Монады и IO

Лекция 16: Работа с побочными эффектами

Преподаватель: [Ваше имя]

Группа: 203

Семестр: Осенний 2024

План лекции

1. **Что такое монады?**
2. **Maybe монада**
3. **IO монада**
4. **Работа с файлами**
5. **Обработка ошибок**
6. **Практический пример: Игровой движок**

Что такое монады?

Определение:

Монада — это способ структурирования вычислений с побочными эффектами в чистом функциональном языке.

Основные операции:

- **return** — помещает значение в монадический контекст
- **>>=** (bind) — связывает монадические вычисления
- **>>** — связывает монадические вычисления без передачи значения

Преимущества:

- **Чистота** функций сохраняется

Maybe монада

Что такое Maybe?

Maybe — монада для представления вычислений, которые могут не дать результата.

Определение:

```haskell
data Maybe a = Nothing | Just a
    deriving (Show, Eq)

-- Maybe является монадой
instance Monad Maybe where
    return = Just
    Nothing >>= _ = Nothing
    Just x >>= f = f x
```

Использование:

Maybe в игровой логике

```haskell
-- Функции для работы с Maybe в игре
findUnitByName :: String -> [Unit] -> Maybe Unit
findUnitByName name units =
    case filter (\unit -> unitName unit == name) units of
        [] -> Nothing
        (unit:_) -> Just unit

findUnitAtPosition :: Position -> [Unit] -> Maybe Unit
findUnitAtPosition pos units =
    case filter (\unit -> unitPosition unit == pos) units of
        [] -> Nothing
        (unit:_) -> Just unit

-- Безопасное движение юнита
safeMoveUnit :: Unit -> Position -> [Unit] -> Maybe Unit
safeMoveUnit unit newPos allUnits = do
    -- Проверяем, что позиция свободна
    guard (isNothing (findUnitAtPosition newPos allUnits))
    -- Проверяем, что юнит может двигаться
    guard (canMoveTo unit newPos)
    -- Возвращаем обновленный юнит
    return unit { unitPosition = newPos }

-- Безопасная атака
safeAttack :: Unit -> Unit -> Maybe Int
safeAttack attacker target = do
    -- Проверяем, что атакующий жив
    guard (isAlive attacker)
    -- Проверяем, что цель жива
    guard (isAlive target)
    -- Проверяем, что атака возможна
    guard (canAttack attacker target)
    -- Вычисляем урон
    damage <- calculateDamage attacker target
    return damage

-- Безопасное получение информации
getUnitInfo :: String -> [Unit] -> Maybe String
getUnitInfo name units = do
    unit <- findUnitByName name units
    return $ formatUnit unit

-- Обработка цепочки Maybe операций
processUnitAction :: String -> Position -> [Unit] -> Maybe String
processUnitAction unitName targetPos units = do
    unit <- findUnitByName unitName units
    updatedUnit <- safeMoveUnit unit targetPos units
    return $ "Unit " ++ unitName unit ++ " moved to " ++ show targetPos
```

IO монада

Что такое IO?

IO — монада для работы с вводом-выводом и другими побочными эффектами.

Особенности:

- **Чистые функции** не могут выполнять IO
- **IO действия** выполняются только в main или других IO функциях
- **Последовательность** операций гарантирована

