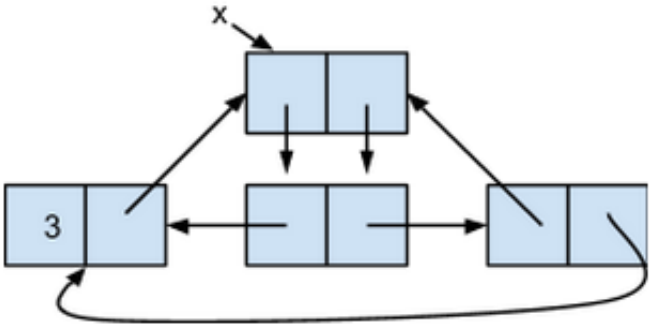
Práctica 7: Programación Imperativa en Scheme

Ejercicio 1

a) Dibuja el diagrama box & pointer resultante de las siguientes instrucciones:

```
(define p1 (cons 1 '()))
(define p2 (cons p1 p1))
(define p3 (cons p1 p2))
(set-car! p3 (cddr p2))
```

b) Escribe las instrucciones en Scheme que han generado la siguiente estructura:

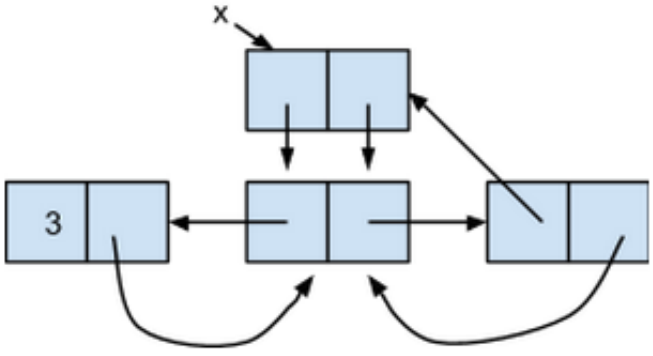


image

c) Dibuja el diagrama *box & pointer* resultante al evaluar la siguiente expresión:

```
(set-car! (cddr x) (caaar x))
```

d) Escribe las expresiones mutadoras en Scheme que modifiquen la estructura representada en el apartado b) para obtener la siguiente:



image

Ejercicio 2

Vamos a definir dos funciones adicionales que sirven para completar la barrera de abstracción de la listas ordenadas mutables que hemos visto en teoría.

a) Escribe el procedimiento auxiliar `(delete-next-item! ref)` que elimine la siguiente pareja de la referencia a la que apunta `ref`. Escribe después el procedimiento `(delete-olist! n lista)` que borre el número `n` de la lista ordenada con cabecera que pasamos como segundo parámetro, llamando a la función anterior. Si el número no está en la lista, no hará nada.

Ejemplo:

```
(define lista '(*olist* 10 20 30 40))
(delete-next-item! lista)
lista
⇒ (*olist* 20 30 40)
(delete-olist! 30 lista)
lista
⇒ (*olist* 20 40)
```

b) Escribe el procedimiento `filter-olist!` que tome una lista ordenada con cabecera y un predicado como argumento y, llamando a la función `delete-next-item`, elimine los elementos de la lista que no cumplan el predicado.

Ejemplo:

```
(define lista '(*olist* 1 2 3 5 6 7 8 9 10 11 12))
(filter-olist! lista odd?)
lista
⇒ (*olist* 1 3 5 7 9 11)
```

Ejercicio 3

Implementa en Scheme el procedimiento mutador `aplana-lista!` que reciba una lista estructurada y la transforme en una lista plana. Para simplificar, suponemos que el primer elemento de las listas y sublistas es atómico. Explica utilizando diagramas de caja y puntero el funcionamiento de tu solución.

Ejemplo:

```
(define lista1 '(1 2 3 (4 5) 6 7))
(define lista2 '(1 (2 (3 (4 5))) 6 7)
(aplana-lista! lista1)
(aplana-lista! lista2)
lista1
⇒ (1 2 3 4 5 6 7)
lista2
```

```
⇒ (1 2 3 4 5 6 7)
```

Ejercicio 4

Implementa un nueva versión de una lista ordenada que contenga en la parte izquierda de la cabecera un lista con la longitud de la lista, el predicado que comprueba el tipo de datos de la lista y la función de comparación del tipo de datos de la lista. Llamamos a la lista `tlist` (*typed list*).

Implementa la *barrera de abstracción* de esta nueva estructura con las siguientes funciones. Cuando lo consideres necesario, utiliza funciones auxiliares para que el código se más legible.

