LENGUAJES, TECNOLOGÍAS Y PARADIGMAS DE PROGRAMACIÓN

TEMA 3:

PROGRAMACIÓN FUNCIONAL

PARTE I

(SOLUCIONES EJERCICIOS DE AULA)

Salvador Lucas,
Javier Piris, María José Ramírez, María José Vicent y Germán Vidal

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería Informática etsinf

PARTE I: TIPOS EN PROGRAMACIÓN FUNCIONAL

1. Indica cuál de las siguientes afirmaciones es **CIERTA** para las siguientes definiciones:

```
data Color = Rojo | Amarillo | Blanco
data Flor = Rosa Color | Margarita Color | Geranio Color
type Ramo = [Flor]
```

   - [A] El tipo definido `Ramo` es un tipo algebraico paramétrico.
   - [B] El tipo definido `Flor` es un tipo algebraico recursivo.
   - [×] El tipo definido `Color` es un tipo algebraico, cuyos valores son `Rojo`, `Amarillo` y `Blanco`.
   - [D] `[Rosa Rojo, Amarillo]` es un valor de tipo `Ramo`.

2. Dada la siguiente expresión de tipo

$$[\mathtt{a}] \to \mathtt{Int}$$

   los constructores de tipo que aparecen en ella son:

   - [×] $[\_], \to$
   - [B] `a, Int`
   - [C] $[\mathtt{a}], \to, \mathtt{Int}$
   - [D] `a`

3. Indica cuál de las siguientes expresiones define un tipo polimórfico.

   - [×] $[\mathtt{a}] \to \mathtt{Bool}$
   - [B] $[\mathtt{Int}] \to \mathtt{Bool}$
   - [C] $\mathtt{Int} \to \mathtt{Bool}$
   - [D] `Bool`

4. Indica cuál de las siguientes afirmaciones es cierta en relación al siguiente programa funcional:

```
pippo x
   | x > 0          = 0
   | otherwise      = "double"
```

   [A] La función `pippo` es de orden superior.

   [×] El programa da un error de tipo.

   [C] La función `pippo` está indefinida para los negativos.

   [D] La función `pippo` está definida para datos de cualquier tipo.

5. Dadas las funciones estándar `head` y `tail`, que computan la cabeza y cola de una lista, indica cuál de las siguientes funciones [*no*] computa una tupla formada por los dos primeros elementos de una lista de dos o más elementos:

   [A] `f (x:y:xs) = (x,y)`

   [×] `f ((x,y):xs) = (x,y)`

   [C] `f (xs) = ((head xs),(head (tail xs)))`

   [D] `f (x:xs) = (x,(head xs))`

6. Indica para cuál de las siguientes funciones Haskell infiere el tipo `f :: [[a]] ->(a, [a])`:

   [×] `f ((x:ys):xs) = (x,ys)`

   [B] `f ((x:ys),xs) = (x,xs)`

   [C] `f [[0]]= (x,[0])`

   [D] `f ([x]:[])= (x,[])`

7. Indica qué representa la expresión $[\texttt{x} \mid \texttt{x} \leftarrow [\texttt{1..n}], \texttt{n `mod` x} == 0]$

   [×] los divisores de `n`.

   [B] los múltiplos de `n`.

   [C] los pares comprendidos entre `1` y `n`.

   [D] los impares comprendidos entre `1` y `n`.

8. Indica cuál de las siguientes afirmaciones es **falsa**:

   [×] El tipo de la función `id x = x`, inferido por cualquier intérprete de Haskell es `id:: Int → Int`

   [B] En notación *curryficada*, las funciones de más de un argumento se interpretan como funciones que toman un único argumento y devuelven como resultado otra función con un argumento menos

   [C] En un lenguaje con orden superior los argumentos de una función pueden ser, a su vez, funciones.

