Programación Funcional

Curso ~~2017/2018~~, Ejercicios – Sesión práctica (Lote 3)

Esto es un repertorio de actividades sugeridas para la sesión de prácticas

1. Definir un tipo para representar números complejos y declararlo como instancia de las clases *Eq*, *Num*, usando `deriving` cuando sea conveniente. Declararlo también como instancia de *Show* de modo que, por ejemplo, `show` del término Haskell que represente al complejo $2 - 3i$ sea el string `"2-3i"`. *(Para declarar un tipo t como instancia de Show basta con definir la función* `show::t ->String`*, no hace falta definir las otras funciones de la clase Show)*

2. (i) Definir un tipo enumerado `Direccion` con cuatro valores que representen movimientos (*arriba, abajo, izquierda, derecha*) por una cuadrícula en el plano con coordenadas enteras. Convertirlo en instancia de `Eq, Ord, Show` usando `deriving`.

   (ii) Definir una función `mueve movs punto` que al aplicarse a un punto del plano y una lista de movimientos, devuelva el punto final al que se llega.

   (ii) Definir una función `trayectoria movs punto` que al aplicarse a un punto del plano y una lista de movimientos, devuelva la lista de puntos por los que se pasa al aplicar `movs` a `punto`.

   (iii) Definir una función `inferior movs movs'` que devuelva `True` si la trayectoria determinada por `movs` a partir de cualquier `punto` nunca sube por encima de la determinada por `movs'`.

3. Definir un tipo de datos polimórfico para representar árboles generales, en los que cada nodo tiene una información y n hijos ($n \geq 0$, y puede variar con cada nodo). No se consideran árboles vacíos.

   - Programar las siguientes funciones:
     - `listaHojas t`, que obtiene la lista de las informaciones de todas las hojas del árbol `t`.
     - `listaNodos t`, que obtiene la lista de las informaciones de todos los nodos del árbol `t`.
     - `repMax t`, que devuelve el árbol resultante de poner como información de todos los nodos del árbol `t` la información más grande que aparece en `t`.
   - Declarar explícitamente el tipo de los árboles como instancia de la clase `Ord`, de manera que el orden definido sea el mismo que resultaría de usar `deriving Ord`.
   - Declarar el tipo de los árboles como instancia de la clase `Show`, de manera que la vista en pantalla de un árbol sea visualmente más atractiva que lo que nos da el poner simplemente `deriving Show`

4. Definir una clase de tipos `Medible` que disponga de una función `tamanyo::a ->Int` que se pueda aplicar a cada tipo `a` de dicha clase. Declara algunos tipos como instancia de la clase `Medible`, por ejemplo: `Bool`, `[a]` , `(a,b)`.

5. Definir un tipo de datos `Exp` para representar expresiones aritméticas, que pueden ser enteros, sumas, restas, multiplicaciones y divisiones. Definir funciones sujetas a las siguientes especificaciones:

```
eval:: Exp ->Int
-- eval e = resultado de evaluar e

cuentaOps:: Exp ->Int
-- cuentaOps e = número de operaciones aritméticas en e

cambiaOps:: Exp ->Exp
-- cambiaOps e = resultado de cambiar en e 'más´ por 'menos' y 'por' por 'entre'

operandos:: Exp ->[Int]
-- operandos e = lista de los enteros que aparecen en e
```

6. Considérese el siguiente tipo de datos para representar conjuntos finitos de elementos de un tipo cualquiera: `data Conjunto a = Con Int [a]`
donde en un dato `Con n xs` que represente a un conjunto C, el argumento `n` representa el cardinal de C y `xs` es la lista de sus `n` elementos.

   - Definir una función `toC` que convierta listas en conjuntos.
   - Definir la intersección y la unión de conjuntos.
   - Definir la función `mapset f c`, que calcula la imagen por `f` de un conjunto `c`, es decir, el conjunto resultado de aplicar la función `f` a cada elemento de `c`.
   - Declarar `Conjunto a` como instancia de las clase `Eq` y `Ord`, de modo que `==` y `<=` reflejen la noción matemática de igualdad de conjuntos y de inclusión de conjuntos, respectivamente.

7. Definir un tipo de datos polimórfico `ASec a b` para representar secuencias (posiblemente vacías) con valores alternos de tipos a y b. Y una vez definido el tipo:

   - Definir la función longitud de una secuencia de `ASec a b`.
   - Definir la función `nelem n abs` que devuelve el elemento $n-$simo de la secuencia `abs`.
   (¿Qué tipo puede tener esa función?)
   - Definir la función `separa abs` que devuelve una pareja de listas, la primera con los elementos de tipo `a` que hay en `abs` y la segunda con los de tipo `b`.
   - Examina qué problemas hay para definir las operaciones de concatenación e inversa de secuencias.
   - Declarar `ASec a b` como instancia de las clases `Eq` y `Ord`, de modo que una secuencia sea igual que otra si lo es elemento a elemento, y lo mismo para la relación 'menor o igual'.
   - Declarar `ASec a b` como instancia de la clase `Show`, de modo que la visualización de una secuencia sea en notación estándar de listas (con corchetes y comas).

8. *(Para mentes algo perversas)*
Hacer las declaraciones de instancia de clase adecuadas para conseguir los siguientes efectos exóticos, cuando no indeseables:

   - La evaluación de `1 && True` da `True` y no un error.
   - La evaluación de `map and [1,[2],3]` da `[True,False,True]` y no un error.

9. Programar funciones que realicen los siguientes procesos interactivos. Serán útiles las funciones `getLine, readFile, writeFile` del `Prelude`, entre otras.

   - `palabras:: String -> IO Int`
   `palabras FileIn` obtiene como valor asociado el número de palabras de `FileIn`.
   - `palabras':: IO ()`
   Como la anterior, pero leyendo el nombre del fichero de la entrada y mostrando el resultado con un mensaje de la forma: `El fichero` *Fichero* `tiene` n `palabras`.
   - `promedia :: IO ()`
   Proceso que va leyendo un entero de cada línea de la entrada y mostrando la suma y el promedio de los valores leídos hasta ese momento. El proceso se detiene al leer `-1` en la entrada
   - `calculadora :: IO ()` *(Intérprete de expresiones aritméticas)*
   Proceso que lee de la entrada una expresión aritmética (formada por números y operadores infijos `+,*,-,/`) y la evalúa.
   ¿A que no es tan fácil?
   - `formatea:: String -> String -> Int -> IO ()`
   `formatea FileIn FileOut n` formatea a `n` columnas de ancho cada línea de `FileIn` y escribe el resultado en `FileOut`. Para formatear se meten espacios intermedios repartidos de manera uniforme entre palabras de modo que la línea quede justificada a izquierda y derecha. Puede suceder que, por su longitud, la línea quede con más de `n` columnas.
   Nota: pueden ser útiles las funciones `words, unwords, lines, unlines` del `Prelude`.