Мини-проект на языке Haskell

В этом проекте вам предлагается реализовать мини-язык, встроенный в Haskell, для спецификации и запуска форм ввода и небольших тестов.

2.1. Разбор ввода

```haskell
-- Вид ошибок при разборе ввода пользователя.
data InputError
  = EmptyInput          -- Пустой ввод.
  | CannotParse String  -- Ввод нельзя разобрать в значение.
  | InvalidInput String -- Ввод можно разобрать,
                        -- но полученное значение не является корректным.
  deriving (Show)

-- Результат ввода пользователя.
data InputResult a
  = Success a
  | Failure InputError
  deriving (Show)
```

Упражнение 2.1. Реализуйте следующие функции, считывающие строку текста:

1. Чтение непустой строки:

```haskell
textInput :: String -> InputResult String
```

```
>>> textInput ""
Failure EmptyInput
```

```
>>> textInput "привет"
Success "привет"
```

2. Чтение целого числа:

```haskell
numericInput :: String -> InputResult Int
```

```
>>> numericInput "123"
Success 123
```

```
>>> numericInput ""
Failure EmptyInput
```

```
>>> numericInput "1.5"
Failure (CannotParse "1.5")
```

3. Чтение любой строки “как есть”:

```haskell
inputAsIs :: a -> InputResult a
```

```
>>> inputAsIs "привет"
Success "привет"
```

4. Чтение произвольного значения из класса типов `Read`:

```
readInput :: Read a => String -> InputResult a
```

```
>>> readInput "(123,False)" :: InputResult (Int, Bool)
Success (123,False)
```

```
>>> readInput "" :: InputResult (Int, Bool)
Failure EmptyInput
```

```
>>> readInput "(15," :: InputResult (Int, Bool)
Failure (CannotParse "(15,")
```

Упражнение 2.2 (валидация входных данных)**.** Реализуйте следующие функции, валидирующие корректность входных данных:

1. Валидация положительного числа:

```
newtype Positive = Positive Int deriving (Show)

positive :: Int -> Maybe Positive
```

```
>>> positive 123
Just (Positive 123)
```

```
>>> positive (-123)
Nothing
```

```
>>> positive 0
Nothing
```

2. Разбор и валидация строки по предикату:

```
validateWith :: String -> (a -> Bool) -> InputResult a -> InputResult a
```

```
>>> validateWith "Число должно быть больше 0." (> 0) (numericInput "-123")
Failure (InvalidInput "Число должно быть больше 0.")
```

```
>>> validateWith "Число должно быть больше 0." (> 0) (numericInput "123")
Success 123
```

```
>>> validateWith "Число должно быть больше 0." (> 0) (numericInput "1.23")
Failure (CannotParse "1.23")
```

```
>>> validateWith "Число должно быть больше 0." (> 0) (numericInput "")
Failure EmptyInput
```

3. Разбор строки и валидация результата:

```
validateInput :: String -> (a -> Maybe b) -> InputResult a -> InputResult b
```

```
>>> validateInput "Число должно быть положительным." positive (readInput "-123")
Failure (InvalidInput "Число должно быть положительным.")
```

```
>>> validateInput "Число должно быть положительным." positive (readInput "123")
Success (Positive 123)
```

```
>>> validateInput "Число должно быть положительным." positive (readInput "1.23")
Failure (CannotParse "1.23")
```

```
>>> validateInput "Число должно быть положительным." positive (readInput "")
Failure EmptyInput
```

Упражнение 2.3 (комбинаторы для разбора строк)**.** Реализуйте следующие комбинаторы, работающие с результатами разбора строк:

1. Применить функцию к результату разбора строки:

```
mapInputResult :: (a -> b) -> InputResult a -> InputResult b
```

```
>>> mapInputResult length (textInput "hello")
Success 5
```

