

Nombre: _____ Grupo: _____

Lenguajes y Paradigmas de Programación

Curso 2012-2013

Segundo parcial (turno de tarde)

Normas importantes

- La puntuación total del examen es de 10 puntos.
- Se debe contestar cada pregunta en las hojas que entregamos. Utiliza las últimas hojas para hacer pruebas. No olvides poner el nombre.
- La duración del examen es de 2 horas.

Ejercicio 1 (1,5 puntos)

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

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))

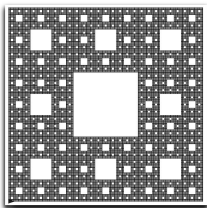
(define (misterio w)
  (if (> w 20)
      (begin
        (misterio (/ w 2))
        (move (/ w 4)) (turn 90) (move (/ w 4)) (turn -90)
        (misterio (/ w 2))
        (turn -90) (move (/ w 4)) (turn 90) (move (/ w 4))
        (misterio (/ w 2))
        (turn 180) (move (/ w 2)) (turn -180))
     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

Si realizamos la llamada:

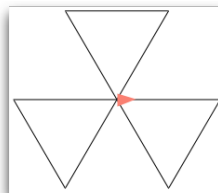
(misterio 400)

Indica el gráfico resultante:

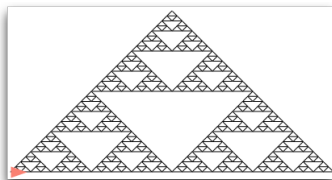
1)



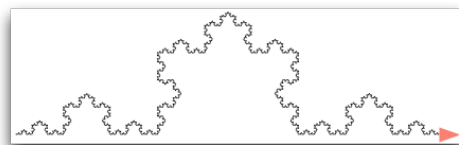
2)



3)



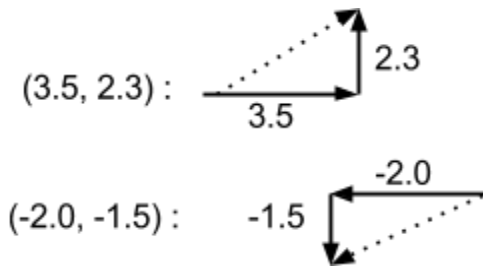
4)



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.

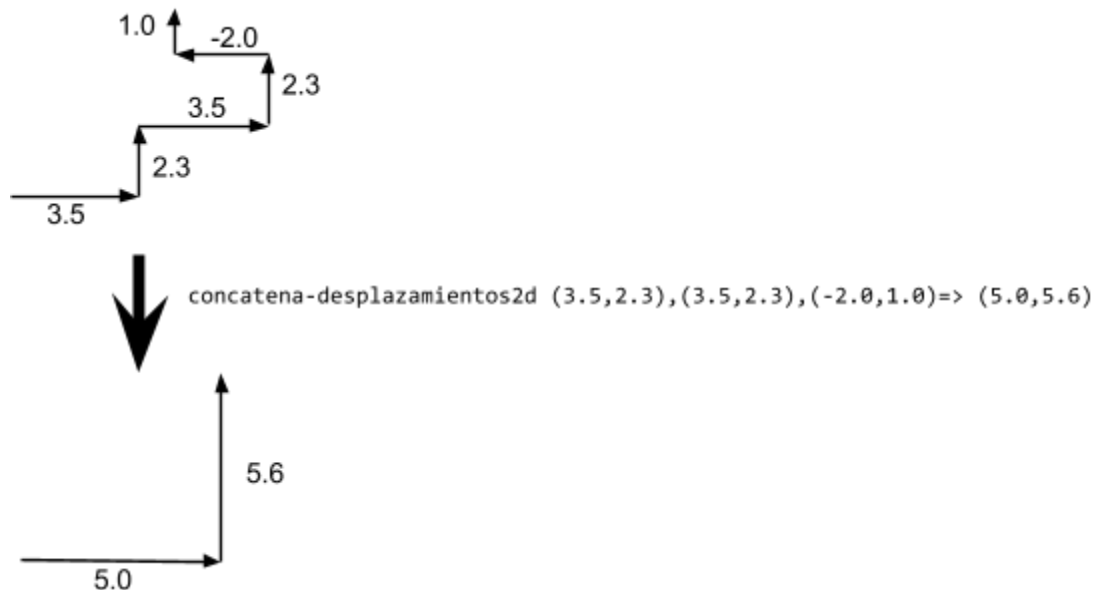
Ejercicio 2 (1,75 puntos)

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.

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.



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, es decir, si está formada sólo por datos atómicos. Puedes utilizar la función `es-hoja?` vista en teoría.

Ejemplo:

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

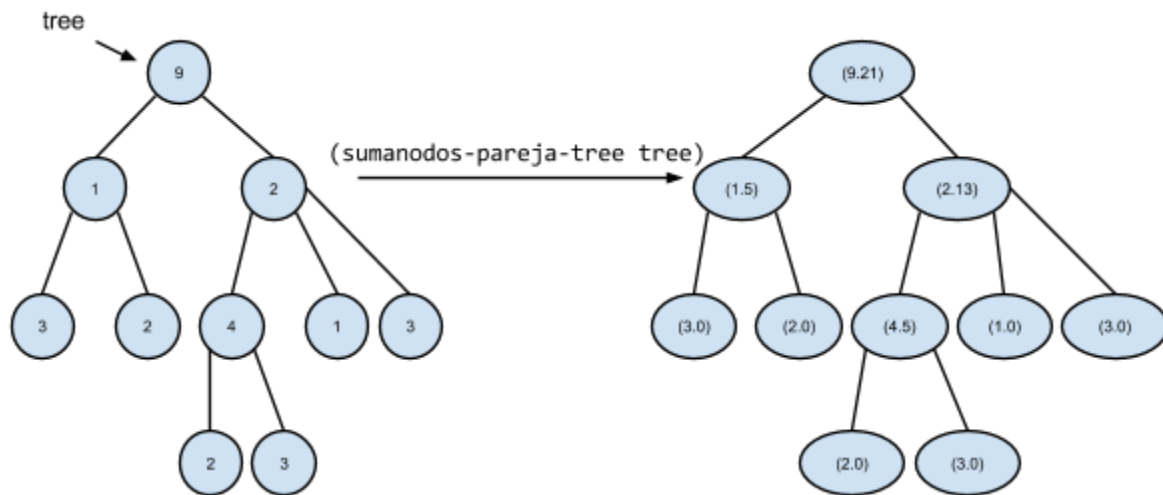
b) (1,25 puntos) Define una función recursiva (`reducir lista`) que reciba una lista estructurada y devuelva una nueva lista estructurada como resultado de eliminar las listas planas contenidas en `lista`. Puedes utilizar la función definida en el apartado anterior.

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)
```

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).



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))
```

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)
```

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))`