Programming in Haskell – Homework Assignment 6

UNIZG FER, 2015/2016

Handed out: February 8, 2016. Due: ∞, 2016 at 17:00

Note: Define each function with the exact name and the type specified. You can (and in most cases you should) define each function using a number of simpler functions. Provide a type signature above each function definition and comment the function above the type signature. Unless said otherwise, a function may not cause runtime errors and must be defined for all of its input values. Use the `error` function for cases in which a function should terminate with an error message. Problems marked with a star ($\star$) are optional.

Each problem is worth a certain number of points. The points are given at the beginning of each problem or subproblem (if they are scored independently). These points are scaled, together with a score for the in-class exercises, if any, to 10. Problems marked with a star ($\star$) are scored on top of the mandatory problems, before scaling. The score is capped at 10, but this allows for a perfect score even with some problems remaining unsolved.

1. *(2 pt)* Now that you have learned more about input and output, let's do more unix like utilities.

   (a) Define a simple utility for measuring the time an action spent executing. One way to get the current time is to use `Data.Time.Clock`.

   ```
   time ::  IO () -> IO Double
   time (putStrLn "printing this line takes time") ⇒ return 0.00204s
   time (mapM_ print [1..100000]) ⇒ return 4.608441s
   ```

   (b) Define a simple grep-like utility. Read a file and output only the lines containing some given string. (Hint: use `Data.List.isInfixOf`.)

   ```
   grep :: String -> FilePath -> IO ()

   $ echo \some\ntext\nhere\nokay\nnot okay" > lines.txt

   ghci> grep "o" "lines.txt"
   some
   okay
   not okay

   ghci> grep "ok" "lines.txt"
   okay
   not okay
   ```

(c) Use `System.Environment.getArgs` to read the command-line arguments. Pass them into the previously-defined grep action. You can use `System.Environment.withArgs` to simulate command-line arguments within the interpreter.

```
grepWithArgs :: IO ()

ghci> withArgs ["o", "lines.txt"] grepWithArgs
some
okay
not okay
```

(d) Lists of characters are not a very memory efficient representation of text. For most purposes, a better way to store text is by way of a mostly contiguous array. Reimplement `grep` using the `Text` type from the `Data.Text.Lazy` and `Data.Text.Lazy.IO` modules. Where needed, use `Data.Text.Lazy.pack` to convert a `String` into a `Text` value. (You may need to install the `text` package to use the module.)

```
grepText :: Data.Text.Lazy.Text -> FilePath -> IO ()

ghci> grepText (Data.Text.Lazy.pack "okay") "lines.txt"
okay
not okay
```

Use the previously defined timing action to see which of the two greps is faster for large files.

2. *(2 pt)* Create an API for a simple text-based database (saved in a textual file). All database fields are of type `String`, while each row is a `[String]`. A database is abstracted as

```
type Table = (FilePath,[String])
```

where `FilePath` is filepath of the database file and `[String]` are the column labels. Feel free to define `Table` in some other way if it suits you better.

Example of a database file:

```
name surname age
John James 23
Robert Robertson 48
Linda Lindson 31
```

(a) Define a function that takes the table name and the column labels (which must be unique) and creates a file named `<tableName> ++ ".tbl"`. You may assume that column names don't contain spaces.

```
dbCreateTable :: String -> [String] -> IO Table
```

(b) Define a function that deletes the file holding the table.

```
dbDeleteTable :: Table -> IO ()
```

(c) Define a function that inserts a row into the table. Check whether the input is of correct length. If yes, write it to the file, otherwise throw an exception.

```
dbInsert :: Table -> [String] -> IO ()
```

(d) Define a select operation that takes a predicate and returns all rows that satisfy it.

```
dbSelect :: Table -> ([String] -> Bool) -> IO [[String]]
```

(e) Define a delete operation that deletes all rows that satisfy the given predicate.

```
dbDelete :: Table -> ([String] -> Bool) -> IO ()
```

(f) Define an update operation that updates all rows that satisfy the given predicate using the given update function.

```
dbUpdate ::
  Table -> ([String] -> Bool) -> ([String] -> [String]) -> IO ()
```

(g) Define a print function that prints the table to standard output. (Hint: Use `dbSelect`.)

```
dbPrintTable :: Table -> IO ()
```

3. *(2 pt)* In the class exercises you have implemented a simple, persistent dictionary. Let's extend this program and make it more robust.

   Your task is to detect a modificition of the dictionary file outside of the program and reload it for instant use (take a look at `System.Directory.getModificationTime`). You don't need to check the file asynchronously. It is enough to perform the check and reload the file if needed before accessing the in-memory translation dictionary.

