Mañana

Ejercicio 1 (1,5 puntos)

a) (0,5 puntos) Diferencias entre las características funcionales de Scala y Scheme.

- Fuertemente tipeado: hay que declarar tipos en variables, funciones y parámetros
- Definición de variables no reasignables con val
- Comprensión de expresiones y currying
- Uso de placeholders para crear funciones anónimas

b) (0,2 puntos) Supongamos la siguiente definición de una función en Scala

```
def foo(a: String, x: Int, c: (String) => String): Int = { … }
```

¿Cuál de las siguientes invocaciones es correcta (sólo una)?

1) `foo(1,2, (x) => {x+3})`
2) `foo("1",2, (x) => {x+3})`
***** 3) `foo("1",2, _+"3")` *****
4) `foo("1",2, (x: String) => {x.setString("3")}`

c) (0,2 puntos) En una barrera de abstracción de un tipo valor (de un `Punto2D`, por ejemplo) podríamos definir un constructor de copia de la siguiente forma:

```
(define (copia-punto2D p)
    (make-punto2D (getX-punto2D p) (getY-punto2D p))
```

¿Cuál de las siguientes afirmaciones es cierta (sólo una)?

1) La definición de un constructor de copia en el tipo valor es algo muy recomendable para evitar los efectos laterales
2) No es posible definir un constructor de copia, porque los objetos valor son inmutables
3) No es posible definir un constructor de copia en una barrera de abstracción, porque en programación funcional no se pueden crear objetos
***** 4) No es necesario definir un constructor de copia porque los objetos valor son inmutables y nunca van a provocar efectos laterales *****

d) (0,2 puntos) Dado el siguiente código en Scheme:

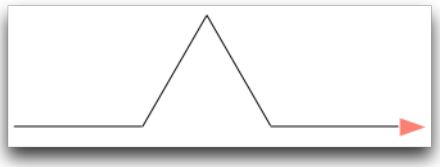
```
(define (misterio nivel n)
  (if (= nivel 0)
      (draw n)
      (begin (misterio (- nivel 1) (/ n 3))
             (turn 60)
             (misterio (- nivel 1) (/ n 3))
             (turn -120)
```

```
            (misterio (- nivel 1) (/ n 3))
            (turn 60)
            (misterio (- nivel 1) (/ n 3)))))
```

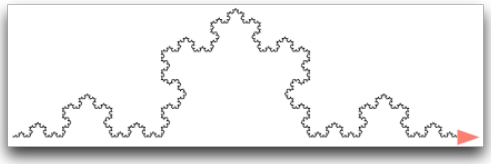
Si realizamos la llamada:
`(misterio 5 400)`

Indica el gráfico resultante:

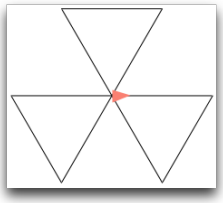
1)



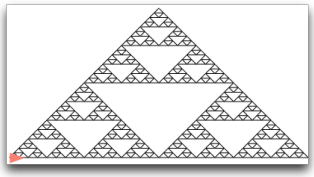
***** 2) *****



3)



4)



e) (0,2 puntos) Explica las diferencias entre el `define` de Scheme y el `def` de Scala. Pon un ejemplo en el que el funcionamiento sea distinto.

```
def z = t / 0
```

Ejercicio 2 (1,75 puntos)

a) (0,75 puntos) Diseña e implementa en Scheme la barrera de abstracción del tipo de dato `intervalo` que representa un valor mínimo y un valor máximo en un eje de coordenadas de números reales. Incluye en la barrera de abstracción un mínimo de 2 operadores. Explica algún ejemplo de aplicación en la que podrías utilizar este nuevo tipo de dato y los operadores definidos.

Solución:

Diseño de la barrera de abstracción del tipo de dato "intervalo"

Constructor y selectores

(make-int min max) : devuelve un intervalo con un valor mínimo y un valor máximo, ambos números reales
(min-int int) : devuelve el valor mínimo del intervalo
(max-int int) : devuelve el valor máximo del intervalo

