Nombre:_____ Grupo:_____

Lenguajes y Paradigmas de Programación
Curso 2012-2013
Tercer parcial

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 3 horas.**

Ejercicio 1 (1,5 puntos)

a) (0,5 puntos) Reescribe el siguiente código en Scala utilizando *pattern matching*:

```
val lista = List("esto","es","el","examen","de","lpp")
if (lista.head.charAt(0).equals('e')) lista.head + " empieza por e" else
    if (lista.head.charAt(0).equals('d')) lista.head + " empieza por d" else
        lista.head + " empieza por otra letra"
```

b) (0,5 puntos) Utilizando *case class*, implementa una jerarquía de clases que funcione como los siguientes ejemplos, teniendo en cuenta que no se pueden añadir más palos en otro fichero fuente:

```
val carta1 = Carta(3,Copas)
val carta2 = Carta(11,Oros)
val carta3 = Carta(1,Espadas)
val carta4 = Carta(4,Bastos)
```

c) (0,5 puntos) Dada la definición de la clase explicaMe, explica qué ocurre cuando se ejecutan las siguientes sentencias:

```
val a = new explicaMe(15)
a.start
```

Indica además si el actor "a" siempre termina su ejecución y por qué.
Aclaración: la invocación a la función de Scala `Random.nextInt(10)` genera un número aleatorio entre 0 y 9, mientras que `Random.nextPrintableChar()` genera un carácter aleatorio.

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

```
import scala.actors.Actor._
import scala.util.Random

class explicaMe(n: Int) extends Actor {
   def act() {

      val actual = self

      for (i <- 1 to n) {
         actor { actual ! Random.nextInt(10) }
         actor { actual ! Random.nextPrintableChar() }
      }

      while (true) {
         receive {
            case 0   => exit()
            case num : Int => println("Numero: " + num)
            case car : Char => println("Caracter: " + car)
        }
      }
   }
}
```

Ejercicio 2 (1,75 puntos)

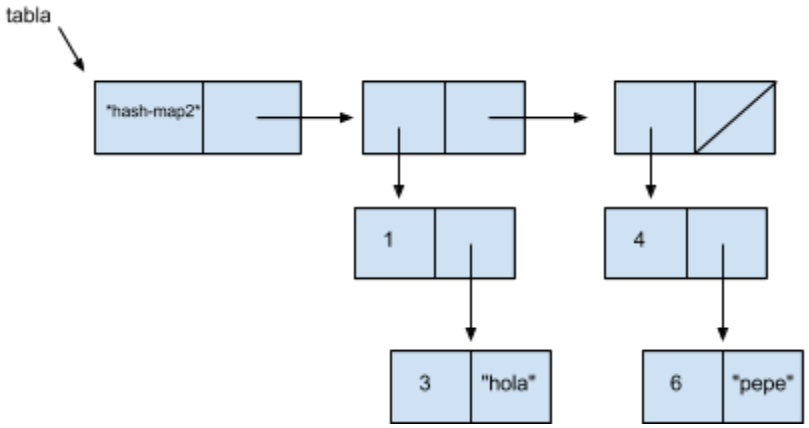
Vamos a definir un tipo de dato mutable `map2` que permite asociar un valor a dos claves. Para ello definimos las funciones `make-map2`, `put-map2!` y `get-map2`:

- `(make-map2)`: construye y devuelve un map2 vacío
- `(put-map2! map2 key1 key2 value)`: guarda (con mutación) en el `map2` un valor asociado a las claves `key1` y `key2`
- `(get-map2 map2 key1 key2)`: devuelve el valor asociado a las claves `key1` y `key2` o `#f` si no existe

Un ejemplo de uso:

```
(define tabla (make-map2))
(get-map2 tabla 1 3) ; -> #f
(put-map2! tabla 1 3 "hola")
(get-map2 tabla 1 3) ; -> "hola"
(put-map2! tabla 4 6 "adios")
(put-map2! tabla 4 6 "pepe")
(get-map2 tabla 4 6) ; -> "pepe"
```

El diagrama *box&pointer* de la estructura resultante de las operaciones anteriores es el siguiente:



a) (0,25 puntos) Implementa la función `(make-map2)` que construye una tabla vacía.

