Nombre:_____

Examen convocatoria extraordinaria (Julio)
Lenguajes y Paradigmas de Programación (Curso 2016-17)

Normas importantes del examen

- El examen consta de una parte de ejercicios **(6 puntos)** y otra preguntas cortas **(2 puntos)** y otra de tipo test **(2 puntos)**
- Los profesores no contestarán ninguna cuestión durante la realización del examen, exceptuando aquellas que estén relacionadas con algún posible error en el enunciado de alguna pregunta.
- La duración del examen es de **2'5 horas**

Ejercicio 1 (1,25 puntos)

a) (0,25 puntos) Escribe el **predicado** `(menor-punto? p1 p2 coord)` que recibe dos parejas que representan dos puntos con sus coordenadas 2D, y un símbolo que denota la coordenada `x` o `y`, y devuelve `true` si la coordenada especificada del primer punto es menor que su correspondiente coordenada del segundo punto.

```
(menor-punto? (cons 2 4) (cons 1 5) 'x)  ⇒ #f
(menor-punto? (cons 2 4) (cons 1 5) 'y)  ⇒ #t
```

```
(define (menor-punto? p1 p2 coord)
  (if (equal? coord 'x)
     (< (car p1) (car p2))
     (< (cdr p1) (cdr p2))))
```

b) (0,5 puntos) Escribe la **función recursiva** `(inserta-punto punto lista-puntos coord)` que recibe una pareja que representa un punto 2D, una lista de puntos 2D y un símbolo que denota la coordenada `x` o `y`, y devuelve una lista de puntos en la que el punto especificado se ha insertado en la lista original en la primera posición que cumpla que su coordenada especificada es menor que la del siguiente punto.

Debes utilizar la función `menor-punto?` definida en el apartado anterior.

```
(define lista-puntos
           (list (cons 10 20) (cons 3 4) (cons 80 9) (cons 60 50)))
(inserta-punto (cons 40 30) lista-puntos 'x)
⇒ {{10 . 20} {3 . 4} {40 . 30} {80 . 9} {60 . 50}}
(inserta-punto (cons 4 70) lista-puntos 'y)
⇒ {{10 . 20} {3 . 4} {80 . 9} {60 . 50} {4 . 70}}
```

```
(define (inserta-punto punto lista-puntos coord)
  (if (null? lista-puntos)
     (list punto)
     (if (menor-punto? punto (car lista-puntos) coord)
        (cons punto lista-puntos)
        (cons (car lista-puntos) (inserta-punto punto (cdr lista-puntos) coord)))))
```

c) (0,5 puntos) Escribe la **función recursiva** `(ordena-puntos lista-puntos coord)` que recibe una lista de puntos 2D y un símbolo que denota la coordenada x o y, y devuelve una lista de puntos en la que se han ordenado los puntos de la lista original con respecto a la coordenada especificada.

Debes utilizar la función `inserta-punto` definida en el apartado anterior.

```
(define lista-puntos
    (list (cons 10  20) (cons 3  4) (cons 80  9) (cons 60  50)))
(ordena-puntos lista-puntos 'x)
⇒ {{3 . 4} {10 . 20} {60 . 50} {80 . 9}}
(ordena-puntos lista-puntos 'y)
⇒ {{3 . 4} {80 . 9} {10 . 20} {60 . 50}}
```

```
(define (ordena-puntos lista-puntos coord)
  (if (null? lista-puntos)
      '()
      (inserta-punto (car lista-puntos)
                     (ordena-puntos (cdr lista-puntos) coord)
                     coord)))
```

Ejercicio 2 (1,5 puntos)

a) (0,75 puntos) Escribe la función `(add n pareja-listas pivote)` que recibe una pareja con dos listas de números y añade el número n a la lista izquierda o a la derecha dependiendo de si es menor o mayor o igual que el número `pivote`.

```
;; Supongamos que la variable pareja-listas contiene la pareja
;; {{10 20 1} . {-10 -2 -30}}
(add 8 pareja-listas 0) ⇒  {{8 10 20 1} . {-10 -2 -30}}
```

```
(define (add n pareja-listas pivote)
  (if (< n pivote)
      (cons (cons n (car pareja-listas))
            (cdr pareja-listas))
      (cons (car pareja-listas)
            (cons n (cdr pareja-listas)))))
```

b) (0,75 puntos) Define la **función recursiva** `(divide lista pivote)` que recibe una lista de un números y un pivote (otro número) y devuelve una pareja con dos listas: la izquierda con todos los números menores que el pivote y la derecha con todos los números mayores o iguales.

