Proyecto 1

Algoritmos y Estructuras de Datos I - Laboratorio
Tipos de datos en Haskell

17 de marzo de 2010

1. Dado el tipo de datos que representa títulos nobiliarios

```
data Titulo = Duque | Marques | Conde
            | Vizconde | Baron deriving Show
```

programar la función
`dama :: Titulo -> String`
que dado un título nobiliario devuelve su femenino (de `Duque` es "duquesa", etc.).

Nota: utilizar pattern matching sobre el tipo `Titulo`.

2. Dado el siguiente tipo de datos[1]

```
type Territorio = String
type Nombre = String

data Persona = Rey | Noble Titulo Territorio | Caballero Nombre
             | Aldeano Nombre deriving Show
```

programar las funciones

   a) `personaAStr :: Persona -> String`
   que dada una persona devuelve la forma en que se lo menciona en la corte. Esto es, al rey se lo nombra simplemente "Su majestad el rey", a un noble se lo nombra "El `<titulo>` de `<territorio>`", a un caballero "Sir `<nombre>`" y a un aldeano simplemente con su nombre.

   b) `sirs :: [Persona] -> [String]`
   que dada un lista de personas devuelve los nombres de los caballeros únicamente.

3. Implementar el tipo `Figuras` las cuales pueden ser

   - un triángulo con su base y su altura,
   - un rectángulo con su base y su altura,
   - una linea con su largo, o
   - un círculo con su diámetro.

   Implementar también una función que devuelva el área de una figura.

[1] Consultar en `http://www.lcc.uma.es/~blas/pfHaskell/gentle/goodies.html` sección "2.3 Type Synonyms" por sentencia `type`.

4. Un problema bastante común en programación es el almacenamiento de datos que poseen una clave que los identifican unívocamente. Ejemplo del mismo es el almacenamiento de palabras en un diccionario (la clave es la palabra y el dato es la definición), información sobre ciudadanos (la clave podría ser el DNI y el dato serían nombre, apellido, sexo, etc.). Una forma posible de representar esta situación es con el tipo de datos *lista de asociaciones* definido como:

```
data ListAssoc a b = Empty | Node a b (ListAssoc a b)
```

donde el parámetro de tipo `a` se instancia con una clave y `b` con el dato.

A partir del mismo programar las funciones

   a) `la_length :: Integral c => ListAssoc a b -> c` que devuelve la cantidad de datos en una lista.

   b) `la_toListPair :: ListAssoc a b -> [(a,b)]` que devuelve la lista de pares contenida en la lista de asociaciones.

   c) `la_search :: Eq a => ListAssoc a b -> a -> Maybe b`[2] que dada una lista y una clave devuelve el dato asociado si es que existe.

5. Dado el tipo datos

```
data Arbol a = Hoja | Rama (Arbol a) a (Arbol a) deriving Show
```

programar las funciones

   a) `aLength :: Integral b => Arbol a -> b` que dado un árbol devuelve la cantidad de datos almacenados.

   b) `aCantHojas :: Integral b => Arbol a -> b` que dado un árbol devuelve la cantidad de hojas.

   c) `aInc :: Num a => Arbol a -> Arbol a` que dado un árbol que contiene números, los incrementa en uno.

   d) `aPersonaAStr :: Arbol Persona -> Arbol String` que dado un árbol de personas, devuelve el mismo árbol con las personas representadas en la forma en que se las menciona en la corte (usar la función `personaAStr`).

   e) `aMap :: (a -> b) -> Arbol a -> Arbol b` que dada una función y un árbol, aplica la función a todos los elementos del árbol.

   Programar las funciones `aInc` y `aPersonaAStr` utilizando `aMap`.

   f) `aSum :: Num a => Arbol a -> a` que suma los elementos de un árbol.

   g) `aProd :: Num a => Arbol a -> a` que multiplica los elementos de un árbol.

6. La expresiones aritméticas (sin variables) están definidas como:

   - Una constante de tipo número es una expresión aritmética.
   - Si E es una expresión aritmética, también lo es $-E$
   - Si E_1 y E_2 son expresiones aritméticas, también lo es $E_1 + E_2$
   - Si E_1 y E_2 son expresiones aritméticas, también lo es $E_1 * E_2$

   a) Definir el tipo de dato en haskell `ExprArith a` que represente las expresiones aritméticas con constantes en el tipo a.

[2] Ver como esta definido el tipo de datos `Maybe` en el prelude de Haskell.

Nota: El tipo `ExprArith a` deberá tener cuatro constructores, uno para la suma, otro para la multiplicación, otro para el cambio de signo y otro para almacenar valores.

b) Programar la función
`eval :: Num a => ExprArith a -> a`
que dada un expresión devuelve su valor.

c) **(Punto $\star$)** Programar la función
`nnf :: Num a => ExprArith a -> ExprArith a`
que dada una expresión aplique **únicamente** la regla de los signos para que los signos "–" aparezcan únicamente en las variable. Por ejemplo, `nnf` aplicado a $-((3+4)*(2*5))$ devuelve $((-3)+(-4))*(2*5)$. Esto quiere decir que la expresión resultado debe evaluar al mismo valor y no debe tener el constructor de cambio de signo "–". Las reglas de los signos son las siguiente:

$$\begin{aligned} --E &= E \\ -(E_1+E_2) &= ((-E_1)+(-E_2)) \\ -(E_1*E_2) &= ((-E_1)*E_2) \end{aligned}$$