b) (0,75 puntos) Implementa la función `(get-map2 tabla key1 key2)` que devuelve el valor asociado a dos claves o `#f` si no existe

c) (0,75 puntos) Completa las siguientes funciones auxiliares para que puedan ser usadas en `put-map2!`

```
;; Inserta una asociación nueva a continuación de una posición (referencia)
;; del map2
(define (insert-new-node! pos key1 key2 value)
  (let ((node (cons key1 (cons key2 value))))
    (...)))

;; Modifica una asociación con un nuevo valor en una posición
;; del map2
(define (change-node! pos value)
  (...))
```

Ejercicio 3 (1,75 puntos)

Supongamos que queremos codificar un juego de rol en el que aparecen distintos tipos de personas, como magos o guerreros, con las siguientes características:

- Las personas tienen un nombre que no cambia y una puntuación que puede ir modificándose con el tiempo.
- La puntuación es una tupla de 3 elementos que representan los valores de poder, resistencia y velocidad.
- Dependiendo del tipo de persona, los valores iniciales de la puntuación son los siguientes:

Tipo de persona	**Puntos iniciales**
Mago	Poder = 5, Resistencia = 2, Velocidad= 2
Guerrero	Poder = 3, Resistencia = 4, Velocidad= 4

- La puntuación de una persona se puede aumentar añadiéndole una cantidad a alguna de sus características.

Lee detalladamente el siguiente ejemplo de código en el que mostramos el funcionamiento de estas clases:

```
val gandalf = new Mago("Gandalf")
val aragorn = new Guerrero("Aragorn")
val persona = new Persona("Sin nombre")
<console>:8: error: class Persona is abstract; cannot be instantiated
val persona = new Persona("Sin nombre") { var puntos = (0,0,0) }
persona.saluda // -> java.lang.String = Hola, soy Sin nombre
gandalf.saluda // -> java.lang.String = Hola, soy el mago Gandalf
gandalf.puntos // -> (Int, Int, Int) = (5,2,2)
gandalf.aumentaPuntos("Velocidad",2)
gandalf.puntos // -> (Int, Int, Int) = (5,2,4)
```

a) (0,75 puntos) Codifica en Scala las clases `Persona`, `Mago` y `Guerrero`, sus atributos y sus métodos para que cumplan las características anteriores.

b) (0,5 puntos) Define en Scala la función `sumaPuntos` que suma todos los puntos de un equipo de personas. Su perfil es el siguiente:

```
// Devuelve una tupla con la suma de todos los puntos de un equipo
def sumaPuntos(equipo: List[Persona]) : (Int,Int,Int)
```

Importante: haz una implementación *no imperativa*. Puedes hacerla recursiva (llamándose a si misma) o implementarla con alguna llamada a alguna función de orden superior de Scala.

c) (0,5 puntos) Crea una nueva clase `Puntuacion` que represente una puntuación de poder, resistencia y velocidad. Explica cómo tendrías que cambiar la clase `Persona` y sus derivadas para que trabajaran con esta nueva clase en lugar de con tuplas.

Escribe una nueva implementación de la función `sumaPuntos`, ahora con este nuevo perfil:

```
	// Devuelve la puntuación con la suma de todos los puntos de un equipo
	def sumaPuntos(equipo: List[Persona]) : Puntuacion
```

Ejercicio 4 (1,25 puntos)

Dadas las siguientes definiciones:

```
abstract class A {
	def f(x:Int):Int
}
abstract class B extends A {
	def f(x:Int) = x+5
	def g(x:Int):Int
}
class C extends A{
	def f(x:Int) = x+2
}
trait T1 extends A {
	abstract override def f(x:Int) = super.f(x+5)
}
trait T2 extends B {
	abstract override def f(x:Int) = 1 + super.f(x+3)
	def g(x:Int) = 4 + x
}
```

a) (0,5 puntos) Indica si se produce algún error, y si es así explica por qué, en las siguientes instrucciones:

```
val a = new C with T1
val b = new A with T1
val c = new B with T2
val d = new B with T2 with T1
```

b) (0,25 puntos) Indica el resultado (tacha las que provoquen error):

