Функции высшего порядка

Лекция 15: Функции как данные

Преподаватель: [Ваше имя]

Группа: 203

Семестр: Осенний 2024

План лекции

1. **Что такое функции высшего порядка?**
2. **Map, filter, fold**
3. **Композиция функций**
4. **Частичное применение**
5. **Лямбда-функции**
6. **Практический пример: Игровая логика**

Что такое функции высшего порядка?

Определение:

Функция высшего порядка — функция, которая принимает другие функции в качестве аргументов или возвращает функции.

Преимущества:

- **Переиспользование** кода
- **Абстракция** общих паттернов
- **Читаемость** и выразительность
- **Композиция** функций

Примеры:

Map функция

Определение:

```haskell
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (x:xs) = f x : map f xs
```

Использование:

```haskell
-- Применение функции к каждому элементу
doubleList :: [Int] -> [Int]
doubleList xs = map (*2) xs

-- Преобразование типов
getNames :: [Unit] -> [String]
getNames units = map unitName units

-- Работа с позициями
getXCoordinates :: [Position] -> [Int]
```

Filter функция

Определение:

```haskell
filter :: (a -> Bool) -> [a] -> [a]
filter _ [] = []
filter p (x:xs)
    | p x = x : filter p xs
    | otherwise = filter p xs
```

Использование:

```haskell
-- Фильтрация по условию
aliveUnits :: [Unit] -> [Unit]
aliveUnits units = filter isAlive units

-- Фильтрация по типу
warriors :: [Unit] -> [Unit]
warriors units = filter (\unit -> unitType unit == Warrior) units

-- Фильтрация по позиции
```

Fold функции

Foldr (справа налево):

```haskell
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr _ z [] = z
foldr f z (x:xs) = f x (foldr f z xs)

-- Примеры использования
sumList :: [Int] -> Int
sumList xs = foldr (+) 0 xs

productList :: [Int] -> Int
productList xs = foldr (*) 1 xs

concatList :: [[a]] -> [a]
concatList xs = foldr (++) [] xs

-- Сложные примеры
countAliveUnits :: [Unit] -> Int
countAliveUnits units = foldr (\unit count ->
    if isAlive unit then count + 1 else count) 0 units

getTotalHealth :: [Unit] -> Int
getTotalHealth units = foldr (\unit total ->
    total + unitHealth unit) 0 units

findStrongestUnit :: [Unit] -> Maybe Unit
findStrongestUnit units = foldr (\unit strongest ->
```

Композиция функций

Оператор композиции:

```haskell
(.) :: (b -> c) -> (a -> b) -> a -> c
(f . g) x = f (g x)

-- Примеры использования
-- Композиция простых функций
doubleAndIncrement :: Int -> Int
doubleAndIncrement = (+1) . (*2)

-- Композиция с функциями для работы со списками
getAliveUnitNames :: [Unit] -> [String]
getAliveUnitNames = map unitName . filter isAlive

getStrongUnitPositions :: [Unit] -> [Position]
getStrongUnitPositions = map unitPosition . filter (\unit -> unitHealth unit > 100)

-- Сложная композиция
formatAliveUnits :: [Unit] -> String
formatAliveUnits =
    concat . map (\unit -> unitName unit ++ " (HP: " ++ show (unitHealth unit) ++ ")\n")
    . filter isAlive

-- Композиция с преобразованием типов
getUnitHealths :: [Unit] -> [Int]
getUnitHealths = map unitHealth . filter isAlive
```

Частичное применение

Что такое частичное применение?

Частичное применение — создание новой функции путем фиксации части аргументов существующей функции.

Примеры:

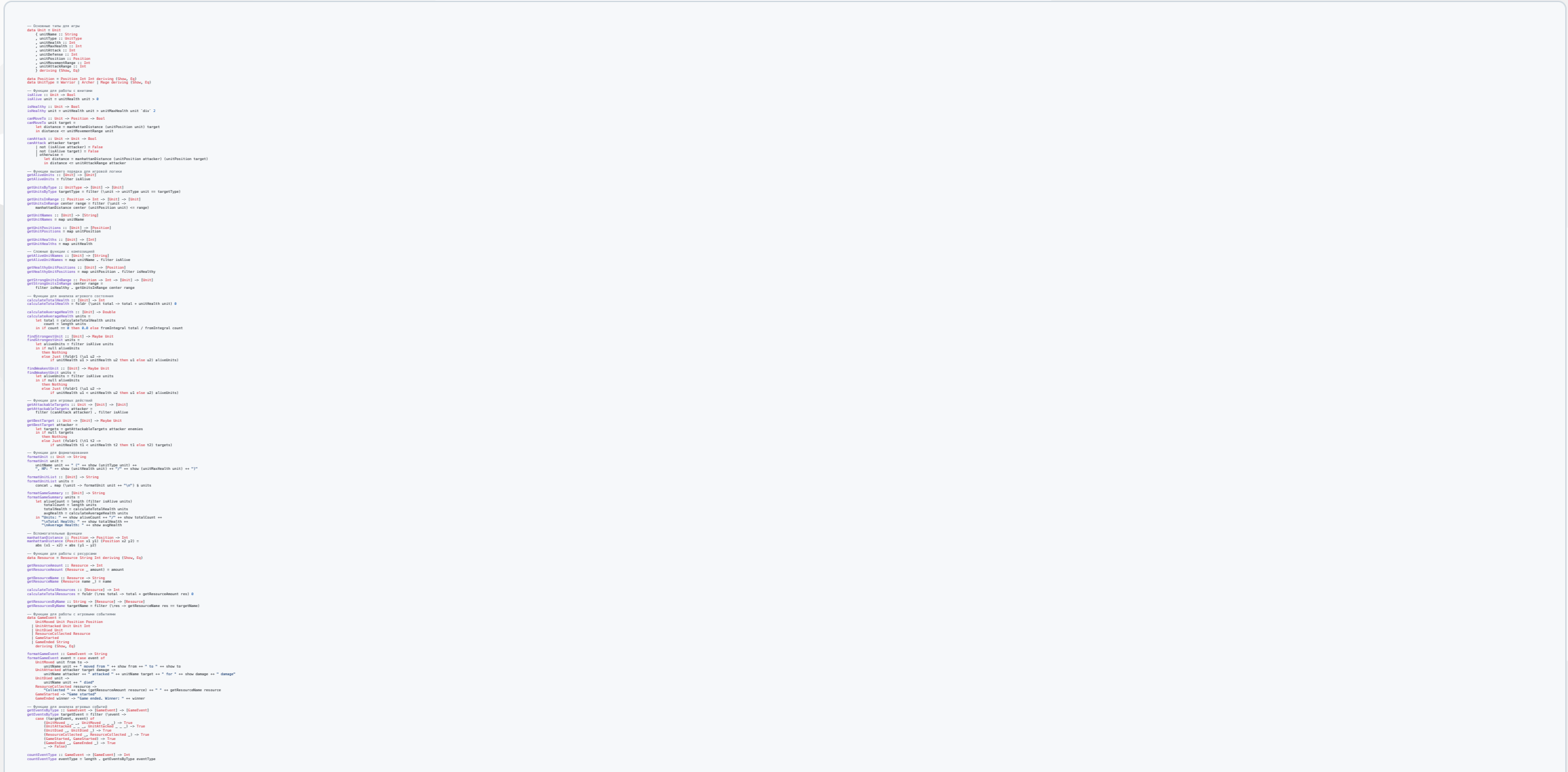
```haskell
-- Частичное применение с двумя аргументами
add :: Int -> Int -> Int
add x y = x + y

addFive :: Int -> Int
addFive = add 5

-- Частичное применение с функциями высшего порядка
filterAlive :: [Unit] -> [Unit]
filterAlive = filter isAlive

mapNames :: [Unit] -> [String]
mapNames = map unitName

-- Комбинирование частичного применения
getAliveUnitNames :: [Unit] -> [String]
getAliveUnitNames = map unitName . filter isAlive

-- Частичное применение с операторами
isGreaterThan :: Int -> Int -> Bool
```

Лямбда-функции

Синтаксис:

```haskell
-- Лямбда-функции (анонимные функции)
\x -> x + 1                    -- добавляет 1 к аргументу
\x y -> x + y                  -- складывает два аргумента
\unit -> unitHealth unit > 50  -- проверяет здоровье юнита

-- Использование с map
incrementList :: [Int] -> [Int]
incrementList xs = map (\x -> x + 1) xs

-- Использование с filter
strongUnits :: [Unit] -> [Unit]
strongUnits units = filter (\unit -> unitHealth unit > 100) units

-- Использование с fold
totalAttack :: [Unit] -> Int
totalAttack units = foldr (\unit total -> total + unitAttack unit) 0 units

-- Сложные лямбда-функции
formatUnitDetailed :: Unit -> String
formatUnitDetailed unit =
    let healthStatus = if unitHealth unit > 50 then "Healthy" else "Weak"
    in unitName unit ++ " (" ++ healthStatus ++ ", HP: " ++ show (unitHealth unit) ++ ")"

-- Лямбда-функции с pattern matching
getUnitPositions :: [Unit] -> [Position]
getUnitPositions = map (\(Unit _ _ _ _ _ _ pos _ _) -> pos)

-- Лямбда-функции с guards
classifyUnit :: Unit -> String
```

Практический пример: Игровая логика



Лучшие практики функций высшего порядка

✅ Что делать:

- **Использовать map, filter, fold** для работы со списками
- **Применять композицию функций** для создания цепочек преобразований
- **Использовать частичное применение** для создания специализированных функций
- **Создавать читаемые лямбда-функции** с понятными именами
- **Комбинировать функции** для сложных операций

❌ Чего избегать:

- **Создавать слишком сложные** композиции
- **Использовать лямбда-функции** там, где можно использовать именованные

Домашнее задание

Задача 1:

Реализовать функции для работы с игровыми картами

Задача 2:

Создать функции для анализа игрового состояния

Задача 3:

Реализовать функции для игровых действий

Что дальше?

На следующей лекции:

- **Монады и IO**
- **Работа с файлами**
- **Обработка ошибок**
- **Тестирование**

Подготовка:

- Изучить главу 27-28 из учебника
- Выполнить домашнее задание
- Подготовить вопросы по текущей теме

Вопросы?

Контакты:

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

Следующая лекция: Монады и IO