Nombre:_____

Lenguajes y Paradigmas de Programación
Convocatoria extraordinaria C4, curso 2015-16
Examen final

Normas importantes

- **Se debe contestar cada pregunta en las hojas que entregamos. No olvides poner el nombre. Puedes usar lápiz.**
- **El profesor que está en el aula no contestará ninguna pregunta, salvo aquellas que se refieran a posibles errores en los enunciados de los ejercicios.**
- **La duración del examen es de 2,5 horas**

Ejercicio 1 (1 punto)

Escribe la **función recursiva** `(invertir-lista lista-parejas)` que recibe una lista de parejas y devuelve la lista invertida, y en la que además aparecen intercambiados los elementos de todas sus parejas.

Ejemplo:

```
(invertir-lista '((1 . 2) (3 . 4) (5 . 6))) ⇒ {{6 . 5} {4 . 3} {2 . 1}}
```

```
(define (intercambia-pareja pareja)
  (cons (cdr pareja)
        (car pareja)))

(define (invertir-lista lista-parejas)
  (if (null? lista-parejas)
      '()
      (append (invertir-lista (cdr lista-parejas))
              (list (intercambia-pareja (car lista-parejas))))))
```

Ejercicio 2 (1 punto)

Escribe la **función recursiva** `(cuenta-a-b lista)` que recibe una lista que contiene los símbolos a y b repetidos y devuelve una pareja cuyos elementos son el número de as y bs de la lista que aparecen en la lista

Ejemplo:

```
(cuenta-a-b '(a b a b a b)) ⇒ {3 . 3}
(cuenta-a-b '(a a a a)) ⇒ {4 . 0}
```

```
(define (suma-1-izq pareja)
  (cons (+ 1 (car pareja))
        (cdr pareja)))

(define (suma-1-der pareja)
  (cons (car pareja)
        (+ 1 (cdr pareja))))

(define (cuenta-a-b lista-a-b)
  (if (null? lista-a-b)
      (cons 0 0)
      (if (equal? 'a (car lista-a-b))
          (suma-1-izq (cuenta-a-b (cdr lista-a-b)))
          (suma-1-der (cuenta-a-b (cdr lista-a-b))))))
```

Ejercicio 3 (1 punto)

Suponemos la función `(aplica-operadores lista-operadores pareja)` que recibe una lista de símbolos que definen operadores aritméticos (pueden ser cualquiera de los símbolos `+`, `-`, `*`, `/`) y una pareja y que devuelve la lista resultante de aplicar cada operador a los elementos de la pareja (no hace falta que la implementes).

Ejemplo:

```
(aplica-operadores '(- - + *) (5 . 4)) ⇒ {1 1 9 20}
```

Usando la función anterior implementa la función `(aplica-lista-op lista-operadores lista-parejas)` que recibe una lista de operadores y una lista de parejas y devuelve una lista resultante de aplicar todos los operadores de la lista a todas y cada una de las parejas. Para implementar esta función y cualquier otra función auxiliar que necesites, **debes usar funciones de orden superior**.

Ejemplo:

```
(aplica-lista-op '(+ - / *) '((5 . 4) (2 . 3))) ⇒ {9 1 5/4 20 5 -1 2/3 6}
```

```
(define (aplana-lista lista)
  (fold-right append '() lista))

(define (aplica-lista-op lista-op lista-parejas )
  (aplana-lista (map (lambda (pareja) (aplica-operadores lista-op pareja))
                     lista-parejas)))
```

Ejercicio 4 (2 puntos)

Escribe el predicado `(valida-tree? tree)` que recibe un árbol de símbolos `'abierto` y `'cerrado`, y recorre el árbol devolviendo `#t` en caso de que para todos sus nodos (excepto los nodos hoja) se cumpla que el dato del nodo sea el que más ocurrencias tiene en las raíces de los hijos, y `#f` en caso contrario (ver figura). Puedes usar funciones de orden superior y/o recursión mutua.