   [D] En notación *curryficada*, si $t_1, t_2, \ldots, t_n, t_r$ son tipos válidos, entonces $t_1 \to t_2 \to \ldots \to t_n \to t_r$ es el tipo de una función con n argumentos

9. Indica cuál de las siguientes ecuaciones define una función `f :: Int− > Int` que haga válida `mitad(f x) = x` siendo `mitad = (‘div‘ 2)`

   [A] `f x = x`

   [B] `f x = f(f x)`

   [×] `f = 2∗`

   [D] `f x = x ∗ x`

10. Indica en cuál de las siguientes ecuaciones se usa una función aplicada parcialmente:

    [A]
    ```
    potencia x (2*n) = square (potencia x n)
            where square x = x*x
    ```

    [×]
    ```
    suma2 l = map (+1) (map (+1) l)
    ```

    [C]
    ```
    facAux 0 p = p
    facAux n p = facAux (n-1) (n*p)
    ```

    [D]
    ```
    nor x y = not (x || y)
    ```

11. Si un lenguaje tiene comprobación estática de tipos, entonces:

    [A] la comprobación de tipos se hace mediante evaluación perezosa.

    [×] se conoce el tipo de todas las expresiones en tiempo de compilación.

    [C] se conoce el tipo de todas las expresiones en tiempo de ejecución.

    [D] es necesario declarar el tipo de todas las expresiones.

12. Indica cuál de las siguientes afirmaciones es $\boxed{falsa}$, en relación con la regla
```
fst (x,f) = x
```

- [A] dicha regla define una función de un único argumento, de tipo "tupla de dos elementos"
- [B] Dicha regla se define por *pattern matching*.
- [×] dicha regla define una función de orden superior
- [D] dicha regla define una función cuyo tipo es `fst :: (a, b) → a`

13. Completar la definición de la siguiente función de orden superior que, dada una función binaria f y dos listas, devuelve una lista cuyos elementos se calculan aplicando la función f a los correspondientes elementos de la lista:

```
zipWith f (a : as) (b : bs) = ____
zipWith _ _ _ = [ ]
```

- [A] f (a,b) ++ zipWith f as bs
- [B] [f a b] : zipWith f as bs
- [×] f a b : zipWith f as bs
- [D] ninguna de las anteriores

14. Indica cuál de las siguientes funciones tiene el tipo:

$$(\texttt{a} -> \texttt{Bool}) -> [\texttt{a}] -> \texttt{Bool}$$

- [A]
```
until p f x
| p x = x
| otherwise = until p f (f x)
```
devuelve lista
- [×]
```
any p xs = or (map p xs)
```
- [C]
```
dropWhile p [] = []
dropWhile p (x:xs) = if (p x) then dropWhile p xs
```
devuelve lista falta else
- [D]
```
filter p xs = [k | k <- xs, p k]
```
devuelve lista: k <- xs

15. Indica cuál de las siguientes declaraciones define un tipo de datos cuyos elementos son o bien valores enteros o bien caracteres:

[A] `data LetraOEntero = Int | Char`

[B] `type LetraOEntero = Int | Char`

[×] `data LetraOEntero = Entero Int | Letra Char`

Entero/Letra son constructores de dato necesarios

[D] `type LetraOEntero = Entero Int | Letra Char`

16. Completa el siguiente programa funcional, que toma como argumentos una lista de funciones de tipo (`a ->b`) y un elemento `e` (de tipo `a`), y devuelve la lista de elementos de tipo `b` que resulta de aplicar cada función de la lista al elemento de entrada `e`:

```
applylist :: [a → b] → a → [b]
applylist [ ] y = [ ]
applylist (x : xs) y = ____________
```

[×] `(x y):(applylist xs y)`

[B] `[x(y)]:applylist(xs y)`

[C] `[(x y):applylist(xs,y)]`

[D] `[x:y:applylist(xs)]`

17. Indica qué computa la siguiente función:

```
examen x xs = [i | (e,i) <- zip xs [1.. length xs], e==x]
```

[A] los elementos de la lista `xs` que son iguales a `x`.

[B] una lista con el elemento `x` repetido tantas veces como `length xs`.

[×] las posiciones en las que `x` aparece en la lista `xs`.

[D] multiplica todos los elementos de la lista `xs` por `x`.

