Nombre: _____

Titulación: _____

Lenguajes y Paradigmas de Programación
Curso 2011-2012
Examen convocatoria extraordinaria C4
Grado en Ingeniería Informática / plan antiguo

Normas importantes

- **La puntuación total del examen es de 9 puntos (la nota de prácticas completa el punto adicional) para el Grado en Informática. Para el plan antiguo se escalará hasta 10 (multiplicando por 1,11)**
- **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 3 horas.**

Ejercicio 1 (1 punto)

Explica los siguientes conceptos:

- **(0,25 puntos)** Definición e implementación de las listas en Scheme. Ejemplos.
- **(0,25 puntos)** Explica las diferentes formas de definir actores en Scala. Ejemplos.
- **(0,25 puntos)** Diferencia entre valor y referencia. Ejemplos.
- **(0,25 puntos)** Tipos de variables en Scala. Ejemplos

Ejercicio 2 (1,25 puntos)

a) **(0,5 puntos)** Define dos implementaciones en Scheme, una recursiva pura y otra iterativa, de la función `(concat-mayores-de lista n)` que tome una lista de cadenas y un número `n` y devuelva la concatenación de las cadenas con tamaño mayor o igual que `n`.

Ejemplo:

```
(concat-mayores-de '("hola", "pep", "automovil", "alicante"), 4) -
> "holaautomovilalicante"
(concat-mayores-de '("hola", "adios", "barcelona", 10) -> ""
```

b) **(0,75 puntos)** Define una función de orden superior que generalice[1] la función anterior `concat-mayores-de`. Cuanto más general sea, mejor. Piensa bien el nombre que le darías, explica lo que haría y define su implementación en Scheme. Por último, escribe cómo implementarías `concat-mayores-de` usando la función general.

[1]Ejemplo de generalización: la función `(filter lista predicado)` permite generalizar todas las funciones concretas que filtran los elementos de una lista. Así, podemos definir una función concreta como `(filtra-numeros-pares lista)` usando la función general `filter`:

```
(define (filtra-numeros-pares lista)
   (filter lista even?))
```

Ejercicio 3 (1,5 puntos)

Supongamos que estamos diseñando una aplicación gráfica en la que codificamos puntos de coordenadas de pantalla (x,y) en forma de tuplas de dos enteros.

a) **(0,35 puntos)** Define una implementación recursiva en Scala de la función `contarPuntos` que toma como parámetros:

- una lista de puntos
- un predicado que se aplica a dos enteros

y que devuelve el número de puntos de la lista que cumplen el predicado.

Por ejemplo, supongamos que la función `enCentroPantalla(x,y)` determina si la coordenada 'x' está en el rango [250,350] y la coordenada 'y' en [150,250]. Un ejemplo de llamada a `contarPuntos` con esta función:

```
contarPuntos(List((330,170), (340,400), (10,100), (290,190)), enCentroPantalla_) => 2
```

b) **(0,4 puntos)** Define en Scala la función `construyeComparadorRango` que tome como parámetros:

- límite inferior y superior de 'x'
- límite inferior y superior de 'y'

y que devuelva un predicado que tome dos coordenadas 'x' e 'y' y compruebe si están dentro de los rangos.

Ejemplo:

```
val p = construyeComparadorRango(0,100,0,100)
contarCoords(List((20,30), (200,3), (3,90)), p) => 2
```

c) **(0,75 puntos)** Utilizando las funciones anteriores, define la función `rangoConMasPuntos` que tome:

- lista de puntos
- lista de 4-tuplas de 4 enteros que representan los rangos

Y que devuelva la 4-tupla que define el rango de la lista de rangos en la que se encuentran más número de puntos:

```
val listaPuntos = List((330,170), (340,400), (10,10), (290,190))
val listaRangos = List((0,100,0,100), (100,200,100,200))
rangoConMasPuntos(listaPuntos, listaRangos) => (0,100,0,100)
```

Ejercicio 4 (1 punto)

Dadas las siguientes definiciones:

```
abstract class Clase1 {
	def f(x:Int) : Int
	def g(x:Int,y:Int) = x*y
}

class Clase2 extends Clase1 {
	def f(x:Int) = x * 2
}

class Clase3 extends Clase2 {
	override def g(x:Int,y:Int) = super.g(x,y) + 100
	override def f(x:Int) = super.f(x+3)
}

trait Trait1 extends Clase2 {
	abstract override def f(x:Int) = super.f(x+2)
}

trait Trait2 extends Clase1 {
	abstract override def f(x:Int) = super.f(x*2)
	abstract override def g(x:Int,y:Int) = super.g(x+2, y)
}
```

a) **(0,5 puntos)** Indica y explica el resultado de las siguientes instrucciones. Si hay algún error, corrígelo y da una versión correcta de la expresión:

