MEMORIA HASKELL

DAVID LÓPEZ VALDIVIA,JAIME FUENTES MARTÍNEZ

5A.Análisis y resultados de las primeras funciones del código.

Las funciones ToString piden un solo parámetro, que debe ser del tipo que se indica en el nombre de la propia función. Debe devolvernos un texto con toda la información o atributos que poseen. A continuación se muestran los outputs de las funciones productoToString, pedidoToString y compraToString:

```
Ok, one module loaded.
ghci> pedidoToString(order0)
"El pedido esta formado por 4.0 unidad(es) del Producto 0: asp
iradora de precio 100.0. El precio total es 400.0"
ghci> compraToString(pur0)
"El pedido esta formado por 4.0 unidad(es) del Producto 0: asp
iradora de precio 100.0. El precio total es 400.0"
ghci> productoToString(product0)
"Producto numero 0: aspiradora de precio 100.0"
ghci>
```

Respecto a las funciones donde devolvemos precios, con precioCompra nos pareció sencillo hacer recursión con la implementación de función llamada precioPedido, aún no pidiendose esta explícitamente.
Por tanto, precioProducto recibe un producto y precioCompra una compra y en ambos devolveremos un número. Se puede comprobar que precioCompra funciona bien ya que suma el precio un producto de 30 millones más uno de 250(la compra tiene varios productos).

```
ghci> precioProducto(product1)
3.0e7
ghci> precioCompra(pur2)
3.000025e7
```

En cuanto a la función FusionaCompras, la cual recibe un par de compras, se nos han ocurrido dos funciones similares y los resultados son compras al igual que los parámetros :

```
ghci> fusionaCompras pur0 pur1
[((0,"aspiradora",100.0),4.0),((1,"un yate super guapo",3.0e7),1.0)]
ghci> fusionaCompras2 pur1 pur2
[((1,"un yate super guapo",3.0e7),1.0),((1,"un yate super guapo",3.0e7),1.0),((2,"batidora",250.0),1.0)]
ghci>
```

precioProductoCompra recibe un producto y una compra para devolver el precio total que suponen todos los productos del tipo indicado dentro de la compra.

Las funciones buscaPedidosConProducto y buscaPedidosConProductos tienen como objetivo devolver un conjunto de pedidos que incluyan al producto o productos(como es el caso de la segunda función,es decir,introducimos como parámetro una lista de productos) dentro de la compra.

```
ghci> precioProductoCompra pur2 product1
3.0e7
ghci> precioProductoCompra pur0 product0
400.0
ghci> buscaPedidosConProducto pur2 product1
[((1,"un yate super guapo",3.0e7),1.0)]
```

Como se puede apreciar en la siguiente imagen, introducimos una lista de productos (podría contener un único elemento) para que se devuelvas todos aquellos pedidos de la compra en los que figure uno de estos productos.

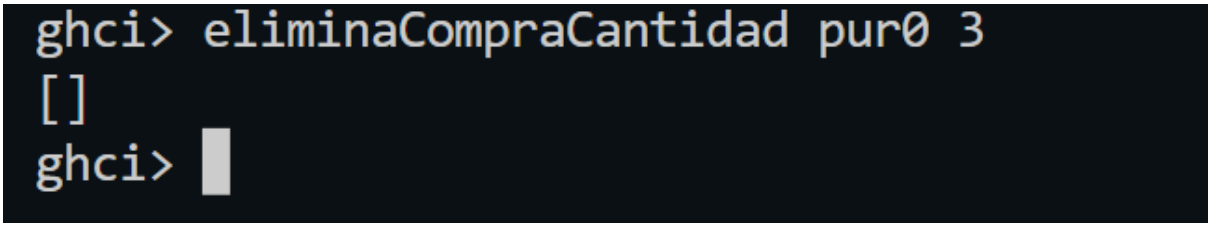
```
ghci> buscaPedidosConProductos pur2 [product1,product2]
[[((1,"un yate super guapo",3.0e7),1.0)],[((2,"batidora",250.0),1.0)]]
ghci>
```

