

Функциональное программирование

Лекция 4. Введение в Haskell

Денис Николаевич Москвин

СПбАУ РАН, CS Center

05.03.2015

- 1 Язык Haskell
- 2 Основы программирования
- 3 Базовые типы
- 4 Система модулей
- 5 Операторы и сечения

- 1 Язык Haskell
- 2 Основы программирования
- 3 Базовые типы
- 4 Система модулей
- 5 Операторы и сечения

- Haskell — *чистый* функциональный язык программирования с «*ленивой*» семантикой и полиморфной *статической* типизацией.
- Сайт языка: <http://haskell.org>
- Назван в честь американского логика и математика Хаскелла Б. Карри.
- Первая реализация: 1990 год.
- Текущий стандарт языка: [Haskell 2010](#).
- Девиз (неофициальный): *Avoid Success at All Costs!*

- Основная реализация: [GHC](#).
- Упаковка библиотек в пакеты и дистрибуция: [Cabal](#).
- Хранилище пакетов: [HackageDB](#).
- Среда разработки: [Haskell Platform 2014.2.0.0](#):
 - стабильная версия компилятора GHC (7.8.3);
 - интерпретатор GHCi;
 - Cabal;
 - лучшие библиотеки;
 - вспомогательные инструменты.

- Инсталлируем Haskell Platform:
`http://hackage.haskell.org/platform/.`
- Создаём файл `hello.hs` содержащий:

```
main = putStrLn "Hello, world!"
```
- Компилируем в исполняемый файл...

```
$ ghc --make hello
```
- ...или запускаем в интерпретаторе GHCi

```
$ ghci hello.hs
```

- 1 Язык Haskell
- 2 Основы программирования**
- 3 Базовые типы
- 4 Система модулей
- 5 Операторы и сечения

- Знак равенства задаёт *связывание*:

имя (слева) связывается со значением (справа)

```
x = 2           -- глобальное
y = 42          -- глобальное
foo = let z = x + y -- глобальное (foo), локальное (z)
      in print z  -- отступ (layout rule)
```

- Первый символ идентификатора должен быть в нижнем регистре.
- В GHCi `let` используют для глобального связывания

Сессия GHCi:

```
Prelude> let fortyTwo = 42
Prelude> fortyTwo
42
```


Определение функций

- Равенство может задавать функцию.

Ниже `add` связывается глобально, а `x` и `y` — локально

```
add x y = x + y           -- определение add  
  
fortyTwo = add 40 2      -- вызов add
```

- Допустимо использовать лямбда-выражения для определения функций.

Все три определения эквивалентны

```
add x y = x + y  
add' x  = \y -> x + y  
add''   = \x y -> x + y
```

- *Соглашения об ассоциативности вызовов* — такие же как в λ -исчислении.

Первый пример даст ошибку из-за арности

```
oops = print add 1 2
good = print (add 1 2)
```

- *Независимость от порядка.*

Не важно, что определено раньше, а что позже

```
fortyTwo = add 40 2  -- вызов add
add x y = x + y      -- определение add
```

- *Иммутабельность.*

Связывание происходит единожды

```
z = 1           -- ок, связали
z = 2           -- ошибка
q q = \q -> q   -- ок, но...
```

- *Ленивость.*

Сессия GHCi

```
Prelude> let k = \x y -> x
Prelude> k 42 undefined
42
```

Факториал на языке C: цикл и изменяемые переменные

```
long factorial (int n)
{
    long res = 1;
    while (n > 1)
        res *= n--;
    return res;
}
```

Факториал на языке Haskell: рекурсия и повторное связывание имени в новой области видимости

```
factorial n = if n > 1
              then n * factorial (n-1)
              else 1
```

Факториал рекурсивно

```
factorial n = if n > 1 then n * factorial (n-1) else 1
```

Это менее эффективно, чем цикл на C — на каждом шаге рекурсии монтируется новый кадр стека (stack frame).

Однако имеется оптимизация *хвостовой* рекурсии — преобразование её в цикл.

Приведём рекурсивный вызов к хвостовому

```
factorial' n = helper 1 n
helper acc n = if n > 1
                then helper (acc * n) (n - 1)
                else acc
```

Стандартная техника обеспечения хвостового вызова — вспомогательная функция с аккумулярующим параметром.

- `where` и `let ... in ...` позволяют обеспечить локальное связывание вспомогательных конструкций.