2. Совместить два результата разбора строк (применить функцию к аргументу):

```
combineInputResult :: InputResult (a -> b) -> InputResult a -> InputResult b
```

```
>>> combineInputResult (mapInputResult (+) (numericInput "2")) (numericInput "3")
Success 5
```

3. Связать результат разбора строки с функцией-обработчиком:

```
bindInputResult :: InputResult a -> (a -> InputResult b) -> InputResult b
```

```
>>> let f = \(xs :: [Int]) -> readInput (concatMap show xs)
>>> bindInputResult (readInput "[1, 2, 3]") f
Success 123
```

2.2. Простые формы

В этом разделе вы реализуете простые формы ввода — программы, позволяющие вводить данные в интерактивном формате. В отличие от `IO`, формы автоматически отслеживают ошибки, связанные с проблематичным вводом пользователей, а также названия полей.

```
newtype Form a = Form { runForm :: [String] -> IO (InputResult a) }
```

Упражнение 2.4 (подготовка для форм)**.** Реализуйте следующие вспомогательные функции:

1. Ввод пользователя с приглашением ко вводу:

```
prompt :: String -> IO String
```

```
>>> prompt "What is your name? "
What is your name? Jack
"Jack"
```

2. Строка приглашения ко вводу, полученная из списка названий вложенных полей:

```
breadcrumbs :: [String] -> String
```

```
>>> breadcrumbs ["one","two","three"]
"[one > two > three]: "
>>> breadcrumbs []
"[]: "
```

3. Строка приглашения ко вводу, полученная из списка названий вложенных полей:

```
inputForm :: (String -> InputResult a) -> Form a
```

```
>>> runForm (inputForm numericInput) ["one", "two"]
[one > two]: 123
Success 123
```

Упражнение 2.5 (примитивные формы)**.** Реализуйте следующие примитивные формы:

1. Форма ввода (непустого) текстового поля:

```
textForm :: Form String
```

```
>>> runForm textForm ["user","name"]
[user > name]: Некий Кирилл
Success "Некий Кирилл"
```

2. Форма ввода числового поля:

```
numericForm :: Form Int
```

```
>>> runForm numericForm ["user","age"]
[user > age]: 25
Success 25
```

3. Форма ввода произвольного поля (для типов класса `Read`):

```
readForm :: Read a => Form a
```

```
>>> runForm (readForm :: Form Double) ["user","height"]
[user > height]: 123.45
Success 123.45
```

Упражнение 2.6 (повторный ввод)**.** Реализуйте следующие комбинаторы, приглашающие пользователя ввести данные повторно, при возникновении ошибки ввода:

1. Запустить форму, при необходимости повторить, до получения корректного ввода:

```
retryForm :: [String] -> Form a -> IO a
```

```
>>> retryForm ["user","age"] numericForm
[user > age]: 1.23
[Ошибка] Непонятный ввод: 1.23
[user > age]:
[Ошибка] Ввод не должен быть пустым (обязательное поле).
[user > age]: 123
123
```

2. Комбинатор, запускающий форму повторно, до получения корректного ввода:

```
retry :: Form a -> Form a
```

```
>>> runForm (retry numericForm) []
[]: 1.23
[Ошибка] Непонятный ввод: 1.23
[]:
[Ошибка] Ввод не должен быть пустым (обязательное поле).
[]: 123
Success 123
```

Упражнение 2.7 (комбинаторы форм)**.** Реализуйте следующие комбинаторы для форм:

1. Сделать форму опциональной (пустой ввод означает отсутствие ввода):

```
optionalForm :: Form a -> Form (Maybe a)
```

```
>>> runForm (optionalForm textForm) []
[]:
Success Nothing
>>> runForm (optionalForm numericForm) []
[]: 123
Success (Just 123)
>>> runForm (optionalForm numericForm) []
[]: 1.23
Failure (CannotParse "1.23")
```

2. Создать пустую форму:

```
emptyForm :: a -> Form a
```

```
>>> runForm (emptyForm 123) []
123
```

