Ejercicios de Programación Declarativa

Curso 2019/20
Hoja 4

1. Supongamos que definimos el tipo `data Pila = P [a]` para representar pilas. Define funciones `crea Pila` para crear una pila vacía, `esPilaVacia` para determinar si una pila dada está vacía o no, `apilar` para apilar un elemento, `cima` para consultar la cima de una pila no vacía y `desapliar` para eliminar la cima de una pila no vacía. Determina el significado de la siguiente definición:

```
r :: [a] -> [a]
r xs = ys
  where P ys = foldl (\p x -> apilar x p) creaPila xs
```

```
data Pila a = P [a]
  deriving Show

creaPila :: Pila a
creaPila = P []

esPilaVacia :: Pila a -> Bool
esPilaVacia (P []) = True
esPilaVacia pila   = False

apilar :: a -> Pila a -> Pila a
apilar x (P xs) = P (x:xs)

cima :: Pila a -> a
cima (P (x:xs)) = x

desapilar :: Pila a -> Pila a
desapilar (P (x:xs)) = P xs

invierte :: [a] -> [a]
invierte xs = ys
  where P ys = (foldl (flip apilar) creaPila xs)
```

`invierte` = `r` Dada una lista cualquiera la invierte.

2. Define una función `primeroQueCumple :: (a -> Bool) ->[a] -> Maybe a` que dada una propiedad y una lista devuelva el primer elemento de la lista que cumple la propiedad. Devuelve `Nothing` en el caso de que ninguno la cumpla.

```
primeroQueCumple :: (a ->  Bool) -> [a] ->  Maybe a
primeroQueCumple p xs
```

```
  | null cumplen = Nothing
  | otherwise    = Just (head cumplen)
  where cumplen  = filter p xs
```

3. Define un tipo de datos `Cj` para representar conjuntos de elementos del mismo tipo. Define funciones para crear un conjunto vacío, para determinar si un conjunto dado está vacío o no, para determinar si un elemento pertenece o no a un conjunto y para devolver la lista con todos los elementos que pertenecen a un conjunto.

```
data Cj a = Cj [a]
 deriving (Eq,Show)

creaCjVacio :: Cj a
creaCjVacio = Cj []

esCjVacio :: Eq a => Cj a -> Bool
esCjVacio c = c == (Cj [])

incluirElem :: Eq a => a -> Cj a -> Cj a
incluirElem x c =
    case c of Cj [] -> Cj [x]
              Cj xs -> if elem x xs then c else Cj (x:xs)

{- Para asegurar que se trata de un conjunto,
hay que comprobar que no hay elementos repetidos -}

eliminaRepCj :: Eq a => Cj a -> Cj a
eliminaRepCj (Cj xs) = foldr incluirElem creaCjVacio xs
```

4. Define un tipo para representar matrices de números reales. Escribe una función que calcule la transpuesta de una matriz rectangular dada. Escribe una función para calcular la operación de suma de matrices.

```
type Vector = [Float]
type Matriz = [Vector]

transpuesta :: Matriz -> Matriz
transpuesta [] = []
transpuesta [fila] = map (:[]) fila
transpuesta (fila:fs) = zipWith (:) fila (transpuesta fs)

sumaMats :: Matriz -> Matriz -> Matriz
sumaMats m1 m2 = zipWith sumafilas m1 m2
       where sumafilas f1 f2 = zipWith (+) f1 f2

-- Y ya puestos, el producto de matrices
```

```
escalar :: Vector -> Vector -> Float
escalar us vs = sum $ zipWith (*) us vs

prodVecMat :: Vector -> Matriz -> Vector
prodVecMat v m = [escalar v fila | fila <- m]

prodMats :: Matriz -> Matriz -> Matriz
prodMats m1 m2 = [prodVecMat fila m2' | fila <- m1] where m2' = transpuesta m2
```