   There are a couple of potential errors that need to be handled. As the file may get deleted you need to catch the appropriate exception and print it on screen. Another error that might happen is when parsing the file if it contains invalid syntax. This also needs to be handled and printed to the user. In both cases your program must continue working. (See: Catching exceptions in `Control.Exception` module.)

   Example usage:

```
interactiveDict :: FilePath -> IO ()

ghci> interactiveDict "dict.txt"
```

```
Input a word:
repair
Input a translation:
reparieren
Input a word:
repair
Translation: reparieren
Input a word:
```

Add a translation of 'mother' to 'mutter' by hand to dict.txt before writing the next word

```
mother
Translation:
mutter
Input a word:
```

Translation for 'mother' was freshly read from the file and existing dictionary updated.

Make an error in the file before writing the next word.

```
repair
Error reloading the dictionary: Invalid syntax.
Translation: reparieren
Input a word:
```

The program reported an error but continued working, produced a translation and asked for the next word.

The program exits when it encounters the end of file, which can be produced by pressing Ctrl-D, on Windows Ctrl-Z might work better. When the program exits it needs to store the full dictionary to the given file.

4. (⋆) *(4 pt)* You have seen how a Monad instance can be used to handle things that can fail. Another use case for a Monad is dealing with non-determinism. The simplest way to model non-deterministic computation is using a list. Think of the list as holding all possible results of some computation. When you use the well known `map` function, the function you are applying produces the outputs for all the possible inputs.

   But before we get to a Monad we have to start from the top. A Monad is just a type class, for example, like Ord. When you want to implement the instance of a type class, you need to implement its functions. You may recall that for Ord, there is another requirement, you have to implement the instance for Eq. Monad also has a similiar requirement, you have to implement instances of 2 type classes: Functor and Applicative. Don't let the names scare you, we will go through each one, step by step.

   Define your list type like so:

```
data MyList a = Cons a (MyList a) | Nil deriving Show
```

(a) Make your list an instance of Functor. The most common Functor instances are containers. A list is a natural fit. You could also make your Tree and Queue types from previous homeworks an instances of Functor. The type class is defined this way:

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

The idea is to lift a regular function to work inside a context, here being a list of elements.

```
instance Functor MyList where
    fmap = ...
```

```
fmap id $ Cons 'a' (Cons 'b' (Cons 'c' Nil))
⇒ Cons 'a' (Cons 'b' (Cons 'c' Nil))
fmap (+1) $ Cons 1 (Cons 2 (Cons 3 Nil))
⇒ Cons 2 (Cons 3 (Cons 4 Nil))
```

For easier use, `fmap` has an infix operator defined: `<$>`. You can think of it as regular application, `$`, but inside some context.

```
('A':) <$> Cons "gda" (Cons "rc" Nil)
⇒ Cons "Agda" (Cons "Arc" Nil)
```

(b) Functor instance is useful when you want to use a regular function on your list elements. But what if you have a list of functions and want to apply the functions to elements of another list? This is where the Applicative type class comes in.

```
class Functor f => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

As you can see from the definition, it requires the type to be an instance of Functor. First function takes a regular value and wraps it inside your list. The function `<*>` can be read as "apply" or "app", and it used to apply a wrapped function to a wrapped value.

```
instance Applicative MyList where
    pure = ...
    (<*>) = ...
```

```
pure 3 ::  MyList Int ⇒ Cons 3 Nil
Cons (+1) (Cons (*4) Nil) <*> Cons 1 (Cons 2 Nil)
⇒ Cons 2 (Cons 3 (Cons 4 (Cons 8 Nil)))
```

(c) At last, it is time to implement Monad instance. This is easy, as the Applicative does most of the work.

```
class Applicative m => Monad m where
  return :: a -> m a
  return = pure
  (>>=) :: m a -> (a -> m b) -> m b
  (>>) :: m a -> m b -> m b
```

You can notice that `return` has the same type signature as `pure`. Indeed, they are the same. The operator `(>>)` is defined in terms of the *bind* function, `(>>=)`. This means that you only need to define the *bind* function. A subset of type class functions that need to be implemented is called *minimal complete definition*. The Monad type class provides multiple functions, but its minimal complete definition is `(>>=)`.

```
instance Monad MyList where
    (>>=) = ...
```

```
fun ::  Int -> MyList Int
fun x = Cons (x + 1) (Cons (x * 2) Nil)

Cons 1 (Cons 3 Nil) >>= fun ⇒ Cons 2 (Cons 2 (Cons 4 (Cons 6 Nil)))
Cons 1 (Cons 2 Nil) >> return 3 ⇒ Cons 3 (Cons 3 Nil)
```

You can read more about type classes on the Typeclassopedia.