CS119 – Module 8: Data Types

Purpose: We are not restricted to data types already defined for us in Haskell, but can also define our own types to suit our needs.

Knowledge: This module will help you become familiar with the following content knowledge:

- Defining data types
- Writing functions to manipulate data types

Activity: With your group perform the following tasks and answer the questions. You will be reporting your answers back to the class in 30 minutes.

1. We can create our own data types by simply listing the possible values that the type may have. Consider this example contained in the `lab8` directory:

   ```
   data Thing = Shoe | Ship | SealingWax | Cabbage | King deriving Show
   ```

   This declares a new type called `Thing` with five possible values (Shoe, Ship, etc) which are the only values of type `Thing`. The `deriving Show` is a magical incantation which tells Haskell to automatically generate default code for printing values of type `Thing`.

2. We can write functions on type `Thing` by *pattern matching*:

   ```
   isSmall:: Thing -> Bool
   isSmall Shoe = True
   isSmall Ship = False
   isSmall SealingWax = True
   isSmall Cabbage = True
   isSmall King = False
   ```

   Predict the result of:
   `> isSmall Cabbage`

3. In a function, the cases are tried in order from top to bottom, so we could also make the definition of isSmall a bit shorter by using a default pattern `_`. You can read the `_` as meaning "everything else".

   ```
   isSmall2 :: Thing -> Bool
   isSmall2 Ship = False
   isSmall2 King = False
   isSmall2 _ = True
   ```

   Verify the result of:
   `>isSmall2 Cabbage`

4. We can combine previously defined data types into a new type as a tuple, which is just a parenthesized grouping. Consider the type `Quantity` which keeps track of how many of a `Thing` we have:

   ```
   type Quantity = (Int,Thing)
   ```

   We can keep determine if we have enough of our `Thing` with the function `haveEnough`. This function indicates that we always have enough kings but only have enough of everything else if we have at least 42 of them.

   ```
   haveEnough :: Quantity -> Bool
   haveEnough (n,King) = True
   haveEnough (n,_) = n >= 42
   ```

   Verify the results of:

   ```
   > haveEnough (0,King)
   > haveEnough (3,Shoe)
   > haveEnough (42,Shoe)
   ```

   Modify this function so that we have enough shoes only if we have an even number of them.

5. Data type values need not be a simple list like we saw above. The types may also include arguments. Consider the following data type:

   ```
   data FailableDouble = Failure | OK Double deriving Show
   ```

   This says that the `FailableDouble` type has two values. The second case, OK, takes an argument of type Double. So OK by itself is not a value of type `FailableDouble`; we need to give it a Double. For example, OK 3.4 is a value of type `FailableDouble`.

   Here's one way we might use our new FailableDouble type:

   ```
   safeDiv:: Double -> Double -> FailableDouble
   safeDiv _ 0 = Failure
   safeDiv x y = OK (x / y)
   ```

   Try

   ```
   > safeDiv 2 0
   > safeDiv 3 4
   ```

   What happened and why?

6. Complete the function `failureToZero` which converts a `FailableDouble` to a regular double by changing the Failure values to zero but leaving the OK values to be the double value that has been deemed OK.

```
failureToZero :: FailableDouble -> Double
failureToZero __________ = 0
failureToZero (OK d) = __________
```

You can test your function with:

```
>failureToZero (safeDiv 2 0)
>failureToZero (safeDiv 3 4)
```

Complete the following assignments to be submitted for grading. Each should be done individually but you can consult with a classmate to discuss your strategies or if you get an error message that you do not understand.
Write all of your functions in the file `Shape.hs` contained in the `lab8` directory. You might want to start by creating a couple of shapes and computing their areas.

Assignment 1:
Add another case to the `Shape` data type which will create a Right Triangle by providing the two sides that meet at the right angle. Then modify the `area` function so that it computes the area of this new shape.

Criteria for Success: Calculate the area of a couple of right triangles and verify the results.

Assignment 2:
Write a function `scale :: Float -> Shape -> Shape`

The expression `scale factor s` will return a new shape that is scaled by the given factor.

Criteria for Success: Scale shapes of all three kinds and verify the results.

Assignment 3:
Define another data type `LocatableShape` that contains x and y coordinate values for the center for the center of the shape, as well as a given shape. You can do this by creating a tuple containing the three components.

Then write a function:
`translate :: (Int,Int) -> LocatableShape- > LocatableShape`
which will move the shape. For example, if the first parameter is (10,10), this would increase both the x and y coordinates of the center of the shape by 10.

Criteria for Success: Translate a LocatableShape value and verify the the location has changed.

Submit your `Shape.hs` file in Canvas for grading.