Advanced Programming (I00032) 2016
Functors, Applicative, and Monad

Assignment 4

Version issues

In this exercise it is assumed that you use the recommended Clean 2.4 version. Newer, iTask, versions of Clean have classes like `Functor` in the standard libraries. You can solve name classes by carefully import only the modules really needed, renaming our own classes and functions, or using the definitions from the library.

We use the manual version of generics instead of the native generics since there are compiler versions that cause nasty problems for the combination of generics and monads.

Goals of this exercise

After making this exercise you can use and implement Functors, Applicative and Monad.

1 Appetizer

On Blackboard you will find the code to read the data for a student record from the console. The version using monads is `f2 :: IO Student`. Write a function `f3 :: IO Student` that reads the data using applicative instead of monad.

2 Serialization using Functor, Applicative and Monad

In the previous exercise we implemented serialization using kind indexed classes. The solution was somewhat clumsy due to the explicit handling of the list of strings. In this exercise we improve this by hiding this list in a state. We require that we can `read` and `write` from the same state. This implies that we can write some value to the state and read exactly the same value back again.

Serialized

Define a type `Serialized` to hold the serialized version of objects. Try to make an implementation were reading and writing can be done in amortized $O(1)$ time. The given code contains a dummy definition to make it compilable.

State access for primitive types

We will use a state monad that accounts for failures in this exercise:

```
:: State s a = S (s → (Maybe a,s))
```

As basic access functions to this state we have

```
:: Serialize a :== State Serialized a

wrt :: a → Serialize String | toString a
rd :: Serialize String
```

The function `wrt` writes primitive types like integers and Booleand to the state. The function `rd` yields the next string and removes it from the state (if it is there). Improved the definitions given in the template.

Monad for State

Define proper instances of the classes `Functor`, `Applicative Monad`, and `OrMonad` for `State` by improving the dummy definitions from the template.

Matching Strings

Improve the functions `match` and `pred` such that they succeed if the first string matches the given value or predicate. Use only the access functions above for the state, do not use any explicit control to the state.

Serialization of Generic Types

Use the tools developed above to improve the instance of the serialization classes. It is never necessary to access the state directly, everything can and should be done by the Functor, Applicative and Monad operations.

Tests

Check that all tests succeed. When there are failing test you might want to change the `test` function to obtain more information.

Deadline

The deadline for this exercise is September 27 2016, 13:30h (just before the next lecture). Add the output of your programs as a comment.