

Лекция 1: введение и основы синтаксиса

Функциональное программирование на Haskell

Алексей Романов

6 марта 2023 г.

МИЭТ

Организация курса

- 8 лекций
- 4 лабораторных
- Итоговый проект

Парадигмы программирования

- Что такое парадигма?

Парадигмы программирования

- Что такое парадигма?
«Совокупность идей и понятий, определяющих стиль написания компьютерных программ.»
(Wikipedia)

Парадигмы программирования

- Что такое парадигма?
«Совокупность идей и понятий, определяющих стиль написания компьютерных программ.»
(*Wikipedia*)
- Основные парадигмы:

Парадигмы программирования

- Что такое парадигма?
«Совокупность идей и понятий, определяющих стиль написания компьютерных программ.»
(Wikipedia)
- Основные парадигмы:
 - Структурное программирование
 - Процедурное программирование
 - **Функциональное программирование**
 - Логическое программирование
 - Объектно-ориентированное программирование

Парадигмы программирования

- Что такое парадигма?
«Совокупность идей и понятий, определяющих стиль написания компьютерных программ.»
(Wikipedia)
- Основные парадигмы:
 - Структурное программирование
 - Процедурное программирование
 - **Функциональное программирование**
 - Логическое программирование
 - Объектно-ориентированное программирование
- В парадигме важно не только то, что используется, но то, использование чего не допускается или минимизируется.

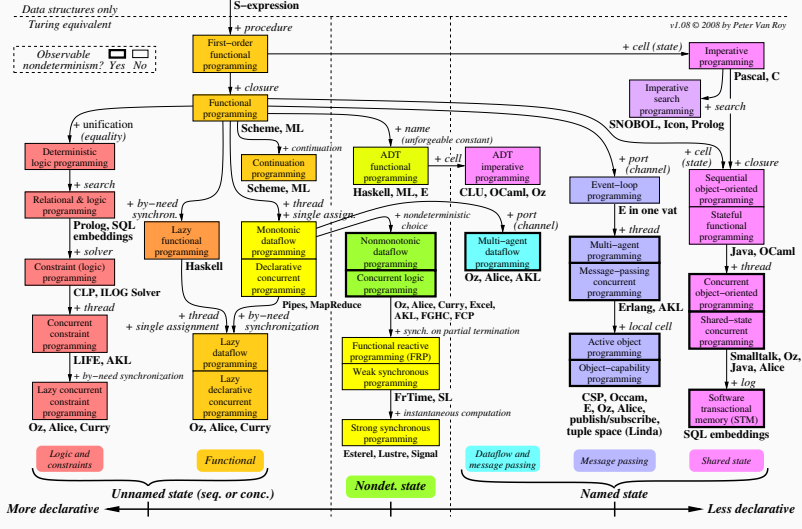
Парадигмы программирования

- Что такое парадигма?
«Совокупность идей и понятий, определяющих стиль написания компьютерных программ.»
(Wikipedia)
- Основные парадигмы:
 - Структурное программирование
 - Процедурное программирование
 - **Функциональное программирование**
 - Логическое программирование
 - Объектно-ориентированное программирование
- В парадигме важно не только то, что используется, но то, использование чего не допускается или минимизируется.
- Например, **goto** в структурном программировании, глобальные переменные в ООП.

The principal programming paradigms

"More is not better (or worse) than less, just different."

v1.08 © 2008 by Peter Van Roy



Peter Van Roy, "Programming Paradigms for Dummies: What Every Programmer Should Know", 2009

- Значения лучше переменных.
 - Переменная даёт имя значению или функции, а не адресу в памяти.
 - Переменные неизменяемы.
 - Типы данных неизменяемы.
- Выражения лучше инструкций.
 - Аналоги `if`, `try-catch` и т.д. — выражения.
- Функции как в математике (следующий слайд)

- Функции как в математике

- Функции как в математике
 - Чистые функции: аргументу соответствует результат, а всё прочее от лукавого.
 - Нет побочных эффектов (ввода-вывода, обращения к внешней памяти, не связанной с аргументом, и т.д.)
 - При одинаковых аргументах результаты такой функции одинаковы
 - Функции являются значениями (функции первого класса)
 - Функции часто принимают и возвращают функции (функции высших порядков)

- Функции как в математике
 - Чистые функции: аргументу соответствует результат, а всё прочее от лукавого.
 - Нет побочных эффектов (ввода-вывода, обращения к внешней памяти, не связанной с аргументом, и т.д.)
 - При одинаковых аргументах результаты такой функции одинаковы
 - Функции являются значениями (функции первого класса)
 - Функции часто принимают и возвращают функции (функции высших порядков)
- Опора на математические теории:
лямбда-исчисление, теория типов, теория категорий

- семейство Lisp: первый ФП-язык и один из первых языков высокого уровня вообще
- Erlang и Elixir: упор на многозадачность (модель акторов), надёжность
- Scala, Kotlin, F#: гибриды с ООП для JVM и для CLR
- Purescript, Elm, Ur/Web: для веба
- Семейство ML: OCaml, SML, F#

- семейство Lisp: первый ФП-язык и один из первых языков высокого уровня вообще
- Erlang и Elixir: упор на многозадачность (модель акторов), надёжность
- Scala, Kotlin, F#: гибриды с ООП для JVM и для CLR
- Purescript, Elm, Ur/Web: для веба
- Семейство ML: OCaml, SML, F#
- **Haskell:**
 - Чисто функциональный

- семейство Lisp: первый ФП-язык и один из первых языков высокого уровня вообще
- Erlang и Elixir: упор на многозадачность (модель акторов), надёжность
- Scala, Kotlin, F#: гибриды с ООП для JVM и для CLR
- Purescript, Elm, Ur/Web: для веба
- Семейство ML: OCaml, SML, F#
- **Haskell:**
 - Чисто функциональный
 - Строго статически типизированный (с очень мощной и выразительной системой типов)

- семейство Lisp: первый ФП-язык и один из первых языков высокого уровня вообще
- Erlang и Elixir: упор на многозадачность (модель акторов), надёжность
- Scala, Kotlin, F#: гибриды с ООП для JVM и для CLR
- Purescript, Elm, Ur/Web: для веба
- Семейство ML: OCaml, SML, F#
- **Haskell:**
 - Чисто функциональный
 - Строго статически типизированный (с очень мощной и выразительной системой типов)
 - Ленивый

Язык Haskell: начало

- Установите ghcup (<https://www.haskell.org/ghcup/>)
- Запустите GHCi
- Это оболочка или REPL (Read-Eval-Print loop)
 - Read: Вы вводите выражения Haskell (и команды GHCi)
 - Eval: GHCi вычисляет результат
 - Print: и выводит его на экран

- Пример:

GHCi, version 9.2.5: <http://www.haskell.org/ghc/> :?

↪ for help

```
ghci> 2 + 2
```

```
4
```

```
ghci> :t True -- команда GHCi
```

```
True :: Bool
```

Язык Haskell: начало

- `2 + 2`, `True` — выражения
- `4`, `True` — значения
- `Bool` — тип

Язык Haskell: начало

- `2 + 2`, `True` — выражения
- `4`, `True` — значения
- `Bool` — тип
- Значение — «вычисленное до конца» выражение.
- Тип (статический) — множество значений и выражений, построенное по таким правилам, что компилятор может определить типы и проверить отсутствие ошибок в них без запуска программы.
- От типа зависит то, какие операции допустимы:
`ghci> True + False`
`<interactive>:12:1: error:`
`No instance for (Num Bool) arising from a use of '+'`
`In the expression: True + False`
`In an equation for 'it': it = True + False`
- Это ошибка компиляции, а не выполнения.

Вызов функций

- Вызов (применение) функции пишется без скобок:
`f x`, `foo x y`.
- Скобки используются, когда аргументы — сложные выражения: `f (g x)`
- И внутри сложных выражений вообще.
- Бинарные операторы (как `+`) это просто функции с именем из символов вместо букв и цифр.
 - Можно писать их префиксно, заключив в скобки:
`(+) 2 2`.
 - А любую функцию двух аргументов с алфавитным именем можно писать инфиксно между обратными апострофами: `4 `div` 2`.
 - Единственный небинарный оператор — унарный `-`.
- Названия переменных и функций начинаются со строчной буквы (кроме операторов).

Определение функций и переменных

- Определение переменной выглядит как в математике, даже без ключевых слов:
`название = значение`
- Определение функции почти такое же:
`название параметр1 параметр2 = значение`
- Тело функции это не блок, а одно выражение (но сколь угодно сложное).

```
ghci> x = sin pi
ghci> x
1.2246063538223773e-16
ghci> square x = x * x
ghci> square 2
4
```

Определение функций и переменных

- Определение переменной выглядит как в математике, даже без ключевых слов:

название = значение

- Определение функции почти такое же:

название параметр1 параметр2 = значение

- Тело функции это не блок, а одно выражение (но сколь угодно сложное).

```
ghci> x = sin pi
```

```
ghci> x
```

Определение функций и переменных

- Определение переменной выглядит как в математике, даже без ключевых слов:
`название = значение`
- Определение функции почти такое же:
`название параметр1 параметр2 = значение`
- Тело функции это не блок, а одно выражение (но сколь угодно сложное).

```
ghci> x = sin pi
ghci> x
1.2246063538223773e-16
ghci> square x = x * x
ghci> square 2
4
```


Базовые типы

- Названия типов всегда с заглавной буквы.
- **Bool**: логические значения **True** и **False**.
- Целые числа:
 - **Integer**: неограниченные (кроме размера памяти);
 - **Int**: машинные¹, **Word**: машинные без знака;
 - **Data.Int.Int/Word.Word}{8/16/32/64}**: фиксированного размера в битах, со знаком и без.
- **Float** и **Double**: 32- и 64-битные числа с плавающей точкой, по стандарту IEEE-754.
- **Character**: символы Unicode.
- **()**: «Единичный тип» (unit) с единственным значением **()**.

¹по стандарту минимум 30 бит, но в GHC именно 32 или 64 бита

Тип функций и сигнатуры

- Типы функций записываются через `->`. Например, `Int -> Char` это тип функции из `Int` в `Char`.
- Для нескольких аргументов это выглядит как

Тип функций и сигнатуры

- Типы функций записываются через `->`. Например, `Int -> Char` это тип функции из `Int` в `Char`.
- Для нескольких аргументов это выглядит как `Bool -> Bool -> Bool`.
- `::` читается как «имеет тип».
- Запись `выражение :: тип` — «сигнатура типа».
- При объявлении экспортируемой функции или переменной сигнатура обычно указывается явно:
`foo :: Int -> Char`
`foo x = ...`
- Компилятор обычно может вывести типы сам, но это защищает от *непреднамеренного* изменения.

- Это упрощённая версия.
- Полное объяснение требует понятия, которое будет введено позже.
- Например, команды `:type 1` или `:type (+)` дадут тип, который понимать пока не требуется.
- То же относится к ошибкам вроде `No instance for (Num Bool)...` на более раннем слайде.
- Пока достаточно понимать, что есть несколько числовых типов.
- Они делятся на целочисленные (`Int`, `Integer`) и дробные (`Float`, `Double`).

Числовые литералы

- Числовые литералы выглядят, как в других языках: `0`, `1.5`, `1.2E-1`, `0xDEADBEEF`.
- Целочисленные литералы могут иметь любой числовой тип, а дробные любой дробный.
- Но это относится *только* к литералам.
- Неявного приведения (например, `Int` в `Double`) в Haskell *нет*.

Приведение числовых типов

- Используйте `fromIntegral` для приведения из любого целочисленного типа в любой числовой.

Тип-цель можно указать явно:

```
ghci> let {x :: Integer; x = 2}
```

```
ghci> x :: Double
```

```
<interactive>:22:1: error:
```

```
Couldn't match expected type 'Double' with actual  
  ↳ type 'Integer' ...
```

```
ghci> fromIntegral x :: Double
```

```
2.0
```

- А может выводиться из контекста:

```
ghci> :t fromIntegral 2 / (4 :: Double)
```

```
fromIntegral 2 / (4 :: Double) :: Double
```

Приведение числовых типов

- `toInteger` переводит любой целочисленный тип в `Integer`.
- `fromInteger` — наоборот. Если аргумент слишком велик, возвращает значение по модулю:

```
ghci> fromInteger (2^64) :: Int  
0
```
- `toRational` и `fromRational` — аналогично для `Rational` и дробных типов.
- `ceiling`, `floor`, `truncate` и `round` — из дробных типов в целочисленные.

Арифметические операции

- Операции `+`, `-`, `*` — как обычно (унарного `+` нет).
- `/` — деление *дробных* чисел.
 - Использование `/` для целых чисел даст ошибку
`No instance... arising from a use of '/'`. Нужно
сначала использовать `fromIntegral`.
- `div` — деление нацело
- `mod` — остаток
- `quot` и `rem` тоже, но отличаются от них поведением на отрицательных числах.
- `^` — возведение любого числа в неотрицательную целую степень.
- `^^` — дробного в любую целую.
- `**` — дробного в степень того же типа.

Операции сравнения и логические операции

- Большинство операций сравнения выглядят как обычно: `==`, `>`, `<`, `>=`, `<=`.
- Но `≠` обозначается как `/=`.
- Функция `compare` возвращает `Ordering`: тип с тремя значениями `LT`, `EQ` и `GT`.
- Есть функции `min` и `max`.
- «и» это `&&`, а «или» — `||`, как обычно.
- «не» — `not`.