c d a a a f)
> (repetir 3 '4 '(1 4 2 3 4 5))
(1 4 4 4 2 3 4 4 4 5)
```

> (repetir 4 ‘es ‘(hoy es jueves))
 (hoy es es es es jueves)

Ejercicio N° 11

Construya una función que se llame (*sub-nil ele lista*). Esta función debe sustituir cualquier aparición de la lista nula por ele en la lista. Por ejemplo:

> (sub-nil ‘5 ‘(1 2 3 4 ()))
 (1 2 3 4 5)
 > (sub-nil ‘x ‘(a b () c d ()))
 (a b x c d x)

Ejercicio N° 12

Construya una función que se llame (*sub-or sust ele lista*). Esta función debe sustituir cualquier elemento que aparezca en la lista subst por el elemento ele en la lista. Por ejemplo:

> (sub-or ‘(a b c) ‘x ‘(a t u b c))
 (x t u x x)
 > (sub-or ‘(5 7 9) 100 ‘(1 2 3 4 5 6 7 8 9 10))
 (1 2 3 4 100 6 100 8 100 10)

Ejercicio N° 13

Construya una función que se llame (*sub-orr buscados elementos lista*). Esta función debe sustituir cualquier elemento que aparezca en la lista buscados por el elemento de nuevos que se encuentre a la misma posición. Por ejemplo:

> (sub-orr ‘(x y z) ‘(xx yy zz) ‘(a b x t z))
 (a b xx t zz)
 > (sub-orr ‘(x y z) ‘(1 2 3) ‘(a b x t z))
 (a b 1 t 3)

Ejercicio N° 14

Construya una función que se llame (*eliminar-ref index lista*). Esta función debe eliminar el elemento que se encuentra en la posición index de la lista. La lista iniciará con la posición 1. A continuación se presentan algunos ejemplos.

```

> (eliminar-ref 4 '( ))
()
> (eliminar-ref 4 '(a b c))
(a b c)
> (eliminar-ref 2 '(a b c))
(a c)
> (eliminar-ref 2 '(a b c d e))
(a c d e)

```

Ejercicio N° 15

Construya una función que se llame (*primeros lista*). Esta función recibe una lista de listas y debe devolver una lista con el primer elemento de cada una de ellas.

```

> (primeros '((1 2 3) (1 2 3)))
(1 1)
> (primeros '((a b c) (d e f) (g h i)))
(a d g)
> (primeros '((1 2 3) (4 5 6) (7 8 9)))
(1 4 7)
> (primeros '((0 1 2) (10 11 12) (20 21 22) (30 31 32)))
(0 10 20 30)

```

Ejercicio N° 16

Construya una función que se llame (*segundos lista*). Esta función recibe una lista de listas y debe devolver una lista con el segundo elemento de cada una de ellas. A continuación se presentan algunos ejemplos.

```

> (segundos '((1 2 3) (1 2 3)))
(2 2)
> (segundos '((a b c) (d e f) (g h i)))
(b e h)
> (segundos '((1 2 3) (4 5 6) (7 8 9)))
(2 5 8)

```

```
> (segundos `((0 1 2) (10 11 12) (20 21 22) (30 31 32)))
(1 11 21 31)
```

Ejercicio N° 17

Una función (*punto index lista*) recibe un valor llamado index. Una vez que llega a index saca los elementos que aparecen cada dos espacios. Por ejemplo:

```
> (punto 1 `'(a x n y a t))
(a n a)

> (punto 5 `'(a b c d j h e i f h f))
(j e f f)

> (punto 6 `'(a b c d e j h e i f h f j s))
(j e f f s)
```

Ejercicio N° 18

Escriba una función de nombre (*subseq ini fin lista*). Esta función recibe una lista y devuelve la subsecuencia que se encuentra entre los índices de ini y fin.

```
> (subseq 1 5 `'(a b c d e))
(a b c d e)

> (subseq 2 4 `'(a b c d e))
(b c d)

> (subseq 2 7 `'(a b c d e))
#f
```

Ejercicio N° 19

Escriba una función de nombre (*seq lista1 lista2*). Esta función recibe dos listas. Si la lista1 se encuentra dentro de la lista2 devuelve los elementos posteriores a la primera aparición de la lista1. En caso contrario devuelve #f.

Por ejemplo

```
> (subseq `'(a b c) `'(x1 y1 a b x2 y2 a b c x3 y3 z4))
(x3 y3 z4)

> (subseq `'(a b c) `'(x1 y1 a b x2 y2 a b x3 y3 z4))
#f
```

```
> (subseq '(manzanas rojas)
           '(la bruja le dio manzanas rojas a blanca nieves))
(a blanca nieves)
```

Ejercicio N° 20

Construya una función que se llame (*insertar-d ref ele lista*). Esta función busca el elemento ref e inserta después de él a ele.