- `(make-tlist predicado-tipo predicado-menor)` : construye una *tlist* sin elementos con el predicado que comprueba el tipo de datos y el que compara si un elemento es *menor* que otro. Suponemos que `equal?` funciona correctamente para indicar si dos datos son iguales.
- `(insert-tlist! dato lista)` : inserta (con mutación) de forma ordenada el dato en la lista, comprobando si el dato se corresponde con el tipo. Devuelve el símbolo `ok` si la inserción se realiza correctamente, `mal-tipo` si se ha intentado introducir un dato que no es el correcto y `repetido` si el dato ya existía. Actualiza la longitud de la lista.
- `(longitud-tlist lista)` : devuelve la longitud de la lista

Ejemplo 1 (lista de números):

```
(define lista (make-tlist number? <))
(insert-tlist! 20 lista) ⇒ ok
(insert-tlist! 10 lista) ⇒ ok
(insert-tlist! 'hola lista) ⇒ mal-tipo
(longitud-tlist lista) ⇒ 2
lista ⇒ ((3 #<procedure:number?> #<procedure:<>) 10 20)
```

Ejemplo 2 (lista de símbolos):

```
(define lista2 (make-tlist symbol? (lambda (x y)
                                     (string<? (symbol->string x)
                                               (symbol->string y)))))
(insert-tlist! 'b lista2) ⇒ ok
(insert-tlist! 'a lista2) ⇒ ok
lista2 ⇒ ((2 #<procedure:symbol?> #<procedure>) a b)
```

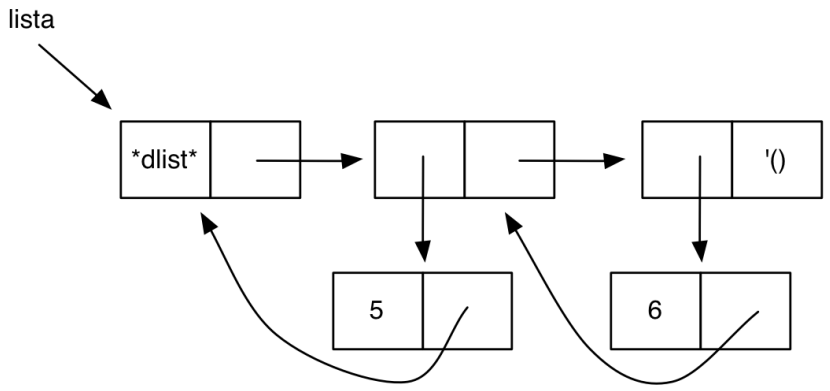
Ejercicio 5

Implementa las funciones `make-dlist` e `insert-dlist!` que construyan una lista ordenada de números doblemente enlazada con cabecera (*dlist*) y que inserten con mutación un elemento en ella.

Por ejemplo, después de realizar las siguientes instrucciones:

```
(define lista (make-dlist))
(insert-dlist! 6 lista)
(insert-dlist! 5 lista)
```

la lista doblemente enlazada debe tener la estructura que indica el siguiente diagrama de parejas:



Ejercicio 6

a) Queremos implementar en Scheme un procedimiento llamado `make-parquimetro` que cree y devuelva una función con estado local mutable que permita gestionar un parquímetro. La función con estado local debe aceptar **mensajes** que permitan consultar y modificar su estado local para hacer la siguientes funciones:

- Ingresar una cantidad de dinero
- Consultar la cantidad de tiempo disponible (suponiendo que cada minuto cuesta 0.05€)
- Decrementar el tiempo un minuto con el mensaje `"tick"`
- Cuando el tiempo llegue a 0 no se debería decrementar más y un mensaje llamado `"multar"` debería devolver `#t`
- Consultar la cantidad de dinero recaudado

Ejemplo del funcionamiento:

```
(define p1 (make-parkimetro))
(p1 "ingresar" 10) ⇒ 10
(p1 "tiempo") ⇒ 5.0
(p1 "tick") ⇒ 4.0
(p1 "ingresar" 20) ⇒ 30
(p1 "cantidad") ⇒ 30
(p1 "tiempo") ⇒ 14.0
```

Piensa en cuál sería el estado local mutable necesario para resolver el problema e implementa la función `make-parquimetro` .

b) Dibuja el diagrama de ámbitos de un ejemplo de ejecución sencillo, que muestre el funcionamiento del estado local mutable.

Ejercicio 7

Dibuja y explica diagrama de ámbitos creado por el siguiente código ¿Se crea alguna clausura?

```
(define x 2)

(define (g i)
        (let ((x i))
           (lambda ()
                   (set! x (* x x)))))

(define h (g 5))

(define (f x func)
   (func)
   (func)
    x)

(f 16 h))
```
