Курс: Функциональное программирование
Практика 11. Монада Error, трансформеры монад

Разминка

► Устно вычислите значения выражений и проверьте результат в GHCi:

```
do {x <- Just 5; guard (x>10)}

do {x <- Just 5; guard (x<10)}

msum [Just 1,Just 2,Just 3]

msum [Nothing,Nothing,Just 1,Just 2]

msum [[1,2,3],[10,20]]

mfilter (<10) $ Just 12

mfilter (<10) [1,3..]
```

Класс типов MonadPlus

► Какие из законов класса типов `MonadPlus` выполняются для списка? типа `Maybe`? Приведите доказательство или опровергающий пример.

Монада Error

► (1 балл) Введём тип данных для представления ошибки обращения к списку по недопустимому индексу.

```
data ListIndexError =
  ErrTooLargeIndex Int
  | ErrNegativeIndex
  | OtherErr String
  deriving (Eq, Show)
```

Реализуйте оператор `(!!!)` доступа к элементам массива по индексу, отличающийся от стандартного `(!!)` поведением в исключительных ситуациях. В этих ситуациях он должен выбрасывать подходящее исключение типа `ListIndexError`.

```
infixl 9 !!!
(!!!) :: (MonadError ListIndexError m) => [a] -> Int -> m a
xs !!! n = undefined
```

Ожидаемое поведение:

```
GHCi> let Right x = [1,2,3] !!! 0 in x
1
GHCi> let Left e = [1,2,3] !!! 42 in e
ErrTooLargeIndex 42
GHCi> let Left e = [1,2,3] !!! (-10) in e
ErrNegativeIndex
```

► (2 балла) Реализуйте собственную монаду обработки ошибок (взамен `Either e`) со строковым типом информации об ошибке на основе типа данных

```
data Excep a =  Err String | Ok a
  deriving (Eq, Show)
```

Сделайте этот тип представителем классов типов `Monad`, `MonadPlus` и `MonadError String` (в последнем случае потребуются прагмы `FlexibleInstances` и `MultiParamTypeClasses`). Протестируйте работу на примере оператора деления:

```
(?/) :: (MonadError String m)
            => Double -> Double -> m Double
x ?/ 0 = throwError "Division by 0."
x ?/ y = return $ x / y
```

Представители классов типов `Monad` и `MonadPlus` должны обеспечивать следующее поведение: при вызовах функции

```
example :: Double -> Double -> Excep String
example x y = action  `catchError` return where
  action = do
    q <- x ?/ y
    guard (q >=0)
    if q  > 100 then do
      100 <- return q
      undefined
    else
      return $ show q
```

должны возвращаться такие результаты:

```
GHCi> example 5 2
Ok "2.5"
GHCi> example 5 0
Ok "Division by 0."
GHCi> example 5 (-2)
Ok "MonadPlus.mzero error."
GHCi> example 5 0.002
Ok "Monad.fail error."
```

▶ (2 балла) Введём тип данных для представления ошибки синтаксического разбора и зададим синоним типа для монады-обработчицы ошибок

```
data ParseError = ParseError {location::Int, reason::String}

type ParseMonad = Either ParseError
```

Разработайте следующие функции

```
parseHex :: String -> ParseMonad Integer
parseHex = undefined

printError :: ParseError -> ParseMonad String
printError = undefined
```

Функция `parseHex` пытается разобрать переданную ей строку как шестнадцатеричное число. При удачном исходе она возвращает это число, а при неудачном — генерирует исключение. Функция `printError` выводит информацию об этом исключении в удобном текстовом виде. Для тестирования используйте

```
test s = str where
  (Right str) = do
      n <- parseHex s
      return $ show n
    `catchError` printError
```

Ожидаемое поведение:

```
GHCi> test "DEADBEEF"
"3735928559"
GHCi> test "DEADMEAT"
"At pos 5: M: invalid digit"
```

Совет: воспользуйтесь вспомогательными функциями из `Data.Char`.

Трансформеры монад

▶ (2 балла) Разберитесь в работе следующего кода

```
import Control.Monad.Trans.Maybe
import Data.Char (isNumber, isPunctuation)

askPassword :: MaybeT IO ()
askPassword = do
  liftIO $ putStrLn "Enter your new password:"
  value <- msum $ repeat getValidPassword
  liftIO $ putStrLn "Storing in database..."

getValidPassword :: MaybeT IO String
getValidPassword = do
```

```
  s <- liftIO getLine
  guard (isValid s)
  return s

isValid :: String -> Bool
isValid s = length s >= 8
            && any isNumber s
            && any isPunctuation s
```

вызывая его в интерпретаторе:

```
GHCi> runMaybeT askPassword
```

Используя пользовательский тип ошибки и трансформер `ErrorT`, модифицируйте приведенный выше код так, чтобы он выдавал пользователю сообщение о причине, по которой пароль отвергнут.

```
data PwdError = PwdError String

type PwdErrorMonad = ErrorT PwdError IO

askPassword' :: PwdErrorMonad ()
askPassword' = do
  liftIO $ putStrLn "Enter your new password:"
  value <- msum $ repeat getValidPassword'
  liftIO $ putStrLn "Storing in database..."

getValidPassword' :: PwdErrorMonad String
getValidPassword' = undefined
```

Ожидаемое поведение:

```
GHCi> runErrorT askPassword'
Enter your new password:
qwerty
Incorrect input: password is too short!
qwertyuiop
Incorrect input: password must contain some digits!
qwertyuiop123
Incorrect input: password must contain some punctuations!
qwertyuiop123!!!
Storing in database...
GHCi>
```