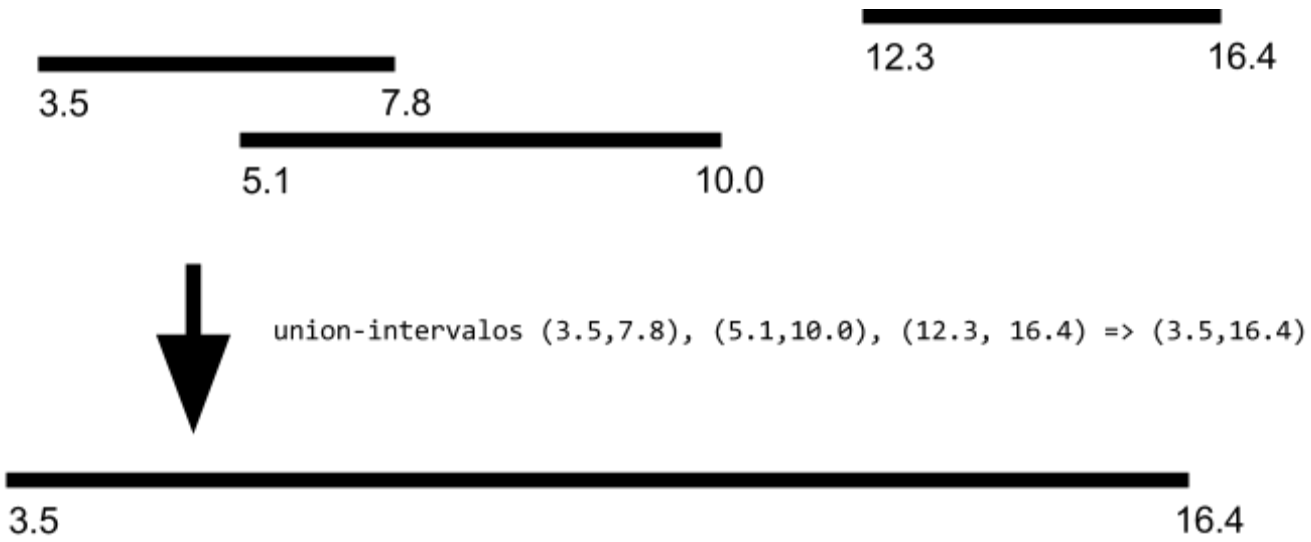
Operadores

(une-int int1 int2) : construye un nuevo intervalo uniendo el primer intervalo y el segundo
(desplaza-int int d) : construye un nuevo intervalo desplazando el intervalo una cantidad "d"

Implementación

```
(define (make-int min max)
  (cons min max))

(define (min-int int)
  (car int))

(define (max-int int)
  (cdr int))

(define (une-int int1 int2)
  (make-int (min (min-int int1) (min-int int2))
            (max (max-int int1) (max-int int2))))

(define (desplaza-int int d)
  (make-int (+ (min-int int1) d)
            (+ (max-int int1) d)))
```

b) (1 punto) Supongamos la función `(union-intervalos lista-intervalos)` que recoge como parámetro una lista de intervalos y devuelve el intervalo resultante de la unión de todos ellos (ver dibujo). Realiza dos implementaciones recursivas de la función `unión-intervalos`, una recursiva pura y otra con tail-recursion.



Solución:

Recursiva pura

```
(define (une-intervalos lista)
  (if (null? (cdr lista))
      (car lista)
      (une-int (car lista) (une-intervalos (cdr lista)))))
```

Tail-recursion

```
(define (une-intervalos lista)
  (une-intervalos-aux (car lista) (cdr lista)))

(define (une-intervalos-aux result lista)
  (if (null? lista)
      result
      (une-intervalos-aux (une-int result (car lista)) (cdr lista))))
```

Ejercicio 3 (1,75 puntos)

a) (0,5 puntos) Define una función recursiva `(plana lista)` que reciba una lista estructurada y que compruebe si es plana. Puedes utilizar la función `es-hoja?` vista en teoría.

Ejemplo:

```
(plana? '(1 2 3 4)) → #t
(plana? '(1 (2 3) 4) → #f
```

Solución:

```
(define (plana? lista)
  (if (null? lista)
      #t
      (and (es-hoja? (car lista))
           (plana? (cdr lista)))))
```

b) (1,25 puntos) Define una función recursiva `(contar-planas lista)` que reciba una lista y devuelva el número de sublistas planas que contiene.

Ejemplo:

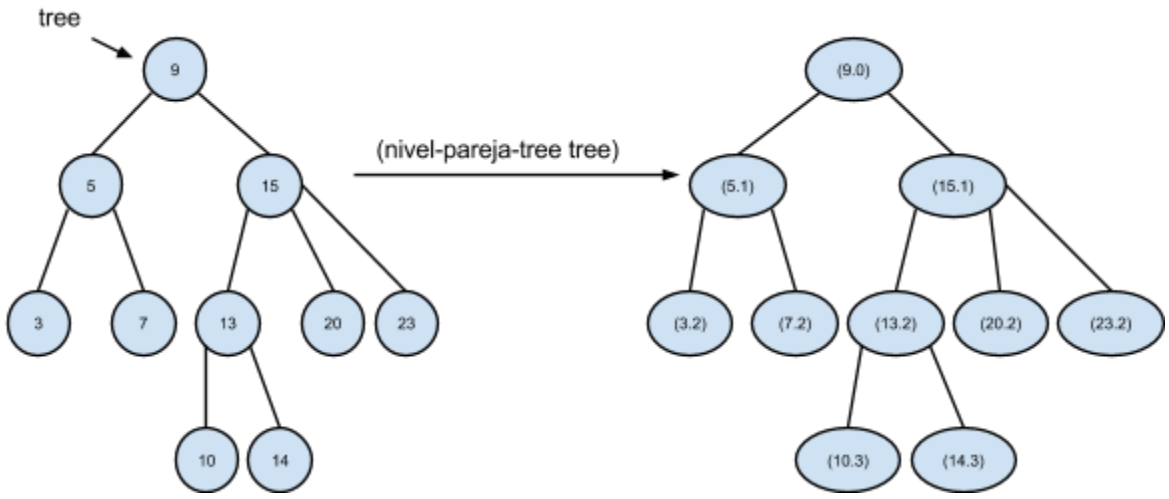
```
(contar-planas '(1 2 3 4)) → 0
(contar-planas '(1 2 (3 4) 5 6) → 1
(contar-planas '(1 2 (3 4) (5 6 (7 8) 9) 10)) →  2
```