La función eliminaProductoCompra recibe una compra y un producto, el cual no tiene por qué estar incluido en uno de los pedidos de compra. En este caso devolvería la compra que se ha introducido. Si el producto está presente, devolveremos la compra sin aquellos pedidos que se refieran al producto:

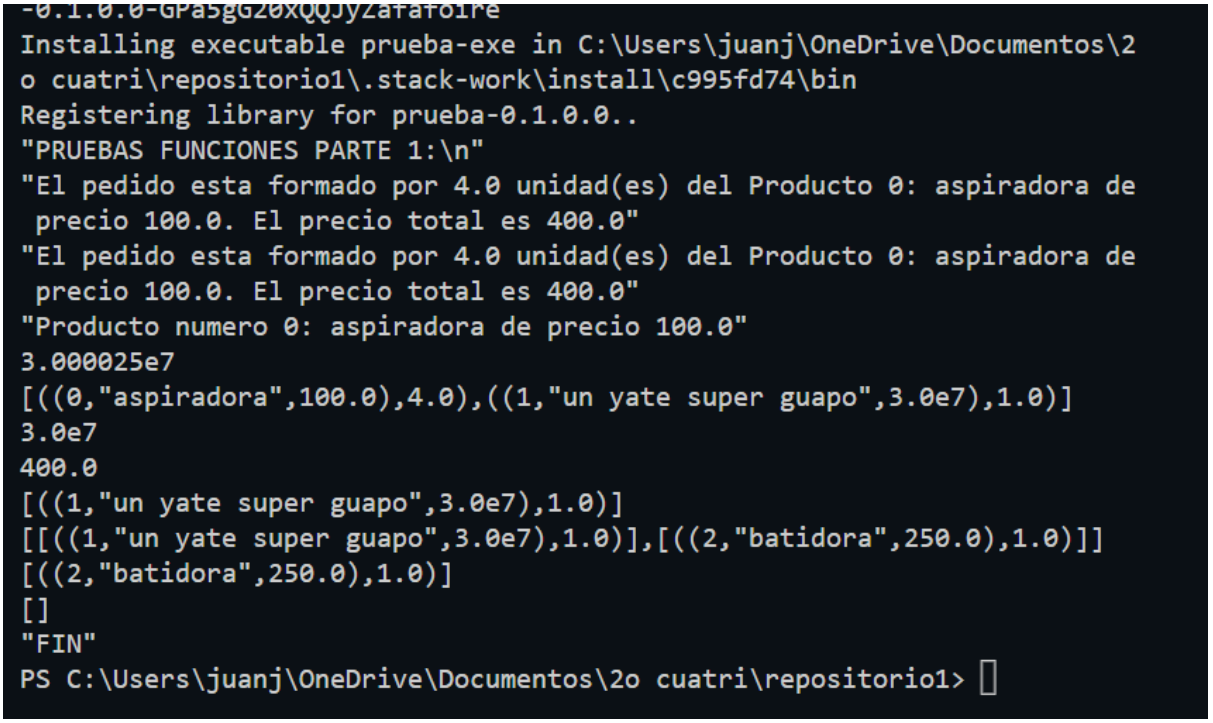
```
ghci> eliminaProductoCompra pur2 product1
[((2,"batidora",250.0),1.0)]
ghci> z = eliminaProductoCompra pur2 product1
ghci> show z
"[((2,\"batidora\",250.0),1.0)]"
```

La función eliminaCompraCantidad recibe una compra y una cantidad. Recibiremos la compra sin aquellos pedidos cuya cantidad sobrepase la del parámetro. Por ejemplo, la compra pur0 incluye un

solo pedido de cuatro aspiradores, por lo que esperamos que la compra a devolver no incluya pedido alguno:



Todas estas pruebas fueron realizadas con ghci. Usando stack, al ejecutar el main debe de salir:



9A.Análisis de resultados de pedido en el 8A

Se nos ocurren dos maneras de gestionar el tratamiento de errores:

Una de ellas sería ir modificando cada función repetitivamente comprobando que los pedidos y productos introducidos como parámetros fueran válidos. Para hacerlo tendríamos que hacer algún razonamiento como modificar en las listas de compresión con una condición adicional que debe complirse: dos funciones de comprobación (una para productos y otra para pedidos) que devuelva True o False dependiendo de si los datos de la instancia están en orden. En caso de ser incorrectos, la lista de compresión omitiría ese objeto y pasará al siguiente,es decir, hará como si no estuviese en la compra. Esta idea no nos convencía del todo ya que de esta forma podríamos seguir creando objetos de forma errónea, así que probamos otra manera:

Esta sería instanciar cada pedido y producto llamando a las funciones pedidoS y productoS. Estas funciones reciben como argumentos todos aquellos datos que introducimos para instanciar un pedido o un producto, respectivamente. Si alguno de estos datos es una cantidad/ precio negativa o nula, un identificador incorrecto, o un nombre con un string vacío (“”) lanzamos un error.

Si no sucede nada de esto, lo que hacemos es crear la instancia. Esto implica que cada vez que queremos crear uno de estos en, por ejemplo, una función, vamos a recurrir siempre a esta función.

```haskell
productoS :: Int -> String -> Float -> Producto
productoS id nombre precio
  | precio <= 0 = error  "el precio del producto no puede ser negativo"
  | id <= 0 =  error "el identificador introducido no es correcto"
  | nombre == "" = error  "el nombre del producto no se ha establecido"
  otherwise (Producto id nombre precio)

pedidoS :: Producto -> Cantidad -> Pedido
--no hace falta una función de comprobación para crear un pedidoUnitario ya que
--producto SABEMOS que está creado con los datos correctos
pedidoS (Producto codigo nombre precio) cantidad
  | cantidad < 0 = error "el pedido no puede tener cantidad negativa"
  otherwise Pedido (Producto codigo nombre precio) cantidad
```

Como estos funciones no las podemos poner el main (no queremos que salte un error al hacer stack run) ponemos por aquí algunas de las pruebas.
En primer lugar, si escribimos en el main `print(productoS 2 "" 9)` y ejecutamos obtenemos:

```
Ok, one module loaded.
ghci> main2
"Comienzo de las ejecuciones del main2"
""
*** Exception: el nombre del producto no se ha establecido
CallStack (from HasCallStack):
  error, called at pruebahaskell2.hs:62:20 in main:Main
ghci>
```

Si ponemos `print(pedidoS (product -1 )` obtenemos:

```
Ok, one module loaded.
ghci> main2
"Comienzo de las ejecuciones del main2"
""
*** Exception: el pedido no puede tener cantidad negativa
CallStack (from HasCallStack):
  error, called at pruebahaskell2.hs:69:20 in main:Main
ghci>
```

De esta forma tendremos una mayor seguridad con nuestras instancias al saber que tienen datos válidos. Lo único que habría que recordar es que las instancias las tenemos que crear con PedidoS y

