Exercise (Week 7)

DUE: Tue 23 July 2019 14:00:00

Controlling Effects

Basic I/O (2 Marks)

CSE | Stack

Download the exercise tarball and extract it to a directory on your local machine. This tarball contains a file, called `Ex05.hs`, wherein you will do all of your programming.

To test your code, run the following shell commands to open a GHCi session:

```
$ stack repl
Configuring GHCi with the following packages: Ex05
Using main module: 1. Package 'Ex05' component exe:Ex05 ...
GHCi, version 8.2.2: http://www.haskell.org/ghc/  :? for help
[1 of 2] Compiling Ex05             (Ex05.hs, interpreted)
[2 of 2] Compiling Main             (Main.hs, interpreted)
Ok, two modules loaded.
*Main Ex05> capitalise "input.txt" "output.txt"
...
```

Calling `IO` actions in `GHCi` as above will execute them, including their side effects.

Note that you will only need to submit `Ex05.hs`, so only make changes to that file.

First, there is the I/O function `capitalise`, of the following type:

```haskell
capitalise :: FilePath -> FilePath -> IO ()
```

This function reads the text file determined by the first argument and capitalises each character while writing the result to the file given in the second argument. The program must read all available text from the first file. You may find the functions in `System.IO` and `Data.Char` useful.

More Intricate I/O (3 Marks)

Write an IO action called `sumFile`, of the following type:

```haskell
sumFile :: IO ()
```

This should retrieve two filenames which are given as a command line arguments, and read the file specified in the first command line argument. This file will contain a list of integers, one per line. The `sumFile` action should write the sum of these numbers into the second file given as a command line argument. If you like an additional challenge, do not use any explicit recursion in the program, just the list operators etc. from the libraries (Note: There are no extra marks for this, but it is a nice puzzle).

Some hints:

- To get the command line arguments, look at `System.Environment`
- To read files, you can just use the functions `readFile` for this simple exercise; instead of fussing around with handles.
- The standard `Prelude` (which is the module implicitly imported into any Haskell programs), has a lot of functions for list manipulation.

To run your program, you can use a helper module, `Main.hs`, provided for you in the code bundle.

CSE | **Stack**

In a `3141` subshell, type `cabal build` to build the executable, which you should be able to invoke by typing `./dist/build/Ex05/Ex05 <command line args>`. For example:

```
$ 3141
newclass starting new subshell for class COMP3141...

$ cabal build
Resolving dependencies...
Configuring Ex05-1.0...
Preprocessing executable 'Ex05' for Ex05-1.0..
Building executable 'Ex05' for Ex05-1.0..
[1 of 2] Compiling Ex05             ( Ex05.hs, ...)
[2 of 2] Compiling Main             ( Main.hs, ...)
Linking dist/build/Ex05/Ex05 ...

$ ./dist/build/Ex05/Ex05 "input.txt" "output.txt"
```

State and Testing IO (4 Marks)

There is a well-known guessing game where, given a number of guesses, the player attempts to guess a number that has been randomly chosen within a certain range. If the player guesses incorrectly, the only information the player receives is whether the target number is lower or higher than the player's guess.

We will model this using Haskell, defining a `Player` as a type that consists of a function to make a guess, and a function that can act on the response given by the game to the player's guess:

```
data Player m = Player { guess :: m Int
                       , wrong :: Answer -> m ()
                       }
data Answer = Lower | Higher
```

We allow a `Player` to perform arbitrary effects in order to make a guess or to handle responses. For example, the `human` player is defined using `IO`, so as to ask the user for a number, and reports feedback to the user by printing a message to the terminal:

```
human :: Player IO
human = Player { guess = guess, wrong = wrong }
  where
    guess = do
      putStrLn "Enter a number (1-100):"
      x <- getLine
      case readMaybe x of
        Nothing -> guess
```

```
        Just i  -> pure i

    wrong Lower  = putStrLn "Lower!"
    wrong Higher = putStrLn "Higher!"
```

The guessing game itself is defined *generically* for any monad `m`, so long as we can provide a `Player` that acts in that monad. You can play the guessing game yourself in the `IO` monad with the `human` player by calling `play`:

```
-- x is the number we're trying to guess
-- n is the number of guesses we get
-- p is the player
-- Returns whether or not the player managed to guess correctly
-- in the time limit
guessingGame :: (Monad m) => Int -> Int -> Player m -> m Bool
guessingGame x n p = go n
  where
   go 0 = pure False
   go n = do
     x' <- guess p
     case compare x x' of
       LT -> wrong p Lower  >> go (n-1)
       GT -> wrong p Higher >> go (n-1)
       EQ -> pure True

play :: IO ()
play = do
  x <- randomRIO (1,100)
  b <- guessingGame x 5 human
  putStrLn (if b then "You got it!" else "You ran out of guesses!")
```

Your task is to define a new player, `ai`, which plays the game automatically, by acting in the `State (Int, Int)` monad instead of the `IO` monad:

```
ai :: Player (State (Int,Int))
```

We would like it to satisfy some properties. Firstly, we would like the `ai` player never to repeat a guess, that is, the `ai` player should guess the number in at most `n` guesses if the number lies between 1 and `n`:

```
prop_no_repeat (Positive n)
  = forAll (choose (1,n)) $ \x -> evalState (guessingGame x n ai) (1,n)
```

We would further like it to play *optimally* – i.e. making as few guesses as possible. By bisecting the range each time, we can make a logarithmic number of guesses, proportional to the size of the initial range:

```
prop_optimality (Positive n)
    = forAll (choose (1,n)) $ \x ->
        evalState (guessingGame x (bound n) ai) (1,n)
  where bound n = ceiling (logBase 2 (fromIntegral n)) + 1
```

Submission instructions

You can submit your exercise by typing:

```
$ give cs3141 Ex05 Ex05.hs
```

on a CSE terminal, or by using the `give` web interface. Your file *must* be named `Ex05.hs` (case-sensitive!). A dry-run test will partially autotest your solution at submission time. To get full marks, you will need to perform further testing yourself.