Базовые IO операции:

```haskell
-- Чтение строки
getLine :: IO String
```

Работа с IO

```haskell
-- Простая программа с IO
main :: IO ()
main = do
    putStrLn "Добро пожаловать в игру!"
    putStrLn "Введите имя игрока:"
    playerName <- getLine
    putStrLn $ "Привет, " ++ playerName ++ "!"

    putStrLn "Введите уровень сложности (1-5):"
    difficultyStr <- getLine
    let difficulty = read difficultyStr :: Int

    if difficulty >= 1 && difficulty <= 5
        then putStrLn $ "Уровень сложности установлен: " ++ show difficulty
        else putStrLn "Неверный уровень, установлен средний (3)"

    putStrLn "Игра начинается!"

-- Функция для получения пользовательского ввода
getPlayerInput :: String -> IO String
getPlayerInput prompt = do
    putStr prompt
    getLine

-- Функция для получения числового ввода
getNumberInput :: String -> IO Int
getNumberInput prompt = do
    putStr prompt
    input <- getLine
    case reads input of
        [(n, "")] -> return n
        _ -> do
            putStrLn "Неверный ввод, попробуйте снова"
            getNumberInput prompt

-- Функция для выбора из меню
showMenu :: [String] -> IO Int
showMenu options = do
    putStrLn "Выберите опцию:"
    mapM_ (\i -> putStrLn $ show i ++ ". " ++ options !! (i-1)) [1..length options]
    choice <- getNumberInput "Ваш выбор: "
    if choice >= 1 && choice <= length options
        then return choice
        else do
            putStrLn "Неверный выбор, попробуйте снова"
            showMenu options
```

Работа с файлами

Чтение файлов:

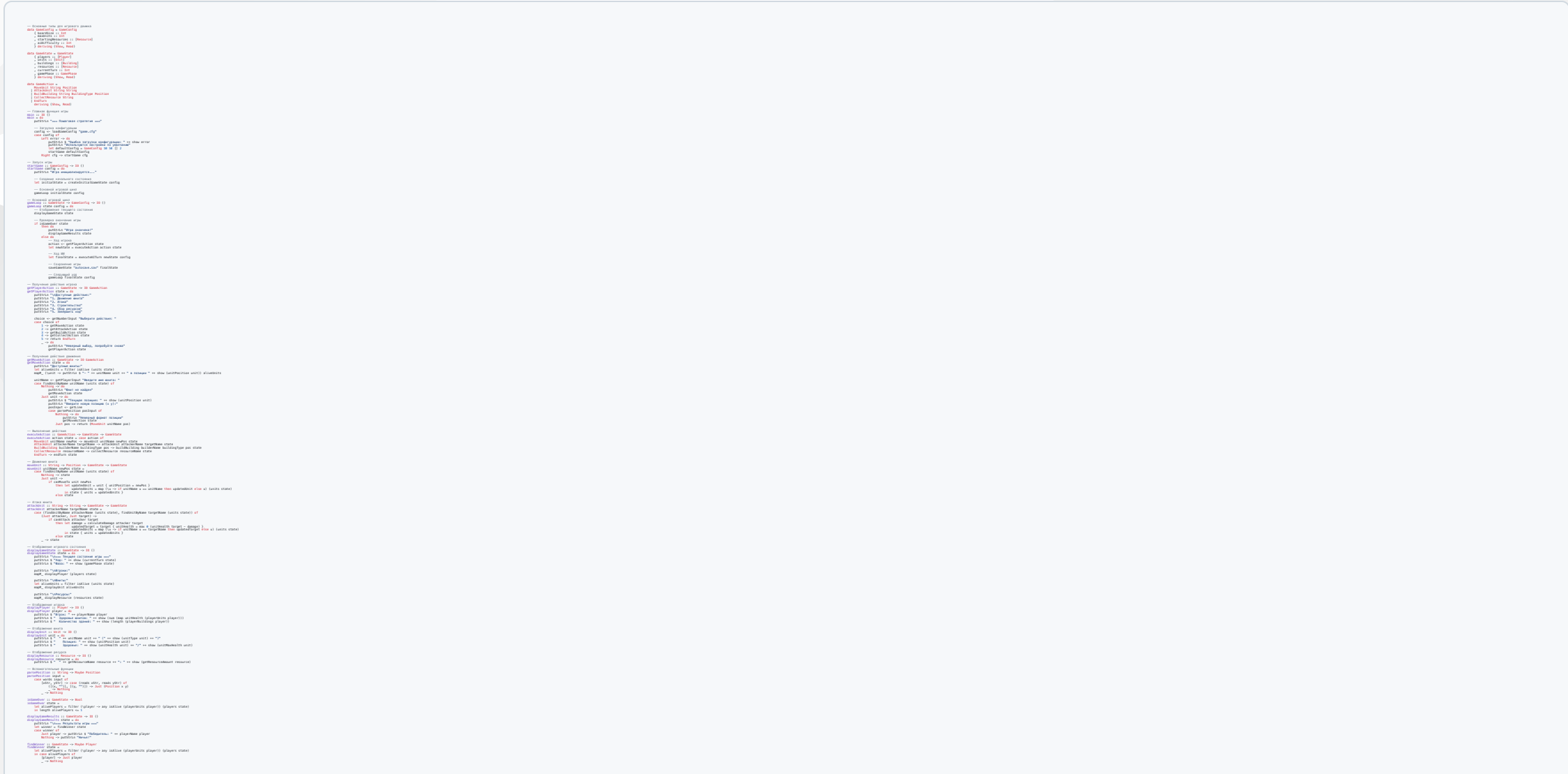
```haskell
-- Чтение содержимого файла
readGameFile :: FilePath -> IO String
readGameFile filename = do
    content <- readFile filename
    return content

-- Чтение файла с обработкой ошибок
safeReadFile :: FilePath -> IO (Either String String)
safeReadFile filename = do
    catch (do
        content <- readFile filename
        return (Right content))
        (\e -> return (Left ("Ошибка чтения файла: " ++ show (e :: IOError))))

-- Чтение игрового состояния из файла
loadGameState :: FilePath -> IO (Maybe GameState)
loadGameState filename = do
    result <- safeReadFile filename
    case result of
        Left error -> do
            putStrLn $ "Ошибка загрузки: " ++ error
            return Nothing
        Right content -> do
            case parseGameState content of
```