ProductoS y no usar Pedido o Pedido Unitario por nuestra cuenta.De esta forma, nos quedan las siguientes ejecuciones del main2 habiendo definido las siguientes instancias:

```
product0 :: Producto
product0 = productoS 1 "la copa" 2000
product1 :: Producto
product1 = productoS 2 "yate" 30000000 --yate
product2 :: Producto
product2 = productoS 3 "naranja" 2
order0 :: Pedido
order0 = PedidoUnitario product0
order1 :: Pedido
order1 = pedidoS product1 3
order2 :: Pedido
order2 = pedidoS product2 5
pur0 :: Compra
pur0 = Compra [order1,order1,order0]
pur1 :: Compra
pur1 = Compra [order2]
```

```
"Comienzo de las ejecuciones del main2"
""
1.80002e8
Pedido (2,"yate",3.0e7)con cantidad3.0 , Pedido (2,"yate",3.0e7)con cantidad3.0 ,

9.0e7
"fusionaCompras"
Pedido (2,"yate",3.0e7)con cantidad3.0 , Pedido (2,"yate",3.0e7)con cantidad3.0 ,

"buscaPedidosConProducto pur0 product0"
Pedido (2,"yate",3.0e7)con cantidad3.0 , Pedido (2,"yate",3.0e7)con cantidad3.0 ,
"buscaPedidosConProducto pur0 product0"
Pedido Unitario (1,"la copa",2000.0) ,
"buscaPedidosConProductos pur0 [product0,product1]"
Pedido (2,"yate",3.0e7)con cantidad3.0 , Pedido (2,"yate",3.0e7)con cantidad3.0 ,
"eliminaProductoCompra pur0 product0"
Pedido (2,"yate",3.0e7)con cantidad3.0 , Pedido (2,"yate",3.0e7)con cantidad3.0 ,
"eliminaCompraCantidad pur0 1"
Pedido (2,"yate",3.0e7)con cantidad3.0 , Pedido (2,"yate",3.0e7)con cantidad3.0 ,
"ELIMINAR REPES"
Pedido (2,"yate",3.0e7)con cantidad6.0 , Pedido (1,"la copa",2000.0)con cantidad1
()
```

La última función que se ejecuta es eliminarRepeticiones pur0.

La función elimina Compra Cantidad si necesitará un pequeño cambio, indicando simplemente que la cantidad del parámetro debe ser 1 como mínimo.

2B. (MEMORIA) Comenta en la memoria los aspectos más relevantes del código implementado, el resultado de la ejecución para un mínimo de 4 turnos y las pruebas realizadas para comprobar su correcto funcionamiento:

El código en Haskell se centra en la implementación de una simulación del Juego de la Vida de Conway.

Las primeras funciones que vemos en **P2B_EX1** son las funciones que se encargan del tamaño del tablero, de definir lo que es un tablero y una posición, y de la función *cls* que limpia la pantalla. Luego tenemos *glider* que indica que células se encuentran vivas en el momento de iniciar el tablero. Estas funciones vienen dadas por el código **life.hs.**

```
6
7   cls :: IO ()
8   cls = putStr "\ESC[2J"
9
10  type Pos = (Int,Int)
11
12  width :: Int
13  width = 9
14
15  height :: Int
16  height = 9
17
18  type Board = [Pos]
19
20  glider :: Board
21  glider = [(4,2),(2,3),(4,3),(3,4),(4,4)]
22
```

Después tenemos la función *showcells*, la función principal del código que se encarga de imprimir las células por pantalla. Para ello, simplemente comprueba mediante la función *isAlive* si una célula está viva o muerta para a continuación añadir a un string su correspondiente carácter, utilizando unlines para juntar el array de strings de tamaño width/height para juntarlo todo en un solo string. Esto es una modificación del *showcells* dado por el código, con la cual no necesitamos las funciones auxiliares w*writeat* y *goto* que usaba el mismo.

```
showcells :: Board -> IO ()
showcells b = putStrLn (unlines [ [if isAlive b (x,y) then 'X' else '_' | x <- [1..width]] | y <- [1..height]])

isAlive :: Board -> Pos -> Bool
isAlive b p = elem p b

isEmpty :: Board -> Pos -> Bool
isEmpty b p = not (isAlive b p)
```

Después tenemos las funciones *neighbs*, que define los vecinos de una célula nada mediante coordenadas y *wrap*, que se asegura de que en caso de que acabemos el tablero volvamos atrás. Ambas son idénticas a las del código dado.

```
32  neighbs :: Pos -> [Pos]
33  neighbs (x,y) = map wrap [(x-1,y-1), (x,y-1), (x+1,y-1), (x-1,y),
34                            (x+1,y), (x-1,y+1), (x,y+1), (x+1,y+1)]
35
36  wrap :: Pos -> Pos
37  wrap (x,y) = (((x-1) `mod` width) + 1, ((y-1) `mod` height) + 1)
38
```