18. Sabiendo que `f :: a → a`, indica cuál es el tipo inferido por `Haskell` para la función `fix` que se define mediante la siguiente ecuación

$$\texttt{fix f} = \texttt{f(fix f)}$$

[×] `fix :: (a → a) → a`

[B] `fix :: (a → a) → a → a`

[C] `fix :: a → a`

[D] `fix :: a`

19. Dadas las funciones estándar `fact` y `sqr` definidas sobre los números enteros, indica cuál es el tipo de la siguiente expresión `[fact, sqr]`:

[A] `[Int]`

[B] `[String, String]`

[×] `[Int ->Int]`

[D] `[(Int, Int)]`

20. El tipo abstracto de datos `Pila a` (que representa una pila de valores de tipo `a`) consta de las siguientes operaciones:

```
pilaVacia :: Pila a
meteEnPila :: a → Pila a → Pila a
sacaDePila :: Pila a → (a, Pila a)
topeDePila :: Pila a → a
estaVaciaPila :: Pila a → Bool
```

Suponiendo la implementación de las pilas como listas

$$\texttt{type Pila a} = [\texttt{a}]$$

indica cuál de las siguientes ecuaciones define de forma más eficiente (menor coste) la operación `meteEnPila`.

[A] `meteEnPila x xs = x ++ xs` ++ concatena dos listas

[×] `meteEnPila x xs = (x : xs)` sintaxis : -> elemento:lista

[C] `meteEnPila x xs = xs ++ [x]`

[D] `meteEnPila x xs = xs`

21. Indica cuál de las siguientes afirmaciones es **FALSA**:

- [×] Los tipos renombrados no pueden ser paramétricos, por ejemplo no es correcto `type Par a = (a,a)`.
- [B] Los renombramientos (alias o sinónimos) de tipo se definen usando la palabra clave `type`.
- [C] Pueden definirse renombramientos de tipos predefinidos o de tipos definidos por el usuario.
- [D] Los tipos renombrados no pueden ser recursivos, por ejemplo no es correcto `type Arbol = [Arbol]`.

22. Dado el siguiente tipo algebraico que define un árbol de valores de tipo `a`

$$\texttt{data Arbol a} = \texttt{Vacio} \mid \texttt{Nodo a [Arbol a]}$$

indica qué calcula la siguiente función

```
sobreArbol :: (a → Bool) → Arbol a → Bool
sobreArbol p Vacio = True
sobreArbol p (Nodo x xs) = p x && and (map(sobreArbol p) xs)
```

donde `&&` y `and` son los operadores predefinidos de la conjunción sobre valores booleanos y sobre listas de valores booleanos respectivamente.

- [A] Devuelve `True` si algún elemento del árbol cumple la condición `p`
- [×] Devuelve `True` si todos los elementos del árbol cumplen la condición `p`
- [C] Devuelve `True` si la raíz del árbol cumple la condición `p` o si el árbol es vacío
- [D] Devuelve `True` si el árbol es vacío y `False` en caso contrario

23. Dada la siguiente función

```
triplas :: Int → [(Int, Int, Int)]
triplas n = [(x, y, z)|x ← [1..n], y ← [x..n], z ← [y..n]]
```

indica cuál es el resultado de evaluar la expresión `triplas 3`.

- [A] `[(1,1,1),(2,2,2),(3,3,3)]`
- [×] `[(1,1,1),(1,1,2),(1,1,3),(1,2,2),(1,2,3),(1,3,3),(2,2,2), (2,2,3), (2,3,3),(3,3,3)]`
- [C] `[(1,1,1),(1,1,2),(1,1,3),(1,2,1),(1,2,2),(1,2,3),(1,3,1), (1,3,2),(1,3,3),(2,1,1),(2,1,2),(2,1,3),(2,2,1),(2,2,2), (2,2,3),...]`
- [D] `[(1,1,1),(1,2,2),(1,3,3),(2,2,2),(2,3,3),(3,3,3)]`

24. Dada la siguiente definición

```
multiploDe :: Int → Int → Bool
multiploDe p n = n `mod` p == 0
```

33 `mod` 12 = 9

que comprueba si su segundo argumento es múltiplo del primero, podemos obtener una función `esPar :: Int → Bool` que comprueba si un número es par mediante la siguiente aplicación parcial:

- [×] `esPar = multiploDe 2`
- [B] `esPar x = multiploDe x 2`
- [C] `esPar = multiploDe 0`
- [D] ninguna de las anteriores