Debes usar la función add definida en el apartado anterior.

```
(divide '(8 -10 10 20 -2 -30 1) 0) ⇒ {{-10 -2 -30} . {8 10 20 1}}
```

```
(define (divide lista pivote)
  (if (null? lista)
      (cons '() '())
      (add (car lista) (divide (cdr lista) pivote) pivote)))
```

Ejercicio 3 (1,25 puntos)

Escribe la función `(mayor-rama tree)` que devuelve la lista de los datos de la rama más larga del árbol. Puedes usar recursión o funciones de orden superior. Debes usar las funciones de la barrera de abstracción de árboles: `dato-tree` e `hijos-tree`.

```
(define tree '(35 (10 (4) (6)) (25 (22 (12 (3)) (8)))))
(mayor-rama tree) => {35 25 22 12 3}
```

```
; Versión con recursión mutua

; función auxiliar
(define (mayor-lista l1 l2)
  (if (> (length l1) (length l2))
      l1
      l2))

(define (mayor-rama tree)
  (if (hoja-tree? tree)
      (list (dato-tree tree))
      (cons (dato-tree tree)
            (mayor-rama-bosque (hijos-tree tree)))))

(define (mayor-rama-bosque bosque)
  (if (null? bosque)
      '()
      (mayor-lista (mayor-rama (car bosque))
                   (mayor-rama-bosque (cdr bosque)))))
```

```
; versión FOS

(define (mayor-rama-fos tree)
  (if (hoja-tree? tree)
      (list (dato-tree tree))
      (cons (dato-tree tree) (fold-right mayor-lista '()
                                         (map mayor-rama-fos (hijos-tree tree))))))
```

Ejercicio 4 (1,25 puntos)

Define la función `(comprueba-quiniela resultados quiniela)` que recibe una lista `resultados` con parejas que indican resultados de partidos de fútbol y la lista `quiniela` con valores `1`, `X` o `2`. Ambas listas tienen el mismo número de elementos. La función debe devolver cuántos aciertos tiene la quiniela. Debes utilizar **alguna función de orden superior y definir funciones auxiliares** (no lo hagas todo en una única función).

```
(define resultados
        (list (cons 1 1) (cons 2 1) (cons 2 0) (cons 0 0) (cons 1 2)))
(comprueba-quiniela resultados '(X 1 1 X 2)) ⇒ 5
(comprueba-quiniela resultados '(1 X X 2 1)) ⇒ 0
```

```
(define (convierte-a-quiniela resultado)
  (if (> (car resultado) (cdr resultado))
      1
      (if (< (car resultado) (cdr resultado))
          2
          'X)))

(define (comprueba-aciertos quiniela1 quiniela2)
  (if (null? quiniela1)
      0
      (if (equal? (car quiniela1) (car quiniela2))
          (+ 1 (comprueba-aciertos (cdr quiniela1) (cdr quiniela2)))
          (comprueba-aciertos (cdr quiniela1) (cdr quiniela2)))))

(define (comprueba-quiniela resultados quiniela)
  (comprueba-aciertos (map convierte-a-quiniela resultados) quiniela))
```

Ejercicio 5 (0,75 puntos)

Vamos a representar un cartón de bingo como una lista de sublistas, donde cada sublista representa una línea del cartón. Rellena los huecos para completar la función mutadora `tacha-num!` que reciba un número y un cartón y, si se encuentra en el cartón, lo sustituya por un asterisco. Consideramos que los cartones no tienen números repetidos, pero pueden tener distintos tamaños.

```
(define (tacha-num! num carton)
  (cond ((null? carton) #t)
        ((list? (car carton)) _____________________)
        ((equal? num (car carton)) _____________________) ;
        (else _____________________)))

(define carton '((3 40 2)
                 (25 12 33)
                 (20 10 22)))
(tacha-num! 40 carton)
carton ⇒  {(3 * 2)
           (25 12 33)
           (20 10 22)}
```

```
(define (tacha-num! num carton)
  (cond ((null? carton) #t)
        ((list? (car carton)) (tacha-num! num (car carton)))
        ((equal? num (car carton)) (set-car! carton '*))
        (else (tacha-num! num (cdr carton)))))
```

Preguntas cortas (2 puntos)

- Cada pregunta tiene una puntuación de **0,4 puntos**.

