Proyecto 2

Derivaciones y Tipos de datos en Haskell

Algoritmos y Estructuras de Datos I

11 de abril de 2013

Este proyecto consta de dos partes. La primera (Ejercicios 1,2,7) consiste en implementar los resultados de algunas derivaciones realizadas en el práctico.

La segunda parte consiste en definir nuestros propios tipos de datos. 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 aprender a declarar nuevos tipos de datos en Haskell y definir funciones para manipular expresiones que correspondan a los nuevos tipos.

Recordá:

- definir las funciones con su declaración de tipo
- poner como comentarios las decisiones que tomás a medida que resolvés los ejercicios
- usar archivos distintos de manera coherente (no todo en un mismo archivo)

Se recomienda hacer primero los ejercicios sin punto estrella.

1. En Haskell, programar las funciones `fac`, `sum` y `fib` del ejercicio 1 del práctico 2 de la materia, y verificar que las propiedades que se piden demostrar (de nombre `P`) se verifiquen.

   Ayuda: *Para comprobar la propiedad sobre las funciones utilizar* `cuantGen`.

2. Implementar en Haskell las funciones obtenidas en el formalismo básico en los siguientes ejercicios del práctico 2 de la materia.

Muy importante! No escriba 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).

   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.

   a) Ejercicio 3 función `exp`
   b) Ejercicio 3 función `sum_cuad`
   c) Ejercicio 3 función `cuantos`
   d) Ejercicio 3 función `busca`

   Una vez programadas las funciones ejecútelas y pruebe para diferentes casos que el resultado devuelto es correcto con respecto a sus correspondientes especificaciones.

3. **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 la función `hombre :: Titulo -> String` que devuelve la forma masculina del título.

   b) Programar la función `dama` que devuelve la inflexión femenina.

4. **Tipos enumerados; constructores con argumentos.** En este ejercicio, introducimos dos conceptos: los sinónimos de tipos (ver Sec. 2.3 del tutorial) 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 sinonimos 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 "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? Ayuda: agregarle un argumento a los constructores.
   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` .

5. 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.

6. **Tipos recursivos (y polimórficos).** Consideremos las siguientes situaciones:

   - 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 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, el tipo que estamos definiendo (`ListAssoc a b`) aparece como un argumento de uno de sus constructores (`Node`); 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_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?

7. Implementar en Haskell las funciones obtenidas en el formalismo básico en los siguientes ejercicios del práctico 2 de la materia.

   Muy importante! No escriba 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).

   Tener en cuenta que 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.

   a) Ejercicio 5: `minimo`, `creciente`, `iguales`

   b) Ejercicio 7: `pi` con costo lineal. Comparar con la constante `pi` definida en `Prelude.hs`

   c) Ejercicio 8: `psum` y `sum8`.

   d) Ejercicio 9: `cuad` y `n8`

   Una vez programadas las funciones pruebe para diferentes casos que el resultado devuelto es correcto con respecto a sus correspondientes especificaciones.

8. (**Punto** $\star$) 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. Sugerimos hacer un esquema gráfico del árbol `can`:

```
type Prefijos = Arbol String

can, cana, canario, canas, cant, cantar, canto :: Prefijos
can     = Rama cana "can" cant
cana    = Rama canario "a" canas
canario = Rama Hoja "rio" Hoja
canas   = Rama Hoja "s" Hoja
cant    = Rama cantar "t" canto
cantar  = Rama Hoja "ar" Hoja
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) `aSum :: Num a => Arbol a -> a`
   que suma los elementos de un árbol.

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

9. (**Punto** $\star$) Programar el ejercicio de cálculo de `pi` del punto 7*b*, utilizando la aproximación mediante el método Bailey 1997, que aparece en este link.

   Compara esta implementación con la anterior.