# Типы данных и алгебраические типы

## Лекция 14: Система типов Haskell

**Преподаватель:** [Ваше имя]

**Группа:** 203

**Семестр:** Осенний 2024

# План лекции

1. **Система типов Haskell**
2. **Базовые типы**
3. **Алгебраические типы данных**
4. **Pattern matching**
5. **Type classes**
6. **Практический пример: Игровые типы**

# Система типов Haskell

## Особенности:

- **Статическая типизация** — типы проверяются на этапе компиляции
- **Вывод типов** — компилятор автоматически определяет типы
- **Полиморфизм** — функции могут работать с разными типами
- **Type classes** — интерфейсы для типов

## Преимущества:

- **Безопасность** — ошибки типов обнаруживаются на этапе компиляции
- **Читаемость** — типы служат документацией
- **Оптимизация** — компилятор может оптимизировать код
- **Рефакторинг** — изменения типов безопасны

# Базовые типы

## Примитивные типы:

```haskell
-- Числовые типы
42          :: Int          -- 32-битные целые
3.14        :: Double       -- числа с плавающей точкой
42          :: Integer      -- произвольной точности
3.14        :: Float        -- одинарная точность

-- Символьные типы
'a'         :: Char         -- символ
"Hello"     :: String       -- строка (список символов)

-- Логический тип
True        :: Bool         -- логическое значение
False       :: Bool

-- Функциональный тип
(+)         :: Num a => a -> a -> a  -- функция сложения
```

# Функции и типы

## Аннотации типов:

```haskell
-- Явное указание типа функции
add :: Int -> Int -> Int
add x y = x + y

-- Функция с полиморфным типом
length :: [a] -> Int
length [] = 0
length (_:xs) = 1 + length xs

-- Функция высшего порядка
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (x:xs) = f x : map f xs

-- Функция с ограничением типа
sum :: Num a => [a] -> a
```

# Алгебраические типы данных

## Что такое ADT?

**Алгебраические типы данных** — способ создания новых типов путем комбинирования существующих.

## Виды ADT:

- **Product types** — типы-произведения (кортежи, записи)
- **Sum types** — типы-суммы (перечисления, варианты)
- **Recursive types** — рекурсивные типы (списки, деревья)

# Product Types

## Кортежи:

```haskell
-- Кортежи фиксированной длины
(1, "hello")           :: (Int, String)
(1, 2, 3)             :: (Int, Int, Int)
(True, 42, "test")     :: (Bool, Int, String)

-- Функции для работы с кортежами
fst :: (a, b) -> a
snd :: (a, b) -> b

-- Пример использования
getFirst :: (a, b, c) -> a
getFirst (x, _, _) = x

getSecond :: (a, b, c) -> b
getSecond (_, y, _) = y
```

# Sum Types

## Перечисления:

```haskell
-- Простое перечисление
data Direction = North | South | East | West
    deriving (Show, Eq)

-- Перечисление с данными
data UnitType = Warrior | Archer | Mage
    deriving (Show, Eq)

-- Перечисление с параметрами
data GameEvent =
    UnitMoved Unit Position Position
  | UnitAttacked Unit Unit Int
  | UnitDied Unit
  | GameStarted
  | GameEnded String
    deriving (Show, Eq)

-- Функции для работы с перечислениями
isWarrior :: UnitType -> Bool
isWarrior Warrior = True
isWarrior _ = False

getEventDescription :: GameEvent -> String
getEventDescription (UnitMoved unit from to) =
    "Unit " ++ unitName unit ++ " moved from " ++ show from ++ " to " ++ show to
getEventDescription (UnitAttacked attacker target damage) =
    "Unit " ++ unitName attacker ++ " attacked " ++ unitName target ++ " for " ++ show damage ++ " damage"
```