5. Dada la declaración:

   `data Temp = Kelvin Float | Celsius Float | Fahrenheit Float`

   para representar temperaturas en diferentes escalas, escribe una función para realizar conversiones de una escala a otra y otra para determinar la escala en la que está representada una temperatura. El nuevo tipo tiene que ser instancia de las clases `Ord` y `Eq`. Define adecuadamente los métodos `compare` y `==` para la nueva estructura de datos.

```
data Temp = Kelvin Float | Celsius Float | Fahrenheit Float
  deriving (Show,Read)

ceroAbs :: Float
ceroAbs = -273.15

escala :: Temp -> String
escala (Kelvin x)     = "Kelvin"
escala (Celsius x)    = "Celsius"
escala (Fahrenheit x) = "Fahrenheit"

-- Cambio a Kelvin
-- x C = x + 273,15 K
-- x F = (x  32) * 5 / 9 + 273,15 K
toKelvin :: Temp -> Temp
toKelvin (Celsius x) = Kelvin (x-ceroAbs)
toKelvin (Fahrenheit x) = Kelvin (((x-32)/1.8)-ceroAbs)
toKelvin k = k

-- Cambio a Fahrenheit
-- x C = (x * 9 / 5) + 32 F
-- x K = (x  273,15) * 9 / 5 + 32 F
toFahrenheit :: Temp -> Temp
toFahrenheit (Celsius x) = Fahrenheit (32+ x*1.8)
toFahrenheit (Kelvin x) = Fahrenheit (32+ (x+ceroAbs)*1.8)
toFahrenteit f = f

-- Cambio a Celsius
-- x K = x - 273,15 C
-- x F = (x  32)  5 / 9 C
```

```
toCelsius :: Temp -> Temp
toCelsius (Kelvin x) = Celsius (x+ceroAbs)
toCelsius (Fahrenheit x) = Celsius ((x-32)/1.8)
toCelsius t = t

instance Eq Temp where
  x == y = cx == cy
    where Celsius cx = (toCelsius x); Celsius cy = (toCelsius y)

instance Ord Temp where
  compare x y = compare cx cy
    where Celsius cx = (toCelsius x); Celsius cy = (toCelsius y)
```

6. Declara adecuadamente un tipo de datos para representar árboles binarios de búsqueda con valores en los nodos pero no en las hojas. Programa en Haskell la ordenación de una lista por el algoritmo `treeSort`, consistente en ir colocando uno a uno los elementos de la lista en un árbol binario de búsqueda inicialmente vacío. A continuación devuelve la lista resultante de recorrer el árbol en *inOrden*.

```
data Arbol a = AVacio | Nodo a (Arbol a) (Arbol a)
  deriving (Eq, Ord, Show)

creaVacio :: Ord a => Arbol a
creaVacio = AVacio

treeSort :: Ord a => [a] -> [a]
treeSort xs = arbolToLista(listaToarbolOrd xs)

inOrden :: Ord a => Arbol a -> [a]
inOrden AVacio = []
inOrden (Nodo x arbolL arbolR) =
     inOrden arbolL ++ [x] ++ inOrden arbolR

listaToarbolOrd :: Ord a => [a] -> Arbol a
listaToarbolOrd xs = foldl colocaord creaVacio xs

colocaord :: Ord a =>  Arbol a -> a -> Arbol a
colocaord AVacio x = Nodo x AVacio AVacio
colocaord (Nodo y arbolL arbolR) x
      | x <= y = (Nodo y (colocaord arbolL x) arbolR)
      | otherwise = (Nodo y arbolL (colocaord arbolR x))

estaVacio :: Ord a => Arbol a -> Bool
estaVacio AVacio = True
estaVacio arbol = False
```

7. Escribe una función `adivina n` para jugar a adivinar un número. Debe pedir que el usuario introduzca un número hasta que acierte con el valor de `n`. Devuelve mensajes de ayuda

indicando si el número introducido es menor o mayor que el número `n` a adivinar. Observa que el tipo de la función será `adivina :: Int -> IO ()`.

```
adivina :: Int -> IO ()
adivina num = do
   putStrLn "Elige un numero:"
   guess <- getLine
   if (read guess) < num
     then do putStrLn "El numero buscado es mayor"
             adivina num
     else if (read guess) > num
            then do putStrLn "El numero buscado es menor"
                    adivina num
            else putStrLn "Acertaste"
```

---

8. Escribe un programa que lea una línea introducida por teclado y muestre el número de palabras que contiene.

```
numpal :: IO ()
numpal = do
  putStrLn "Escribe una frase"
  frase <- getLine
  n <- return (length (words frase))
  putStrLn ("Tu frase tiene " ++ show n ++ " palabras")
```