1. Indica brevemente 4 características funcionales del lenguaje Swift **(0,1 punto por característica correcta)**

- Posibilidad de definir funciones
- Funciones como objetos de primera clase: expresiones lambda y clausuras
- Funciones de orden superior sobre colecciones: map, filter, reduce, etc.
- Inmutabilidad: let
- Semántica de tipos valor con struct

2. Ordena las siguientes parejas de lenguajes de programación, indicando en cada caso cuál es anterior y cuál posterior **(0,1 puntos por ordenación correcta, -0,1 por ordenación incorrecta)**:

Lenguajes	Anterior	Posterior
Java, Python	Python	Java
Smalltalk, Prolog	Prolog	Smalltakl
C++, Java	C++	Java
C#, Java	Java	C#

3. Rellena el hueco siguiente con un **bucle ~~for~~** que guarde en la variable `suma` la suma de todos los valores devueltos por la función `obtenerValores()`:

```
let valores: [Int] = obtenerValores()
var suma = 0

    for i in valores {
        suma += i
    }

print("La suma de todos los valores es \(suma)")
```

4. Escribe qué aparece por pantalla al ejecutar el siguiente código?

```
let parejas = [(2,7), (3,5), (10,4), (5,5)]
let resultado =
     parejas.filter({$0.0 % 2 == 0}).map({($0.1, $0.1)})
print(resultado)
```

```
[(7, 7), (4, 4)]
```

5. Escribe el código que debería ir en el `reduce` para que se devuelva una tupla cuya parte izquierda tenga el número de elementos del array y su parte derecha la suma de todos sus elementos:

```
let nums = [1,2,3,4,5,6,7]
nums.reduce(                          ) ⇒ (7, 28)
```

```
    (0,0), {($0.0+1, $0.1+$1)}
```

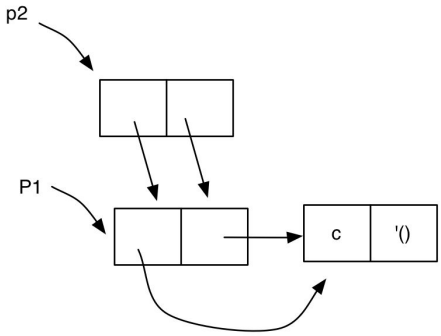
Preguntas de tipo test (2 puntos)

- Cada pregunta tiene una puntuación de **0,4 puntos**.
- Todas las preguntas de tipo test tienen **una única respuesta correcta**.
- Redondea claramente la respuesta que consideres correcta.
- Las respuestas incorrectas tienen una **penalización de 0,1 puntos**.

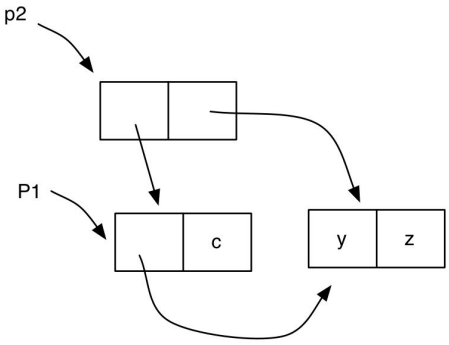
1. ¿Cuál es el diagrama box & pointer resultante de ejecutar las siguientes instrucciones? (Utiliza la última página del examen para hacerlo en sucio)

```
1. (define p1 (cons (list 'a) 'c))
2. (define p2 (list p1 (cons 'y 'z)))
3. (set-car! (car p1) (cadr p2))
4. (set-cdr! p2 (car p1))
```

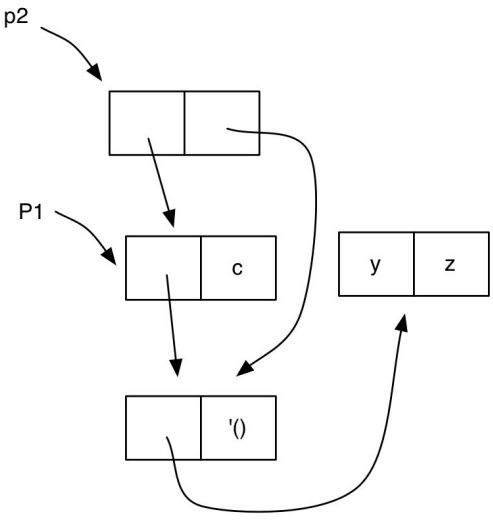
1.