Solución:

```
(define (contar-planas lista)
  (cond
    ((null? lista)
     0)
    ((es-hoja? (car lista))
      (contar-planas-exp-s (cdr lista)))
    ((plana? (car lista))
      (+ 1 (contar-planas (cdr lista))))
    (else
      (+ (contar-planas (car lista)) (contar-planas (cdr lista))))))
```

Ejercicio 4 (1,75 puntos)

Define en Scheme una función `(nivel-pareja-tree tree)` que reciba un árbol genérico y devuelva un nuevo árbol genérico donde cada dato sea una pareja que contenga el dato original (izquierda) y el nivel en el que se encuentra (derecha). Utiliza la barrera de abstracción de los árboles genéricos vista en teoría (no es necesario implementarla).



Solución:

```
(define (nivel-pareja-tree tree)
    (nivel-pareja-aux tree 0))

(define (nivel-pareja-aux-tree tree n)
    (make-tree (cons (dato-tree tree) n)
                      (nivel-pareja-aux-bosque (hijos-tree tree) (+ n 1))))

(define (nivel-pareja-aux-bosque bosque n)
    (if (null? bosque) bosque
        (cons (nivel-pareja-aux-tree (car bosque) n)
                  (nivel-pareja-aux-bosque (cdr bosque) n))))
```

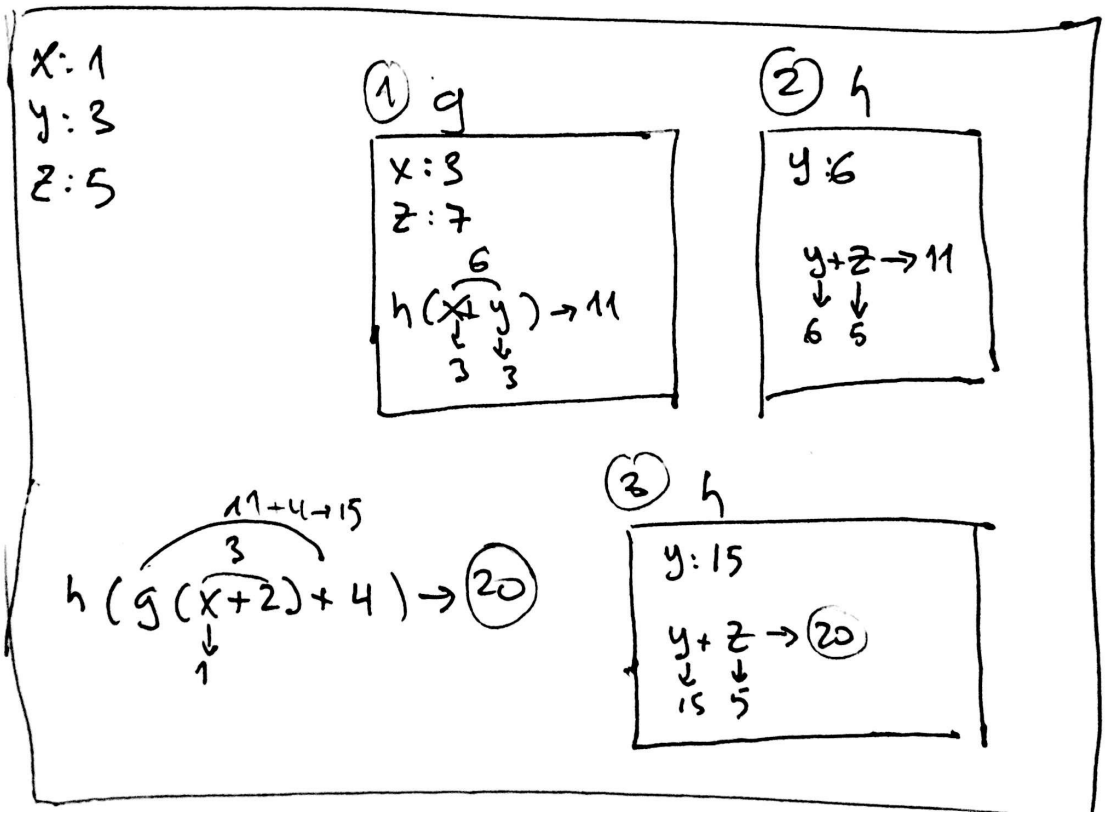
Ejercicio 5 (1,75 puntos)

a) (0,5 puntos) Dibuja y explica los ámbitos que se crean al ejecutar las siguientes expresiones en Scala. ¿Cuántos ámbitos locales se crean? ¿En qué orden? ¿Qué valor devuelve la última expresión?

```
val x=1
val y=3
val z=5

def h(y:Int) = {
       y+z
}

def g(x:Int) = {
       val z=7
       h(x+y)
}
```

`h(g(x+2)+4)`

Solución:

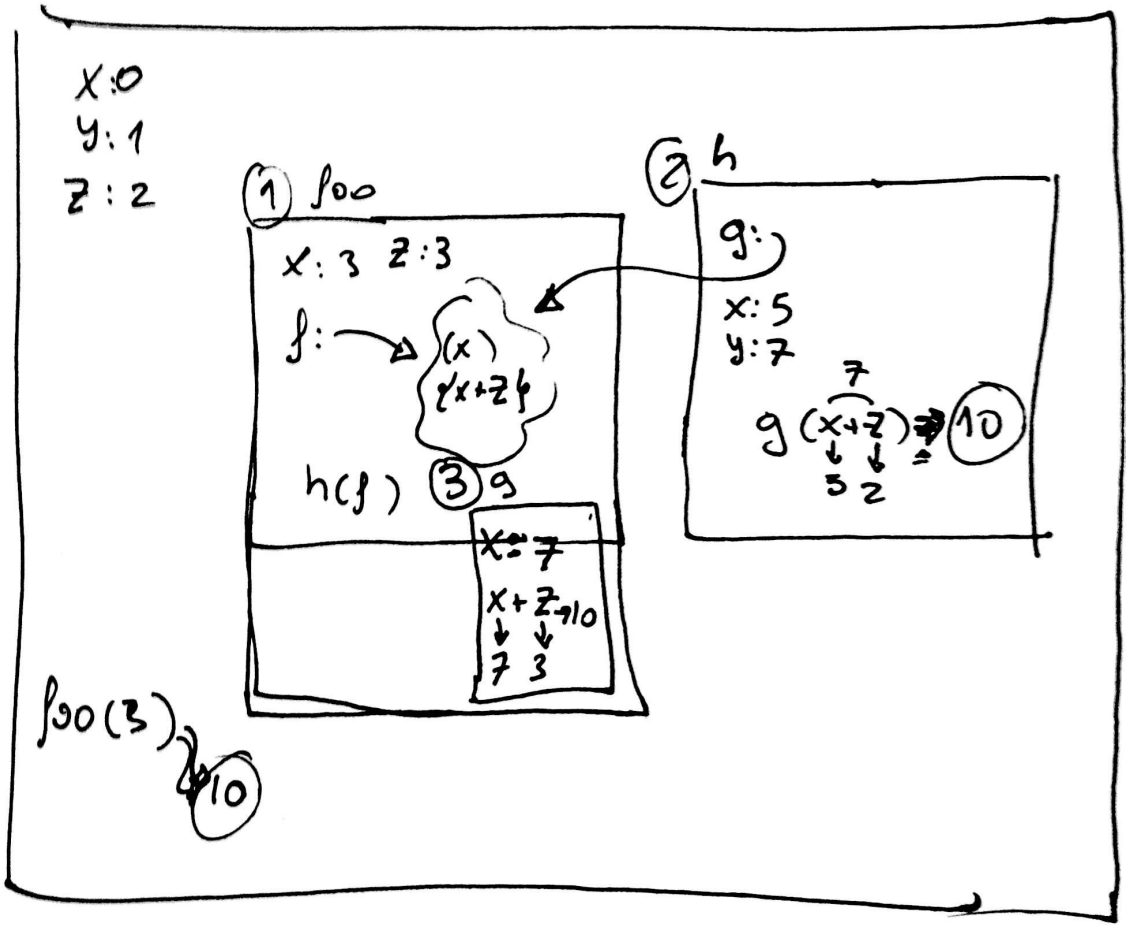


b) (1,25 puntos) Dibuja y explica los ámbitos que se crean al ejecutar las siguientes expresiones en Scala. ¿Cuántos ámbitos locales se crean? ¿En qué orden? ¿Qué valor devuelve la última expresión?

```
val x = 0
val y = 1
val z = 2

def h(g: (Int) => Int): Int = {
    val x = 5
    val y = 7
    g(x+z)
}

def foo (x: Int): Int = {
    val z = 3
    val f = (x: Int) => {x+z}

    h(f)
```

```
}

foo(3)
```

Solución:



Ejercicio 6 (1,5 puntos)

Escribe en Scala la función `cuenta-preds(pred1,pred2,lista)` que reciba una `lista de enteros` y dos predicados y devuelva una tupla de dos elementos donde su parte izquierda contiene el número de elementos de la lista que cumplen el `pred1` y la parte derecha contiene los que cumplen el `pred2`. La función `cuenta-preds` debe ser **recursiva** y sólo debe hacer un **único** recorrido a la lista. Define completo el prototipo de la función.

Ejemplo:

`cuenta-preds(_>5,_%2!=0,List(1,2,3,4,5,6,7,8,9))` → *(4,5)*

Solución:

```
def cuenta-preds(p1:(Int)=>Boolean, p2:(Int)=>Boolean, lista: list[Int]):(Int,Int)={
    if(lista.isEmpty) (0,0) else {
        val tupla = cuenta-preds(p1,p2,lista.tail)
        if (p1(lista.head)==true && p2(lista.head)==true)
            (1+tupla._1, 1+tupla._2)
        else if (p1(lista.head)==true)
            (1+tupla._1,tupla._2)
        else
            (tupla._1,1+tupla._2)
    }
}
```

Tarde

Ejercicio 1 (1,5 puntos)

a) (0,7 puntos) Características funcionales comunes en Scheme y Scala. Ejemplos en Scala.