Пример использования `where`

```
factorial'' n' = helper 1 n'
  where helper acc n = if n > 1
                        then helper (acc * n) (n - 1)
                        else acc
```

Пример использования `let ... in ...`

```
factorial''' n' =
  let helper acc n = if n > 1
                    then helper (acc * n) (n - 1)
                    else acc
  in helper 1 n'
```

Предохранители (Guards)

Просматриваются сверху вниз до первого истинного

```
factorial'' n' = helper 1 n'  
  where helper acc n | n > 1      = helper (acc * n) (n - 1)  
                    | otherwise = acc
```

```
factorial''' n' =  
  let helper acc n | n > 1      = helper (acc * n) (n - 1)  
                  | otherwise = acc  
  in helper 1 n'
```

Конструкция `where` может быть общей для предохранителей

```
f x y | y > z = ...  
      | y == z = ...  
      | y < z = ...  
  where z = x * x
```

- 1 Язык Haskell
- 2 Основы программирования
- 3 Базовые типы**
- 4 Система модулей
- 5 Операторы и сечения

Каждое выражение имеет тип

- Базовые типы:
 - `Bool` — булево значение;
 - `Char` — символ Юникода;
 - `Int` — целое фиксированного размера;
 - `Integer` — целое произвольного размера;
 - `type1 -> type2` — тип функции;
 - `(type1, type2, ..., typeN)` — тип кортежа;
 - `()` — единичный тип, с одной константой `()`;
 - `[type1]` — тип списка с элементами типа `type1`.
- В GHCi для определения типа используют команду `:type`.
- Можно явно указывать тип выражения (`42::Integer`).

Булев тип представляет собой перечисление (enumeration)

Пример

```
data Bool = True | False
```

Здесь `Bool` — *конструктор типа*,
а `True` и `False` — *конструкторы данных*.

Их имена должны начинаться с символа в верхнем регистре!

Можно задавать функции несколькими равенствами:

Пример

```
not      :: Bool -> Bool
not True  = False
not False = True
```

Объявление типа необязательно, но приветствуется.

Пример функции двух аргументов

```
mult :: Integer -> (Integer -> Integer)
mult x1 x2 = x1 * x2
```

Применение последовательно: `mult 2 3 == (mult 2) 3`.
Конструкция `mult 2` — это *частично применённая* функция.

Сессия GHCi

```
*Fp04> :type mult 2
mult 2 :: Integer -> Integer
*Fp04> let foo = mult 2
*Fp04> :type foo
foo :: Integer -> Integer
*Fp04> foo 3
6
```

Возможная реализация комбинатора K на Haskell

```
*Fp04> let k = \x y -> x
*Fp04> :type k
k :: t -> t1 -> t
```

В стрелочный тип входят не конкретные типы (должны начинаться с символа в верхнем регистре), а *переменные типа*.
Можем применять к **любым** типам

Сессия GHCi

```
*Fp04> :type k 'x' False
k 'x' False :: Char
```

Все переменные типа находятся под (неявно подразумеваемым) квантором всеобщности $k :: \text{forall } t \ t1. t -> t1 -> t$

Ограниченная квантификация

Классы типов позволяют наложить специальные ограничения на полиморфный тип

Сессия GHCi

```
*Fp04> :type add  
add :: Num a => a -> a -> a
```

Контекст `Num a` накладывает на тип `a` ограничения: для него должны быть определены операторы сложения, умножения и т.п. `Int` и `Double` — представители класса типов `Num`:

Сессия GHCi

```
*Fp04> add (2::Int) (3::Int)  
5  
*Fp04> add (2.0::Double) (3.0::Double)  
5.0
```

- 1 Язык Haskell
- 2 Основы программирования
- 3 Базовые типы
- 4 Система модулей**
- 5 Операторы и сечения

- Программа состоит из набора модулей.
- Модули позволяют управлять пространствами имён.
- Инкапсуляция через списки экспорта и импорта.

Пример модуля

```
module A (foo, bar) where
import B (f, g, h)
foo = f g
bar = ...
bas = ...
```

- Конфликты имён разрешаются через полные имена

Квалифицированный импорт

```
import qualified B (f, g, h)
foo = B.f B.g
```

Загрузка модулей в GHCi

- Команда `:load` отвечает за загрузку модуля.
- Команда `:module` управляет областью видимости.

Сессия GHCi

```
Prelude> :load Fp04
[1 of 1] Compiling Fp04      ( Fp04.hs, interpreted )
Ok, modules loaded: Fp04.
*Fp04> isUpper 'A'
<interactive>:1:1: Not in scope: 'isUpper'
*Fp04> :module +Data.Char
*Fp04 Data.Char> isUpper 'A'
True
*Fp04 Data.Char> :module -Data.Char
*Fp04>
```

- Модуль `Prelude` всегда в области видимости (пока его явно не выгрузили).

