EBERHARD KARLS UNIVERSITÄT TÜBINGEN

Functional Programming
Wintersemester 2021/2022
Prof. Dr. T. Grust, D. Hirn

Assignment 11 (28.01.2022)

Handin until: Friday, 04.02.2022, 00:00

Exercise 1: Guessing Numbers **[8 Points]**

With monads, we finally have the tools to express and combine computations which perform I/O. We will implement a simple interactive application. The interaction is defined by the following rules:

1. The computer picks a random number between 1 and 2^{10}.
2. The *player* has 13 attempts to guess the right number.
3. After every wrong guess, the computer tells whether the guessed number was too small or too large.
4. If the player can not guess the correct number in 13 attempts, he loses and the computer makes fun of him/her.
5. If the player manages to guess the correct number, he/she wins the game.

Open `Guess.hs` and complete the missing function definitions to implement the game according to the rules described above:

- Function `prompt :: IO Int` – Ask the player for a guess. Parse his input string (use function `Text.Read.readMaybe`). If the input is invalid, ask again until a valid input is provided. Otherwise return the guessed number. A basic example how to use I/O combinators `getLine` and `putStrLn` is given below.
- Function `hint :: Int -> Int -> IO ()` – Given a target number and a guess, print a hint to the player whether his guess is to small, to large or correct.
- Function `gameLoop :: Guess Bool` – Interact with the player and finally return whether the player won or not. Ask the player for a guess. If he is correct or is out of attempts, end the game. Otherwise enter the loop again.
- Function `main :: IO ()` – Pick a random number (use function `System.Random.randomRIO`[1]) and enter the game loop with an initial `GameState`.

Example interactions in GHCi:

```
*Guess> main
Guess a number:
invalid
Guess a number:
512
Your guess is too large!
Guess a number:
42
You won within 2 attempts.

*Guess> main
Guess a number:
1
Your guess is too small!
Guess a number:
2
Your guess is too small!
[...]
Guess a number:
14
Your guess is too small!
You ran out of attempts.
```

Example how to use basic I/O combinators, namely `getLine` and `putStrLn` (you can also find it in `io-example.hs`):

```haskell
check :: String -> String -> IO Bool
check q a = putStrLn (q ++ "Wait...") >>
            return (a == "42")

main :: IO ()
main =
  putStrLn "Please give a question:" >>
  getLine >>= \q ->
  putStrLn "And an answer:" >>
  getLine >>= \a ->
  if last q == '?'
    then check q a >>= \res ->
         putStrLn $ "Your answer is " ++ show res
    else putStrLn "Your question should end with an ?. Retry!" >>
         main
```

[1] To import `System.Random` you may have to install package `random` first: `cabal install --lib random` or `stack install random`

Exercise 2: Monadic Evaluation of Expressions **[12 Points]**

Consider the following (incomplete) type of expressions over integers and booleans. The complete code can be found in module `Eval`.

```haskell
data IntOp = Mul | Div | Eq

data Val = IntV Int | BoolV Bool

data BoolOp = And | Or

data Expr = Lit Val
          | AppInt IntOp Expr Expr
          | AppBool BoolOp Expr Expr
          | Var String
          | Let (String, Expr) Expr
```

The language supports *let*-bindings (`Let`) and variable references (`Var`). In a term `Let [("x", x)] e`, expression *e* is evaluated in an **environment** in which the name `"x"` is bound to the result of expression *x*. The following term represents the computation 42^2.

```haskell
Let ("x", Lit $ IntV 42) (AppInt Mul (Var "x") (Var "x"))
```

During evaluation of terms, a number of things might go wrong, causing the evaluation to **fail**. We might encounter division by zero, ill-typed expressions (*e.g.* `Mul` applied to booleans), or references to unbound variables.
We could define an evaluation function which passes an environment explicitly and wraps its result in `Maybe` to account for failing evaluations. However, doing all this manually quickly proves tedious and verbose. Instead, we will define an *evaluation monad* which provides for both requirements (passing an environment, and possibility to fail).

An environment `type Env = [(String, Val)]` is a mapping from variable names to values. An evaluation of expressions (`Expr`) in such an environment can be modeled by a function of type `Env -> Val`, *i.e.* a function that receives an environment and returns a value, which is based on the variable bindings of the environment. Since the evaluation in the environment might fail, we enhance the return type to `Env -> Maybe Val`. Extending the idea to any evaluations in the context of an environment (not only of return type `Val`), we define the data type `Eval` as follows:

```haskell
-- | Evaluation in an environment.
newtype Eval a = E { runEval :: Env -> Maybe a }
```

1. Before approaching the monadic aspect, we will define a few helper functions for expression evaluation.

   (a) Define a function `lookupEnv :: String -> Eval Val` that looks up a variable name in the environment.

   Example:

   ```haskell
   > runEval (lookupEnv "x") [("x", IntV 1)]
   Just (IntV 1)
   ```

   (b) Define a function `local :: (Env -> Env) -> Eval Val -> Eval Val` that evaluates an expression in a modified environment. `local f m` modifies the environment using function *f* and runs the expression evaluation *m* in the modified environment.

   Example:

   ```haskell
   > runEval (local (const [("x", IntV 2)]) $ lookupEnv "x") [("x", IntV 1)]
   Just (IntV 2)
   ```

2. Now the interesting part: Make the type `Eval` an instance of `Monad`, *i.e.*

```haskell
instance Monad Eval where
    return = ...
    (>>=)  = ...
```

Implementing the (>>=) method is not trivial. You should expect to need some time to figure out how things need to be combined. We strongly recommend that you use the types of `return` and >>= in the `Eval` monad to let you guide you in this part of the assignment.

Example:

```haskell
> runEval (lookupEnv "x") [("x", IntV 42)] >>= \x -> runEval (lookupEnv "y") [("y", x)]
Just (IntV 42)
```

3. We now have everything in place to actually evaluate expressions in the `Eval` monad. Define the core evaluation function `eval :: Expr -> Eval Val`. Also, define a helper function `evalTop :: Expr -> Maybe Val` that evaluates an expression in an empty initial environment using `eval`.

Example:

```haskell
> evalTop $ Let ("x", Lit (IntV 4)) (AppInt Mul (Var "x") (Var "x"))
Just (IntV 16)
```

Hint: Use `local` to modify the environment when you evaluate a `Let` binding.