- Recursión
- Definición declarativa de funciones, sin usar pasos de ejecución
- Funciones como objetos de primer orden
- Funciones anónimas
- Evaluación con deep binding y clausuras
- Tipos de datos inmutables
- Uso de listas y funciones head y tail (car y cdr)

b) (0,2 puntos) Supongamos la siguiente definición de una función en Scala

```
def foo(a: String, x: Int, c: (String) => String): Int = { … }
```

¿Cuál de las siguientes invocaciones es correcta (sólo una)? Explica por qué.

1) `foo("1",2, (x: String) => {x.setString("3")})`
2) `foo(1,2, (x) => {x-3})`
3) `foo("1",2, (x) => {x-3})`
***** 4) `foo("1",2, _+"3")` *****

c) (0,2 puntos) En una barrera de abstracción de un tipo valor (de un `Punto2D`, por ejemplo) podríamos definir un constructor de copia de la siguiente forma:

```
(define (copia-punto2D p)
    (make-punto2D (getX-punto2D p) (getY-punto2D p))
```

¿Cuál de las siguientes afirmaciones es cierta (sólo una)?

1) No es posible definir un constructor de copia, porque los objetos valor son inmutables
2) La definición de un constructor de copia en el tipo valor es algo muy recomendable para evitar los efectos laterales
3) No es necesario definir un constructor de copia porque los objetos valor son inmutables y nunca van a provocar efectos laterales
***** 4) No es posible definir un constructor de copia en una barrera de abstracción, porque en programación funcional no se pueden crear objetos *****

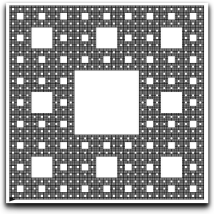
d) (0,2 puntos) Dado el siguiente código en Scheme:

```
(define (h x)
    (* x (sqrt 2)))

(define (t w)
    (draw w)
    (turn 135)
    (draw (h (/ w 2)))
    (turn 90)
    (draw (h (/ w 2)))
    (turn 135))

(define (misterio w)
    (if (> w 20)
        (begin
          (misterio (/ w 2))
          (move (/ w 4)) (turn 90) (move (/ w 4)) (turn -90)
          (misterio (/ w 2))
          (turn -90) (move (/ w 4)) (turn 90) (move (/ w 4))
          (misterio (/ w 2))
          (turn 180) (move (/ w 2)) (turn -180))
        (t w)))
```

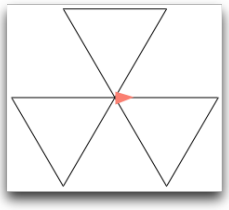
Si realizamos la llamada:
`(misterio 400)`

Indica el gráfico resultante:

1)

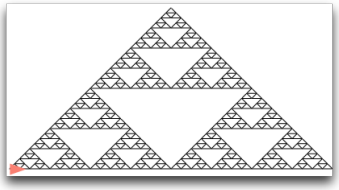


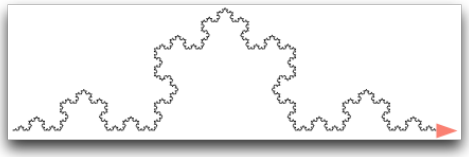
2)



***** 3) *****

4)



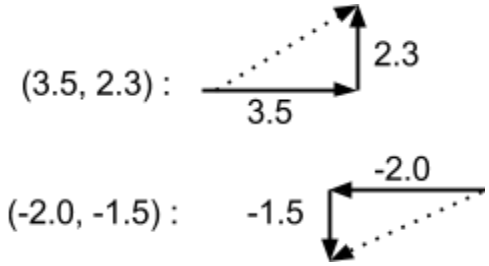


e) (0,2 puntos) Explica las diferencias entre el `define` de Scheme y el `def` de Scala. Pon un ejemplo en el que el funcionamiento sea distinto.

```
def x = 3 / 0
```

Ejercicio 2 (1,75 puntos)

a) (0,75 puntos) Diseña e implementa en Scheme la barrera de abstracción del tipo de dato `desplazamiento2d` que contiene dos números que representan un desplazamiento en el eje x, seguido de un desplazamiento en el eje y. Por ejemplo, el siguiente dibujo representa los desplazamientos 2d (3.5, 2.3) y (-2.0, -1.5):



Incluye en la barrera de abstracción un mínimo de 2 operadores. Explica algún ejemplo de aplicación en la que podrías utilizar este nuevo tipo de dato y los operadores definidos.

