# Tipos de datos en Haskell

## 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) Definir el tipo Titulo como descrito arriba. No tiene que pertenecer a la clase `Eq`.
b) Definir `hombre :: Titulo -> String` que devuelve la forma masculina del título.
c) Definir 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) y tipos algebraicos cuyos constructores llevan argumentos. Los sinónimos de tipos nos permiten definir nombres de tipos nuevo a partir de tipos ya existentes:

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

cba , bsas :: Territorio
cba = "Cordoba"
bsas = "Buenos Aires"

alicia , bob :: Nombre
alicia = "Alicia"
bob = "Bob"
```

Los tipos algebraicos tienen constructores que llevan argumentos. Esos argumentos nos permiten agregar información; por ejemplo, en este ejercicio además de distinguir el rango de cada persona, representamos datos pertinentes a cada tipo de persona:

```
-- Persona es un tipo algebraico
data Persona = Rey                    -- constructor sin argumento
             | Noble Titulo Territorio -- constructor con dos argumentos
             | Caballero Nombre        -- constructor con un argumento
             | Aldeano Nombre          -- constructor con un argumento
```

a) Definir los tipos `Territorio`, `Nombre` y `Persona` como descrito arriba. Este último tipo no tiene que pertenecer a la clase `Eq`.

b) Chequear los tipos de los constructores de `Persona` en `ghci` usando el comando `:t`.

c) 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 de ", a un caballero "Sir " y a un aldeano simplemente con su nombre.

d) ¿ Qué modificaciones se deben realizar sobre el tipo Persona para poder representar personas de distintos géneros sin agregarle más constructores? Realice esta modificación y vuelva a programar la función tratamiento de forma tal de respetar el género de la persona al nombrarla, sin usar guardas.

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

f) Utilizar funciones del preludio para programar sirs. Puede ser necesario definir un predicado de tipo `Persona -> Bool`.

## 3. Tipos recursivos

Considerar el siguiente tipo algebraico siguiente:

```
data IntExp = Const Int      -- constante
     | Op IntExp             -- opuesto
     | Plus IntExp IntExp    -- suma
     | Times IntExp IntExp   -- multiplicación
     | Div IntExp IntExp
```

Este tipo algebraico representa expresiones aritméticas obtenidas a partir de cinco constructores (constantes, opuesto, suma, multiplicación y división). A continuación algunos ejemplos que muestran cómo construir expresiones:

```
Plus ( Const 5 ) ( Const 3 ) -- 5 + 3
Plus ( Const 5 ) ( Times ( Const 3 ) ( Const 2 ) ) -- 5 + (3*2)
Times (Op (Const 3)) (Div (Const 4) (Const 2)) -- (-3) * (4/2)
```

a) Definir una función `eval :: IntExp -> Int` que dada una expresión aritmética retorna su valor. (Para eso, utilizar los operadores aritméticos de Haskell, para la división usar el operador `div`.)
b) Definir la función `substract :: IntExp -> IntExp -> IntExp` que dadas dos expresiones enteras, devuelve otra expresión que representa la resta entre la dos. (Se puede definir usando Plus y Op. La función `eval` no debe cambiar respecto al punto anterior.)
c) Definir el tipo algebraico `BoolExp` (expresiones booleanas) que tenga como constructores las constantes (true y false), el "y" (And), el "o" (Or) y la negación lógica (Not).
d) Definir la función `evalb :: BoolExp -> Bool` que dada una expresión booleana, devuelve su valor de verdad.

## 4. 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 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 característica que se da en ambos ejemplos previos es que, el diccionario no tienen palabras repetidas y el padrón no tiene documentos repetidos.

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_existe :: Eq a => ListAssoc a b -> a -> Bool` que dada una lista y una clave devuelve `True` si la clave está y `False` en caso contrario.
e) `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.
f) `la_agregar :: Eq a => a -> b -> ListAssoc a b -> ListAssoc a b` que dada una clave `a`, un dato `b` lo agrega en la lista si la clave a NO existe. En caso de que la clave a exista, se reemplaza el dato ingresado como parámetro.
g) `la_borrar :: Eq a => a ListAssoc a b -> ListAssoc a b` que dada una clave `a` elimina la entrada en la lista.

Nota: Nota: Tener en cuenta que para que no haya elementos repetidos, debemos asegurarnos que solo agreguemos elementos a la `ListAssoc a b` utilizando la función `la_agrega`.

## 5. Tipos recursivos y polimórficos II: Árboles

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 ) deriving Show
```

Programar las siguientes funciones:

a) `arbolProp :: Int -> Arbol Int` que, dado un entero positivo `n`, construye un árbol de tamaño proporcional a `n`. Esta función va a servir para probar las funciones siguientes; hay varias formas interesantes de árbol que se pueden definir.
b) `aLargo :: Integral b => Arbol a -> b` que dado un árbol devuelve la cantidad de datos almacenados.
c) `aCantHojas :: Integral b => Arbol a -> b` que dado un árbol devuelve la cantidad de hojas.
d) `aInc :: Num a => Arbol a -> Arbol a` que dado un árbol que contiene números, los incrementa en uno.
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 una nueva versión de `aInc` 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.

## 6. Tipos recursivos y polimórficos III: Automatas Funcionales

En este ejercicio trabajamos con una versión simplificada de automata, donde los estados tienen una sola transición a otro estado.

Sea `Estado` un tipo sinónimo de `Int`, y `EstadosFinales` sinónimo de `[Estado]`. Sea `Automata` un sinónimo de `(ListAssoc Estado Estado, EstadosFinales)`.

Queremos escribir la función `alcanza`, que dado un `Automata` y un `Estado`, devuelve un `Bool` indicando si el estado dado puede alcanzar un estado final.

También queremos probar `alcanza` con un ejemplo simple.

a) Definir los tipos de datos descritos más arriba.
b) Definir la función `lineal :: Int -> Automata` tal que `lineal n` sea el automata `(0 -> 1, 1-> 2, ... , n-1 -> n , [n])`.
c) Definir la función `alcanza` (usar `la_search` y `case ... of`). No es necesario tratar de evitar recursión infinita.
d) Probar la función alcanza con automatas construidos con `lineal` y varios estados de partida.