```
> (insertar-d 'x 'x '(a b c x d e f))
(a b c x x d e f)

> (insertar-d 'x 'y '(a b c x d e f))
(a b c x y d e f)

> (insertar-d '4 '5 '(1 2 3 4 6 7 4 8))
(1 2 3 4 5 6 7 4 5 8)
```

Ejercicio N° 21

Construya una función de nombre (*centro lista*). Si la lista tiene una cantidad par de elementos debe devolver el valor de #f, en caso contrario devuelve el número que se encuentre en la mitad o centro de la lista. Por ejemplo:

```
> (centro '())
#f

> (centro '(1))
1

> (centro '(1 2 3 4 5))
3

> (centro '(1 2 3 4 5 6))
#f
```

Ejercicio N° 22

Construya una función de nombre (*sacar num lista*). Esta función debe sacar de la lista los primeros num elementos. A continuación se muestra su funcionamiento:

> (sacar 0 '(1 2 3 4 5 6))

()

> (sacar 2 '(1 2 3 4 5 6))

(1 2)

> (sacar 3 '(1 2 3 4 5 6))

(1 2 3)

Construya una función en Scheme de nombre (*borrar num lista*). Esta función debe borrar de la lista los primeros num elementos:

> (borrar 0 '(1 2 3 4 5 6))

(1 2 3 4 5 6)

> (borrar 2 '(1 2 3 4 5 6))

(3 4 5 6)

> (borrar 3 '(1 2 3 4 5 6))

(4 5 6)

Construya una función de nombre (*espejo lista*). Esta función parte una lista por la mitad e intercambia las mitades. La lista debe tener una cantidad par de elementos:

> (espejo '())

()

> (espejo '(1 2 3 4))

(3 4 1 2)

> (espejo '(1 2 3 4 5 6))

(4 5 6 1 2 3)

Ejercicio N° 23

Escriba una función que se llame (*promedio lista*), esta función recibe como entrada una lista de números de la forma (x(1) x(2) x(3) ... x(4)) y debe producir como salida el promedio ponderado de esa lista.

El promedio de un grupo de datos (x(1) x(2) x(3) ... x(4)) suele representarse de la siguiente manera:

$\bar{x} = \text{promedio de } (x(1) \ x(2) \ x(3) \dots \ x(4))$

La fórmula del promedio está dada por:

$$\bar{x} = \text{promedio } '(x(1) \dots x(n)) = \frac{\left(\sum_{i=1}^n x(i) \right)}{\left(n \right)}$$

Esta función debe producir los siguientes resultados:

> (promedio '(100 90 80))
90

> (promedio '(100 97 82 74))
88.25

> (promedio '(75 68 95 42 100 83))
77.1666...

Ejercicio N° 24

La desviación estándar, denominada (de lista), de un conjunto de datos de la forma $(x(1) x(2) x(3) \dots x(4))$ está dada por la fórmula:

$$\text{de} = \sqrt{\left(\frac{1}{n} \right) * \sum_{i=1}^n \left(x(i) - \bar{x} \right)^2}$$

Construya una función que se llame (de lista) y que calcule la desviación estándar para una lista de números. Se debe invocar de la siguiente manera:

> (de '(100 90 80))
8.1649...

```
> (de '(100 97 82 74))  
10.6858...  
> (de '(75 68 95 42 100 83))  
19.1434...
```

Ejercicio N° 25

Escriba una función que se llame (*contador lista*). Esta función recibe una lista de números y debe producir una lista con tres elementos que indiquen la cantidad de negativos de la lista original, la cantidad de ceros y la cantidad de números positivos. A continuación se muestra un ejemplo de cómo debe operar esta función.

```
> (contador '(0 1 4 5 -4 -2 1))  
(2 1 4)  
> (contador '(0 0 0 1 1 1 1 -1 -1 -1))  
(3 3 4)
```

Ejercicio N° 26

Construya una función que se llame (*iota ini fin*). Esta función genera una secuencia de números en una lista de ini hasta fin, con ini y fin enteros.

```
> (iota 0 5)  
(0 1 2 3 4 5)  
> (iota 1 10)  
(1 2 3 4 5 6 7 8 9 10)  
> (iota 5 11)  
(5 6 7 8 9 10 11)
```

Ejercicio N° 27

Construya una función que se llame (*criba lista*). Esta función selecciona de una secuencia numérica, que inicia en 1 y termina en n, aquellos números que son primos. A continuación se presentan algunos ejemplos.

