LENGUAJES, PARADIGMAS Y TECNOLOGÍAS
DE PROGRAMACIÓN

TEMA 3:

PROGRAMACIÓN FUNCIONAL

(EJERCICIOS DE TRABAJO PERSONAL)

PARTE I: TIPOS EN PROGRAMACIÓN FUNCIONAL

1. Indica cuál de las siguientes expresiones completa la definición de la función de inversión de listas en Haskell:

```
reverse [] = []
reverse (x:xs) = [          ]
```

   A `(reverse xs):x`

   B `(reverse xs)++x`

   C `(reverse xs):[x]`

   D `(reverse xs)++[x]`

2. Indica cuál de las siguientes ecuaciones es **falsa**:

   A `(x:[])= [x]`

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

   C `([]:[xs])= [[],xs]`

   D `([x]++[])= [x]`

3. Indica cuál es el tipo de la siguiente función `loop = loop`:

   A `loop :: loop`

   B `loop :: a`

   C `loop :: a ->a`

   D `a :: loop ->loop`

4. Indica qué computa la siguiente función:

```
qs :: [Int] -> [Int]
qs [] = []
qs (x:xs) = qs [y | y <- xs, y < x] ++ [x] ++ qs [y | y <- xs, y >= x]
```

   A invierte una lista de enteros.

   B ordena en sentido ascendente una lista de enteros.

   C selecciona de una lista de enteros los elementos menores o iguales que `x`, incluido éste.

   D parte una lista de enteros en dos, poniendo los elementos menores que `x` al principo y los iguales o mayores a continuación.

5. Indica cuál de las siguientes funciones **no** es de orden superior:

   A `normalizar :: (Int, Int) → (Int, Int)`

   B `algunCero :: (Int → Int) → Int → Bool`

   C `igual :: (Bool → Int) → (Bool → Int) → Bool`

   D `any :: (a → Bool) → [a] → Bool`

6. Indica cuál de las siguientes ecuaciones define la función
   `todosCeros :: (Int → Int) → Int → Bool`
   tal que `todosCeros f n` devuelve `True` si `f(i) = 0` para todo `i` entre `1` y `n`.

   A `todosCeros f n = and (map (== 0) (map f [1..n]))`

   B `todosCeros f n = and [f x|x ← [1..n]]`

   C `todosCeros f n = foldr (==) 0 (map f [1..n])`

   D `todosCeros f n = f n == 0`

7. Indica cuál de las afirmaciones referentes a la función `pp` es cierta:

```
pp x y = x + 2*y
```

   A su tipo es `(Int → Int) → Int`

   B su tipo es `Int → (Int → Int)`

   C su tipo es `(Int, Int) → Int`

   D `pp` está mal definida

8. Dada la siguiente declaración del tipo de datos árbol de enteros:

```
data ArbolInt = Vacio | Nodo (ArbolInt) Int (ArbolInt)
```

   indica cuál de las siguientes funciones calcula la suma de los elementos del árbol:

   A
```
sumAr Vacio = 0
sumAr (Nodo l x r) = sumAr (Nodo l) + x + sumAr (Nodo r)
```

   B
```
sumAr Vacio = 0
sumAr (Nodo l x r) = x + sumAr l + sumAr r
```

   C
```
sumAr Vacio = 0
sumAr (Nodo (ArbolInt l) x (ArborlInt r)) = x + sumAr l
+ sumAr r
```

   C
```
sumAr Vacio = 0
sumAr (l x r) = x + sumAr l + sumAr r
```

9. Indica cuál de las siguientes funciones $\boxed{no}$ es de orden superior:

   A `splitAt :: Int → [a] → ([a], [a])`

   B `filter :: (a → Bool) → [a] → [a]`

   C `eqList :: (a → a → Bool) → [a] → [a] → Bool`

   D `flip :: (a → b → c) → b → a → c`

10. Si un lenguaje tiene inferencia de tipos, entonces:

    A la comprobación de tipos se hace mediante evaluación perezosa.

    B cuando el intérprete (o compilador) detecta un error de tipos, lo ignora simplemente.

    C no es necesario declarar el tipo de todas las expresiones.