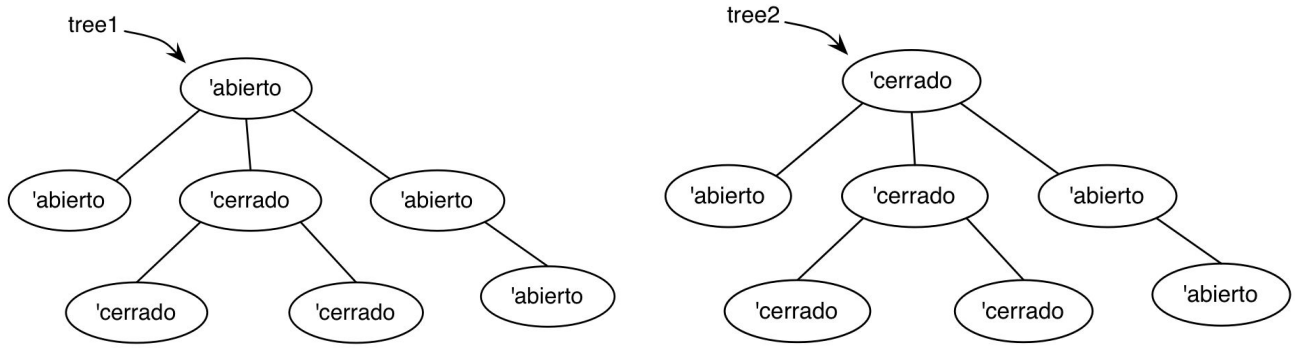
Debes usar las funciones de la barrera de abstracción de árboles (no hace falta que las implementes).Recordamos esta barrera de abstracción:

`(dato-tree tree)`: devuelve el dato de la raíz del árbol
`(hijos-tree tree)`: devuelve la lista de árboles hijos del árbol
`(hoja-tree? tree)`: #t o #f si el árbol es una hoja (no tiene hijos)

También puedes utilizar la función `(ocurrencias lista)` que recibe una lista de símbolos `'abierto` y `'cerrado`, y devuelve una pareja cuya parte izquierda es el número de ocurrencias del símbolo `'abierto`, y su parte derecha el número de ocurrencias del símbolo `'cerrado`. (No tienes que implementar esta función).

Ejemplos:

```
(ocurrencias '(abierto cerrado cerrado cerrado abierto)) ⇒ {2 . 3}
(ocurrencias '(abierto abierto)) ⇒ {2 . 0}
```




```
(valida-tree? tree1)   ⇒ #t
(valida-tree? tree2)   ⇒ #f
```

```
;; Hay varias formas de hacerlo.. esta usa la FOS map para obtener la
;; pareja de las ocurrencias de 'abierto y 'cerrado en las raíces de los hijos:

(define (mayor-car? pareja)
   (> (car pareja) (cdr pareja)))

(define (mayor-cdr? pareja)
   (> (cdr pareja) (car pareja)))

(define (valida-tree? tree)
  (if (hoja-tree? tree)
     #t
     (and (if (equal? (dato-tree tree) 'abierto)
              (mayor-car? (ocurrencias (map dato-tree (hijos-tree tree))))
              (mayor-cdr? (ocurrencias (map dato-tree (hijos-tree tree)))))
          (valida-bosque? (hijos-tree tree)))))

(define (valida-bosque? bosque)
  (if (null? bosque)
     #t
     (and (valida-tree? (car bosque))
          (valida-bosque? (cdr bosque)))))
```

Ejercicio 5 (1 punto)

Escribe la **función recursiva** `(veces-lista dato lista)` que recibe un dato (un símbolo) y una lista estructurada de símbolos y devuelve el número de veces que aparece el dato en la lista. Debes usar la barrera de abstracción de listas estructuradas (no hace falta que la implementes). Recordamos esta barrera de abstracción:

`(car lista)`: devuelve el primer elemento de la lista
`(cdr lista)`: devuelve el resto de la lista
`(hoja? lista)`: devuelve #t o #f si el parámetro es un dato o una lista

Ejemplo:

`(veces-lista 'a '(a j f (k a (b w) a) d c))` ⇒ `3`

```
(define (veces-lista dato lista)
  (cond ((null? lista)
         0)
        ((hoja? lista)
         (if (equal? dato lista)
             1
             0))
        (else
         (+ (veces-lista dato (car lista))
            (veces-lista dato (cdr lista))))))
```

Test (4 puntos)

Cada respuesta errónea **penaliza con 0,12 puntos**. En todas las preguntas (excepto en la primera), **sólo una respuesta es la correcta**. **Redondea la letra** con tu respuesta.

1. (0,4 puntos) Conecta cada lenguaje con el tipo de ejecución de los programas escritos en él (1 fallo: 0,2 puntos, 2 fallos o más: 0 puntos y no descuenta)

1. C -> A
2. Scheme -> B
3. Swift -> A
4. Python -> B
5. C++ -> A
6. Ruby -> A

A. Se ejecuta el código compilado

B. Se ejecuta por un intérprete

2. (0,4 puntos) La definición de “función de orden superior” es:

A. Una función que transforma listas, aplicando otras funciones a sus elementos, como `map`, `filter` o `fold-right`.
B. Una función que contiene en su cuerpo una llamada a la forma especial `lambda`.
C. Una función que recibe como parámetro otra función.
D. Una función que generaliza otra función, añadiendo un parámetro adicional a esta última.

