CMPT 383 Comparative Programming Languages

Homework 4

This homework is due by 11:59pm PT on Wednesday Mar 2, 2022. No late submission is accepted. Please save your Haskell code in a single file called `h4_firstname_lastname.hs` (`h` in lower case, firstname and lastname replaced with your first and last name) and submit it to Canvas.

Requirements of this homework:

- Write type signatures for all functions using the :: operator, including functions required by certain type classes. If your compiler does not support type signatures for functions in an instance, you can write the type signatures in comments.
- Do not use the `if-then-else` expression unless specified in the question.

1. (20 points) Consider the `ErrJst` type constructor defined as follows

```
data ErrJst e j = Err e | Jst j deriving (Show)
```

`ErrJst` is a generalization of `Maybe`, because it can return an error message using the `Err` data constructor when computation fails (as opposed to `Nothing` for `Maybe`). If the computation succeeds, it returns the result using `Jst`. In this question, you need to make `ErrJst e` a functor.

Sample input and output:

```
ghci> fmap (+1) (Jst 1)
Jst 2
ghci> fmap (+1) (Err 1)
Err 1
```

2. (20 points) Consider the `ErrJst` type constructor in Question 1. Make `ErrJst e` an applicative functor.

Sample input and output:

```
ghci> pure (+) <*> (Err 1) <*> (Jst 1)
Err 1
ghci> pure (+) <*> (Jst 2) <*> (Jst 1)
Jst 3
```

3. (20 points) Consider the `ErrJst` type constructor in Question 1 again. Make `ErrJst e` a monad.

Sample input and output:

```
ghci> Jst 1 >>= \x -> return (x*2)
Jst 2
ghci> Err 1 >>= \x -> return (x*2)
Err 1
```

4. (20 points) Write a function called `join` that can join "a monad of monadic values" into a monadic value. For example, it can join a list of lists into a list. It can also join a value of type `Maybe (Maybe Int)` into a value of type `Maybe Int`.

Sample input and output:

```
ghci> join [[1, 2], [3, 4]]
[1,2,3,4]
ghci> join (Just (Just 1))
Just 1
```

5. (20 points) Consider the following definition of binary trees

```
data LTree a = Leaf a | LNode (LTree a) (LTree a) deriving (Show)
```

Make `LTree` an instance of `Foldable`. Whenever you need to process a node with two sub-trees, make sure you first process the left sub-tree and then process the right sub-tree.

Sample input and output:

```
ghci> let t = (LNode (LNode (Leaf 2) (Leaf 3)) (Leaf 4))
ghci> foldl (+) 0 t
9
ghci> foldr (*) 1 t
24
```