    D únicamente se conoce el tipo de todas las expresiones durante la ejecución del programa.

11. Indica qué computa la siguiente función:

```
pippo xs = (reverse xs) == xs
```

    A invierte dos veces una lista, devolviendo la lista original.

    B comprueba si una secuencia de caracteres es capicúa.

    C devuelve la sublista de `xs` que coincide con su inversa.

    D ninguna de las anteriores.

12. Indica cuál de las siguientes funciones de Haskell es genérica:

    A $+$

    B $++$

    C $<$

    D $\&\&$

13. Indica cuál es el tipo inferido por Haskell para la función:

$$\texttt{aplicarPar}\ (\texttt{f}, \texttt{g})\ \texttt{x} = (\texttt{f}\ \texttt{x}, \texttt{g}\ \texttt{x})$$

    A `(a ->b , a ->c) ->a ->(b, c)`

    B `((a ->b) ->(a ->c)) ->a ->(b ->c)`

    C `(a ->b ->a ->c) ->a ->(b, c)`

    D `((a ->b), (a ->c)) ->a ->b ->c`

14. Completa el siguiente programa para que, dada una lista de entrada (`y:ys`) y un entero `x`, la función `menores` devuelva la lista de los elementos de (`y:ys`) que son menores que `x`:

```
menores _ [] = []
menores x (y:ys)
            | ____________________________
            | otherwise = menores x ys
```

A `x >y = y++(menores x ys)`

B `x >y = y:(menores x ys)`

C `x >y = [(menores x ys)]`

D `x >y = (menores x ys)`

15. Asumiendo que disponemos de tres funciones auxiliares `menores`, `mayores` e `iguales` que, dado un entero `e` y una lista de enteros `l`, devuelve la lista con los elementos de `l` menores, mayores e iguales que `e` respectivamente, completa el siguiente programa para que, dada una lista de entrada, la función `misort` devuelva esa misma lista ordenada ascendentemente:

```
misort [] = []
misort (x:xs) = ____________________________
```

A `(menores x xs) ++ (iguales x xs) ++ [x] ++ (mayores x xs)`

B `(misort (menores x xs)) ++ (iguales x xs) ++ [x] ++ (misort (mayores x xs))`

C `(misort (menores x xs)) ++ (misort (mayores x xs))`

D `(misort (mayores x xs)) ++ [x] ++ (misort (menores x xs)))`

16. Completa la siguiente función para convertir en mayúsculas todos los caracteres alfabéticos de un string (descartando los que no son alfabéticos):

```
cem :: String ->String
cem xs = [toUpper c | ____________, isAlpha c]
```

A `c..xs`

B `c << xs`

C `c in xs`

D `c <- xs`

17. Elige la opción adecuada para completar la siguiente función `capitaliza` que, a partir de una lista de palabras, devuelve una lista idéntica salvo que la letra inicial de cada palabra está escrita en mayúsculas:

```
capitaliza :: [String] ->[String]
capitaliza x = map toMayInicial x

toMayInicial :: String ->String
toMayInicial [] = []
toMayInicial x = ____________________________
```

A `(toUpper (head x)):(toMayInicial x)`

B `(toUpper (head x)):x`

C `(toUpper (head x)):(tail x)`

D `(toUpper (head x)):(toMayInicial (tail x))`

18. Considera la siguiente definición de un tipo algebraico árbol de enteros:

```
data NTree = NilT | Node Int NTree NTree
```

e Indica cuál de los siguientes datos **NO** corresponde al tipo `Ntree`:

A `Node 17 (Node 14 NilT (Node 0 NilT NilT)) (Node 20 NilT NilT)`

B `Node 17 (Node 20 NilT NilT) (Node 14 NilT (Node 0 NilT NilT))`

C `Node 14 (Node 0 NilT NilT) NilT`

D `(Node NilT 0 NilT)`

19. Completa la siguiente función para obtener el número de elementos de una lista `xs` que son diferentes a un elemento `e` dado :

```
repetido e xs = length [ x | ____________, e /= x]
```

