Ejercicios de Programación Declarativa

Curso 2018/19
Hoja 3

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, `apliar` 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
   ```

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.

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.

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.

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.

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*.

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 ()`.

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