Функциональное программирование

Лекция 13: Введение в Haskell

Преподаватель: [Ваше имя]

Группа: 203

Семестр: Осенний 2024

План лекции

1. **Что такое функциональное программирование?**
2. **Почему Haskell?**
3. **Основные концепции ФП**
4. **Синтаксис Haskell**
5. **Наш проект: Пошаговая стратегия на Haskell**

Что такое функциональное программирование?

Функциональное программирование — парадигма программирования, основанная на вычислении математических функций.

Ключевые принципы:

- **Функции** — основные строительные блоки
- **Иммутабельность** — данные не изменяются
- **Чистые функции** — без побочных эффектов
- **Высший порядок** — функции как аргументы и результаты

Почему именно Haskell?

Преимущества для изучения ФП:

✅ **Чистый функциональный язык** — без побочных эффектов
✅ **Ленивые вычисления** — вычисления по требованию
✅ **Строгая типизация** — безопасность типов
✅ **Параллелизм** — естественная поддержка
✅ **Математическая основа** — теория категорий
✅ **Академический язык** — много исследований

Основные концепции ФП

1. Иммутабельность

```haskell
-- В Haskell переменные не изменяются
x = 5
y = x + 3  -- y = 8, x остается 5
```

2. Чистые функции

```haskell
-- Функция всегда возвращает одинаковый результат
add :: Int -> Int -> Int
add x y = x + y

-- add 2 3 всегда равно 5
```

3. Функции высшего порядка

Синтаксис Haskell

Объявление функций:

```haskell
-- Простая функция
double :: Int -> Int
double x = x * 2

-- Функция с несколькими параметрами
add :: Int -> Int -> Int
add x y = x + y

-- Функция с pattern matching
factorial :: Integer -> Integer
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

Типы в Haskell

Базовые типы:

```haskell
-- Числа
x :: Int          -- Целые числа
y :: Double       -- Числа с плавающей точкой
z :: Integer      -- Большие целые числа

-- Строки и символы
s :: String       -- Строки
c :: Char         -- Символы

-- Логические значения
b :: Bool         -- True или False
```

Составные типы:

Pattern Matching

Сопоставление с образцом:

```haskell
-- Обработка списков
head :: [a] -> a
head (x:_) = x
head [] = error "Empty list"

-- Обработка кортежей
fst :: (a, b) -> a
fst (x, _) = x

-- Обработка конструкторов
data Shape = Circle Double | Rectangle Double Double

area :: Shape -> Double
area (Circle r) = pi * r * r
area (Rectangle w h) = w * h
```

Рекурсия в Haskell

Рекурсивные функции:

```haskell
-- Сумма элементов списка
sumList :: [Int] -> Int
sumList [] = 0
sumList (x:xs) = x + sumList xs

-- Длина списка
length' :: [a] -> Int
length' [] = 0
length' (_:xs) = 1 + length' xs

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

List Comprehensions

Генерация списков:

```haskell
-- Квадраты чисел от 1 до 10
squares = [x^2 | x <- [1..10]]

-- Четные числа от 1 до 20
evens = [x | x <- [1..20], even x]

-- Пары чисел, где сумма равна 10
pairs = [(x, y) | x <- [1..10], y <- [1..10], x + y == 10]

-- Фильтрация с условием
filtered = [x | x <- [1..100], x `mod` 3 == 0, x `mod` 5 == 0]
```

Наш проект: Пошаговая стратегия на Haskell

Архитектура игры:

```haskell
-- Основные типы данных
data Position = Position { x :: Int, y :: Int }
data Unit = Unit { name :: String, health :: Int, attack :: Int }
data GameState = GameState { units :: [Unit], board :: GameBoard }

-- Игровая логика
moveUnit :: Unit -> Position -> GameState -> GameState
attackUnit :: Unit -> Unit -> GameState -> GameState
updateGame :: GameState -> GameState
```

Пример игровой логики

```haskell
-- Типы для игры
data UnitType = Warrior | Archer | Mage
data Unit = Unit {
    unitType :: UnitType,
    name :: String,
    health :: Int,
    maxHealth :: Int,
    attack :: Int,
    position :: Position
}

-- Функция движения
moveUnit :: Unit -> Position -> Unit
moveUnit unit newPos = unit { position = newPos }

-- Функция атаки
attackUnit :: Unit -> Unit -> Unit
attackUnit attacker target =
    target { health = max 0 (health target - attack attacker) }
```

Функции высшего порядка для игры

```haskell
-- Применение действия ко всем юнитам
updateAllUnits :: (Unit -> Unit) -> [Unit] -> [Unit]
updateAllUnits f = map f

-- Фильтрация юнитов по условию
getAliveUnits :: [Unit] -> [Unit]
getAliveUnits = filter (\unit -> health unit > 0)

-- Поиск юнита по позиции
findUnitAt :: Position -> [Unit] -> Maybe Unit
findUnitAt pos = find (\unit -> position unit == pos)

-- Проверка, может ли юнит атаковать цель
canAttack :: Unit -> Unit -> Bool
canAttack attacker target =
    distance (position attacker) (position target) <= getAttackRange attacker
```

Преимущества Haskell для игр

Безопасность:

✅ **Отсутствие null-ошибок**
✅ **Неизменяемость** — предсказуемость
✅ **Строгая типизация** — ошибки на этапе компиляции

Производительность:

✅ **Ленивые вычисления** — эффективность
✅ **Оптимизация компилятора** — быстрый код
✅ **Параллелизм** — масштабируемость

Домашнее задание

Задача 1:

Создать функцию `distance :: Position -> Position -> Double` для вычисления расстояния между позициями

Задача 2:

Реализовать функцию `isValidMove :: Unit -> Position -> [Unit] -> Bool` для проверки возможности движения

Задача 3:

Создать функцию `getUnitsInRange :: Position -> Int -> [Unit] -> [Unit]` для поиска юнитов в радиусе

Что дальше?

На следующей лекции:

- **Типы данных и алгебраические типы**
- **Монады и ввод-вывод**
- **Работа с файлами**

Подготовка:

- Установить GHC (Glasgow Haskell Compiler)
- Изучить главу 1-2 из учебника по Haskell
- Подготовить вопросы по текущей теме

Вопросы?

Контакты:

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

Следующая лекция: Типы данных и алгебраические типы