A `x <- xs`

B `x..xs`

C `x << xs`

D `x in xs`

20. Considera la siguiente definición de un tipo algebraico árbol de enteros:

```
data NTree = NilT | Node Int NTree NTree
```

y completa la siguiente función que determina si un árbol es simétrico (es decir, imaginando una línea vertical que atravesara el nodo raíz, el subárbol derecho en una imagen especular del subárbol izquierdo):

```
simetrico NilT = True
simetrico (Node _ l r) = [          ]
mirror NilT NilT = True
mirror (Node m a b) (Node n x y)
  | m == n = mirror a y && mirror b x
mirror _ _ = False
```

A `mirror l r`

B `(simetrico l) && (simetrico r)`

C `(Node _ r l)`

D `(mirror l)==r`

21. Indica cuál de los siguientes datos **NO** corresponde al tipo `Ntree` de la pregunta anterior:

A `(Node NilT 0 NilT)`

B `Node 14 (Node 0 NilT NilT) NilT`

C `Node 17 (Node 20 NilT NilT) (Node 14 NilT (Node 0 NilT NilT))`

D `Node 7 (Node 4 NilT (Node 0 NilT NilT)) (Node 2 NilT NilT)`

PARTE II: MODELO COMPUTACIONAL

22. Indica el conjunto de redexes de la expresión `rota (head [3,2,5,7]) [3,2,5,7]` con respecto al siguiente programa funcional:

```
rota n  xs | n>0 = (drop n xs) ++ (take n xs)
```

A {`head [3,2,5,7]`}

B {`rota (head [3,2,5,7]) [3,2,5,7]`}

C {`rota (head [3,2,5,7]) [3,2,5,7], head [3,2,5,7]`}

D {`rota (head [3,2,5,7]) [3,2,5,7], head [3,2,5,7], [3,2,5,7]`}

23. Indica cuál es el redex usado en la reducción (en un paso) de la expresión `rota (head [3,2,5,7]) [3,2,5,7]` con respecto al programa funcional de la pregunta anterior si la evaluación fuera perezosa:

A `head [3,2,5,7]`

B `rota (head [3,2,5,7]) [3,2,5,7]`

C `[3,2,5,7]`

D ninguno ya que la expresión es irreducible (ya está en forma normal).

24. Indica la forma normal del término `either` (`div` 8) `length` (`Right` "64") con respecto al siguiente programa:

```
data Either a b = Left a | Right b

either :: (a ->c) ->(b ->c) ->Either a b ->c
either f g (Left x) = f x
either f g (Right y) = g y
```

A 8

B 2

C 1

D 10

25. Indica cuál es la forma normal de la expresión `fact` (-1) con respecto al siguiente programa:

$$\texttt{fact}\ 0 = 1$$
$$\texttt{fact}\ \texttt{n} = \texttt{if}\ \texttt{n} > 0\ \texttt{then}\ \texttt{n} * \texttt{fact}\ (\texttt{n} - 1)$$

A `fact` (-1)

B $[-1, -2, -6, -24, ...]$

C -1

D no existe

26. Dado el siguiente programa funcional:

```
doble x = 2*x
ones = 1:ones
addFirstTwo (x:y:zs) = x + y
```

indica cuál de las siguientes secuencias de reducciones corresponde a la evaluación perezosa de la expresión

```
doble (addFirstTwo ones)
```

A `doble (addFirstTwo ones)` $\rightarrow$ `doble (addFirstTwo (1 : ones))`
$\rightarrow$ `doble (addFirstTwo (1 : 1 : ones))` $\rightarrow$ `doble (1 + 1)` $\rightarrow$ `doble 2` $\rightarrow 2 * 2 \rightarrow 4$

B `doble (addFirstTwo ones)` $\rightarrow$ `doble (addFirstTwo (1 : ones))`
$\rightarrow$ `doble (addFirstTwo (1 : 1 : ones))` $\rightarrow$ `doble (1 + 1)` $\rightarrow 2 * (1 + 1) \rightarrow 2 * 2 \rightarrow 4$

C `doble (addFirstTwo ones)` $\rightarrow 2 *$ `(addFirstTwo ones)`
$\rightarrow$ `(addFirstTwo ones)` $+$ `(addFirstTwo ones)`
$\rightarrow$ `(addFirstTwo (1 : 1 : ones))` $+$ `(addFirstTwo (1 : 1 : ones))`
$\rightarrow (1 + 1) + (1 + 1) \rightarrow 4$

