Proyecto 2

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

31 de agosto de 2010

En este proyecto definimos nuestros propios tipos de datos, esto implica que agregamos nuevos valores al modelo computacional. La importancia de poder definir nuevos tipos de datos está en la facilidad con la que podemos modelar distintos problemas y resolverlos usando las mismas herramientas que para los tipos pre-existentes en nuestro formalismo.

Uno de los objetivos de este proyecto es conocer la forma en que se declaran nuevos tipos de datos en Haskell; otro objetivo es aprender a definir funciones para manipular expresiones que correspondan a los nuevos tipos; por último, es importante poder definir un nuevo tipo de datos de manera de poder resolver adecuadamente un determinado problema.

Recordá definir las funciones con su declaración de tipo; además no dejés de documentar como comentarios las decisiones que tomás a medida que resolvés los ejercicios.

1. **Tipos enumerados.** Cuando los distintos valores que debemos distinguir en un tipo son finitos, entonces podemos *enumerar* todos los valores distintos para el tipo. Por ejemplo, podríamos representar los títulos nobiliarios de algún país (retrógrado) con el siguiente tipo:

```
data Titulo = Ducado | Marquesado | Condado | Vizcondado
            | Baronia
```

   a) Programar, usando pattern-matching, la función `hombre : Titulo -> String` que devuelve la forma masculina del título.

   b) Programar, usando pattern-matching, la función `dama` que devuelve la inflexión femenina.

2. **Tipos enumerados; constructores con argumentos.** En este ejercicio, introducimos dos conceptos: los sinónimos de tipos (ver Sec. 2.3 del tutorial o Sec. 4.2.2 del reporte) y tipos algebraicos cuyos constructores llevan argumentos. Los sinónimos nos permiten renombrar tipos ya existentes. Los argumentos en los constructores nos permiten agregar información a cada forma de construir un valor; por ejemplo, en este ejercicio además de distinguir el rango de cada persona, representamos datos pertinentes a cada tipo de persona.

```
-- Territorio y Nombre son sinónimos de tipo.
type Territorio = String
type Nombre = String

-- Persona es un tipo enumerado, alguno de cuyos constructores
-- llevan argumentos.
data Persona = Rey | Noble Titulo Territorio | Caballero Nombre
             | Aldeano Nombre
```

   a) Programar la función `tratamiento :: 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) Programar la función `sirs :: [Persona] -> [String]`
que dada un lista de personas devuelve los nombres de los caballeros únicamente.

c) ¿ Qué modificaciones se deben hacer para poder representar personas de distintos géneros en cualquier rango?

d) (**Punto** $\star$) Utilizar funciones del preludio para programar `sirs`. Ayuda: puede ser necesario definir un predicado: `Persona -> Bool` .

3. Considere el tipo `Figura` que debe poder usarse para representar:

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

   a) Definir el tipo de datos `Figura`.

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

4. **Tipos recursivos (y polimórficos).** Consideremos las siguientes situaciones, aparentemente no relacionadas entre sí:

   - Encontrar la definición de una palabra en un diccionario;
   - guardar el lugar de votación de cada persona.

   Tanto el diccionario como el padrón electoral almacenan información interesante que puede ser accedida rápidamente si se conoce la *clave* de lo que se busca; en el caso del padrón será el DNI, mientras que en el diccionario será la palabra (el lexema) en sí. Puesto que reconocemos la similitud entre un caso y el otro, deberíamos esperar poder representar con un único tipo de datos ambas situaciones; es decir, necesitamos un tipo polimórfico sobre las claves y la información almacenada.

   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)
   ```

   En esta definición notamos que el tipo que estamos definiendo aparece como un argumento de uno de sus constructores; por ello se dice que el tipo es recursivo. Los parámetro del constructor de tipo indican que es un tipo polimórfico, donde las variables `a` y `b` se pueden instanciar con distintos tipos; por ejemplo:

   ```
   type Diccionario = ListAssoc String String
   type Padron = ListAssoc Int String
   ```

   a) ¿Como se debe instanciar el tipo `ListAssoc` para representar la información almacenada en una guía telefónica?

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

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

   d) `la_buscar :: Eq a => ListAssoc a b -> a -> Maybe b` que dada una lista y una clave devuelve el dato asociado si es que existe. Puede consultar la definición del tipo `Maybe` en Hoogle.

   e) ¿Qué ejercicio del proyecto anterior se puede resolver usando una lista de asociaciones?

5. Otro tipo de datos muy útil y que se puede usar para representar muchas situaciones es el *árbol*; por ejemplo, el análisis sintáctico de una oración, una estructura jerárquica como un árbol genealógico o la taxonomía de Linneo.

   En este ejercicio consideramos *árboles binarios*, es decir que cada *rama* tiene sólo dos descendientes inmediatos:

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

   Como se muestra a continuación, usando ese tipo de datos podemos representar los prefijos comunes de varias palabras. Para ganar intuición sobre qué se almacena en el árbol `can`, sugerimos hacer un esquema gráfico del mismo.

```
type Prefijos = Arbol String

can :: Prefijos
can = Rama cana "can" cant

cana :: Prefijos
cana = Rama canario "a" canas

canario :: Prefijos
canario = Rama Hoja "rio" Hoja

canas :: Prefijos
canas = Rama Hoja "s" Hoja

cant :: Prefijos
cant = Rama cantar "t" canto

cantar :: Prefijos
cantar = Rama Hoja "ar" Hoja

canto :: Prefijos
canto = Rama Hoja "o" Hoja
```

   Programar las siguientes funciones:

   a) `aLargo :: 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) `aTratamiento :: 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 `tratamiento`).

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

   f) Definir nuevas versiones de `aInc` y `aTratamiento` utilizando `aMap`.

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

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

i) (**Punto** $\star$) Al principio del taller se utilizó el funcional `foldr` para programar varias funciones sobre listas sin escribir explícitamente la recursión (por ej. sumatoria, mínimo, productoria, length, etc.). ¿Existe un funcional `aFold` sobre árboles que se pueda utilizar en el mismo sentido? Si es así prográmelo y escriba las funciones `aSum`, `aProd`, `aLargo`, `aCantHojas` y `aMap` utilizando `aFold` y sin recursión.

6. Los tipos de datos algebraicos son especialmente útiles para representar expresiones simbólicas; es decir, en vez de calcular directamente, por ejemplo, el valor de una suma, podemos querer representar la suma como expresión simbólica y luego definir operaciones sobre ellas. La expresiones aritméticas (sin variables) están definidas como:

   - Una constante de tipo entero 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 `ExprInt` que represente las expresiones aritméticas.

   b) Definir un tipo polimórfico `ExprArith a`, tal que `ExprInt` sea equivalente a una instanciación de `ExprArith` .

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

   d) **(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:

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