EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

Functional Programming
Summer Term 2025
Prof. Torsten Grust, Björn Bamberg
WSI — Database Systems Research Group

Assignment 9

Hand in this assignment until —, —, — at the latest.

🤔 Running out of ideas?
Are you hitting a roadblock? Are some of the exercises unclear? Do you just need that one hint to get the ball rolling? Refer to the `#forum` 🌐 channel on our Discord server—maybe you'll find just the help you need.

Ⓔxam-style Exercises
Exercises marked with Ⓔ are similar in style to those you will find in the exam. You can use these to hone your expectations and gauge your skills.

Task 1: Monoids Ⓔ

Let's talk about *monoids* and trees, more specifically the following type of rose trees with labels of type `a`:

```
1 data Tree a = Node a [Tree a]
```

Please submit a file called `MyMonoids.hs`. Import the module `Data.Monoid` and have a look at the documentation for this module[1].

A Write a function `sumTree :: Num a => Tree a -> a` that computes the *sum* of all node labels in a tree.

B Write a function `treeLabels :: Tree a -> [a]` that computes the *list* of all node labels in a tree.

C We continue our hunt for common patterns in computations that we might abstract over. `sumTree` and `treeLabels` are suspiciously similar: The label of the current node is combined with the results for all subtrees. We abstract over this pattern in a function

```
foldTree :: Monoid m => (a -> m) -> Tree a -> m
```

Given a function that maps a node label to some element of monoid `m`, `foldTree` combines the monoidal result for the node label and the results for all subtrees.

Implement `foldTree`.

D First, use `foldTree` to implement a function `treeLabels' :: Tree a -> [a]` that behaves like `treeLabels`.

Then, do the same for `sumTree`. Remember that there is no single `Monoid` instance for numeric types. Instead, we have `newtype` wrappers `Sum` and `Product`, whose `Monoid` instances implement the additive and multiplicative monoids for numeric type `a`, respectively.

E Finally, implement functions

```
allNodes :: (a -> Bool) -> Tree a -> Bool
someNode :: (a -> Bool) -> Tree a -> Bool
```

that check whether *all* or *some* node labels in a tree satisfy a predicate. Again, we do not have a single `Monoid` instance for `Bool`, but wrappers `Any` and `All` that implement monoids with different behaviors.

Task 2: Guessing Numbers

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.

[1] http://hackage.haskell.org/package/base-4.12.0.0/docs/Data-Monoid.html 🌐

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 too small, too 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`[2]) 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
```

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