```
> (criba '(1 2 3 4 5 6 7))  
(1 2 3 5 7)
```

```
> (criba '( 1 2 3 4 5 6 7 8 9 10
           11 12 13 14 15 16 17 18 19 20))
(1 2 3 5 7 11 13 17 19)
```

Ejercicio N° 28

Construya una función que se llame (*criba-e lista*). Esta función selecciona de una secuencia numérica, que inicia en 1 y termina en n, aquellos números que son primos.

Para realizar este proceso toma el primer número de la lista mayor que uno y elimina a todos los números que son divisibles por él, toma el siguiente elemento de la lista y continua este proceso con los demás números hasta que se acabe la lista. La lista resultante tiene únicamente números primos.

De esta forma criba debe trabajar de la siguiente manera:

```
> (criba-e '(1 2 3 4 5 6 7 8 9))
```

Saca el 1 e inicia el proceso:

Iniciamos con '(2 3 4 5 6 7)

Saca el 2 y deja en la lista únicamente los números NO divisibles por 2: '(3 5 7 9)

Saca el 3 y deja en la lista únicamente los números NO divisibles por 3: '(5 7)

Saca el 5 y deja en la lista únicamente los números NO divisibles por 5: '(7)

Saca el 7 y deja en la lista únicamente los números NO divisibles por 7: '()

Cuando la lista se acaba se tiene el conjunto de números primos: '(1 2 3 5 7)

A continuación se muestran unos ejemplos adicionales de como debe trabajar esta función.

```
> (criba-e '(1 2 3 4 5 6 7))
(1 2 3 5 7)

> (criba-e '( 1 2 3 4 5 6 7 8 9 10
           11 12 13 14 15 16 17 18 19 20))
(1 2 3 5 7 11 13 17 19)
```

Ejercicio N° 29

Construya una función que se llame (*merge-sort lista*). Esta función recibe como argumento una lista desordenada y produce una lista ordenada. Para ello debe implementar las siguientes funciones.

- a) Construya una función que se llame (*merge lista1 lista2*). Esta función recibe dos listas ordenadas y produce una nueva lista ordenada. Observe que el algoritmo que utilice debe tomar en cuenta que las dos listas YA se encuentran ordenadas.

A continuación se presentan algunos ejemplos:

```
> (merge '(1 2 3 4) '(5 6 7 8))  
(1 2 3 4 5 6 7 8)  
  
> (merge '(1 2 3) '(1 2 3 4))  
(1 1 2 2 3 3 4)  
  
> (merge '(1 2 3 5 7) '(4 5 6 8))  
(1 2 3 4 5 5 6 7 8)
```

- b) Construya una función que se llame (*natural lista*). Esta función recibe como elemento de entrada una lista desordenada y produce una lista de listas, donde cada sublistas mantiene el orden natural de la lista original. Por ejemplo:

```
> (natural '(1 2 3))  
((1 2 3))  
  
> (natural '(3 2 1))  
((3) (2) (1))  
  
> (natural '(1 2 3 4 5 6))  
((1 2 3 4 5 6))  
  
> (natural '(3 4 5 1 2 3 4 7 3 2 1))  
((3 4 5) (1 2 3 4 7) (3) (2) (1))
```

- c) Finalmente programe la función que se llame (*merge-sort lista*). Esta función recibe como argumento una lista desordenada. Primero aplica la función (*natural lista*) para construir un conjunto de listas ordenadas. Después de aplicar la función (*merge lista1 lista2*) para ir produciendo listas ordenadas. Por ejemplo:

```
> (merge-sort '(3 4 5 1 2 3 4 7 3 2 1))
```

Primero aplica natural y produce:

```
((3 4 5) (1 2 3 4 7) (3) (2) (1))
```

Luego aplica merge y produce:

```
((1 2 3 3 4 4 5 7) (3) (2) (1))
```

Vuelve a aplicar merge:

```
((1 2 3 3 3 4 4 5 7) (2) (1))
```

Vuelve a aplicar merge:

```
((1 2 2 3 3 3 4 4 5 7) (1))
```

Vuelve a aplicar merge:

```
((1 1 2 2 3 3 3 4 4 5 7))
```

Finalmente produce la lista:

```
(1 1 2 2 3 3 3 4 4 5 7)
```

Ejercicio N° 30

Construya una función que se llame (*bucket-sort lista*). Esta función recibe como argumento una lista desordenada que NO posee elementos repetidos y produce como salida una lista ordenada.