Solución:

Diseño de la barrera de abstracción del tipo de dato "desplazamiento"

Constructor y selectores

(make-desp x y) : construye un nuevo desplazamiento 2d a partir de un valor x y un valor y, ambos números reales
(x-desp desp) : devuelve el valor x de un desplazamiento
(y-desp desp) : devuelve el valor y de un desplazamiento

Operadores

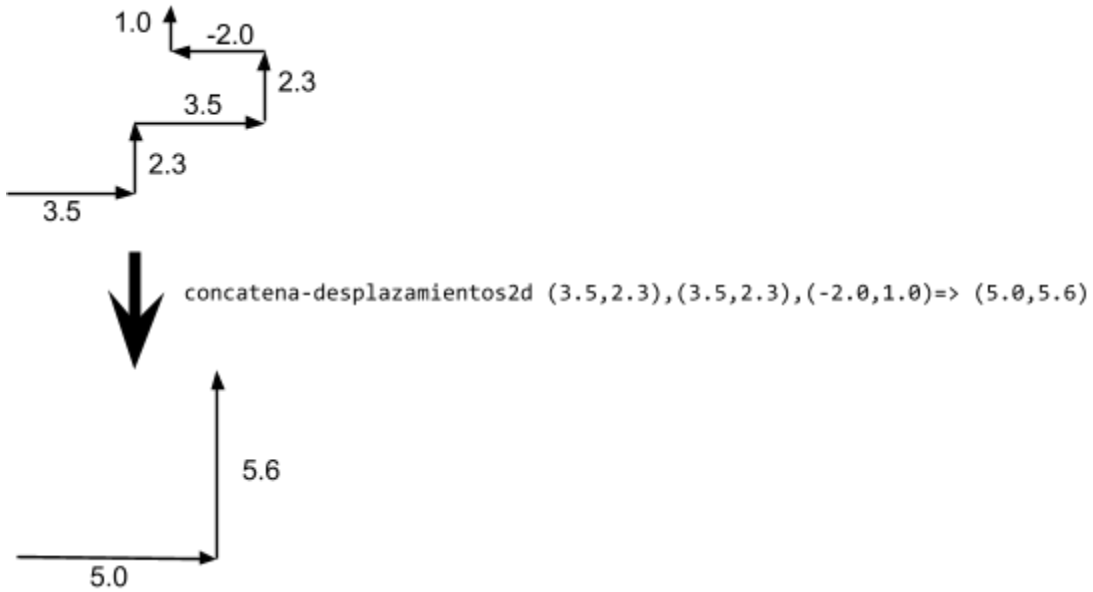
(suma-desp desp1 desp2) : construye un nuevo desplazamiento 2d sumando dos desplazamientos
(longitud-desp desp1) : devuelve la distancia recorrida por el desplazamiento: la distancia

entre la posición inicial y la posición final

Implementación

```
(define (make-desp x y)
  (cons x y))

(define (x-desp desp)
  (car desp))

(define (y-desp desp)
  (cdr desp))

(define (suma-desp desp1 desp2)
  (make-desp (+ (x-desp desp1) (x-desp desp2))
             (+ (y-desp desp1) (y-desp desp2))))

(define (longitud-desp desp)
  (sqrt (* (x-desp desp) (x-desp desp))
        (* (y-desp desp) (y-desp desp))))
```

b) (1 punto) Supongamos la función `(concatena-desplazamientos lista-desplazamientos2d)` que recoge como parámetro una lista de desplazamientos2d y devuelve el desplazamiento2d resultante de aplicar todos los desplazamientos de la lista (ver dibujo). Realiza dos implementaciones recursivas de la función, una recursiva pura y otra con tail-recursion.



Solución:

Recursiva pura

```
(define (concat-desp lista)
  (if (null? (cdr lista))
      (car lista)
      (suma-desp (car lista) (concat-desp (cdr lista)))))
```

Tail-recursion

```
(define (concat-desp lista)
  (concat-desp-aux (car lista) (cdr lista)))

(define (concat-desp-aux result lista)
  (if (null? lista)
      result
      (concat-desp-aux (suma-desp result (car lista)) (cdr lista))))
```

Ejercicio 3 (1,75 puntos)

a) (0,5 puntos) Define una función recursiva `(plana lista)` que reciba una lista estructurada y que compruebe si es plana. Puedes utilizar la función `es-hoja?` vista en teoría.

Ejemplo:

```
(plana? '(1 2 3 4)) → #t
(plana? '(1 (2 3) 4) → #f
```

Solución:

```
(define (plana? lista)
  (if (null? lista)
      #t
      (and (es-hoja? (car lista))
           (plana? (cdr lista)))))
```

b) (1,25 puntos) Define una función recursiva `(reducir lista)` que reciba una lista y devuelva una nueva lista como resultado de eliminar las sublistas planas contenidas en `lista`. Puedes utilizar la función definida en el apartado anterior.

