Proyecto 2

Derivaciones y Tipos de datos en Haskell

Algoritmos y Estructuras de Datos I

6 de septiembre de 2011

Este proyecto consta de dos partes. La primera consiste en implementar los resultados de algunas derivaciones realizadas en el práctico.

La segunda parte consiste en definir 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.

Como tarea adicional se requiere que los ejercicios estén escritos en un módulo de Haskell.

Se recomienda hacer primero los ejercicios sin punto estrella. Una vez finalizados, revisados y testeados intentar recién, en ese momento, hacer los estrella.

1. Derivar los ejercicios 3b), 3c), 3d) y 3e) del práctico 2 de la materia y **después** implementar las funciones obtenidas en el formalismo básico en Haskell.

   > **Muy importante!** No escriba una línea de código Haskell hasta que no termine de derivar la función en lapiz y papel (le recomendamos que mientras deriva apague el monitor de la computadora).

   Para aprobar este ejercicio además hay que tener en cuenta:

   - Al momento de la evaluación hay que presentar en papel las funciones en el formalismo básico y sus derivaciones.
   - Las funciones resultados de las derivaciones deben corresponder a las implementadas en Haskell. Es decir, deben tener los mismos nombres, argumentos, estilo pattern matching o por casos, etc.
   - Al momento de la evaluación debe dar una explicación sobre el comportamiento operacional de las funciones. Esto es, debe poder explicar que hacen las funciones escritas en Haskell, de la misma manera que en los ejercicios siguientes.

2. **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.

3. **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 `<forma masculina del titulo>` de `<territorio>`", a un caballero "Sir `<nombre>`" y a un aldeano simplemente con su nombre.

   b) ¿ Qué modificaciones se deben realizar sobre el tipo `Persona` para poder representar personas de distintos géneros?

   Realice esta modificación y vuelva a programar la función `tratamiento` de forma tal de respetar el género de la persona al nombrarla.

   c) Programar la función `sirs :: [Persona] -> [String]`
   que dada un lista de personas devuelve los nombres de los caballeros únicamente. Utilizar caso base e inductivo.

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

4. 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.

5. **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?

6. 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 por ejemplo representar los prefijos comunes de varias palabras. Para ganar intuición sobre qué se almacena en el árbol `can`, 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** $\star$) En clases vimos el funcional `foldr :: (a -> b -> b) -> b -> [a] -> b` el cual aplica un operador entre los elementos de una lista y el segundo parámetro (elemento neutro), asociando a derecha. Otra forma de explicarlo es pensando que el funcional reemplaza los constructores de lista `:` y `[]` por el operador y el elemento neutro respectivamente:

$$\begin{array}{lll} \texttt{foldr} & (\oplus) & z \quad [a_1, a_2, \cdots, a_n] \\ = & & \\ \texttt{foldr} & (\oplus) & z \quad a_1 : (a_2 : (\cdots (a_n : [\,]) \cdots)) \\ & & \quad\;\; \downarrow \quad\;\; \downarrow \quad\;\; \cdots \quad\;\; \downarrow \downarrow \\ = & & \quad\;\; a_1 \oplus (a_2 \oplus (\cdots (a_n \oplus z\,) \cdots)) \end{array}$$

En este sentido, de reemplazo de constructores, programar una función que haga *fold* sobre árboles en vez de sobre listas. La función debe tomar un operador ternario (ya que el constructor sobre árboles toma 3 parámetros) un caso base (que reemplazará a las hojas) y un arbol, devolviendo un resultado:

```
aFold :: (b -> a -> b -> b) -> b -> Arbol a -> b
```

Programe las funciones `aSum`, `aProd`, `aLargo`, `aCantHojas` y `aMap` utilizando `aFold` en una sola definición (sin recursión).

Ayuda: *Después de encontrar la definición de* `aFold` *testear (o demostrar) que se cumpla:*

```
aFold Rama Hoja a = a
```

(en forma análoga que `foldr (:) [] xs = xs`*).*

7. 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.

   Ayuda: *Cada uno de los cuatro items anteriores que definen una expresión aritmética, define un constructor del tipo* `ExprInt`*. Es decir, el tipo tendrá cuatro constructores, uno para representar valores de tipo* `Int`*, uno para representar el cambio de signo, uno para la suma y otro para la multiplicación.*

   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:

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