# Рекурсивные типы

## Списки:

```haskell
-- Определение списка
data List a = Empty | Cons a (List a)
    deriving (Show, Eq)

-- Функции для работы со списком
listLength :: List a -> Int
listLength Empty = 0
listLength (Cons _ xs) = 1 + listLength xs

listHead :: List a -> Maybe a
listHead Empty = Nothing
listHead (Cons x _) = Just x

listMap :: (a -> b) -> List a -> List b
listMap _ Empty = Empty
listMap f (Cons x xs) = Cons (f x) (listMap f xs)
```

# Pattern Matching

## Сопоставление с образцом:

```haskell
-- Сопоставление с кортежами
getCoordinates :: (Int, Int) -> String
getCoordinates (x, y) = "Position: (" ++ show x ++ ", " ++ show y ++ ")"

-- Сопоставление со списками
listSum :: [Int] -> Int
listSum [] = 0
listSum (x:xs) = x + listSum xs

-- Сопоставление с записями
isAlive :: Player -> Bool
isAlive player = playerHealth player > 0

-- Сопоставление с перечислениями
canMove :: UnitType -> Bool
canMove Warrior = True
canMove Archer = True
canMove Mage = False

-- Сопоставление с вложенными структурами
getUnitInfo :: Unit -> String
```

# Guards и Case выражения

## Guards:

```haskell
-- Использование guards
describeHealth :: Int -> String
describeHealth health
    | health <= 0 = "Dead"
    | health < 25 = "Critical"
    | health < 50 = "Low"
    | health < 75 = "Medium"
    | otherwise = "High"

-- Guards с условиями
canAttack :: Unit -> Unit -> Bool
canAttack attacker target
    | not (isAlive attacker) = False
    | not (isAlive target) = False
    | distance (unitPosition attacker) (unitPosition target) > attackRange attacker = False
    | otherwise = True
```

## Case выражения:

# Type Classes

## Что такое Type Classes?

**Type Classes** — интерфейсы для типов, определяющие набор операций.

## Основные Type Classes:

```haskell
-- Eq - равенство
class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool

-- Show - преобразование в строку
class Show a where
    show :: a -> String

-- Ord - упорядочивание
class Eq a => Ord a where
    compare :: a -> a -> Ordering
    (<) :: a -> a -> Bool
    (<=) :: a -> a -> Bool
    (>) :: a -> a -> Bool
```

# Реализация Type Classes

## Автоматическая реализация:

```haskell
-- deriving автоматически создает реализации
data Position = Position Int Int
    deriving (Show, Eq, Ord)

-- Show: Position 1 2
-- Eq: Position 1 2 == Position 1 2 -> True
-- Ord: Position 1 2 < Position 2 1 -> True
```

## Ручная реализация:

```haskell
-- Ручная реализация Show
data Unit = Unit String Int Int Position

instance Show Unit where
    show (Unit name health level pos) =
```

