REPASAR : Filter

Implementar la función nOf, la cual toma un número entero n y un valor v de un tipo cualquiera. Retorna una lista de largo n, con todos sus elementos iguales a v. Si n es menor o igual que cero se retorna la lista vacía. La firma de la función sería la siguiente:

```
nOf :: Int -> a -> [a]
```

Por ejemplo:

```
(nOf 3 'a') == "aaa"
(nOf 7 7) == [7,7,7,7,7,7,7]
(nOf 0 0) == []
(nOf (-1) 'b') == []
```

```
nOf :: Int -> a -> [a]
nOf n v
  | n <= 0 = []
  | otherwise = [v] ++ nOf (n-1) v
```

Escribir una función de Haskell que tome una lista de enteros (no necesariamente ordenada) y retorne el menor número entero mayor que no esté en la lista y que sea mayor al mínimo número de la lista. Si la lista está vacía se debe arrojar un error.

```
minAfterMin :: [Int] -> Int
```

Por ejemplo:

```
(minAfterMin [1,2,3]) == 4
(minAfterMin [1,3,7,5,6]) == 2
(minAfterMin [17]) == 18
(minAfterMin [77,700,707,70,7,770]) == 8
(minAfterMin [0..10]) == 11
```

```
tieneSucesor :: Int -> [Int] -> Bool
tieneSucesor x [] = False
tieneSucesor x (y:ys)
   | x+1 == y = True
   | otherwise = tieneSucesor x ys

delete :: Int -> [Int] ->[Int]
```

```
delete x [] = []
delete x (y:ys)
  | x == y = ys
  | otherwise = [y] ++ delete x ys

minAfterMin :: [Int] -> Int
minAfterMin [] = error"Lista vacia"
minAfterMin xs
  | tieneSucesor m xs = minAfterMin (delete m xs)
  | otherwise = m+1
  where m = minimum xs
```

Escribir una función de Haskell que tome una tripleta con tres valores del mismo tipo, una posición en esa tripleta y un nuevo valor. El resultado debe ser una nueva tripleta con el valor en la posición dada reemplazado por el nuevo valor. Si la posición no está entre 1 y 3, se debe retornar la tripleta sin modificar.

```
updateTriplet :: (a,a,a) -> Int -> a -> (a,a,a)
```

Por ejemplo:

```
(updateTriplet (1, 2, 3) 1 9) == (9, 2, 3)
(updateTriplet (False, True, False) 2 False) == (False, False,
False)
(updateTriplet ('a', 'b', 'c') 3 'x') == ('a', 'b', 'x')
(updateTriplet (2, 3, 5) 0 77) == (2, 3, 5)
(updateTriplet ('i', 'j', 'k') 7 'z') == ('i', 'j', 'k')
```

```
updateTriplet :: (a,a,a) -> Int -> a -> (a,a,a)
updateTriplet (x,y,z) num valor
  | num <1 || num>3 = (x,y,z)
  | num == 1 = (valor,y,z)
  | num == 2 = (x,valor,z)
  | num == 3 = (x,y,valor)
```

¿Cuál es la forma correcta de llamar una función f con 2 argumentos en Haskell? (1p)

Seleccione una:

a.

```
f(1 2)
```

b.

```
(f, 1, 2)
```

c.

```
(f 1 2)
```
Correcta

d.

```
f(1, 2)
```

¿Cuál es el problema con la siguiente definición de Haskell?

```
isElem :: [a] -> a -> Bool

isElem [] _ = False

isElem [x:xs] y = x == y || (isElem xs y)
```

(1p)

Seleccione una:

a. El patrón de listas [x:xs] no debe usar el dos puntos, sino la barra vertical. Así: [x|xs].

b. El valor booleano falso en Haskell se escribe false y no False.

c. El patrón de listas [x:xs] no debe ir entre paréntesis rectos, sino curvos. Así: (x:xs). Correcta

d. La definición dada es código Haskell perfectamente válido. No tiene problemas.

¿Cuál es el problema con la siguiente definición de Haskell?

```
isEqual :: a -> a -> Bool

isEqual x x = True

isEqual _ _ = False
```

(1p)

Seleccione una:

a. La función siempre evaluará a True. El patrón en la segunda ecuación es más general, por lo que siempre evalúa primero.

b. Los patrones no se pueden repetir en el miembro izquierdo de las ecuaciones, como sucede con x en la primera ecuación. Correcta

c. La definición dada es código Haskell perfectamente válido. No tiene problemas.

d. El patrón _ no se puede aplicar para todos los argumentos de la función, como sucede con la segunda ecuación.

¿Cuál es el problema con la siguiente definición de Haskell?

```
reverse :: [a] -> [a]

reverse [] = []

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

(1p)

Seleccione una:

a. Según la firma la función recibe dos argumentos, pero en las ecuaciones se usa uno sólo.

b. La llamada recursiva debe escribirse reverse(xs) en lugar de (reverse xs).

c. La definición dada es código Haskell perfectamente válido. No tiene problemas.

d. El operador (:) debe tener siempre un elemento a la izquierda y una lista a la derecha y no al revés. Correcta

¿Cuál es el problema con la siguiente definición de Haskell?

```
clamp :: Int -> Int -> Int -> Int

clamp value min max =

| value < min = min

| value > max = max

| otherwise = value
```

(1p)

Seleccione una:

a. La definición dada es código Haskell perfectamente válido. No tiene problemas.

b. Las guardas comienzan con if y no con la barra vertical (|).

c. La firma de la función debería ser (Int, Int, Int) -> Int.

d. El signo de igual = no va antes de las guardas. Correcta

¿Qué función cumple la siguiente definición?

```
fun n x = map (\_ -> x) (take n [0..])
```

(2p)

Seleccione una:

a. El código dado no es Haskell válido.

b. Construye la lista de los primeros *n* números (empezando desde cero) que son iguales a *x*.

c. Construye una lista de largo *n*, con todos sus elementos iguales a *x*. Correcta

d. Construye una lista de *n* ceros.

¿Qué función cumple la siguiente definición?

```
fun x xs = length (filter (== x) xs)
```

(2p)

Seleccione una:

a. Retorna la lista de elementos de xs que son iguales a x.

b. Cuenta la cantidad de elementos de xs que son iguales a x. Correcta

c. El código dado no es Haskell válido.

d. Chequea si la cantidad de elementos de xs es igual a x.

¿Qué función cumple la siguiente definición?

```
fun s = concat (map (\x -> if x == '\t' then "    ") s)
```

(2p)

Seleccione una:

a. Sustituye todos los tabuladores en la cadena s por secuencias de 4 espacios.

b. Sustituye todas las subcadenas de 4 espacios seguidos en la cadena s por un tabulador.

c. El código dado no es Haskell válido. Correcta

d. Sustituye todos los tabuladores en la cadena s por secuencias de 3 espacios.

¿Qué función cumple la siguiente definición?

```
fun [] ys = ys

fun (x:xs) ys = x:(fun xs ys)
```

(2p)

Seleccione una:

a. Concatena dos listas, de la misma forma que lo hace el operador (++). Correcta

b. El código dado no es código Haskell válido.

c. Concatena el reverso de la primer lista con la segunda.

d. Concatena el segundo argumento al final de la lista del primer argumento. Es como el operador (:), pero al revés.