Ejemplo:

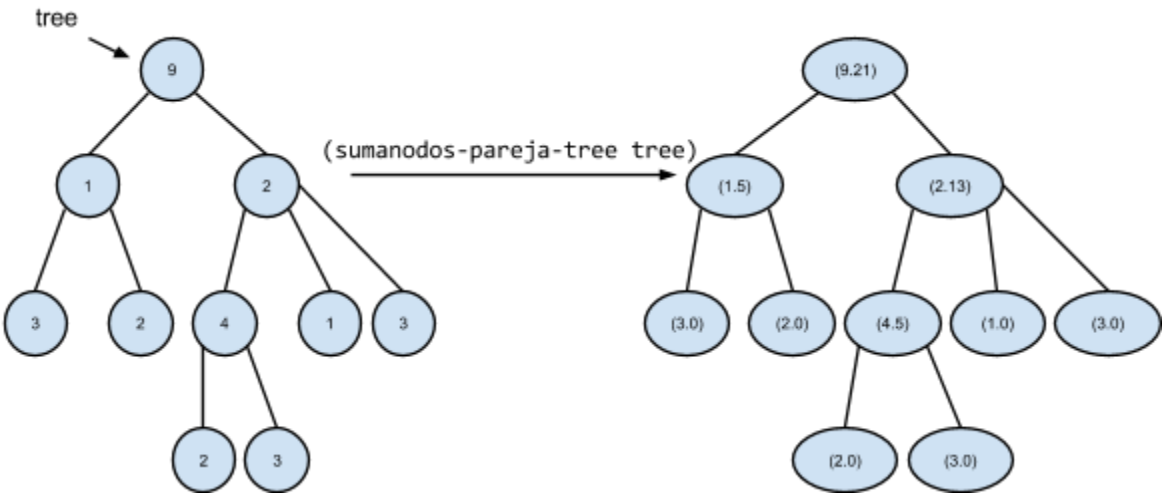
```
(reducir '(1 2 (3 4) 5 6) → (1 2 5 6)
(reducir '(1 2 (3 4) (5 6 (7 8) 9) 10)) → (1 2 (5 6 9) 10)
(reducir ‘(1 2 3)) → (1 2 3)
```

Solución:

```
(define (reducir lista)
  (cond
    ((null? lista)
     '())
    ((es-hoja? (car lista))
      (cons (car lista) (reducir-exp-s (cdr lista))))
    ((plana? (car lista))
      (reducir (cdr lista)))
    (else
      (cons (reducir (car lista)) (reducir (cdr lista))))))
```

Ejercicio 4 (1,75 puntos)

Define en Scheme una función `(sumanodos-pareja-tree tree)` que reciba un árbol genérico y devuelva un nuevo árbol genérico donde cada dato sea una pareja que contenga el dato original (izquierda) y la suma de los nodos de sus hijos (derecha). Utiliza la barrera de abstracción de los árboles genéricos vista en teoría (no es necesario implementarla).



Solución:

```
(define (sumanodos-pareja-tree tree)
  (make-tree (cons (dato-tree tree) (suma-hijos (hijos-tree tree)))
             (sumanodos-pareja-bosque (hijos-tree tree))))
(define (sumanodos-pareja-bosque bosque)
  (if (null? bosque) bosque
      (cons (sumanodos-pareja-tree (car bosque))
            (sumanodos-pareja-bosque (cdr bosque)))))
```

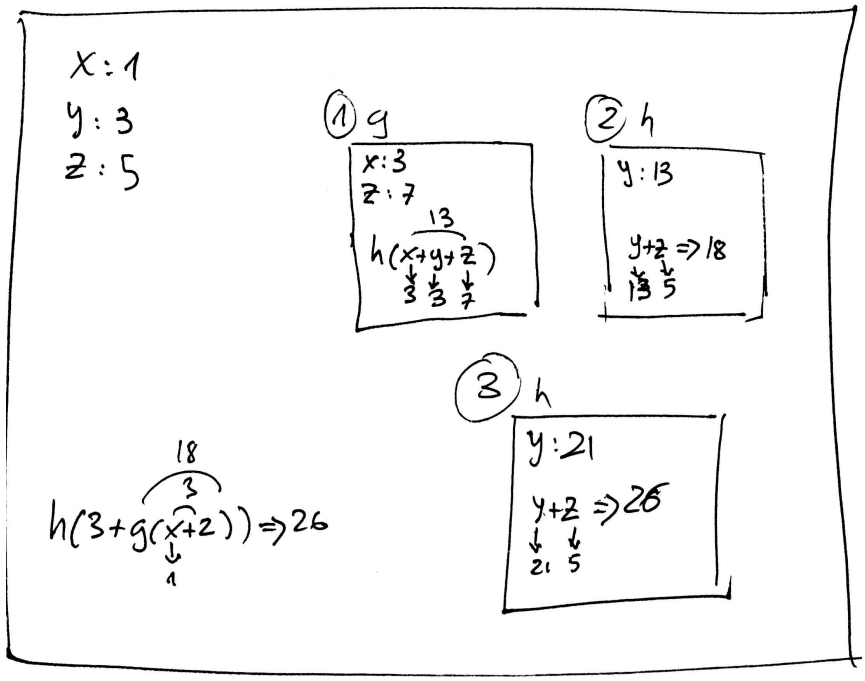
```
(define (suma-hijos bosque)
  (if (null? bosque) 0
      (+ (suma-hijos-tree (car bosque))
         (suma-hijos (cdr bosque)))))
(define (suma-hijos-tree tree)
  (+ (dato-tree tree)
     (suma-hijos (hijos-tree tree))))
```