Despues, tambien sacadas del código original tenemos *liveneighbs, survivors y births* que se encargan de comprobar que vecinos están vivos, que células van a sobrevivir y que nuevos nacimientos va a haber.

```
39  liveneighbs :: Board -> Pos -> Int
40  liveneighbs b = length . filter (isAlive b) . neighbs
41
42  survivors :: Board -> [Pos]
43  survivors b = [p | p <- b, elem (liveneighbs b p) [2,3]]
44
45  births :: Board -> [Pos]
46  births b = [p | p <- rmdups (concat (map neighbs b)),
47                  isEmpty b p,
48                  liveneighbs b p == 3]
49
```

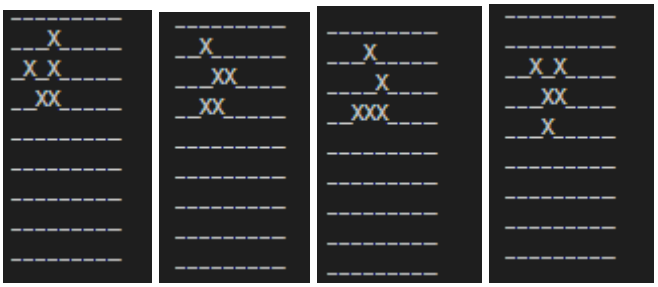
Por último, las funciones *nextgen y life*. La primera se encarga de generar el nuevo tablero y la siguiente de ser el "driver" de todo el código anterior. *Life* recibe un int y un tablero y espera (usando la función *wait*) a que el usuario presione enter para mostrar el siguiente turno, hasta un máximo de turnos n (dónde n viene dado por el int).

```
54  nextgen :: Board -> Board
55  nextgen b = survivors b ++ births b
56
57  life :: Board -> Int -> IO ()
58  life _ 0 = return ()
59  life b n = do
60                showcells b
61                wait
62                life ((nextgen b)) (n-1)
```

ambas son ligeras modificaciones de sus originales en **life.hs**

Por último, vemos el resultado de ejecutar los 4 primeros turnos con el *glider* [(4,2),(2,3),(4,3),(3,4),(4,4)]:

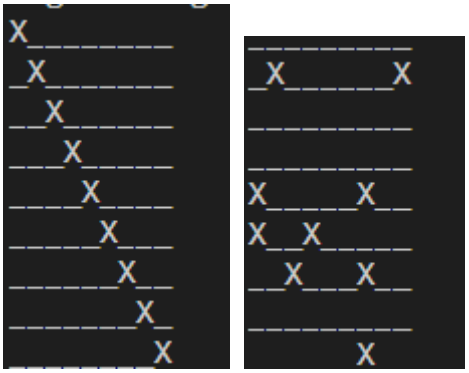
:



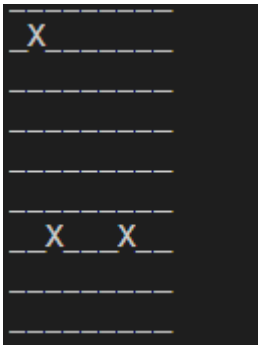
Nótese que después de cada turno hay que presionar enter para llamar al siguiente.

Las comprobaciones hechas han sido:

1. Comprobar a mano que el juego funciona correctamente y hace las evoluciones de las células
2. Comprobar que *showcells* funciona con diferentes *gliders*. Dejo dos ejemplos con los *gliders* [(1, 1), (2, 2), (3, 3), (4, 4), (5, 5), (6, 6), (7, 7),(8,8),(9,9)] y [(3, 7), (1, 5), (4, 6), (7, 5), (7, 7), (2, 2), (1, 6),(9,2), (7,9)] respectivamente



3. Comprobar que wrap funciona, usando *showcells* para *gliders* con elementos mayores de width/height:
   Para el *glider* [(3, 7), (18, 5), (14, 6), (7, 35), (7, 7), (2, 2), (1, 26),(9,16), (7,10)]:



7B. (MEMORIA) Resultado de la ejecución de los apartados anteriores (3B, 4B, 5B y, opcionalmente, 6B) para un mínimo de 4 turnos y pruebas realizadas para comprobar su correcto funcionamiento**.**

Veamos las distintas funciones individualmente:

La función del ejercicio 3B, *life_evo* :

```
life_evo :: Board -> Pos -> String
life_evo b p = if isAlive b p then "X" else "_"
```

Es muy sencilla. Podemos comprobar que funciona introduciendo 2 coordenadas de un *glider* básico (1,1) y pidiendo *life_evo*(1,1), (2,2)

```
ghci> life_evo glider pos
"X"
ghci> life_evo glider pos2
"_"
ghci>
```

La función del ejercicio 4B, life_changes:

```
life_changes :: Board -> IO ()
life_changes b = do
  let numChanges = sum $ zipWith (\p1 p2 -> if p1 == p2 then 0 else 1) b (nextgen b)
  putStrLn $ "Número de cambios : " ++ show numChanges
```

y un resultado de su ejecución con el *glider* [(1,1),(1,3),(1,4),(1,6),(2,3),(2,4),(3,1),(3,6),(4,1),(4,3),(4,4),(4,6),(6,2),(6,3),(6,4),(6,5)].

```
Número de cambios : 16
```

Podemos ver que utiliza zipWith y una función lambda para comparar las células de este tablero con las de su siguiente generación, devolviendo el número de cambios con la anterior.

La modificación de life en el ejercicio 5B:

```
80  life :: Board -> Int -> IO ()
81  life _ 0 = return ()
82  life b n = do
83      let center = (div width 2, div height 2)
84          corners = [(1,1), (1,height), (width,1), (width,height)]
85          posiciones = center:corners  -- creamos una lista con las posiciones relevantes que luego seran devueltas
86      putStrLn $ "Evolución del centro  y de las esquinas: " ++ unwords (map (life_evo b) posiciones)
87      life_changes b
88      wait
89      life ((nextgen b)) (n-1)
90
```

En ella podemos ver como primero hace una lista con los “puntos notables” que debe devolver y luego utilizando *life_evo* devuelve su siguiente estado en forma de lista.

Un resultado de su ejecución con el *glider* [(1,1),(1,3),(1,4),(1,6),(2,3),(2,4),(3,1),(3,6),(4,1),(4,3),(4,4),(4,6),(6,2),(6,3),(6,4),(6,5)], Y n = 5.

```
Evolución del centro  y de las esquinas: _ X X _ _
Número de cambios : 16

Evolución del centro  y de las esquinas: X X X X X
Número de cambios : 4

Evolución del centro  y de las esquinas: _ _ _ _ _
Número de cambios : 4

Evolución del centro  y de las esquinas: _ _ _ _ _
Número de cambios : 15

Evolución del centro  y de las esquinas: X X X X X
Número de cambios : 12
```

Este es el resultado de la ejecución en un mínimo de 4 turnos pedida.
Nótese que después de cada turno hay que presionar enter para llamar al siguiente.

Y el ejercicio 6B, para el cual hemos tenido que crear la siguiente función:

```haskell
celulaSumInvBin :: Board -> Pos -> Bool
celulaSumInvBin b p
    | liveNeighbors > deadNeighbors = False
    | liveNeighbors < deadNeighbors = True
    | otherwise = not (isAlive b p)
  where
    liveNeighbors = liveneighbs b p
    deadNeighbors = 8 - liveNeighbors
```

En la cual, para cada célula, recibiendo el tablero revisa,según las normas de la **CelulaSumInvBinario** explicadas en la práctica 1A y devuelve el estado que tendrá que tener la célula.

y modificar las siguientes funciones:

```haskell
nextgen :: Board -> Board
nextgen b = [p | p <- rmdups (concat (map neighbs b)), celulaSumInvBin b p]
```

En la cuál va por todas las posiciones del tablero revisando cómo va a ser la siguiente generación, usando map y concat para concatenarlos en una lista.