Práctica 8: Programación funcional en Swift (1)

Entrega de la práctica

Para entregar la práctica debes subir a Moodle el fichero `practica08.swift` con una cabecera inicial con tu nombre y apellidos, y las soluciones de cada ejercicio separadas por comentarios. Cada solución debe incluir:

- La **definición de las funciones** que resuelven el ejercicio.
- Una visualización por pantalla de todos los ejemplos incluidos en el enunciado que **demuestren qué hace la función**.

> **Importante**
>
> Antes de proceder a realizar los ejercicios de la práctica te recomendamos que realices y pruebes los ejemplos presentados en el seminario de Swift [https://domingogallardo.github.io/apuntes-lpp/seminarios/seminario2-swift/seminario2-swift.html] y los apuntes de teoría [https://domingogallardo.github.io/apuntes-lpp/teoria/tema05-programacion-funcional-swift/tema05-programacion-funcional-swift.html] de la primera parte del tema de Programación Funcional en Swift.

Ejercicios

Ejercicio 1

a) Implementa en Swift la función recursiva `prefijos(prefijo:palabras:)` que recibe una cadena y un array de palabras. Devuelve un array de `Bool` con los booleanos resultantes de comprobar si la cadena es prefijo de cada una de las palabras de la lista.

Ejemplo:

```swift
let array = ["anterior", "antígona", "antena"]
let prefijo = "ante"
print("prefijos(prefijo: \(prefijo), palabras: \(array))")
print(prefijos(prefijo: prefijo, palabras: array))
// Imprime:
// prefijos(prefijo: ante, palabras: ["anterior", "antígona",
"antena"])
// [true, false, true]
```

b) Implementa en Swift la función recursiva `parejaMayorParImpar(numeros:)` que recibe un array de enteros positivos y devuelve una pareja con dos enteros: el primero es el mayor número impar y el segundo el mayor número par. Si no hay ningún número par o impar se devolverá un 0.

```swift
let numeros = [10, 201, 12, 103, 204, 2]
print("parejaMayorParImpar(numeros: \(numeros))")
print(parejaMayorParImpar(numeros: numeros))
// Imprime:
// parejaMayorParImpar(numeros: [10, 201, 12, 103, 204, 2])
// (201, 204)
```

Ejercicio 2

a) Implementa en Swift la **función recursiva** `compruebaParejas(_:funcion:)` con el siguiente perfil:

```swift
([Int], (Int) -> Int) -> [(Int, Int)]
```

La función recibe dos parámetros: un `Array` de enteros y una función que recibe un entero y devuelve un entero. La función devolverá un array de tuplas que contiene las tuplas formadas por aquellos números contiguos del primer array que cumplan que el número es el resultado de aplicar la función al número situado en la posición anterior.

Ejemplo:

```swift
func cuadrado(x: Int) -> Int {
   return x * x
}
print(compruebaParejas([2, 4, 16, 5, 10, 100, 105], funcion:
```

```
cuadrado))
// Imprime [(2,4), (4,16), (10,100)]
```

b) Implementa en Swift la **función recursiva** `coinciden(parejas: [(Int,Int)], funcion: (Int)->Int)` que devuelve un array de booleanos que indica si el resultado de aplicar la función al primer número de cada pareja coincide con el segundo.

```
let array = [(2,4), (4,14), (4,16), (5,25), (10,100)]
func cuadrado(x: Int) -> Int {
   return x * x
}
print("Resultado coinciden:  \(coinciden(parejas: array,
funcion: cuadrado))\n")
// Imprime: Resultado coinciden:  [true, false, true, true,
true]
```

Ejercicio 3

Supongamos que estamos escribiendo un programa que debe tratar movimientos de cuentas bancarias. Define un enumerado `Movimiento` con valores asociados con el que podamos representar:

- Depósito (valor asociado: `(Double)`)
- Cargo de un recibo (valor asociado: `(String, Double)`)
- Cajero (valor asociado: `(Double)`)

Y define la función `aplica(movimientos:[Movimiento])` que reciba un array de movimientos y devuelva una pareja con el dinero resultante de acumular todos los movimientos y un array de Strings con todos los cargos realizados.

