Algoritmos y Estructuras de Datos I - Laboratorio
Proyecto 2

Tipos de datos

1. Objetivo

En este proyecto nos introducimos en la definición de nuestros propios tipos de datos. La importancia de poder definir nuevos tipos de datos reside en la facilidad con la que podemos modelar problemas y resolverlos usando las mismas herramientas que para los tipos pre-existentes.

El objetivo de este proyecto es aprender a declarar nuevos tipos de datos en Haskell y definir funciones para manipular expresiones que utilizan estos tipos.

Recordá:

- Usar `ghci` con el flag `-Wall`.
- Poner como comentarios las decisiones que tomás a medida que resolvés los ejercicios.
- Nombrar las distintas versiones de una misma función `f`, `f'`, `f''`, ...

2. Ejercicios

1. **Tipos enumerados.** Cuando los distintos valores que debemos distinguir en un tipo son finitos, podemos *enumerar* cada uno de los valores del tipo. Por ejemplo, podríamos representar las carreras que se dictan en nuestra facultad con el siguiente tipo:

```
data Carrera = Matematica | Fisica | Computacion | Astronomia | Profesorado
```

   Cada uno de estos valores es un *constructor*, ya que al utilizarlo en una expresión, generan un valor del tipo `Carrera`.

   a) Implementá el tipo `Carrera` como está definido arriba.

   b) Definí la siguiente función, usando *pattern matching*: `titulo :: Carrera -> String` que devuelve el nombre completo de la carrera en forma de *string*. Por ejemplo, para el constructor `Matematica`, debe devolver "Licenciatura en Matemática".

   c) ¿Podés definir la función anterior usando análisis por casos? ¿Por qué?

2. **Tipos enumerados; constructores con parámetros.** En este ejercicio, introducimos dos conceptos: los sinónimos de tipos y tipos algebraicos cuyos constructores llevan argumentos. Un sinónimo de tipo nos permite definir un nuevo nombre para un tipo ya existente, como el ya conocido tipo `String` que no es otra cosa que un sinónimo para `[Char]`. Por ejemplo, si queremos modelar el año de ingreso de un estudiante a una carrera, podemos definir:

```
-- Ingreso es un sinonimo de tipo.
type Ingreso = Int
```

Los tipos algebraicos tienen constructores que llevan parámetros. Esos parámetros permiten agregar información, generando potencialmente infinitos valores dentro del tipo. Por ejemplo, si queremos modelar los roles de las diferentes personas que son miembros de la comunidad de nuestra facultad, podríamos definir los siguientes tipos:

```
-- Cargo y Area son tipos enumerados
data Cargo = Titular | Asociado | Adjunto | Asistente | Auxiliar
data Area = Administrativa | Ensenanza | Economica | Postgrado

-- Rol es un tipo algebraico
data Rol = Decanx                          -- constructor sin argumento
         | Docente Cargo                   -- constructor con un argumento
         | NoDocente Area                  -- constructor con un argumento
         | Estudiante Carrera Ingreso      -- constructor con dos argumentos
```

a) Implementá los tipos `Ingreso`, `Cargo`, `Area` y `Rol` como están definidos arriba.

b) ¿Cuál es el tipo del constructor `Docente`?

c) Programá la función `cuantos_doc :: [Rol] -> Cargo -> Int` que dada una lista de roles xs, y un cargo c, devuelve la cantidad de docentes incluidos en xs que poseen el cargo c.

d) ¿La función anterior usa `filter`? Si no es así, reprogramala para usarla.

e) ¿Qué modificaciones se deben realizar sobre el tipo `Rol` para poder representar el género del decano/a, sin agregar otro constructor?

f) ¿Podemos representar un alumno que está inscripto en dos carreras? Si no, ¿qué habría que modificar para conseguirlo? Luego programá la función `estudia :: Rol -> Carrera -> Bool` que dado un rol y una carrera, determina si se trata de un estudiante de dicha carrera.

3. Supongamos que queremos definir un tipo para representar a las personas que conforman la comunidad educativa de nuestra facultad. La información que nos interesa almacenar es: el apellido, el nombre, el número de documento, la fecha de nacimiento, y su rol dentro de la institución. Podemos definir entonces el siguiente tipo:

```
data Persona = Per String String Int Int Int Int Rol
```

donde los diferentes parámetros representan la información de interés, en el orden mencionado. La fecha de nacimiento es representada almacenando el día, el mes y el año de manera independiente, en formato numérico.

a) Implementá el tipo `Persona` como está definido mas arriba.

b) ¿Se puede utilizar el mismo identificador para el constructor `Per` y para el nombre del tipo `Persona`?

c) Programá las siguientes funciones:

1) `edad :: Persona -> (Int, Int, Int) -> Int` que dada una persona, y una fecha representada como *dia, mes y año*, calcula la edad de la persona en esa fecha.

