Nombre:_____ **Grupo:**_____

Lenguajes y Paradigmas de Programación
Curso 2011-2012
Tercer parcial Grado en Ingeniería Informática

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 2:30 horas.**

Ejercicio 1 (1 punto)

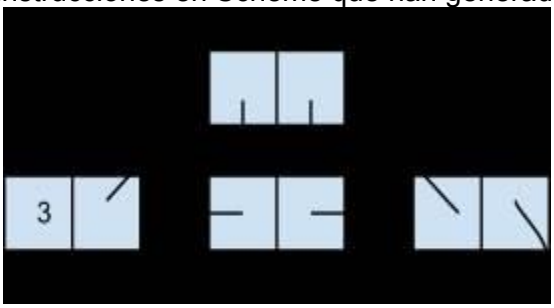
a) **(0,25 puntos)** Explica las características, historia y lenguajes principales del paradigma de programación imperativa.
b) **(0,25 puntos)** Explica las características principales, historia y lenguajes principales del paradigma de programación orientada a objetos.
c) **(0,5 puntos)** Constructores en Scala: sintaxis, primarios y auxiliares, distintas formas de definir los parámetros y ejemplos

Ejercicio 2 (1,75 puntos)

a) **(0,25 puntos)** 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))
```

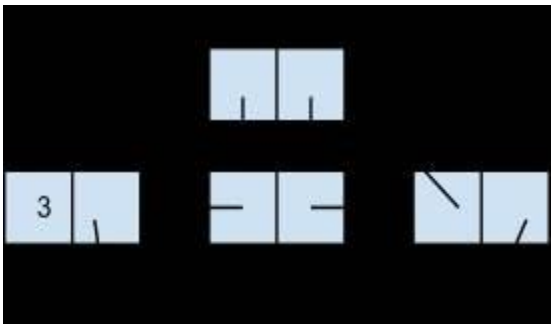
b) **(0,5 puntos)** Escribe las instrucciones en Scheme que han generado la siguiente estructura:



c) **(0,5 puntos)** Dibuja el box&pointer resultante al evaluar la siguiente expresión:

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

d) **(0,5 puntos)** Escribe las expresiones mutadoras en Scheme que modifiquen la estructura representada en el apartado b) para obtener la siguiente:



Ejercicio 3 (2 puntos)

a) **(1,5 puntos)** Define una función `(convertir! exp-s)` que reciba una expresión-S y la convierta, utilizando mutadores, en una lista plana, sustituyendo cada sublista por el carácter < o > en el lugar de los paréntesis. Por simplificar, suponemos que el primer elemento de las listas y sublistas siempre es atómico.

Ejemplo:

```
(define exp-s '(1 (2 3) 4 (5 (6 7) 8) 9))
(convertir! exp-s)
exp-s → (1 < 2 3 > 4 < 5 < 6 7 > 8 > 9)
```

b) **(0,5 puntos)** Explica tu solución con un ejemplo utilizando diagramas box & pointer.

Ejercicio 4 (1,75 puntos)

a) **(1 punto)** Implementa en Scala un contador utilizando estado local que, cada vez que se invoque, devuelva el resultado de aplicar una función al valor del resultado obtenido la vez anterior. Tanto la función a aplicar como el valor de inicio se determinan en el momento de la creación del contador.

Ejemplo:

```
def cuadrado(x:Int) = x*x
val cont = makeContador(2, cuadrado _)
cont() → 4
cont() → 16
```

b) **(0,75 puntos)** Dibuja el diagrama de ámbitos generado tras la ejecución de las siguientes instrucciones:

```
def suma10(x:Int) = x+10
val cont = makeContador(100, suma10 _)
cont()
cont()
```

Ejercicio 5 (1,75 puntos)

Supongamos las siguientes definiciones de clases y traits:

```
class Clase1 {
   def g(x:Int, y:Int) = x+y
}

trait Trait1 {
   def h(x:Int, y:Int) = x*y
}

class Clase2 extends Clase1 with Trait1 {
   override def g(x:Int, y:Int) = x+y+100
   def f(x:Int) = x+30
}

class Clase3 {
   def g(x:Int, y:Int) = x+y+200
}

trait Trait2 extends Clase1 {
   abstract override def g(x:Int, y:Int) = super.g(x,y+10)
}
```

a) **(0,75 puntos)** Indica cuáles de las siguientes definiciones son incorrectas y explica por qué:

```
val a = new Clase2
val b = new Clase3 with Trait2
val c = new Clase2 with Trait2
val d: Clase1 = new Clase2
```

b) **(1 punto)** En las siguientes expresiones tacha las que correspondan a las expresiones incorrectas anteriores. Con las expresiones no tachadas, índica qué valor devuelven o si dan un error y explica por qué devuelven ese valor o por qué resultan en un error.

```
a.h(2,4)
a.g(2,4)
b.g(2,4)
b.h(2,4)
c.g(2,4)
c.h(2,4)
d.f(2)
```

Ejercicio 6 (1,75 puntos)

Supongamos la siguiente definición de las clases `Ping` y `Pong`:

```
import scala.actors.Actor
import scala.actors.Actor._

class Ping(count: Int, pong: Actor) extends Actor {
   def act() {
      for(i <- 1 to count) {
            println("Ping envía Ping")
            pong ! "Ping"
      }
      pong ! "Stop"
   }
}

class Pong extends Actor {
   def act() {
      receive {
         case "Ping" =>
          println("Recibido mensaje")
         case "Stop" =>
            println("Pong se para")
             exit()
      }
   }
}
```

a) **(0,75 puntos)** Di y explica qué aparecería por pantalla al ejecutar las siguientes instrucciones:

```
val pong = new Pong
val ping = new Ping(10,pong)
ping.start
pong.start
```

b) **(1 punto)** Escribe una definición equivalente a las anteriores sin utilizar clases, definiendo `ping` y `pong` como funciones que generan actores usando la palabra **`actor`**.