Expresión	Resultado / Correción
`val a = new Clase1 with Trait2` `a.g(2,3)`	
`val b = new Clase3 with Trait1` `(b with Trait2).g(1,2)`	
`val c = new Clase3 with Trait2` `(c with Trait1).f(3)`	
`val d = new Clase2 with Trait2 with Trait1` `d.f(20)`	
`val e : Clase1 = new Clase2 with Trait2 with Trait1` `e.g(2,3)`	

b) **(0,5 puntos)** Para cada una de las siguientes expresiones, rellena los huecos para que se produzca el resultado esperado:

Expresión	Resultado
`val a : Clase 1 = ______` `a.g(4,2)`	8
`val b = ______` `b.g(3,2)`	110

`val c : Clase1 = _____` `c.f(10)`	30
`val d = _____` `d.f(10)`	54
`val e : Clase2 = _____` `e.f(4)`	26

Ejercicio 5 (1,5 puntos)

a) **(0,75 puntos)** Define la función `(intercambia-si-suma>! exp-s1 exp-s2 n)` que reciba dos expresiones-S que tienen la misma estructura pero datos diferentes y un número `n`. Deberá intercambiar los elementos entre las expresiones-S si la suma de los elementos en la misma posición es mayor o igual que `n`. Explica tu solución utilizando diagramas caja y puntero.

```
(define exps1 '(1 (2 3 (4) (3 (4 (5 (6 7)))))))
(define exps2 '(5 (1 2 (1) (2 (1 (0 (2 0)))))))
(intercambia-si-suma>! exps1 exps2 6)
exps1 → (5 (2 3 (4) (3 (4 (5 (2 0)))))))
exps2 → (1 (1 2 (1) (2 (1 (0 (6 7)))))))
```

b) **(0,5 puntos)** Dada la siguiente expresión-S, dibuja su estructura jerárquica (arbórea) y su diagrama caja y puntero:

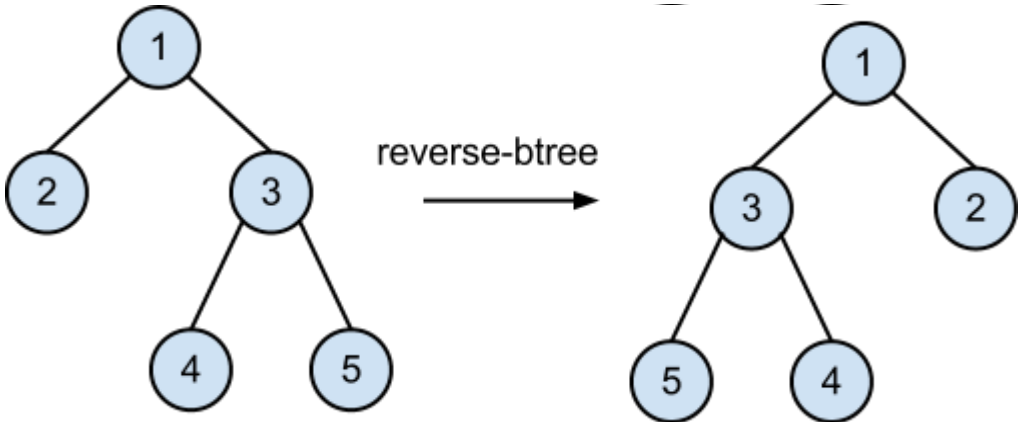
(define x '(1 (2) (3 (4 (5 6) 7) (8))))

c) **(0,25 puntos)** Modifica el diagrama caja y puntero para reflejar la siguiente mutación:

```
(set-cdr! (cdaddr x) (cdddr x))
```

Ejercicio 6 (1,5 puntos)

a) **(0,75 puntos)** Define el procedimiento funcional `(reverse-btree btree)` que reciba un árbol binario y lo invierta.



b) **(0,75 puntos)** Implementa ahora su versión imperativa.

Ejercicio 7 (1,25 puntos)

a) **(1 punto)** Dibuja y explica los ámbitos generados por la ejecución de las siguientes expresiones en Scala. ¿Qué resultado devuelve la última expresión?

```
var a=10
def f(x: Int) = {
   a = x*a
   a
}
def g(x:Int, h:(Int)=>Int) = {
   var cont = x
   ()=>{
      cont = h(cont)
      cont
   }
}
var c = g(10,f)
c()
c()
```

b) **(0,25 puntos)** ¿Se crea alguna clausura? Explica por qué.