Functional Programming

https://proglang.informatik.uni-freiburg.de/teaching/functional-programming/2021/

Exercise Sheet 7 – Monads, Monad transformers

2021-07-06

In this exercise, we will be using various other monads and monads transformers available in the mtl library:

https://hackage.haskell.org/package/mtl

Here is a handy chart[1] of some of them:

Standard Monad	Transformer Version	Original Type	Combined Type
`Maybe`	`MaybeT`	`Maybe a`	`m (Maybe a)`
`State`	`StateT`	`s -> (a,s)`	`s -> m (a,s)`
`Reader`	`ReaderT`	`r -> a`	`r -> m a`
`Writer`	`WriterT`	`(a,w)`	`m (a,w)`
`Error`	`ErrorT`	`Either e a`	`m (Either e a)`
`Cont`	`ContT`	`(a -> r) -> r`	`(a -> m r) -> m r`

Table 1: Some monads and their transformers

You will need to import the following modules:

```
import Control.Monad.Trans
import Control.Monad.Trans.Maybe
import Control.Monad.Trans.Reader
import Control.Monad.Trans.Writer.Strict
```

Exercise 1 (Computations with protected data)
We want to write a program that manipulates protected data. This means that, in various point of our program, we might ask the user for their password in order to the data. Of course, access to the protected data can fail if the password is wrong. For this simple example, we will simply assume the existence of the following functions.[2]

```
data ProtectedData a = ProtectedData String a

accessData :: String -> ProtectedData a -> Maybe a
accessData s (ProtectedData pass v) =
  if s == pass then Just v else Nothing
```

1. Your task is to implement the `Protected` monad that will ask the password to the user when trying to access the data, and fail in case of error. The data is stored in a reader monad.

   ```
   type Protected s a = MaybeT (Reader (ProtectedData s)) a

   run :: ProtectedData s -> Protected s a -> Maybe a

   access :: String -> Protected a a
   ```

[1] Taken from the Haskell wiki.
[2] Warning, passwords should never be stored! Only store salted password: https://en.wikipedia.org/wiki/Salt_(cryptography).

2. Instead of asking the programmer to enter the password, it would be better to ask the user directly. Improve your `Protected` monad like so.

```
type Protected s a = MaybeT (ReaderT (ProtectedData s) IO) a

run :: ProtectedData s -> Protected s a -> IO (Maybe a)

access :: Protected a a
```

Exercise 2 (Structured Logging)

The `Writer` monad (and its transformer) allows to emit some output that will be returned when the computation run. The type `w` in the writer is expected to be a monoid, so that messages in different part of the program can be combined.

Most of the time, logs are just lists of strings. Structured logging allows to introduce sections. We will consider the following types:

```
data Item = Msg String
          | Section String [Item]
            deriving (Show,Eq)
type Log = [Item]

type Logging a = Writer Log a
```

1. Write the following definitions:

```
-- ‘log s‘ logs the messages ‘s‘.
log :: Show t => t -> Logging ()

-- ‘with_section s m‘ executes m and add its log in a section titled ‘s‘.
with_section :: String -> Logging a -> Logging a

runLogging :: Logging a -> (a, Log)
```

   Hint: you might use `tell` and `pass`.

2. Sometimes, it is useful to have *timestamps* in logs. For this purpose, we can use the following haskell functions:

```
import Data.Time.Clock.POSIX
getPOSIXTime :: IO POSIXTime
```

   Extend the `Logging` monad to be able to call IO actions. Do you need to change the type of `runLogging`?

   Extend `Item`, `log` and `with_section` to always register timestamps.

   In the case of `with_section`, you should register two timestamps: one before and one after. Test your implementation to ensure that you record time correctly, by using `System.Posix.Unistd.sleep` for example.