Lab 08 - Haskell Datatypes and Typeclasses

CS 1XA3

March 5^{th}, 2018

Algebriac Datatypes - Enumerated Types

- Think of Enumeration Types as a preferred method to represent different states with unique names rather than matching to corresponding Integers

```
data TypeName = Enum1 | Enum2 | Enum3
   deriving (Show,Eq,Enum)
```

- Note: the deriving statement allows automatic definitions for methods like $==$, $<$, $>$, etc
- Example: encoding traffic light states

```
data Lights = Green | Yellow | Red
   deriving (Show,Eq)
```

Algebriac Datatypes - Product Types

- A Product Type wraps values of other predefined types with a tag or value constructor to construct a new type

  ```
  data TypeName = Tag Type1 Type2
     deriving Show
  ```

- Note: Type1,Type2 are already defined types, whereas Tag is just a label you define
- Example: pairing an Int and Double

  ```
  data Pair = Pair Int Double
     deriving Show
  ```

Algebriac Datatypes - Sum / Union Types

- A Sum Type uses the | operator to define multiple tagged values (an Enum type is a Sum type without Products)

```
data TypeName = Tag1 Type1 | Tag2 Type2
   deriving Show
```

- Example: union of Int and Double

```
data IntOrDouble = I Int | D Double
   deriving Show
```

- Note: access data wrapped in a tag through pattern matching

```
convertToDouble :: IntOrDouble -> Double
convertToDouble iORd = case iORd of
                         I x -> fromIntegral x
                         D x -> x
```

Algebriac Datatypes - Parametric Types

- Algebriac Datatypes can be given type parameters (arguments just like functions)

```
data TypeName a b c = Tag1 a b c
   deriving Show
```

- Example: union of any two types

```
data UnionPair a b = Type1 a | Type2 b
   deriving Show
```

Algebraic Datatypes - Record Syntax

- When dealing with Product Types, it's helpful to write getter functions that retrieve specific values

```
data Student = Student String String Int
studName (Student name _ _) = name
studID (Student _ sID _)    = sID
studNum (Student _ _ num)   = num
```

- Record Syntax provides a cleaner way of defining getters

```
data Student = Student {studName :: String
                       ,studID   :: String
                       ,studNum  :: Int }
```

Algebriac Datatypes - Exercises

Try defining datatypes for representing:

- An x,y coordinate of Int's
- Different simple colors
- An RGB color
- A persons characteristics (name,age,hair color,etc)

TypeClasses - Type Signature Binding

- Recall: Type Signatures can be polymorphic (i.e except any type)

```haskell
-- works on any two types
pair :: a -> b -> (a,b)
pair x y = (x,y)
```

- If a function uses a method belonging to a TypeClass, you can still use polymorphism, but you have to constrain the type variable to be part of the type class

```haskell
-- (+) is part of the Num type class
addNums :: (Num a) => a -> a -> a
addNums x y = x + y
```

- This is vastly preferable to defining different addNums for each type Int,Integer,Float,Double,etc

TypeClasses - Overview

- A class definition provides an interface for a TypeClass. I.e provides the names and type signatures of it's member functions

```
class ClassName a where
  funcName1 :: a -> a
  funcName2 :: ...
```

- An instance definition provides an implementation of each function over a specific typeclass

```
instance ClassName Int where
  funcName1 x = ...
  funcName2 ...
```

TypeClasses - Case Study

- Scenario: imagine you are writing a painting app, you want to create a library of filters that are all built by mixing and inverting colors, thus the core of the library is built upon two

```
mixColors :: Color -> Color -> Color
invColor  :: Color -> Color
```

- Problem: you want your library to work with different color types (primary colors, RGB, RGBA, etc) with the possibility of adding more later. But you don't want to rewrite filters each time

- Solution: create a type class with instances for each color type

```
class MixableColors a where
  mixColors :: a -> a -> a
  invColor  :: a -> a -> a
instance MixableColors RGB where
  ...
```

TypeClasses - Example: The Num Class

- The Num typeclass implements simple arithmetic methods

```
class Num a where
  (+) :: a -> a -> a
  (-) :: a -> a -> a
  (*) :: a -> a -> a
  negate :: a -> a
  abs :: a -> a
  signum :: a -> a
  fromInteger :: Integer -> a
```

- The Prelude provides instances for all the primitive numeric types

```
instance Num Integer
instance Num Int
instance Num Float
instance Num Double
```

TypeClasses - Exercise

- Consider the following datatype for representing 2 dimensional vectors

```haskell
data Vec2 a = Vec2 a a
  deriving Show
```

- Note: Vector2 is parameterized. In order to implement an instance of Num for it we need to restrict it's parameter to also be a member of Num

```haskell
instance Num a => Num (Vec2 a) where
  (Vec2 x1 y1) + (Vec2 x2 y2) = Vec2 (x1+x2) (y1+y2)
  (Vec2 x1 y1) - (Vec2 x2 y2) = Vec2 (x1-x2) (y1-y2)
  ...
```

- Exercise: implement the rest of the instance