3. Применить функцию к результату формы ввода:

```
mapForm :: (a -> b) -> Form a -> Form b
```

```
>>> runForm (mapForm length textForm) ["any text"]
[any text]: Hello, world!
Success 13
```

4. Комбинировать результат двух форм ввода:

```
combineForm :: Form (a -> b) -> Form a -> Form b
```

```
>>> runForm (combineForm (mapForm (+) numericForm) numericForm) ["int"]
[int]: 2
[int]: 3
Success 5
```

5. Собрать заданное число результатов (списком), используя одну и ту же форму ввода:

```
listForm :: Int -> Form a -> Form [a]
```

```
>>> runForm (listForm 3 numericForm) ["int"]
[int]: 1
[int]: 2
[int]: 3
Success [1,2,3]
```

6. Связать результат формы ввода с другой формой:

```
bindForm :: Form a -> (a -> Form b) -> Form b
```

```
>>> runForm (bindForm numericForm (\n -> listForm n numericForm)) ["int"]
[int]: 2
[int]: 13
[int]: 45
Success [13,45]
```

7. Начать вложенную форму с данным названием:

```
subform :: String -> Form a -> Form a
```

```
>>> runForm (subform "age" numericForm) ["user"]
[user > age]: 123
Success 123
```

8. Начать форму с текстового описания:

```
describe :: String -> Form a -> Form a
```

```
>>> runForm (describe "Сколько будет 2 + 2?" numericForm) ["Ответ"]
Сколько будет 2 + 2?
[Ответ]: 4
Success 4
```

Упражнение 2.8 (валидация форм)**.** Реализуйте следующие комбинаторы, валидирующие формы и вспомогательные функции валидации:

1. Валидировать результат формы:

```
validate :: String -> (a -> Maybe b) -> Form a -> Form b
```

```
>>> runForm (validateForm "Число должно быть положительным." positive numericForm) ["int"]
[int]: -123
Failure (InvalidInput "Число должно быть положительным.")
```

2. Валидировать телефонный номер в международном формате (11 цифр):

```
newtype Phone = Phone Integer deriving (Show)

validatePhone :: Integer -> Maybe Phone
```

```
>>> validatePhone 79991234567
Just (Phone 79991234567)
```

```
>>> validatePhone 1234567
Nothing
```

```
>>> validatePhone (-1234567890)
Nothing
```

3. Форма ввода телефонного номера в международном формате (11 цифр):

```
phoneForm :: Form Phone
```

```
>>> retryForm ["тел."] phoneForm
[тел.]:
[Ошибка] Ввод не должен быть пустым (обязательное поле).
[тел.]: 123
[Ошибка] Некорректный ввод: Ожидается номер в международном формате (11 цифр).
[тел.]: 79991234567
Phone 79991234567
```

4. Валидировать адрес электронной почты:

```
newtype Email = Email String deriving (Show)

validateEmail :: String -> Maybe Email
```

```
>>> validateEmail "someone@ispring.com"
Just (Email "someone@ispring.com")
```

```
>>> validateEmail "best_email"
Nothing
```

5. Форма ввода адреса электронной почты:

```
emailForm :: Form Email
```

```
>>> retryForm ["email"] emailForm
[email]: best_email
[Ошибка] Некорректный ввод: Должен быть корректный адрес эл. почты.
[email]:
[Ошибка] Ввод не должен быть пустым (обязательное поле).
[email]: admin@ispring.com
Email "admin@ispring.com"
```

Упражнение 2.9 (поля)**.** Реализуйте следующие функции для именованных форм-полей:

1. Обязательное (непустое) текстовое поле:

```
textField :: String -> Form String
```

```
>>> runForm (textField "name") ["user"]
[user > name]: Jack
Success "Jack"
```