Ejercicio 5 (1,75 puntos)

a) (0,5 puntos) Dibuja y explica los ámbitos que se crean al ejecutar las siguientes expresiones en Scala. ¿Cuántos ámbitos locales se crean? ¿En qué orden? ¿Qué valor devuelve la última expresión?

```
val x=1
val y=3
val z=5

def h(y:Int) = {
    y+z
}

def g(x:Int) = {
    val z=7
    h(x+y+z)
}

h(3+g(x+2))
```

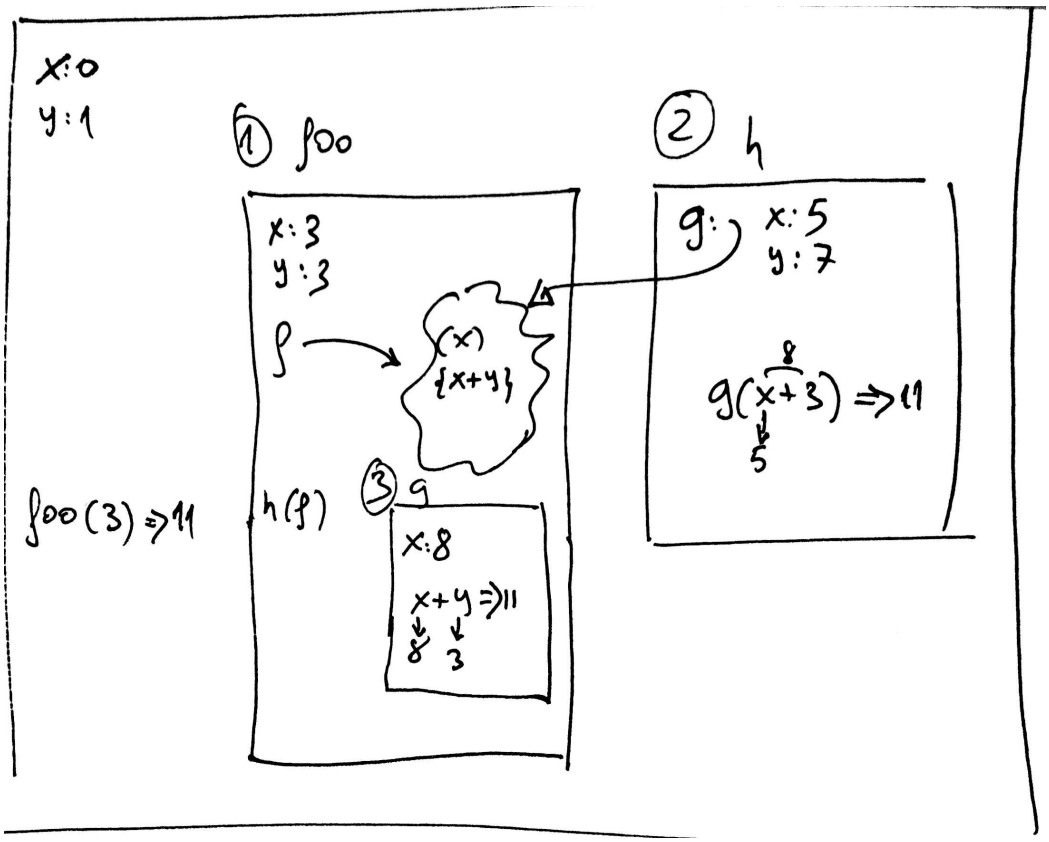
Solución:



b) (1 punto) Dibuja y explica los ámbitos que se crean al ejecutar las siguientes expresiones en Scala. ¿Cuántos ámbitos locales se crean? ¿En qué orden? ¿Se crea alguna clausura? ¿Qué valor devuelve la última expresión?

```
val x = 0
val y = 1

def h(g: (Int) => Int): Int = {
    val x = 5
    val y = 7
    g(x+3)
}

def foo (x: Int): Int = {
    val y = 3
    val f = (x: Int) => {x+y}
    h(f)
}

foo(3)
```

Solución:



Ejercicio 6 (1,5 puntos)

Escribe en Scala la función `divide-preds(pred1,pred2,lista)` que reciba una lista de enteros y dos predicados y devuelva una tupla de dos elementos donde su parte izquierda contiene una lista con los elementos de la lista que cumplen el `pred1` y la parte derecha contiene los que cumplen el `pred2`. La función `divide-preds` debe ser **recursiva** y sólo debe hacer un **único** recorrido a la lista. Define completo el prototipo de la función.

Ejemplo:

`divide-preds(_>5,_%2!=0,List(1,2,3,4,5,6,7,8,9))` → *((6,7,8,9),(1,3,5,7,9))*

Solución:

```
def divide-preds(p1:(Int)=>Boolean, p2:(Int)=>Boolean, lista: list[Int]) :
(List[Int],List[Int]) = {
    if(lista.isEmpty) (Nil,Nil) else {
        val tupla = divide-preds(p1,p2,lista.tail)
        if (p1(lista.head)==true && p2(lista.head)==true)
            (lista.head::tupla._1, lista.head::tupla._2)
        else if (p1(lista.head)==true)
            (lista.head::tupla._1,tupla._2)
        else
```

```
            (tupla._1,lista.head::tupla._2)
    }
}
```