```
> (bucket-sort '(5 1 4 2))
(1 2 4 5)
```

Para construir esta función, debe programar los siguientes procedimientos.

- a) Escriba una función de nombre (*qmenores num lista*). Esta función cuenta la cantidad de elementos menores o iguales que num. Por ejemplo

```
> (qmenores 7 '(1 2 3 4 5))
5
> (qmenores 7 '(8 10 11 12))
0
> (qmenores 7 '(3 8 2 4 8))
3
```

- b) Escriba una función de nombre (*qlista lista*). Esta función recibe una lista y para cada elemento cuenta el número de elementos menores o iguales que existen.

```
> (qlista '(4 7 2))
((4 2) (7 3) (2 1))

> (qlista '(5 1 4 2))
((5 4) (1 1) (4 3) (2 2))

> (qlista '(5 4 1 2 3))
((5 5) (4 4) (1 1) (2 2) (3 3))
```

¶) Escriba un proceso de ordenamiento que realice el siguiente algoritmo:

Toma una lista por ejemplo: '(5 1 4 2)

Le aplica qlista:

```
> (qlista '(5 1 4 2))
((5 4) (1 1) (4 3) (2 2))
```

Luego tenemos la lista anterior y una lista con tantos ceros como la lista original, y se realiza un proceso de colocar cada elemento con base en el segundo valor producido. Por ejemplo sea el elemento (5 4) significa que el 5 va en la posición 4 de la lista.

Lista original	Lista ordenada
((5 4) (1 1) (4 3) (2 2))	'(0 0 0 0)
((1 1) (4 3) (2 2))	'(0 0 0 5)
((4 3) (2 2))	'(1 0 0 5)
((2 2))	'(1 0 4 5)
()	'(1 2 4 5)
> (bucket-sort '(5 1 4 2))	
(1 2 4 5)	

Ejercicio N° 31

Todo lenguaje de programación posee algún mecanismo para la generación de números aleatorios. En Scheme existe una función que se denomina (*random k*). Random trabaja de la siguiente forma:

```
> (random k)
;; produce un valor aleatorio uniforme entero
;; entre 0,...,k-1
```

Por ejemplo

```
> (random 6)
;; generó un número entre 0 y 5
3
> (random 6)
;; generó un número entre 0 y 5
4
> (random 6)
;; generó un número entre 0 y 5
0
```

Utilice la función anterior para construir una función de nombre

(*generar-número inf sup*). Esta función genera un número x entero de manera aleatoria tal que $\text{inf} \leq x \leq \text{sup}$.

Esta función genera un número aleatorio entre dos enteros cualesquiera de la siguiente manera.

```
> (generar-número 5 10)
;; generó un número entre 5 y 10
7
> (generar-número 0 10)
;; generó un número entre 5 y 10
2
> (generar-número 1 6)
;; generó un número entre 1 y 6
;; simula el comportamiento de un dado
5
```

Posteriormente, con la ayuda de la función anterior construya una función que se llame (*generar-lista-números cant inf sup*). Esta función utiliza la variable *cant* para generar esa cantidad de números. Los números generados deben tener un valor que se encuentre entre el valor de *inf* y el valor de *sup*. Por ejemplo:

```
> (generar-lista-números 2 0 10)
;; genera 2 números contenidos entre 0 y 10
(2 6)
```

```
> (generar-lista-números 3 0 10)
;; genera 3 números contenidos entre 0 y 10
(2 7 9)

> (generar-lista-números 5 0 10)
;; genera 5 números contenidos entre 0 y 10
(3 7 9 10 2)
```

Ejercicio N° 32

Para este ejercicio se realizarán pruebas sobre los siguiente métodos de ordenamiento: Ordenar, Insert-Sort, Merge-Sort, Quick-Sort.

Pruebe cada uno de ellos con listas de distinto tamaño. Ejecute y tome el tiempo de cada algoritmo. Debe ejecutar cada algoritmo 10 veces con listas de 1000 elementos. 10 veces con listas de 2000 elementos y 10 veces con listas de 3000 elementos.

Para construir las listas puede utilizar la función que genera valores aleatorios. Por ejemplo:

```
> (ordenar (generar-lista-números 1000 0 1000))
;; Debe tomar el tiempo que tarda esta función
```

Para tomar el tiempo que tarda un proceso puede utilizar la función time:

```
> (time (ordenar (generar-lista-números 1000 0 1000)))
;; devuelve el tiempo que tardó la función en ejecutarse
```

Tome el tiempo de ejecución de cada programa y responda a las siguientes preguntas.

- Cuál es el algoritmo que tarda menos tiempo.
- Cuál es al algoritmo que tarda más tiempo.

Ejercicio N° 33

Programación Avanzada: La Conjetura de Goldbach.

El matemático ruso Christian Goldbach (1690-1764) conjeturó que todo número par es igual a la suma de dos números primos. Por ejemplo:

$$2 = 1 + 1$$

4 = 2 + 2
 6 = 3 + 3
 8 = 3 + 5
 10 = 3 + 7
 12 = 5 + 7
 ...
 100 = 3 + 97
 102 = 5 + 97
 104 = 7 + 97

Construya un programa que dado un número par a encuentre dos números primos p1 y p2 tal que:

$$a = p_1 + p_2$$

Por ejemplo:

```

> (suma-primos 10)
(3 7)

> (suma-primos 12)
(5 7)

> (suma-primos 100)
(3 97)

> (suma-primos 102)
(5 97)
  
```

Ejercicio N° 34. Programación Avanzada. Análisis de Regresión Lineal

Construya una función que se llame (*regresión lista*). Esta función recibe un conjunto de datos de la forma:

```

((x(1) y(1))
 (x(2) y(2))
 (x(3) y(3))
 ...
 (x(n) y(n)))
  
```

y con esos datos deber realizar el siguiente cálculo matemático:

$$b = \frac{\left(\sum_{i=1}^0 x(i) - y(i) \right)}{\left(\sum_{i=1}^0 x(i) - x(i) \right)} - \frac{\left(n * \bar{x} * \bar{y} \right)}{\left(n * \bar{x} * \bar{x} \right)}$$

$$a = y - (b * x)$$

En donde se tiene que:

$$x = (\text{promedio } x) = \frac{\left(\sum_{i=1}^n x(i) \right)}{\left(n \right)}$$

$$y = (\text{promedio } x) = \frac{\left(\sum_{i=1}^n y(i) \right)}{\left(n \right)}$$

Regresión debe recibir la lista de elementos:

((x(1) y(1)) (x(2) y(2)) (x(3) y(3)) ... (x(n) y(n)))

y producir como salida los valores de a y b.

Por ejemplo:

> (regresión '((1 5) (2 7)))

;; produce los valores (a b)

(3 2)

> (regresión '((5 7) (3 7) (3 6) (1 4)))

;; produce los valores (a b)

(3.75 0.75)



Ejercicio N° 35. Programación Avanzada. Permutaciones de Conjuntos

Construya una función que se llame permutaciones que recibe una lista y devuelve todos los posibles ordenamientos de esa lista. Su comportamiento debe ser:

> (permutaciones '())

()

> (permutaciones '(1 2 3))

((1 2 3) (1 3 2) (2 1 3) (2 3 1) (3 1 2) (3 2 1))

> (permutaciones '(a b c))

((a b c) (a c b) (b a c) (b c a) (c a b) (c b a))

> (permutaciones '(a b c d))

((a b c d) (a b d c) (a c b d)

 (a c d b) (a d b c) (a d c b)

 (b a c d) (b a d c) (b c a d)

 (b c d a) (b d a c) (b d c a)

 (c a b d) (c a d b) (c b a d)

 (c b d a) (c d a b) (c d b a)

 (d a b c) (d a c b) (d b a c)

 (d b c a) (d c a b) (d c b a))

Ejercicio N° 36. Programación Avanzada. El Problema de los Vasos de Agua.

Se tiene dos contenedores de agua etiquetados vaso aa y vaso bb. Al vaso aa le caben 3 litros de agua y al vaso bb le caben 5 litros. Inicialmente los vasos se encuentran vacíos. Para representar los vasos se utilizará una lista donde la primera posición representa el contenido del vaso aa y la segunda el contenido del vaso bb. Inicialmente los vasos se encuentran vacíos, es decir, el estado inicial es (0 0).

Las operaciones permitidas para los vasos son:

- a. Vaciar el vaso aa.
- b. Vaciar el vaso bb.
- c. Llenar el vaso aa.
- d. Llenar el vaso bb.
- e. Pasar el contenido de aa hacia bb.
- f. Pasar el contenido de bb hacia aa.

Si el estado inicial (0 0) y se desean encontrar TODAS las posibles formas de llegar al estado final de (1 0), existirían 16 formas posibles. Por ejemplo:

((0 0) (3 0) (0 3) (3 3) (1 5) (1 0))
((0 0) (0 5) (3 2) (3 0) (0 3) (3 3) (1 5) (1 0))
...

Programe una función de nombre (vasos estado-inicial estado-final) que genere todos los movimientos para llegar del estado inicial al estado final. Ejecute este programa de la siguiente manera:

```
> (vasos '(0 0) '(1 0))
;; genera todas las soluciones
```