2. Обязательное числовое поле:

```
numericField :: Read a => String -> Form a
```

```
>>> runForm (numericField "age") ["user"]
[user > age]: 123
Success 123
```

3. Обязательное поле поизвольного типа (из класса `Read`):

```
readField :: Read a => String -> Form a
```

```
>>> runForm (readField "height" :: Form Double) ["user"]
[user > height]: 123.45
Success 123.45
```

4. Произвольное поле-форма:

```
field :: String -> Form a -> Form a
```

```
>>> runForm (field "тел." phoneForm) []
[тел.]: 12345678901
Success (Phone 12345678901)
```

Упражнение 2.10 (формы для двоичных ответов)**.** Реализуйте следующие функции и формы для ввода двоичных ответов (да/нет):

1. Валидировать двоичный ответ (из строки):

```
validateBool :: String -> String -> String -> Maybe Bool
```

```
>>> validateBool "да" "нет" "да"
Just True
>>> validateBool "да" "нет" "наверное"
Nothing
```

2. Форма ввода с двумя вариантами ответа:

```
boolForm :: String -> String -> Form Bool
```

```
>>> runForm (boolForm "да" "нет") []
[]: нет
Success False
>>> runForm (boolForm "да" "нет") []
[]: наверное
Failure (InvalidInput "Ответ должен быть [да] или [нет] (без скобок).")
```

Упражнение 2.11 (ввод данных пользователя)**.** Отметим, что `Form` образует монаду, а значит, формы можно создавать, используя `do`-нотацию:

```
instance Functor Form where
  fmap = mapForm
instance Applicative Form where
  pure = emptyForm
  (<*>) = combineForm
instance Monad Form where
  (>>=) = bindForm
```

Реализуйте следующие функции и форму ввода данных пользователя.

```
import Data.Time (Day)

newtype FullName = FullName String deriving (Show)
data Role = Regular | Admin deriving (Show, Read)

data User = User
  { userName     :: FullName
  , userPhone    :: Phone
  , userBirthday :: Maybe Day
  , userEmail    :: Maybe Email
  , userRole     :: Role
  } deriving (Show)
```

1. Реализуйте функцию, превращающую полное имя пользователя в фамилию с инициалами:

   ```
   shortName :: FullName -> String
   ```

   ```
   >>> putStrLn (shortName (FullName "Иванов Иван Иванович"))
   Иванов И.И.
   ```

   ```
   >>> putStrLn (shortName (FullName "Склодовская-Кюри Мария"))
   Склодовская-Кюри М.
   ```

   ```
   >>> name = "Длинныйчулок Пеппилотта Виктуалия Рульгардина Крисминта Эфраимсдоттер"
   >>> putStrLn (shortName (FullName name))
   Длинныйчулок П.В.Р.К.Э.
   ```

2. Обязательное (непустое) текстовое поле:

   ```
   user :: Form User
   ```

   ```
   >>> runForm user []
   [ФИО]: Петров Иван Фёдорович
   [Петров И.Ф. > Телефон]: 79991234567
   [Петров И.Ф. > Дата рождения]: 1991-01-01
   [Петров И.Ф. > Email]: petrov@ispring.com
   [Петров И.Ф. > Роль]: Admin
   Success
     (User
       {
         userName = FullName "Петров Иван Фёдорович",
         userPhone = Phone 79991234567,
         userBirthday = Just 1991-01-01,
         userEmail = Just (Email "petrov@ispring.com"),
         userRole = Admin
       })
   ```

2.3. Простые тесты

В этом разделе, определите комбинаторы для тестов на основе форм.

```
type Grade = Int
newtype CorrectAnswer a = CorrectIs a deriving (Show)
newtype Quiz a = Quiz { runQuiz :: Form (Grade, a) }

execQuiz :: Quiz a -> IO (Grade, a)
execQuiz q = retryForm [] (runQuiz q)
```

Упражнение 2.12 (примитивные тесты)**.** Реализуйте следующие примитивные тесты:

1. Тест из произвольной формы:

```
gradeWith :: (a -> Grade) -> Form a -> Quiz a
```

```
>>> let grade x = if x == 4 then 100 else 0
>>> runForm (runQuiz (gradeWith grade (field "2 + 2 =" numericForm))) []
[2 + 2 =]: 4
Success (100,4)
```

2. Тест из произвольной формы (с единственным верным ответом за 1 балл):

```
gradeCorrectAnswer :: Eq a => CorrectAnswer a -> Form a -> Quiz a
```

```
>>> form = describe "2 + 2 = ?" (numericField "Ответ")
>>> runForm (runQuiz (gradeCorrectAnswer (CorrectIs 4) form)) []
2 + 2 = ?
[Ответ]: 4
Success (1,4)
```

3. Вопрос с ответом да/нет:

```
trueOrFalse :: String -> CorrectAnswer Bool -> Quiz Bool
```

```
>>> execQuiz (trueOrFalse "2 + 2 = 4?" (CorrectIs True))
2 + 2 = 4?
[Ответ]: 4
[Ошибка] Некорректный ввод: Ответ должен быть [да] или [нет] (без скобок).
2 + 2 = 4?
[Ответ]: нет
(0,False)
```

4. Вопрос с числовым ответом:

```
numerical :: String -> CorrectAnswer Int -> Quiz Int
```

```
>>> execQuiz (numerical "2 + 2 = ?" (CorrectIs 4))
2 + 2 = ?
[Ответ]: 4
(1,4)
```

5. Вопрос выбором варианта из предложенных:

```
singleChoice :: String -> [String] -> CorrectAnswer String -> Quiz String
```

```
>>> execQuiz (singleChoice "2 + 2 = ?" ["3", "4", "5", "не знаю"] (CorrectIs "4"))
2 + 2 = ?
  1) 3
  2) 4
  3) 5
  4) не знаю

[Выберите верный вариант (от 1 до 4)]: 44
[Ошибка] Некорректный ввод: Ответ должен быть от 1 до 4.
2 + 2 = ?
  1) 3
  2) 4
  3) 5
  4) не знаю

[Выберите верный вариант (от 1 до 4)]: 2
(1,"4")
```

Упражнение 2.13 (комбинаторы тестов)**.** Реализуйте следующие комбинаторы для тестов:

1. Создать пустую форму:

```
emptyQuiz :: a -> Quiz a
```

```
>>> execQuiz (emptyQuiz "готово")
(0,"готово")
```

2. Применить функцию к результату теста (не к оценкам!):

```
mapQuiz :: (a -> b) -> Quiz a -> Quiz b
```

```
>>> execQuiz (mapQuiz show (numerical "2 + 2 = ?" (CorrectIs 4)))
2 + 2 = ?
[Ответ]: 123
(0,"123")
```

3. Комбинировать результат двух форм ввода:

```
bindQuiz :: Quiz a -> (a -> Quiz b) -> Quiz b
```

```
>>> execQuiz (bindQuiz q1 q2)
Отвечать будете?
[Ответ]: да
2 + 2 = ?
[Ответ]: 4
(2,4)
```

```
>>> execQuiz (bindQuiz q1 q2)
Отвечать будете?
[Ответ]: нет
(0,0)
```

Упражнение 2.14 (небольшой тест о Haskell)**.** Отметим, что `Quiz` образует монаду, а значит, формы можно создавать, используя `do`-нотацию:

```
import Control.Monad (ap)

instance Functor Quiz where
  fmap = mapQuiz
instance Applicative Quiz where
  pure = emptyQuiz
  (<*>) = ap
instance Monad Quiz where
  (>>=) = bindQuiz
```

Составьте небольшой тест о языке Haskell из 3 вопросов, каждый как 2–5 подпунктами.

```
miniQuiz :: Quiz ()
```