Ejemplo:

```
let movimientos: [Movimiento] = [.deposito(830.0),
.cargoRecibo("Gimnasio", 45.0), .deposito(400.0),
.cajero(100.0), .cargoRecibo("Fnac", 38.70)]
print(aplica(movimientos: movimientos))
//Imprime (1046.3, ["Gimnasio", "Fnac"])
```

Ejercicio 4

Implementa en Swift un tipo enumerado recursivo que permita construir árboles binarios de enteros. El enumerado debe tener

- un caso en el que guardar tres valores: un `Int` y dos árboles binarios (el hijo izquierdo y el hijo derecho)
- otro caso constante: un árbol binario vacío

Llamaremos al tipo `ArbolBinario` y a los casos `nodo` y `vacio`.

Impleméntalo de forma que el siguiente ejemplo funcione correctamente:

```
let arbol: ArbolBinario = .nodo(8, .nodo(2, .vacio, .vacio),
.nodo(12, .vacio, .vacio))
```

Implementa también la función `suma(arbolb:)` que reciba una instancia de árbol binario y devuelva la suma de todos sus nodos:

```
print(suma(arbolb: arbol))
// Imprime: 22
```

Ejercicio 5

Implementa en Swift un tipo enumerado recursivo que permita construir árboles de enteros usando el mismo enfoque que en Scheme: un nodo está formado por un dato (un `Int`) y una colección de árboles hijos. Llamaremos al tipo `Arbol`.

Impleméntalo de forma que el siguiente ejemplo funcione correctamente:

```
/*
Definimos el árbol

    10
   / | \
  3  5  8
  |
  1

*/

```

```
let arbol1 = Arbol.nodo(1, [])
let arbol3 = Arbol.nodo(3, [arbol1])
let arbol5 = Arbol.nodo(5, [])
let arbol8 = Arbol.nodo(8, [])
let arbol10 = Arbol.nodo(10, [arbol3, arbol5, arbol8])
```

Implementa también la función `suma(arbol:cumplen:)` que reciba una instancia de árbol y una función `(Int) -> Bool` que comprueba una condición sobre el nodo. La función debe devolver la suma de todos los nodos del árbol que cumplan la condición.

Implementa la función usando la misma estrategia que ya utilizamos en Scheme de definir una función auxiliar `suma(bosque:cumplen:)` y una recursión mutua.

```
func esPar(x: Int) -> Bool {
    return x % 2 == 0
}

print("La suma del árbol es: \(suma(arbol: arbol10, cumplen:
esPar))")
// Imprime: La suma del árbol genérico es: 18
```

Ejercicio 6

a) Define la función `maxOpt(_ x: Int?, _ y: Int?) -> Int?` que devuelve el máximo de dos enteros opcionales. En el caso en que ambos sean `nil` se devolverá `nil`. En el caso en que uno sea `nil` y el otro no se devolverá el entero que no es `nil`. En el caso en que ningún parámetro sea `nil` se devolverá el mayor.

Ejemplo:

```
let res1 = maxOpt(nil, nil)
let res2 = maxOpt(10, nil)
let res3 = maxOpt(-10, 30)
print("res1 = \(String(describing: res1))")
print("res2 = \(String(describing: res2))")
print("res3 = \(String(describing: res3))")
// Imprime:
// res1 = nil
// res2 = Optional(10)
// res3 = Optional(30)
```

b1) Escribe una nueva versión del ejercicio 1b) que permita recibir números negativos y que devuelva una pareja de `(Int?, Int?)` con `nil` en la parte izquierda y/o derecha si no hay número impares o pares.

Ejemplo:

```
let numeros2 = [-10, 202, 12, 100, 204, 2]
print("parejaMayorParImpar2(numeros: \(numeros2))")
print(parejaMayorParImpar2(numeros: numeros2))
// Imprime:
// parejaMayorParImpar2(numeros: [-10, 202, 12, 100, 204, 2])
// (nil, Optional(204))
```

b2) Escribe la función `sumaMaxParesImpares(numeros: [Int]) -> Int` que llama a la función anterior y devuelve la suma del máximo de los pares y el máximo de los impares. El array de números tendrá como mínimo un elemento, por lo que el valor devuelto por la función será un `Int` (no será `Int?`).

```
print("sumaMaxParesImpares(numeros: \(numeros2))")
print(sumaMaxParesImpares(numeros: numeros2))
// Imprime:
// sumaMaxParesImpares(numeros: [-10, 202, 12, 100, 204, 2])
// 204
```