D `doble (addFirstTwo ones)` $\rightarrow 2 *$ `(addFirstTwo ones)`
$\rightarrow 2 *$ `(addFirstTwo (1 : ones))`
$\rightarrow 2 *$ `(addFirstTwo (1 : 1 : ones))` $\rightarrow 2 * (1 + 1) \rightarrow 2 * 2 \rightarrow 4$

27. indica cuál de las siguientes afirmaciones es cierta en relación a este programa:

```
loop = loop
mult 0 x = 0
```

A La expresión `mult loop loop` no tiene forma normal.

B `loop` está en forma normal.

C La expresión `mult 0 loop` no tiene una forma normal porque su evaluación impaciente no termina.

D La forma normal de la expresión `mult x 0` es 0.

28. Dado el siguiente programa funcional:

```
doble x = 2*x
ones = 1:ones
addFirstTwo (x:y:zs) = x + y
```

indica cuál de las siguientes secuencias de reducciones corresponde a la evaluación impaciente de la expresión

```
doble (addFirstTwo ones)
```

A `doble (addFirstTwo ones)` $\rightarrow 2 * ($`addFirstTwo ones`$)$
$\rightarrow 2 * (\texttt{addFirstTwo}\ (1 : \texttt{ones}))$
$\rightarrow 2 * (\texttt{addFirstTwo}\ (1 : 1 : \texttt{ones})) \rightarrow 2 * (1 + 1) \rightarrow 2 * 2 \rightarrow 4$

B `doble (addFirstTwo ones)` $\rightarrow$ `doble` $(\texttt{addFirstTwo}\ (1 : \texttt{ones}))$
$\rightarrow$ `doble` $(\texttt{addFirstTwo}\ (1 : 1 : \texttt{ones})) \rightarrow$ `doble` $(1 + 1)$
$\rightarrow$ `doble` $2 \rightarrow 2 * 2 \rightarrow 4$

C `doble (addFirstTwo ones)` $\rightarrow$ `doble` $(\texttt{addFirstTwo}\ (1 : \texttt{ones}))$
$\rightarrow$ `doble` $(\texttt{addFirstTwo}\ (1 : 1 : \texttt{ones})) \rightarrow$ `doble` $(1 + 1)$
$\rightarrow 2 * (1 + 1) \rightarrow 2 * 2 \rightarrow 4$

D `doble (addFirstTwo ones)` $\rightarrow$ `doble` $(\texttt{addFirstTwo}\ (1 : \texttt{ones}))$
$\rightarrow$ `doble` $(\texttt{addFirstTwo}\ (1 : 1 : \texttt{ones}))$
$\rightarrow$ `doble` $(\texttt{addFirstTwo}\ (1 : 1 : 1 : \texttt{ones}))$...

29. Indica cuál es el conjunto de redexes de la expresión

`drop` $(\texttt{length}\ [3,5,7,2] - (\texttt{minimumLs}\ [3,5,7,2]))\ [3,5,7,2]$

respecto al siguiente programa funcional:

```
drop n xs     | n <= 0 =  xs
drop _ []              =  []
drop n (_:xs)          =  drop (n-1) xs

minimumLs :: [Integer] -> Integer
minimumLs = foldr1 min
```

siendo `foldr1` una variante de `foldr` que no tiene el segundo argumento (valor inicial) y que, por lo tanto, sólo puede aplicarse a listas no vacías.

A {`length [3,5,7,2] , minimumLs [3,5,7,2]`}

B {`minimumLs [3,5,7,2]`}

C {`length [3,5,7,2] , minimumLs [3,5,7,2], drop (length [3,5,7,2] - (minimumLs [3,5,7,2])) [3,5,7,2]`}

D {`length [3,5,7,2] , drop (length [3,5,7,2] - (minimumLs [3,5,7,2])) [3,5,7,2]`}

30. Indica cuál es el redex usado en la reducción (en un paso) de la expresión

$$\texttt{drop}\ (\texttt{length}\ [3,5,7,2] - (\texttt{minimumLs}\ [3,5,7,2]))\ [3,5,7,2]$$

con respecto al programa funcional de la pregunta anterior si la evaluación fuera impaciente o voraz:

A `drop (length [3,5,7,2] - (minimumLs [3,5,7,2])) [3,5,7,2]`

B `minimumLs [3,5,7,2]`

C `length [3,5,7,2]`

D ninguno ya que la expresión es irreducible (ya está en forma normal).