25. Indica con cuál de las siguientes definiciones de tipo se construyen árboles binarios de elementos de tipo `a`:

- [×] `data Arbol a = Vacio | Nodo (Arbol a) a (Arbol a)`
- [B] `data Arbol a = Vacio | Nodo a [Arbol a]`
- [C] `data Arbol a = Vacio | Nodo a (Arbol a)`
- [D] `data Arbol a = Vacio | a (Arbol a) (Arbol a)`

26. Completar la definición de la función `zip3` que toma tres listas y devuelve una lista de triplas:

```
zip3 (a : x) (b : y) (c : z) = ____
zip3 x y z = [ ]
```

- [A] `[a,b,c] ++ zip3(x,y,z)`
- [B] `(a b c):zip3(x,y,z)`
- [×] `(a,b,c):zip3 x y z` tuplas separadas por comas
- [D] `a:b:c:(zip3 x y z)`

27. Dada la siguiente definición de un tipo algebraico de datos:

```
data MyTree a = Branch (MyTree a) a (MyTree a) | Void
```

indica cuál de las siguientes afirmaciones es **FALSA**:

- [A] Es una definición de tipo paramétrico.
- [B] El término `Branch Void 'a' Void` es de tipo `MyTree Char`, que es una concreción de la definición dada.
- [×] Podríamos tener una definición equivalente de la siguiente forma:

  ```
  type MyTree a = Branch (MyTree a) a (MyTree a) | Void
  ```

- [D] Los constructores de dato de la definición dada en el enunciado son `Branch` y `Void`.

28. La expresión `type Complejo = (Float,Float)` define

- [A] una nueva clase de tipos.
- [×] un sinónimo (renombramiento) del tipo `(Float,Float)`.
- [C] un nuevo tipo de datos algebraico para los números complejos.
- [D] una instancia la clase `Complejo` con el tipo `Float`.

29. Indica el tipo inferido por Haskell para la función `f x y = not x && y`.

- [A] `f :: a ->Bool ->a`
- [B] `f :: Bool ->a ->Bool`
- [C] `f :: Bool ->[Bool] ->[Bool]`
- [×] `f :: Bool ->Bool ->Bool`

30. Supongamos que, como se hace a veces en ciertos textos de diseño lógico, deseamos utilizar los operadores `(+)` y `(*)` para definir la *disyunción* y *conjunción* de booleanos. Marca la respuesta **CORRECTA**.

[×] Deberíamos introducir una instancia de la clase `Num` para el tipo `Bool` definiendo adecuadamente los operadores `(+)` y `(*)` declarados en `Num` para que implementen la disyunción y la conjunción de booleanos.

[B] Puesto que la *disyunción* y *conjunción* de booleanos son funciones predefinidas del tipo `Bool` (`||` y &&, respectivamente), no podemos volver a definirlas ni aunque usemos otros nombres.

[C] Dado que `Bool` es un tipo predefinido de Haskell, no es posible definir funciones nuevas para los booleanos. En particular, no es posible definir instancias de ninguna clase de Haskell para el tipo `Bool`.

[D] Dado que `Bool` no es un tipo numérico, no es posible utilizar operadores (infijos) como `(+)` y `(*)` con los booleanos.

31. Dada la siguiente definición de un tipo algebraico, indica cuál de las siguientes afirmaciones es **CORRECTA**:

```
data Shape = Circle Float | Rectangle Float Float
```

[A] Es una definición de tipo recursiva.

[B] `Circle 2 4` es un valor de tipo `Shape`.

[×] `Circle` y `Rectangle` son constructoras de dato.

[D] Es una definición de tipo paramétrica.

32. ¿Qué opción es **CORRECTA** para una función `primo x` que comprueba si un número `x>1` es primo?

[×] `primo x = 2 == length ([d | d <- [1..x], x ‘mod‘ d == 0])`

[B] `primo x = 1 == length ([d | d <- [1..x], x ‘mod‘ d == 0])`

[C] `primo x = 1 == length ([d | d <- [1..x], x ‘div‘ d == 0])`

[D] `primo x = 2 == length ([d | d <- [1..x], x ‘div‘ d == 0])`

33. La currificación (currying) en Programación Funcional:

- [A] Establece el orden en el que se evalúan los argumentos de una llamada a función.
- [B] Sirve para poder utilizar otros nombres para designar un tipo de datos (e.g., `String` para referirnos a *listas de caracteres* `[Char]`).
- [×] Hace posible la aplicación *parcial* de funciones a sus argumentos, permitiendo así definir y manipular expresiones de tipo funcional.
- [D] Es la operación por la cual se añade un tipo algebraico a una clase de tipos.