2) `existe :: String -> [Persona] -> Bool`, que dado un apellido como primer parámetro, y una lista de personas, verifica si existe una persona con tal apellido.

3) `est_astronomia :: [Persona] -> [Persona]` que dada una lista de personas, devuelve la lista de aquellas que estudian astronomía.

4) `padron_nodocente :: [Persona] -> [(String, Int)]` que dada una lista de personas, devuelve el listado del personal nodocente, donde cada uno es representado con un par de la forma *(Apellido y Nombre, Nº de documento)*.

4. **Tipos recursivos.** Supongamos que queremos representa una *cola* de personas, como aquellas que forma fila para ser atendidas en el comedor universitario. Una persona llega y se coloca al final de la cola. El orden de atención respeta el orden de llegada, es decir, quien primero llega, es atendido primero. Podemos representar esta situación con el siguiente tipo:

```
data Cola = Vacia | Encolada Persona Cola
```

En esta definición, el tipo que estamos definiendo (`Cola`) aparece como un parámetro de uno de sus constructores; por ello se dice que el tipo es *recursivo*. Así una cola o bien está vacía, o bien contiene a una persona encolada, seguida del resto de la cola. Esto nos permite representar colas cuya longitud no conocemos *a priori* y que pueden ser arbitrariamente largas.

   a) Programá las siguientes funciones:

      1) `atender :: Cola -> Cola`, que elimina de la cola a la persona que está en la primer posición de una cola, por haber sido atendida.

      2) `encolar :: Persona -> Cola -> Cola`, que agrega una persona a una cola de personas, en la última posición.

      3) `busca :: Cola -> Cargo -> Persona`, que devuelve el primer docente dentro de la cola que tiene un cargo que se corresponde con el segundo parámetro.

   b) ¿A qué otro tipo se parece `Cola`? ¿Cómo implementarías las funciones anteriores utilizando este otro tipo?

5. **Tipos recursivos y polimórficos.** Consideremos los siguientes problemas:

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

   Ambos problemas se resuelven eficientemente, usando un diccionario o un padrón electoral. Estos almacenan la información *asociandola* a otra que se conoce; en el caso del padrón será el número de documento, 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 que asocie a un dato bien conocido (la clave) la información relevante (el dato).

   Una forma posible de representar esta situación es con el tipo de datos recursivo *lista de asociaciones* definido como:

```
data ListaAsoc a b = Vacia | Nodo a b (ListaAsoc a b)
```

   Los parámetro del tipo tipo `a` y `b` indican que se trata de un tipo *polimórfico*. Tanto `a` como `b` son variables de tipo, y se pueden *instanciar* con distintos tipos, por ejemplo:

```
type Diccionario = ListaAsoc String String
type Padron      = ListaAsoc Int    String
```

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

   b) Programá las siguientes funciones:

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

2) `la_concat :: ListaAsoc a b -> ListaAsoc a b -> ListaAsoc a b` que devuelve la concatenación de dos listas de asociación.

3) `la_pares :: ListaAsoc a b -> [(a, b)]` que transforma una lista de asociación en una lista de pares *clave-dato*.

4) `la_busca :: Eq a => ListaAsoc a b -> a -> Maybe b` que dada una lista y una clave devuelve el dato asociado, si es que existe. En caso contrario devuelve `Nothing`. Podes consultar la definición del tipo `Maybe` en Hoogle.

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

6) `la_borrar :: Eq a => a -> ListaAsoc a b -> ListaAsoc a b` que dada una clave a elimina la entrada en la lista.

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

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
```

   Programá las siguientes funciones:

   a) `a_long :: Integral b => Arbol a -> b` que dado un árbol devuelve la cantidad de datos almacenados.

   b) `a_hojas :: Integral b => Arbol a -> b` que dado un árbol devuelve la cantidad de hojas.

   c) `a_inc :: Num a => Arbol a -> Arbol a` que dado un árbol que contiene números, los incrementa en uno.

   d) `a_nombre :: Arbol Persona -> Arbol String` que dado un árbol de personas, devuelve un árbol con la misma estructura, conteniendo el nombre completo de tales personas.

   e) `a_map :: (a -> b) -> Arbol a -> Arbol b` que dada una función y un árbol, devuele el árbol con la misma estructura, que resulta de aplicar la función a cada uno de los elementos del árbol. Revisá la defincón de las dos funciones anteriores y reprogramalas usando `a_map`.

   f) `a_sum :: Num a => Arbol a -> a` que suma los elementos de un árbol de números.

g) `a_prod :: Num a => Arbol a -> a` que multiplica los elementos de un árbol de números