Haskell

Исключения
Модули

Выполнили:
Костюхин Алексей
Тасаков Антон
Теплов Андрей
Студенты гр. 5030102/10201

Тип Maybe

```haskell
data Maybe a = Nothing | Just a

safeDivide :: Double -> Double -> Maybe Double
safeDivide _ 0 = Nothing
safeDivide x y = Just (x / y)

-- Пример вызова
safeDivide 10 2   -- Результат: Just 5.0
safeDivide 10 0   -- Результат: Nothing
```

Maybe — это параметризованный тип данных в Haskell, используемый для представления значения, которое может отсутствовать

Тип Either

```haskell
data Either a b = Left a | Right b

safeDivide :: Double -> Double -> Either String Double
safeDivide _ 0 = Left "Division by zero!"
safeDivide x y = Right (x / y)

-- Пример вызова
safeDivide 10 2   -- Результат: Right 5.0
safeDivide 10 0   -- Результат: Left 'Division by zero!'
```

Either — это параметризованный тип данных в Haskell, который используется для представления двух альтернативных результатов

Класс Monad

```haskell
type Monad :: (* -> *) -> Constraint
class Applicative m => Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
  (>>) :: m a -> m b -> m b
  return :: a -> m a
  {- # MINIMAL (>>=) # -}

-- 1. Закон левосторонней единицы
return a >>= f == f a
-- 2. Закон правосторонней единицы
m >>= return == m
-- 3. Закон ассоциативности
(m >>= k) >>= h == m >>= (\x -> k x >>= h)
```

Класс монад (Monad) — одна из ключевых абстракций в функциональном программировании, особенно в Haskell. Он обобщает паттерн вычислений, который последовательно обрабатывает значения, поддерживая контекст

Монады Maybe и Either

```haskell
instance Monad Maybe where
    return = Just
    Nothing >>= _ = Nothing
    (Just x) >>= f = f x
```

```haskell
instance Monad (Either e) where
    return = Right
    (Left e) >>= _ = Left e
    (Right x) >>= f = f x
```

Maybe и **Either** - являются монадами, что позволяет удобно комбинировать последовательные вычисления

do-нотация

```haskell
exampleMaybe :: Maybe Int
exampleMaybe = do
    x <- Just 10  -- Получить значение из Just
    y <- Just 20  -- Получить значение из Just
    return (x + y)  -- Вернуть результат

-- Эквивалентное выражение без do:
exampleMaybe' :: Maybe Int
exampleMaybe' = Just 10 >>= (\x -> Just 20 >>= (\y -> return (x + y)))
```

do-нотация — это синтаксический сахар для упрощения работы с монадическими вычислениями. Она позволяет писать последовательные операции в стиле императивного программирования

Модуль Control.Monad.Except

```haskell
newtype ExceptT e m a = ExceptT { runExceptT :: m (Either e a) }
```

ExceptT — это трансформер монад, который добавляет обработку ошибок к любой базовой монаде

```haskell
class Monad m => MonadError e m | m -> e where
    throwError :: e -> m a
    catchError :: m a -> (e -> m a) -> m a
```

MonadError — это класс типов, который предоставляет стандартный интерфейс для работы с ошибками в монадическом контексте

Пример

```haskell
import Control.Monad.Except

type App = ExceptT String IO

divide :: Double -> Double -> App Double
divide _ 0 = throwError "Division by zero"
divide x y = return (x / y)

runApp :: App () -> IO ()
runApp app = do
    result <- runExceptT app
    case result of
        Left err  -> putStrLn $ "Error: " ++ err
        Right val -> putStrLn "Success!"

main :: IO ()
main = runApp $ do
    res <- divide 10 2
    lift $ putStrLn $ "Result: " ++ show res
```

Модуль Control.Exception

```haskell
class (Show e, Typeable e) => Exception e where
    toException   :: e -> SomeException
    fromException :: SomeException -> Maybe e
```

```haskell
throw :: Exception e => e -> a
throwIO :: Exception e => e -> IO a

try :: Exception e => IO a -> IO (Either e a)
catch :: Exception e => IO a -> (e -> IO a) -> IO a
finally :: IO a -> IO b -> IO a

bracket :: IO a         -- Открытие ресурса
        -> (a -> IO b) -- Освобождение ресурса
        -> (a -> IO c) -- Использование ресурса
        -> IO c
evaluate :: a -> IO a
```

Виды исключений

Пользовательский тип исключения. Реализуется через экземпляр класса **Exception**

```haskell
import Control.Exception

data MyException = MyException String
  deriving (Show)

instance Exception MyException
```

```
SomeException
 ├── IOException
 ├── ArithException
 ├── ArrayException
 ├── AssertionFailed
 ├── PatternMatchFail
 ├── ErrorCall
 ├── NonTermination
 ├── AsyncException
 └── ...
```

Все исключения в стандартной библиотеке являются экземплярами **Exception.** **SomeException** — универсальный тип, который может содержать любое исключение (в том числе пользовательское)

Пример

```haskell
riskyAction :: IO ()
riskyAction = do
  putStrLn "Performing risky action..."
  throwIO (MyException "Something went wrong!")

main :: IO ()
main = do
  result <- try riskyAction -- `try` возвращает Either
  case result of
    Left (ex :: MyException) -> putStrLn $ "Caught exception: " ++ show ex
    Right _ -> putStrLn "Risky action completed successfully."

  putStrLn "This will run regardless of exceptions."

  -- Пример с `catch` и `finally`
  (riskyAction
    `catch` \(ex :: MyException) -> putStrLn $ "Handled exception: " ++ show ex)
    `finally` putStrLn "Cleanup: This always runs."
```

Модули

```haskell
module MyModule (hello, double) where

{-      Экспортируемые функции     -}
hello :: String
hello = "Hello, World!"

double :: Int -> Int
double x = x * 2


{-     Неэкспортируемая функция    -}
privateFunction :: Int -> Int
privateFunction x = x + 1
```

```haskell
{-            Полный импорт              -}
import Data.List

{- Импорт только определённых символов -}
import Data.List (sort, nub)

{-     Импорт всех, кроме некоторых    -}
import Data.List hiding (nub)

{-          Импорт с префиксом          -}
import qualified Data.List as L
{-                  ...                 -}
L.sort [3, 1, 2]
```

Cabal

Cabal (Common Architecture for Building Applications and Libraries) это инструмент и формат для управления проектами и зависимостями в Haskell.

Команда	Описание
`cabal init`	Инициализация нового проекта. Создает файл `.cabal`, который описывает проект.
`cabal build`	Сборка проекта. Компилирует исходный код и генерирует исполнимый файл.
`cabal run`	Запуск исполнимого файла после сборки.
`cabal install`	Установка зависимостей, указанных в файле `.cabal`.
`cabal test`	Запуск тестов, если они настроены в проекте.
`cabal update`	Обновление списка доступных пакетов из репозитория пакетов Haskell.
`cabal clean`	Очистка промежуточных файлов и каталогов, созданных при сборке.

Cabal. Простой проект

```
MyProject/
├── src/                    -- Исходный код
│   ├── Main.hs             -- Главный модуль
│   └── Lib.hs              -- Дополнительные модули
├── MyProject.cabal         -- Конфигурация проекта
```

```
name:                MyProject
version:             0.1.0.0
build-type:          Simple
cabal-version:       >= 1.10

executable myproject
  main-is:             Main.hs
  hs-source-dirs:      src
  build-depends:       base >= 4.14 && < 5
  default-language:    Haskell2010
```