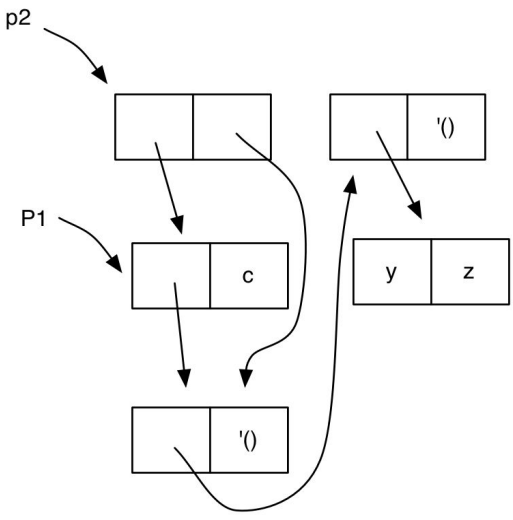
2.



3.



4.



2. En las siguientes instrucciones se asigna a `foo` un valor de una clave que no existe en el diccionario `dic`. ¿Qué sucede?

```
let dic = [1: "Uno", 2: "Dos", 3: "Tres", 4: "Cuatro"]
var foo = dic[5]
```

A. La sentencia provoca un error de compilación
B. La sentencia provoca un error de ejecución
C. Se guarda en foo el valor "" (cadena vacía)
D. **Se guarda en foo el valor `nil`**

3. Supongamos un protocolo `P`:

```
protocol B {
   var a: Int {get}
   var b: Int? {get set}
}
```

A continuación se muestran posibles definiciones de tipos que adoptan `P`. ¿Cuál de ellas **provocaría un error de compilación**?.

A.
`struct A: B {`
`  let a = 10`
`  var b: Int? = 10`
`}`
	B.
	`class A: B {`
	`  var a: Int {`
	`  return b! + 10`
	`  }`
	`  var b: Int? = nil`
	`}`

C.
`class A: B {`
`  var a = 0`
`  var b: Int {`
`  return a + 10`
`  }`
`}`
	D.
	`struct A: B {`
	`  let a = 10`
	`  var b: Int?`
	`}`

4. Dado el siguiente código:

```
let x = 5
func foo(_ x: Int) -> (()->Int, ()->Int) {
      func f() -> Int {
            return x + x
      }
      func g() -> Int {
            return f()
      }
      return (f,g)
}
```

Indica la llamada correcta:

A.
 	
`let y = foo(x+2)`
`y[0]()`
`y[1]()`
	B.
	
	`let y = foo(x+2)`
	`y.0`
	`y.1`

C.
 	
`let y = foo(x+2)`
`y.0()`
`y.1()`
	D.
	
	`let y = foo(x+2)`
	`y.f()`
	`y.g()`

5. Supongamos el siguiente código que define un árbol binario

```
indirect enum ArbolBinario<T>{
      case nodo(T, ArbolBinario<T>, ArbolBinario<T>)
      case vacio
}
```

¿Cuál de las siguientes expresiones es la correcta para crear un árbol binario con 10 en la raíz, 8 como hijo izquierdo y 12 como hijo derecho?

A.

```
let a: ArbolBinario = .nodo(12,
                            .nodo(5, nil, nil),
                           .nodo(18, nil, nil))
```

B.

```
let a: ArbolBinario = .nodo(12,
                            .nodo(5, .vacio, .vacio),
                           .nodo(18, .vacio, .vacio))
```

C.

```
let a = ArbolBinario(12,
                     ArbolBinario(5, .vacio, .vacio),
                     ArbolBinario(18, .vacio, .vacio))
```

D.

```
let a: ArbolBinario = .nodo(12<Int>,
                            .nodo(5<Int>, .vacio, .vacio),
                           .nodo(18<Int>, .vacio, .vacio))
```