34. Dada la siguiente declaración para el tipo de datos `Colores`:

```
data Colores = Blanco | Negro    deriving Show
```

la expresión "`deriving Show`" nos permite sobrecargar automáticamente el operador `show`. Si quisieramos definirlo de forma que `(show Blanco)` mostrara `White` y `(show Negro)` mostrara `Black`, ¿cómo podríamos hacerlo?

- [×] Definiendo una instancia de la clase `Show` como sigue:

```
instance Show Colores where
  show Blanco = "White"
  show Negro = "Black"
```

- [B] Definiendo una instancia de la clase `Show` como sigue:

```
instance Show Colores where
  show White = "White"
  show Black = "Black"
```

- [C] No es necesario hacer nada, `show` ya se comporta de ese modo.
- [D] No es posible, ya que `White` y `Black` son predefinidos.

35. ¿Cuál es el tipo de la expresión `map ("pre"++)`? :t map = (a -> b) -> [a] -> [b]

- [A] `[a] -> [a]` map abs [-1,-3,4,-12] = [1,3,4,12]
- [B] `(String -> String) -> [String] -> [String]`
- [×] `[String] -> [String]`
- [D] `(a -> b) -> ([a] -> [b])`

36. La expresión `[toUpper,toLower]`:

[A] Es de tipo `[Char]`.

[B] Es de tipo `[String]`.

[×] Es una lista de funciones.

[D] Puede ser concatenada con la lista `[1,2,3]` ya que `(++)` tiene tipo polimórfico. (++) :: [a] -> [a] -> [a]

37. La inferencia de tipos:

[A] Es el mecanismo mediante el cual se sabe a qué clases debe pertenecer un tipo algebraico introducido por el usuario.

[B] Consiste en la comprobación de que los parámetros actuales se corresponden en número y tipo con los parámetros formales.

[C] Es un concepto inexistente en programación funcional, ya que la *inferencia* pertenece al ámbito de la *programación lógica*.

[×] Permite obtener el tipo más general que se le puede asociar a una expresión del programa en función de las definiciones de tipos algebraicos y ecuaciones de definición de función dadas en el programa.

38. Indica cuál de las siguientes afirmaciones referentes a la programación funcional en Haskell es **CIERTA**:

[×] La currificación hace posible la aplicación parcial de una función.

[B] La aplicación parcial de una función currificada consiste en invocarla con un número de argumentos mayor que el número de parámetros de su definición. Invocas con un número menor de arg. que en su definición

[C] Una de las características de los lenguajes funcionales son los efectos laterales. Ausencia de efectos laterales

[D] `(2+)` es una función de orden superior.

TEMA 3 - PARTE I

Soluciones Ejercicios de Aula: Test

1. C
2. A
3. A
4. B
5. B
6. A
7. A
8. A
9. C
10. B
11. B
12. C
13. C
14. B
15. C
16. A
17. C
18. A
19. C
20. B
21. A
22. B
23. B
24. A
25. A
26. C
27. C
28. B
29. D
30. A
31. C
32. A
33. C
34. A
35. C
36. C
37. D
38. A

TEMA 3 - PARTE I

Soluciones Ejercicios de Aula: Cuestiones

---

1. Para árboles de enteros

```
data Arbol = Hoja Int | Nodo Arbol Arbol

instance Show Arbol where
  show (Hoja n) = show n
  show (Nodo n m) = ``(N '' ++(show n)++`` '' ++(show m)++``)''
```

   Para árboles de cualquier tipo `a`

```
data Arbol a = Hoja a | Nodo (Arbol a)  (Arbol a)

instance  (Show a) => Show (Arbol a) where
   show (Hoja n) = show n
   show (Nodo n m) = ``(N '' ++(show n)++`` '' ++(show m)++``)''
```