PARTE III: CARACTERISTICAS AVANZADAS

31. Indica el valor infinito que corresponde a la evaluación de la expresión inicial `iter` $(2*)$ 2 :

```
iter f x = x:iter f (f x)
```

A $[2,2,2,2,2,2,2,...]$

B $[2,4,8,16,32,64,128,256,512,1024,2048,4096,...]$

C $[2,4,6,8,10,12,14,...]$

D $[4,4,4,4,4,4,4,...]$

32. Indica cuál es el tipo de la función `iter` de la pregunta anterior:

A `iter :: (a,a)→ [a]`

B `iter :: (a → a) → a → [a]`

C `iter :: (a → a) → a → (a → a)`

D `iter :: (a → a) → a → (a → a → a)`

33. Indica cuál de las siguientes funciones no computa la suma de una lista de enteros:

A `suma xs = foldr (+) 0 xs`

B `suma xs = map (+ 0) xs`

C `suma = foldr (+) 0`

D
```
suma [] = 0
suma (x:xs) = x+(suma xs)
```

34. Indica qué función computa el siguiente programa funcional `pippo`:

```
pippo = foldr prefijar [] where prefijar x xs = x:xs
```

A la extracción de todos los prefijos de una lista.

B la inversión de listas.

C la función identidad, que deja inalterada la lista a la que se aplica.

D la extracción de todos los sufijos de una lista (es decir, los prefijos de la lista invertida).

35. Indica cuál de las siguientes funciones no es de orden superior:

A `dosVeces x = (2*) x`

B `flip f x y = f y x`

C `(f.g) x = f (g x)`

D `zipWith z (a:as) (b:bs) = z a b : zipWith z as bs`

36. Indica qué computa la siguiente función:

```
desconocida l = foldr (++) [] (map sing l)
           where sing a = [a]
```

A devuelve la lista argumento

B suma los elementos de la lista argumento

C invierte la lista argumento

D aplana una lista de listas

37. Indica cuál de las siguientes funciones anónimas se corresponde con la función:

```
f x y = (x + y) * 2
```

A $\backslash\, \mathtt{y} -> (*)((+)\, \mathtt{x}\, \mathtt{y})2$

B $\backslash -> (*)((+)\, \mathtt{x}\, \mathtt{y})2$

C $\backslash\, \mathtt{x} -> (*)((+)\, \mathtt{x}\, \mathtt{y})2$

D $\backslash$ x y $->$ $(*)((+)$ x y$)2$

38. Considerando la función `isEven`, que comprueba si un número entero es par, indica cuál de las siguientes funciones **NO** comprueba que son pares todos los números de una lista:

A
```
allEven xs = xs == filter isEven xs
```

B
```
allEven xs = and [isEven x | x <- xs]
```

C
```
allEven xs = foldr f True xs where f x e = isEven x &&
e
```

D
```
allEven [ ] = True
allEven (x:xs) = (isEven x) || (allEven xs)
```

39. Indica qué hace la función h:

```
f (u,v) n
   | u == v    = n+1
   | otherwise = n
g xs = foldr f  0 xs
h xs ys = g (zip xs ys)
```

A dadas las listas `xs` e `ys` calcula la longitud de la más pequeña, por ejemplo, `h [2,1,3] [1,1,2,5] = 3`.

B calcula cuántos elementos aparecen en ambas listas `xs` e `ys`, por ejemplo, `h [2,1,3] [1,1,2,5] = 2`.

C dadas las listas `xs` e `ys` calcula la longitud de la más grande, por ejemplo, `h [2,1,3] [1,1,2,5] = 4`.

D calcula cuántos elementos coinciden en la misma posición en las listas `xs` e `ys`, por ejemplo, `h [2,1,3] [1,1,2,5] = 1`.