- Hoogle — это Google для Haskell.
- Позволяет осуществлять поиск по API стандартных библиотек.
 - Переходим на <http://www.haskell.org/hoogle/>;
 - вводим, например, `digitToInt`;
 - смотрим описание;
 - можем посмотреть исходный код.

Описание `digitToInt`

```
digitToInt :: Char -> Int
```

Convert a single digit `Char` to the corresponding `Int`. This function fails unless its argument satisfies `isHexDigit`, but recognises both upper and lower-case hexadecimal digits (i.e. `'0'..'9'`, `'a'..'f'`, `'A'..'F'`).

- 1 Язык Haskell
- 2 Основы программирования
- 3 Базовые типы
- 4 Система модулей
- 5 **Операторы и сечения**

Оператор — это комбинация из одного или более символов

Список символов для операторов

! # \$ % & * + . / < = > ? @ \ ^ | - ~ :

Все операторы *бинарные* и *инфиксные*.
Исключение: унарный префиксный минус,
который всегда ссылается на `Prelude.negate`.

Пример: определим оператор для суммы квадратов

```
a *** b = a * a + b * b

-- использование
res = 3 *** 4    --    == 25
```

Инфиксная и префиксная нотация

- Операторы могут определяться и использоваться в префиксном (функциональном) стиле.

Оператор для суммы кубов

```
(**+**) a b = a * a * a + b * b * b
```

```
res1 = (**+**) 2 3 -- == 35
```

```
res2 = 2 **+++ 3 -- == 35
```

- Функции, в свою очередь, могут определяться и использоваться в инфиксном (операторном) стиле.

Функция суммирования (в операторном стиле)

```
x 'plus' y = x + y
```

```
res3 = 2 'plus' 3 -- == 5
```

Чему равны значения выражений?

1 `*** 2 + 3`

1 `*** 2 *** 3`

Чему равны значения выражений?

1 `**` 2 + 3

1 `**` 2 `**` 3

Инфиксные операторы требуют определения

- приоритета (какой оператор из цепочки выполнять первым);
- ассоциативности (какой оператор из цепочки выполнять первым при равном приоритете).

Приоритет и ассоциативность (fixity)

С помощью объявлений `infixl`, `infixr` или `infix` задаётся приоритет и ассоциативность операторов и функций.

Пример:

```
infixl 6  ***, **+**, 'plus'
```

Теперь введённые нами операторы левоассоциативны и имеют тот же приоритет, что и обычный оператор сложения.

Задача: расставьте скобки и вычислите

```
1 *** 2 + 3
```

```
3 + 1 *** 2 * 3
```

Приоритет стандартных операторов

```
infixl 9  !!
infixr 9  .
infixr 8  ^, ^^, **
infixl 7  *, /, 'quot', 'rem', 'div', 'mod'
infixl 6  +, -
infixr 5  ++, :
infix 4   ==, /=, <, <=, >=, >, 'elem', 'notElem'
infixr 3  &&
infixr 2  ||
infixl 1  >>, >>=
infixr 1  =<<
infixr 0  $, $!, 'seq'
```

- В GHCi можно подглядеть, набрав `:info (&&)`.
- Аппликация имеет наивысший (10) приоритет.

Сечения (Sections)

Операторы на самом деле просто функции и, поэтому, допускают частичное применение.

Левое сечение

```
(2 ***) == (***) 2 == \y -> 2 *** y
```

Правое сечение

```
(*** 3) == \x -> x *** 3
```

Наличие скобок при задании сечений обязательно!

Стандартный оператор (\$)

- Оператор (\$) задаёт аппликацию, но с наименьшим возможным приоритетом

Из Prelude

```
infixr 0 $  
($)      :: (a -> b) -> a -> b  
f $ x    = f x
```

Используется для элиминации избыточных скобок:

```
f (g x) == f $ g x
```

```
f (g x (h y)) == f $ g x (h y) == f $ g x $ h y
```

- Из примера ясна причина правоассоциативности.
- (\$) используют также для передачи аппликации в ФВП.

Бесточечный (Pointfree) стиль

В Haskell можно сделать η -редукцию в определении функции.
Если

Пример комбинаторного определения

```
foo x = bar bas x
```

или, что то же самое

Пример определения через λ -выражения

```
foo = \x -> bar bas x
```

то можно написать определение `foo` в *бесточечном стиле*

Пример бесточечного определения

```
foo = bar bas
```