2.
```
data Arbol = Hoja Int | Nodo Arbol Arbol

sumaNodos:: Arbol -> Int
sumaNodos (Hoja n) = n
sumaNodos (Nodo n m) = sumaNodos n + sumaNodos m
```

3. Usando la conversión a listas vista en prácticas

```
data Arbol = Hoja Int | Nodo Arbol Arbol

maxArbol:: Arbol -> Int
maxArbol a = maximum (tolist a)
             where tolist (Hoja n) = [n];
                   tolist (Nodo n m) = (tolist n) ++ (tolist m)
```

   Usando recursión:

```
maxArbol:: Arbol -> Int
maxArbol (Hoja x) = x
maxArbol (Nodo i d) = max (maxArbol i) (mazArbol d)
```

4.
```
data Arbol = Hoja Int | Nodo Arbol Arbol

flipA :: (Arbol a) -> (Arbol a)
flipA (Hoja n) = Hoja n
flipA (Nodo x y) = Nodo (flipA y) (flipA x)
```

5.
```
data Arbol a = Hoja a | Nodo (Arbol a) a (Arbol a)

ramaIzq:: Arbol a -> [a]
ramaIzq (Hoja n) = n
ramaIzq Nodo i n _ = n: ramaIzq  i
```

6.
```
reverses :: [a] -> [a]
reverses xs = rev xs []

rev :: [a] -> [a] -> [a]
rev [] acc = acc
rev (x:xs) acc = rev xs (x:acc)
```

7.
```
mapB :: (a -> b -> c) -> [a] -> [b] -> [c]

mapB f [] _ = []
mapB f _ [] = []
mapB f (x:xs) (y:ys) = f x y : mapB f xs ys
```

8.
```
t :: Int -> Int
t x = x * (x+1)

trans :: [Int] -> [Int]
trans xs = map t xs
```

9.
```
myUnzip :: [(a,b)] -> ([a],[b])

myUnzip [] = ([],[])
myUnzip ((x,y):resto) = (x:xs,y:ys)
         where (xs,ys) = myUnzip resto
```

10.
```
coords :: Int -> Int -> [(Int,Int)]

coords n m = [(x,y) | x <- [1..n], y <- [1..m] ]
```

11.
```
piramide :: Int -> [[Int]]

piramide n = [[1..x] | x <- [1..n]]
```

12.
```
foo :: IO ()
foo = do {
    putStrLn "Nombre?";
    nombre <- getLine;
    putStrLn "Apellido?";
    apellido <- getLine;
    putStrLn ("Hola " ++ nombre ++ " " ++ apellido ++ ", como estas? ")
    }
```

13.
```
revelar2 :: IO()
revelar2 = do {
          putStrLn "Introduce un caracter: ";
          c <- getChar;
          if c==‘q’ then putStrLn "Adios"
                    else do {
                              putStrLn (show (ord c));
                              revelar2 }}
```

14.
```
class MyArea a where
     area::  a -> Area
class MyVolumen  a where
      volumen:: a -> Volumen

instance   MyArea  Square where
    area (ASquare l)=   l * l
instance MyArea Rectangle where
    area (ARectangle l m) = l * m
instance MyArea Circle where
    area (ACircle r) = 3.14 * r * r
instance MyVolumen Cube where
    volumen (ACube l) = l * l * l
instance MyVolumen Sphere where
    volumen (ASphere r) =  3.14 * r * r * r
```

15.
```
data PrimaryColor = Violet|Indigo|Blue|Green|Yellow|Orange|Red
                                                deriving (Eq, Ord)
```

16.
```
type Nombre = String
type Edad = Int
type Empleado =  (Nombre, Edad)

masJoven::[Empleado] -> Empleado
masJoven x = masJovenAux (head x) (tail x)
masJovenAux x [] = x
masJovenAux (n1,e1) ((n2, e2):xs)
               | e1>e2 = masJovenAux (n2,e2) xs
               |otherwise =   masJovenAux (n1,e1) xs

joven::Empleado -> Empleado -> Empleado
joven (n1, e1) (n2, e2) = if e1<e2 then (n1,e1) else (n2, e2)

masJoven2::[Empleado] -> Empleado
masJoven2 [x]=x
masJoven2 (x:y:xs) = masJoven2 ((joven x y):xs)
```