Algoritmos y Estructuras de Datos I - Laboratorio
Proyecto 2

Derivaciones y Tipos de datos en Haskell

1. Objetivo

Este proyecto consta de dos partes. La primera (Ejercicios 1) 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:

- poner como comentarios las decisiones que tomás a medida que resolvés los ejercicios
- nombrar las distintas versiones de una misma función `f`, `f''`, `f'''`, ...
- Se verificará que no hay warnings al cargar el archivo en `ghci`.

2. Ejercicios

Se recomienda hacer primero los ejercicios sin punto estrella.

1. 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 2.c función `exp`

   b) Ejercicio 2.b función `iga`

   c) Ejercicio 2.e función `cuantos`

   d) Ejercicio 2.f función `busca`

   Una vez programadas las funciones, probarlas y chequear que el resultado devuelto es correcto con respecto a sus correspondientes especificaciones.

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) 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.

3. **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 tipos 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 <forma masculina del titulo> de <territorio>", a un caballero "Sir <nombre>" 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) (**Punto** $\star$) Utilizar funciones del preludio para programar `sirs`. Ayuda: puede ser necesario definir un predicado: `Persona -> Bool` .

4. **Tipos recursivos** Considerar el siguiente tipo algebraico en Haskell:

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

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

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

a) Definir una función `eval :: IntExp -> Int` que dada una expresión aritmética retorna su valor. A continuación algunos ejemplos de cómo debe funcionar `eval`:

```
$> eval (Plus (Const 5) (Const 3))
8

$> eval (Plus (Const 5) (Times (Const 3) (Const 2)))
11

$> eval (Times (Op (Const 3)) (Div (Const 4) (Const 2)))
-6
```

Ayuda: 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. Algunos ejemplos:

```
$> eval (substract (Const 4) (Const 1))
3

$> eval (substract (Op (Const 1)) (Const 1))
-2
```

Ayuda: 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.

5. **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 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.

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

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

6. (**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.

7. (**Punto** $\star$) En este ejercicio trabajamos con una versión simplificada de automata, donde los estados tienen una sola transición a otro estado.

   Sea `Estado` un 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 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.