3. (0,4 puntos) ¿Qué aparece por pantalla al ejecutar el siguiente código?

```
let array = [3, 1, 4, 2, 2]
var suma = 0
for i in 1...array[0] {
   suma += array[i]
}
print("La suma es \(suma)")
```

A. La suma es 8
B. La suma es 5
C. La suma es 0
D. La suma es 7

4. (0,4 puntos) El siguiente código produce un error:

```
01  let array = [2, 2, 5, 1]
02  var array2: [String] = []
03  let dic = [1: "Uno", 2: "Dos", 3: "Tres", 4: "Cuatro"]
04  for i in 0..<array.count {
05    array2.append(dic[array[i]])
06  }
07  print(array2)
```

¿Cuál de los siguientes cambios permitiría ejecutar el código sin errores?

A. Sustituir la línea 4 por:
```
for i in 0..<array.count - 1 {
```
B. Sustituir la línea 5 por:
```
if let elem = dic[array[i]] {
      array2.append(elem)
}
```
C. Sustituir la línea 5 por:
```
array2.append(dic[array[i]]!)
```
D. Sustituir la línea 2 por:
```
var array2: [String]? = []
```

5. (0,4 puntos) Supongamos el siguiente código:

```
func foo(opcional: Int?) -> Int {
   if let i = opcional {
      return i
   } else {
      return 30
   }
}
let a: Int? = 10
print(foo(10)+foo(Int("abc"))+foo(a))
```

¿Qué valor se imprime?

A. Da error
B. 30
C. 70
D. 50

6. (0,4 puntos) Supongamos el siguiente código:

```
func cuadrado(x: Int) -> Int {return x*x}
let numeros = [Int](0...5)
```

Y supongamos las siguientes expresiones con funciones de orden superior:

1. `numeros.map(cuadrado).reduce(0, combine: +)`
2. `numeros.map {$0 * $0}.reduce(0, combine: {$0 + $1})`
3. `numeros.map {$0 * $0}.reduce(0, combine: +)`
4. `numeros.reduce(0, combine: +).map(cuadrado)`

¿Cuáles de las expresiones anteriores son correctas y devuelven la suma de los números del array elevados al cuadrado, es decir el valor 55?

A. 1 y 2
B. 2 y 3
C. Sólo 4
D. 1, 2 y 3

7. (0,4 puntos) Supongamos la siguiente función:

```
func foo<A,B>(x: (A) -> B, y: A) -> (A,B) {
    ...
}
```

¿Cuál sería una expresión correcta de la sentencia return?

A. return (y, x(y))
B. return (x, x(y))
C. return (y, x(x))
D. Ninguna de las anteriores

8. (0,4 puntos) Supongamos el siguiente código:

```
struct MiStruct {
   var x = 0
}
class MiClass {
   var x = 0
}
var s = MiStruct()
var c = MiClass()
var s2 = s
var c2 = c
s2.x = 10
c2.x = 10
print("c:\(c.x), s:\(s.x)")
```

¿Qué aparece por pantalla?

A. c:0, s:10
B. c:10, s:10
C. c:0, s: 0
A. c:10, s: 0

9. (0,4 puntos) Supongamos la siguiente función:

```
func prueba(i: Int) -> Int? {
   if i < 100 {
      return i
   } else {
      return nil
   }
}
```

¿Cuál de las siguientes definiciones de una clase A es correcta?

A.

```
class A {
   var a: Int = prueba(0)
   var b = 0
   var c: Int
   init(valor: Int) {
      c = valor
   }
}
```

B.

```
class A {
   var a = 0
   var b = 0
   let c = 10
   init(valor: Int) {
      b = valor
   }
}
```

C.

```
class A {
   var a: Int? = prueba(200)
   var b = 0
   var c: Int
}
```

D.

```
class A {
   var a: Int? = prueba(0)
   var b = 0
   var c: Int
   init(v1: Int, v2: Int) {
      a = v1
      b = v2
   }
}
```

10. (0,4 puntos) Supongamos el siguiente código:

```
protocol P {
    var x : Int { get set }
    var y : Int { get }
}

class ClaseA {
    var x : Int = 0
}
```

¿Cuál de las siguientes definiciones es **errónea**?

A.
`extension ClaseA: P {`
`  var x : Int {`
`  get {`
`  return x*x`
`  }`
`  }`
`}`
	B.
	`class ClaseB: P {`
	`  var x : Int = 0`
	`  var y : Int = 0`
	`  var z : Int = 0`
	`}`

C.
`class ClaseC: P {`
`  var x : Int = 0`
`  var y : Int = 0`
`}`
	D.
	`class ClaseD: ClaseA {`
	`  var z : Int = 0`
	`}`