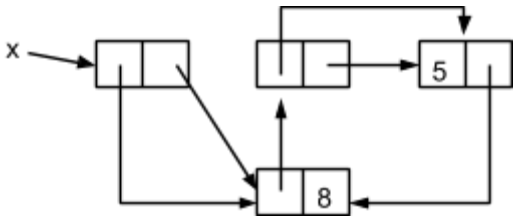
```
a.f(5)
b.f(3)
c.f(1)
c.g(4)
d.f(1)
d.g(2)
```

c) (0,5 puntos) Rellena los huecos para obtener el resultado esperado:

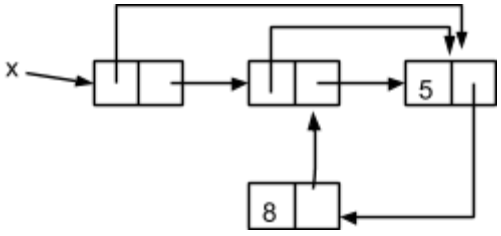
```
val q = new ____ with ____
q.f(5) → 19

val p = new B ____
p.f(2) → 9
```

Ejercicio 5 (1 punto)

a) (0,5 puntos) Escribe las instrucciones que generan el siguiente diagrama box & pointer:



b) (0,5 puntos) Dado el diagrama anterior, escribe las instrucciones que lo han modificado (utilizando como única referencia la x y sin añadir ningún valor ni pareja nueva) de la siguiente forma:



Ejercicio 6 (1,5 puntos)

Implementa en Scala, utilizando **estado local**, una función que construya un almacén. Para utilizar el almacén, son necesarios dos elementos: un mensaje ("guardar" u "obtener") y un entero. En el caso del mensaje "guardar", el entero es el artículo a guardar y devuelve el entero que has guardado. En el mensaje "obtener", el entero indica qué artículo se extrae del almacén (el orden en que se almacenó). En caso que el número de orden exceda la cantidad de artículos del almacén, se devuelve 0 y en caso contrario, se obtiene de forma **recursiva**. Existe un artículo especial a "guardar", el 0, que limpia el almacén (se queda vacío).

Ejemplo de funcionamiento:

```
val almacen1 = make-almacen()
almacen1("guardar", 10) → 10
almacen1("guardar",15) → 15
almacen1("obtener",2) → 15
almacen1("guardar",30) → 30
almacen1("obtener",1) → 10
almacen1("obtener",3) → 30
almacen1("obtener",4) → 0  //No existe, devuelvo 0

val almacen2 = make-almacen()
almacen2("guardar", 100)
almacen2("obtener",1) → 100
almacen2("guardar",0) → 0// Se limpia el almacén
almacen2("guardar",2) → 2
almacen2("obtener,1) → 2
```

Ejercicio 7 (1,25 puntos)

Dado el siguiente objeto singleton y la clase HashMap que implementa una estructura en la que se guardan asociaciones entre claves y valores:

```
object Utils {
   def construirMap[T](lista1: List[String], lista2: List[T]) : List[(String, T)] = {
      if (lista1.isEmpty) Nil
      else (lista1.head, lista2.head) :: construirMap(lista1.tail, lista2.tail)
   }
}

class HashMap[T](val tabla:List[(String, T)]) {
   def this(lista1:List[String], lista2:List[T]) =
                  this(Utils.construirMap(lista1, lista2))
```

```
   // Devuelve el valor asociado a una determinada clave
   // si la clave no existe, devuelve None
   def get(clave:String) : T = {
      buscarClave(tabla, clave)
   }

   // Añade el valor a una clave
   // si la clave ya existe, se actualiza su valor
   // si la clave no existe, se añade la nueva asociación clave/valor
   def put(clave:String, valor:T) : HashMap[T] = {
      val nuevaLista = addClave(tabla, clave, valor)
      new HashMap(nuevaLista)
   }

   private def buscarClave(...) : ... {
      ...
   }

   private def addClave(...) : ... {
      ...
   }
}
```

Responde a las siguientes cuestiones sobre la clase `HashMap`:

a) (0,25 puntos)

- ¿Qué funcionalidad aporta `this()` en esta clase?
- ¿Por qué en la función `put` se hace un `new` de la propia clase?

b) (0,5 puntos) Completa la implementación de las funciones buscarClave() y addClave() de forma recursiva

c) (0,5 puntos) Escribe código de pruebas para esta clase intentando que sea exhaustivo y que pruebe todas sus funcionalidades.