

### Pregunta 1 (6 puntos)

Define un procedimiento llamado set-double! que reciba un árbol numérico de cualquier profundidad como parámetro y reemplace cada hoja por el doble de cada número. El método debe recorrer cada elemento del árbol.

```
>(define arbol (list 2 (list 3 4) (list 2 3 (list 3 3))))
>arbol
>(2 (3 4) (2 3 (3 3)))
>(set-double! arbol)
>arbol
>(4 (6 8) (4 6 (6 6)))
```

#### Solución:

```
(define (set-double! l)
  (cond ((null? l) )
        ((atom? (car l)) (begin (set-car! l (* 2 (car l)))
                                (set-double! (cdr l))))
        (else (begin (set-double! (car l))
                      (set-double! (cdr l))))))
```

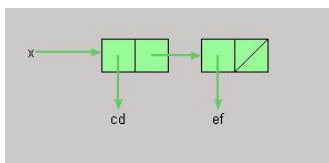
### Pregunta 2 (6 puntos)

Representa mediante diagramas box & pointer el resultado de evaluar las siguientes expresiones:

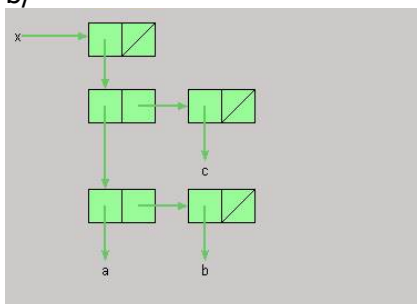
- a/ (cadadr '(a (b (cd ef) d)))
- b/ (define lista (list (cons (append '(a) '(b)) '(c)))
- c/ (set-car! (cdar lista) (caar lista))

#### Solución:

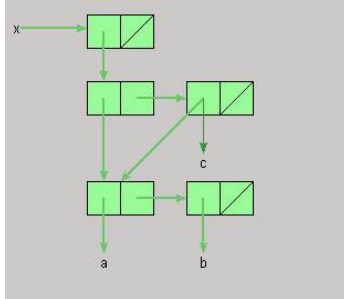
a/



b/



c/



### Pregunta 3 (6 puntos)

Escribe una función `multicompose` que tome como argumento una lista de cualquier longitud, cuyos elementos son funciones de un argumento. Debería devolver una función de un argumento que realiza la composición de todas las funciones dadas.

Nota: No usar mutación (`set!`)

```
>((multicompose (list sqrt sqrt 1+)) 80)
3
```

```
>((multicompose (list first bf)) `(la sinuosa carretera))
sinuosa
```

#### Solución:

```
(define (multicompose funciones)
  (lambda (x)
    (if (null? funciones)
        x
        ((car funciones)((multicompose (cdr funciones)) x))))))
```

### Pregunta 4 (6 puntos)

Vamos a crear un TAD para las horas del día (horas y minutos). Usaremos la representación de 24 horas, en que 3 PM es la hora 15.

La implementación se hará usando dos representaciones internas diferentes.

- El constructor para las horas del día es `time`. Toma dos argumentos: las horas y los minutos. 4:12 debería ser `(time 4 12)`

- También queremos los siguientes tres operadores:

`(hour t)` → devuelve la parte de hora de un `time` dado

`(minute t)` → devuelve la parte de minutos de un `time` dado

`(hour? t)` → devuelve `#t` si el `time t` es una hora exacta (como 6:00); devuelve `#f` en cualquier otro caso

a/ Implementa `time`, `hour`, `minute` y `hour?` usando una representación interna en la que la hora se representa como una lista de dos números. Esto es, 4:12 debería representarse internamente como `(4 12)`

b/ Implementa `time`, `hour`, `minute` y `hour?` usando una representación interna en la que la hora se representa como un entero, a saber, el número de minutos transcurridos desde medianoche. Esto es, 4:12 debería representarse internamente como el número 252  $((4 \times 60) + 12)$

Solución:

a/

```
(define time list)

(define hour car)

(define minute cadr)

(define (hour? t)
  (= (minute t) 0))
```

b/

```
(define (time h m)
  (+ (* h 60) m))

(define (hour t)
  (div t 60))

(define (minute t)
  (remainder t 60))

(define (hour? t)
  (= (minute t) 0))
```

### **Pregunta 5 (6 puntos)**

Define un procedimiento recursivo llamado (`sufijo wd char`) que admita como argumento una palabra `wd` y un carácter `char` y que devuelva el resto de la palabra `wd` desde que se encuentra el carácter `char`. En el caso en que la palabra no contenga a `char`, entonces la función devuelve la cadena vacía.

Indica si el procedimiento que has escrito es recursivo o iterativo y explica por qué.

```
> (sufijo 'hola 'o)
'la
> (sufijo 'resto 'e)
'sto
> (sufijo 'pepita 's)
""
```

**Solución:**

```
(define (sufijo wd c)
  (cond
    ((empty? wd) wd)
    (else
     (if (equal? (first wd) c)
         (butfirst wd)
         (sufijo (butfirst wd) c)))))
```

El procedimiento es iterativo porque al final de la recursión la solución está ya disponible.

**Pregunta 6 (6 puntos)**

Indica qué valor devuelven las siguientes expresiones. En el caso en que el valor devuelto sea un procedimiento explica qué hace el procedimiento. En el caso en que la expresión produzca un error, indícalo y da una descripción concisa del error..

a/

```
(define foo (lambda (x)
              (if (equal? x 0)
                  1
                  (* x (foo (- x 1))))))
```

(foo 3)

b/

```
(define (foo3 x)
  (lambda (y) (+ y x)))
```

```
(define foo4 (foo3 1))
```

foo4

c/

```
(let ((x 3) (y 4))
  (let ((x 5) (y (+ x 3)))
    (let ((x (+ x 7)))
      (+ x y))))
```

**Solución:**

a/ 6

b/ Un procedimiento de un argumento que suma 1 al valor pasado como argumento

c/ 18

### Pregunta 7 (6 puntos)

Explica la evaluación de las siguientes expresiones y dibuja el entorno resultante.

```
(define (foo y)
  (lambda (x)
    (set! y (+ x y))))
```

```
(define w (foo 5))
(w 30)
```

Solución:

