

# **Лекция 2: основы синтаксиса (продолжение)**

Функциональное программирование на Haskell

---

Алексей Романов

6 марта 2023 г.

МИЭТ

# Существенные отступы

- Вспомним виденное раньше

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

По правилам Haskell можно написать это же без фигурных скобок и точки с запятой:

```
let x :: Integer  
    x = 2
```

- Если два выражения (или других фрагмента кода) относятся к одному уровню одного блока, то они должны иметь одинаковый отступ (начинаться в одном и том же столбце).
- Фрагмент кода, являющийся частью другого, должен иметь отступ больше той строки, где тот начинается.
- Но не обязательно больше начала его самого.

# Правило преобразования отступов

- Если после ключевых слов **where**, **let**, **do**, **of** нет открывающей фигурной скобки {<sup>1</sup>, то такая скобка вставляется и начинается новый блок.
- Его базовый отступ — столбец, содержащий следующий пробельный символ.
- После этого каждая строка, которая начинается:
  - в том же столбце: относится к этому блоку и перед ней ставится ;
  - правее: продолжает предыдущую, перед ней ничего не ставится.
  - левее: блок закончился, перед ней ставится } (одновременно может закончиться несколько блоков).

---

<sup>1</sup>Если она есть, то правило перестаёт действовать до соответствующей ей закрывающей.

# Условные выражения

- В любом языке нужна возможность выдать результат в зависимости от какого-то условия.
- В Haskell это выражения `if` и `case`.
- Синтаксис `if`:  
`if` условие `then` выражение1 `else` выражение2  
Варианта без `else` нет.
- Многострочно пишется так:  
`if` условие  
    `then` выражение1  
    `else` выражение2
- Тип `условия` обязательно `Bool`.
- Типы `выражения1` и `выражения2` должны совпадать.
- Это же и тип всего `if ... then ... else ...`
- Ближе к `?:`, чем к `if` в C-подобных языках.

# Сопоставление с образцом

- Синтаксис **case**:

**case** выражение **of**

образец1 -> выражение1

образец2 -> выражение2

- Образец это «форма» для значений типа, которая может содержать несвязанные переменные. Для конкретного значения он либо подходит (и связывает эти переменные), либо не подходит.
- Процедура вычисления:
  - Вычислить значение **выражения**.
  - Сопоставить его с каждым образцом по очереди.
  - Если первым подошёл **образецN**, вычислить **выражениеN** и вернуть его значение.
  - Если ни один не подошёл, выкинуть ошибку.

# Образцы для известных нам типов

- `True` и `False` — образцы для `Bool`. Они подходят, только если значение совпадает с ними.
- Аналогично `LT`, `EQ` и `GT` для `Ordering`, а числовые литералы для числовых типов.
- Переменная — образец для любого типа, который подходит для любого значения.
  - В Haskell все переменные в образцах «свежие» и перекрывают видимые снаружи переменные с тем же названием.
- `_` тоже подходит для любого значения любого типа и означает, что это значение не важно.
- Образцы похожи на выражения, но ими не являются: это новая синтаксическая категория!

- Пример:

```
if условие
  then выражение1
  else выражение2
```

это ровно то же самое, что

```
case условие of
  True  -> выражение1
  False -> выражение2
```

или

```
case условие of
  True  -> выражение1
  _     -> выражение2
```

# Охраняющие условия

- У каждого образца могут быть дополнительные условия (зависящие от его переменных):

образец

```
| условие1 -> выражение1  
| условие2 -> выражение2
```

При удачном сопоставлении они проверяются по порядку. Если все оказались ложны, сопоставление переходит к следующему образцу.

- Последнее условие часто `otherwise` (синоним `True`), тогда хотя бы одно условие точно истинно.
- Чтобы сравнить переменную в образце с внешней, нужно использовать `==` в условии:

```
case foo x of
```

```
  y | y == x -> ... -- не то же, что x -> ...  
  _ -> ...
```



# Многоветвенные `if`

- Цепочка `if ... else if ...`, оформленная по правилам отступа, напоминает лестницу:

```
if условие1
  then результат1
  else if условие2
    then результат2
    else результат3
```

- С расширением `MultiWayIf` можем написать:

```
if | условие1 -> результат1
   | условие2 -> результат2
   | otherwise -> результат3
```

- Это можно сделать и с `case` без расширений: как?

# Многоветвенные `if`

- Цепочка `if ... else if ...`, оформленная по правилам отступа, напоминает лестницу:

```
if условие1
  then результат1
  else if условие2
    then результат2
    else результат3
```

- С расширением `MultiWayIf` можем написать:

```
if | условие1 -> результат1
   | условие2 -> результат2
   | otherwise -> результат3
```

- Это можно сделать и с `case` без расширений:

```
case () of _ | условие1 -> ...
```

# Определение функций по случаям

- Часто тело функции — `case` по аргументу, например.

```
not x = case x of
  True  -> False
  False -> True
```

- Такие функции можно записать несколькими равенствами, по одному на ветвь `case`:

```
not True = False
not False = True
```

- Это работает и с охранными условиями:

```
not x | x = False
      | otherwise = True
```

- и для нескольких параметров:

```
nand True True = False
nand _      _  = True
```

## Локальные определения: `let`

- Локальные определения дают две выгоды:
  - Более читаемый код.
  - Избавление от повторяющихся вычислений.
- В Haskell два способа их задать: `let` и `where`.
- Синтаксис `let`:

```
let переменная1 = выражение1
    функция2 x = выражение2
    ...
in выражение3
```

- Всё `let ... in ...` — выражение.
- Первое проявление ленивости: будут вычислены только те переменные, которые понадобятся для вычисления `выражения3`.

## Локальные определения: **where**

- Синтаксис **where**:

```
функция образец1 | условие1 = выражение3  
                  | условие2 = выражение4  
  where переменная1 = выражение1  
        функция2 x = выражение2
```

- Видимы в условиях и в правых частях (но не для других образцов).
- Можно применить только к определениям.
- В том числе к локальным.

# Модули

- Программа на Haskell состоит из модулей.
- Модуль **Модуль** (названия с заглавной буквы) определяется в файле **Модуль.hs**:
  - Для названия вида **A.B.C** будет файл **A/B/C.hs**.

```
module Модуль(функция1) where
```

```
import ...
```

```
функция1 :: ...
```

```
функция1 x = ...
```

- **функция1** экспортирована, она доступна другим модулям и GHCi. Все остальные — нет.
- Можно экспортировать всё, опустив список экспортов (включая скобки).

# Импорт из другого модуля

- У директивы `import` есть много вариантов.
- По адресу <https://wiki.haskell.org/Import> есть полный перечень.
- Нам пока достаточно простейших:
- `import Модуль (функция1, функция2)`  
импортирует конкретные функции.
- `import Модуль`  
импортирует всё, что возможно.
- Импортированные функции доступны как `функция` и как `Модуль.функция`.

- В GHCi можно скомпилировать и загрузить модуль (вместе с зависимостями) командой `:load Модуль`. Подробности в документации:
  - [The effect of :load on what is in scope](#)
- Под Windows можно просто дважды щёлкнуть на файл или нажать Enter в Проводнике.
- Потом при изменениях файла повторить `:load` или сделать `:reload`.
- Многие редакторы позволяют это автоматизировать.



- Для сборки лабораторных мы используем **Stack**.
- Ещё одна распространённая система сборки: **Cabal**.
- Обе ставятся вместе с Haskell через `ghcup`.
- Основные команды можно посмотреть в документации, но нам пока достаточно
  - `stack repl` для запуска GHCi с загруженными файлами проекта.
  - `stack test` для запуска тестов.