# Практический пример: Игровые типы

```haskell
-- Основные типы для игры
data Position = Position
    { x :: Int
    , y :: Int
    } deriving (Show, Eq, Ord)

data UnitType = Warrior | Archer | Mage
    deriving (Show, Eq)

data Unit = Unit
    { unitName :: String
    , unitType :: UnitType
    , unitHealth :: Int
    , unitMaxHealth :: Int
    , unitAttack :: Int
    , unitDefense :: Int
    , unitPosition :: Position
    , unitMovementRange :: Int
    , unitAttackRange :: Int
    } deriving (Show, Eq)

data BuildingType = Barracks | MageTower | Farm | Mine
    deriving (Show, Eq)

data Building = Building
    { buildingName :: String
    , buildingType :: BuildingType
    , buildingHealth :: Int
    , buildingPosition :: Position
    , buildingLevel :: Int
    } deriving (Show, Eq)

data ResourceType = Gold | Wood | Stone | Food
    deriving (Show, Eq)

data Resource = Resource
    { resourceType :: ResourceType
    , resourceAmount :: Int
    } deriving (Show, Eq)

data GameState = GameState
    { players :: [Player]
    , units :: [Unit]
    , buildings :: [Building]
    , resources :: [Resource]
    , currentTurn :: Int
    , gamePhase :: GamePhase
    } deriving (Show, Eq)

data GamePhase = PlayerTurn | AITurn | GameOver
    deriving (Show, Eq)

data Player = Player
    { playerName :: String
    , playerResources :: [Resource]
    , playerUnits :: [Unit]
    , playerBuildings :: [Building]
    } deriving (Show, Eq)

-- Функции для работы с типами
isAlive :: Unit -> Bool
isAlive unit = unitHealth unit > 0

canMoveTo :: Unit -> Position -> Bool
canMoveTo unit target =
    let currentPos = unitPosition unit
        distance = manhattanDistance currentPos target
    in distance <= unitMovementRange unit

canAttack :: Unit -> Unit -> Bool
canAttack attacker target
    | not (isAlive attacker) = False
    | not (isAlive target) = False
    | otherwise =
        let distance = manhattanDistance (unitPosition attacker) (unitPosition target)
        in distance <= unitAttackRange attacker

getUnitsInRange :: Position -> Int -> [Unit] -> [Unit]
getUnitsInRange center range units =
    filter (\unit -> manhattanDistance center (unitPosition unit) <= range) units

findNearestEnemy :: Unit -> [Unit] -> Maybe Unit
findNearestEnemy unit enemies =
    let aliveEnemies = filter isAlive enemies
        distances = map (\enemy -> (enemy, manhattanDistance (unitPosition unit) (unitPosition enemy))) aliveEnemies
        sorted = sortBy (comparing snd) distances
    in case sorted of
        [] -> Nothing
        ((enemy, _):_) -> Just enemy

-- Вспомогательные функции
manhattanDistance :: Position -> Position -> Int
manhattanDistance (Position x1 y1) (Position x2 y2) =
    abs (x1 - x2) + abs (y1 - y2)

euclideanDistance :: Position -> Position -> Double
euclideanDistance (Position x1 y1) (Position x2 y2) =
    sqrt (fromIntegral ((x1 - x2)^2 + (y1 - y2)^2))

-- Функции для работы с ресурсами
hasEnoughResources :: Player -> [Resource] -> Bool
hasEnoughResources player required =
    all (\req ->
        let available = find (\r -> resourceType r == resourceType req) (playerResources player)
        in case available of
            Just avail -> resourceAmount avail >= resourceAmount req
            Nothing -> False
    ) required

spendResources :: Player -> [Resource] -> Maybe Player
spendResources player required
    | not (hasEnoughResources player required) = Nothing
    | otherwise = Just $ player { playerResources = updatedResources }
    where
        updatedResources = map updateResource (playerResources player)
        updateResource playerRes =
            case find (\r -> resourceType r == resourceType playerRes) required of
                Just req -> playerRes { resourceAmount = resourceAmount playerRes - resourceAmount req }
                Nothing -> playerRes
```

# Лучшие практики работы с типами

## ✅ Что делать:

- **Использовать описательные имена** для типов и конструкторов
- **Применять deriving** для стандартных Type Classes
- **Создавать специфичные типы** вместо использования базовых
- **Использовать Pattern Matching** для безопасной работы с данными
- **Группировать связанные данные** в записи

## ❌ Чего избегать:

- **Использовать базовые типы** для доменных понятий
- **Создавать слишком сложные** типы
- **Игнорировать Type Classes** при проектировании

# Домашнее задание

## Задача 1:

Создать типы для игровых карт и местности

## Задача 2:

Реализовать функции для работы с игровыми объектами

## Задача 3:

Создать Type Class для игровых сущностей

# Что дальше?

## На следующей лекции:

- **Функции высшего порядка**
- **Map, filter, fold**
- **Композиция функций**
- **Частичное применение**

## Подготовка:

- Изучить главу 25-26 из учебника
- Выполнить домашнее задание
- Подготовить вопросы по текущей теме

# Вопросы?

## Контакты:

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

## Следующая лекция: Функции высшего порядка