Programación Declarativa. Grado Matemáticas. UCM.
HOJA 2 DE EJERCICIOS

1. Dado un número entero, escribe una función que devuelva la lista de sus dígitos. Por ejemplo, para 1345 debería devolver [1,3,4,5]. Escribe otra función que dado un número entero devuelva su imagen especular. Es decir, para 1345 debería devolver 5431.

2. Supón que definimos el tipo `data Natural = Cero | SUC Natural`. Define funciones para hacer la suma, la resta propia, el producto, el factorial y para elevar un natural a otro.

3. Dada la declaración

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

   para representar una temperatura con la opción de hacerlo en distintas escalas, escribe funciones para realizar conversiones de una escala a otra, así como para devolver en qué escala está representada una cierta temperatura. Incluye definiciones adecuadas para incluir el nuevo tipo dentro de las clases `Eq` y `Ord`. Ten en cuenta que para introducir un tipo en la clase Ord tienes que definir de forma adecuada la función `compare`:

   ```
   instance Ord Temp where
     compare x y = ...
   ```

   Por su parte, para incluirlo dentro de la clase `Eq` tienes que definir adecuadamente el operador `==`.

4. Define un tipo de datos adecuado para manejar árboles de búsqueda binarios que no tengan elementos en las hojas pero sí en los nodos intermedios. Define funciones adecuadas para crear un árbol vacío, para saber si un árbol está vacío, para añadir un elemento, para eliminarlo y para hacer recorridos en preOrden, inOrden y postOrden.

5. Implementa el algoritmo de ordenación `treeSort`. Dicho algoritmo parte de una lista de elementos y su funcionamiento consiste en ir añadiendo uno a uno todos los elementos a un árbol binario de búsqueda. Una vez finalizado el proceso, se realiza un recorrido en inOrden del árbol y se obtiene la lista ordenada.

6. Suponiendo que definimos un nuevo tipo de datos para representar pilas, y suponiendo que para ello nos basamos en la siguiente declaración:

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

   Define funciones para crear una pila vacía, para apilar un elemento, para determinar si una pila está vacía o no, para consultar la cima de una pila no vacía y para eliminar la cima de una pila no vacía.

7. Suponiendo que definimos un nuevo tipo de datos para representar conjuntos, y suponiendo que para ello nos basamos en la siguiente declaración:

   ```
   data Eq a => Cjto a = Cj [a]
   ```

   Define funciones para crear un conjunto vacío, para añadir un elemento a un conjunto, para determinar si un conjunto está vacío o no, para eliminar un elemento, para determinar si un elemento pertenece o no a un conjunto y para devolver una lista con todos los elementos que pertenecen al conjunto.

8. Repite el ejercicio anterior pero basando la nueva implementación en el uso de árboles binarios de búsqueda. ¿Tienes que cambiar algo en el contexto de clases de tipos?

9. Escribe una función `primeroQueCumple :: (a->Bool) -> [a] -> Maybe a` que dada una propiedad y una lista de elementos devuelva el primer elemento de la lista que cumple la propiedad. Si ningún elemento de la lista cumplía la propiedad, debe devolver `Nothing`.

10. Escribe una función `inits :: [a] -> [[a]]` que devuelva la lista de todos los segmentos iniciales de la lista de entrada. Por ejemplo, `inits [8,5,2,3]` debe devolver `[[],[8],[8,5],[8,5,2],[8,5,2,3]]`.

11. Escribe una función `quitaDups :: Eq a => [a] -> [a]` que dada una lista de entrada devuelva esa misma lista pero eliminando de ella los elementos repetidos adyacentes. Es decir, `quitaDups [3,5,5,5,3,8,8,5]` debe devolver `[3,5,3,8,5]`.

12. Escribe dos funciones `lineas` y `antiLineas` análogas a las funciones `lines` y `unlines` que están predefinidas en el preludio. La función `lines :: String -> [String]` convierte un texto en la lista de líneas que lo forman, mientras que la función `unlines :: [String] -> String` crea un único String a partir de una lista de líneas.

13. Escribe una función `justifDr :: Int -> String -> String` que dado un entero `n` y un texto cualquiera (probablemente con múltiples líneas) devuelva ese mismo texto pero justificado a la derecha asumiendo que la anchura de toda línea es de `n` caracteres.

    Nota: Si quieres probarlo con un texto de entrada grande leído directamente de un fichero de texto, puedes utilizar el siguiente fragmento de código (que entenderás dentro de unos días):

    ```
    procesa ancho = do texto <- readFile "c:\directorio\fichero.txt"
                       writeFile "c:\directorio\ficheroDr.txt" (justifDr ancho texto)
    ```

    Lógicamente, los nombres y rutas de los ficheros deben ser los que tú quieras, y para justificar el fichero con una anchura de 80 caracteres sólo tienes que ejecutar en el intérprete `procesa 80`.

14. Repite el ejercicio anterior para una función `justifCen :: Int -> String -> String` que centre el texto de entrada.

15. Las funciones predefinidas `words` y `unwords` son similares a `lines` y `unlines`, pero considerando la lista de palabras del texto en lugar de la lista de líneas. Teniendo esto en cuenta, escribe una función `justifAmbos` que se comporte de forma análoga a las dos funciones de los ejercicios anteriores, pero justificando el texto tanto por la izquierda como por la derecha. Fíjate en que para justificar de dicha forma una línea sólo necesitas añadir espacios en blanco entre las distintas palabras (en vez de añadirlos al principio o al final). Si tienes que añadir 15 blancos y tienes 9 palabras, tendrás 8 posiciones donde añadir blancos, así que usarás 2 blancos en cada una de las 7 primeras posiciones y uno entre las dos últimas palabras.

16. Suponiendo una representación de los vectores como `[Float]`, escribe funciones que sumen dos vectores, resten dos vectores y realicen el producto escalar de dos vectores.

17. Suponiendo una representación de las matrices como `[[Float]]`, escribe funciones que sumen dos matrices, resten dos matrices, calculen la traspuesta de una matriz y realicen el producto de dos matrices.

18. Suponiendo una representación de los polinomios como una lista de coeficientes, de modo que [1,-4,0,2.3,5,-7] represente el polinomio $x^5 - 4x^4 + 2.3x^2 + 5x - 7$, escribe una función que dado un polinomio y un valor para la x evalúe el polinomio para dicho valor.

19. Si el tipo de la función `f` es `f :: a -> b -> c -> d`, ¿cuál será el tipo de `uncurry f`? ¿y el tipo de `(uncurry . uncurry) f`?

    Recuerda que las funciones `tester` (y `tester2`) vistas en clase esperaban que las funciones de entrada tuvieran un único parámetro de entrada. ¿Cómo podemos usarla si tienen dos parámetros de entrada?¿y si tienen tres?¿y si tienen cuatro?

    Usa tu respuesta para probar que efectivamente funcionaba alguna de las funciones que definiste para resolver los ejercicios de la Hoja 1.

    ¿Crees que hay alguna diferencia entre usar `(uncurry . uncurry)` y usar `(twice uncurry)`? Pregunta el tipo de ambos a Hugs (después de definir adecuadamente la función `twice`) e intenta deducir el motivo por el que difieren tanto).

20. Reescribe los ejercicios 8, 9, 11, 13, 24 y 25 de la Hoja 1 de modo que ninguno utilice definiciones recursivas, sino sólo llamadas a funciones de orden superior predefinidas.