Обработка ошибок

Either монада:

```haskell
-- Either для обработки ошибок
data GameError =
    FileNotFound String
  | ParseError String
  | ValidationError String
  | NetworkError String
    deriving (Show, Eq)

-- Функции с обработкой ошибок
loadGameConfig :: FilePath -> IO (Either GameError GameConfig)
loadGameConfig filename = do
    result <- safeReadFile filename
    case result of
        Left ioError -> return (Left (FileNotFound (show ioError)))
        Right content -> case parseConfig content of
            Just config -> return (Right config)
            Nothing -> return (Left (ParseError "Неверный формат конфигурации"))

-- Обработка цепочки Either операций
initializeGame :: FilePath -> IO (Either GameError GameState)
initializeGame configFile = do
    config <- loadGameConfig configFile
    case config of
        Left error -> return (Left error)
        Right cfg -> do
            let gameState = createInitialGameState cfg
            if validateGameState gameState
                then return (Right gameState)
                else return (Left (ValidationError "Неверное начальное состояние игры"))

-- Функции для работы с Either
mapLeft :: (e -> e') -> Either e a -> Either e' a
mapLeft f (Left e) = Left (f e)
mapLeft _ (Right a) = Right a

mapRight :: (a -> b) -> Either e a -> Either e b
```

Практический пример: Игровой движок



Лучшие практики работы с монадами

✅ Что делать:

- **Использовать Maybe** для безопасных вычислений
- **Применять Either** для обработки ошибок
- **Разделять IO и чистую логику** в программах
- **Использовать do-нотацию** для читаемости
- **Обрабатывать ошибки** на соответствующем уровне

❌ Чего избегать:

- **Смешивать IO и чистые функции** без необходимости
- **Игнорировать обработку ошибок** в Maybe и Either
- **Создавать слишком сложные** монадические цепочки

Домашнее задание

Задача 1:

Реализовать систему сохранения/загрузки игры

Задача 2:

Создать интерактивное меню для игры

Задача 3:

Реализовать обработку ошибок в игровом движке

Что дальше?

На следующей лекции:

- **Тестирование**
- **JUnit 5**
- **Mock объекты**
- **Тестирование архитектуры**

Подготовка:

- Изучить главу 29-30 из учебника
- Выполнить домашнее задание
- Подготовить вопросы по текущей теме

Вопросы?

Контакты:

- **Email:** [ваш.email@university.edu]
- **Telegram:** [@username]
- **Офис:** [номер кабинета]

Следующая лекция: Тестирование