Базовые конструкции и типы в Haskell

Михаил Беляев

7 октября 2015

Синтаксис языка: функции

Обычная функция: имя — это стандартный идентификатор

```
foo :: Integer -> Integer
foo x = x + 2
```

Можно использовать символ «'»

```
foo' :: Integer -> Integer
foo' x = x - 2
```

Синтаксис языка: функции

Имена могут состоять из нелитеральных символов (кроме скобок, кавычек и запятых), но тогда нельзя использовать буквы и цифры

```
(<@???@>) :: Integer -> Integer
(<@???@>) x = x + 2

(<@???@>) 42            -- 44
```

Синтаксис языка: функции

Функции от **двух** параметров можно вызывать в инфиксной форме:

- Для обычных функций: в обратных кавычках

```
divmod x y = (x / y, x % y)
z = 45 `divmod` 16
```

- Для операторных функций: просто убираем скобки

```
(</@%>) x y = (x / y, x % y)
z = 45 </@%> 16
```

Синтаксис языка: приоритетность

- Для инфиксной формы важны *ассоциативность* операции и её *приоритет*

Синтаксис языка: приоритетность

- Для инфиксной формы важны *ассоциативность* операции и её *приоритет*

- Ассоциативность:
  - Левая: `a + b + c === (a + b) + c`
  - Правая: `a += b += c === a += (b += c)`
  - Никакая: операции нельзя использовать в цепочке
- Приоритет: `a + b * c === a + (b * c)`

Синтаксис языка: приоритетность

В haskell нет неявной ассоциативности и неявных приоритетов. Для любой функции всё задаётся вручную:

```
(</@%>) x y = (x / y, x % y)
infixr 7 </@%>
z = 45 </@%> 16
```

Оператор имеет приоритет 7 и правоассоциативен

Задаваыемых приоритетов 10: от 0 до 9. Префиксное применение функции имеет приоритет 10.

Секции

Вспомним, что такое **каррирование**

```
foo x y = x * y + 3
foo 4          -- \ y -> 4 * y + 3
```

Секции

Вспомним, что такое **каррирование**

```
foo x y = x * y + 3
foo 4          -- \ y -> 4 * y + 3
```

Секции делают то же самое, но для операторов:

```
let divBy3 = (/ 3)
divBy3 15              -- 5
let div15By = (15 /)
div15By 5              -- 3
```

Let-bindings

```
factorial x =
    if x <= 1 then 1 else x * factorial (x - 1)
strfact sx =
    let param = read sx in
    show (factorial param)
```

Where-bindings

```
factorial x =
    if x <= 1 then 1 else x * factorial (x - 1)
strfact sx =
    let param = read sx in
    show result
    where result = factorial param
```

Байндинги могут принимать параметры

```
function x =
    let factorial x =
        if x <= 1 then 1 else factorial (x - 1) in
    let param = read x in
    show result
    where result = factorial param
```

Базовые типы данных

```haskell
char :: Char
char = 'a'

integer :: Integer
integer = 42

int :: Int
int = 42
```

Отличие Int и Integer

`Int` — соответствует `int` в C/Java/etc

`Integer` — тип «бесконечной» точности, примерно соответствует `BigInteger` в Java

Списки в Haskell

- Основная структура данных — линейный односвязный список
- Два конструктора — `[]` и `h:t`

```
lst :: [Int]
lst = 0 : (1 : (2 : (3 : (4 : []))))
lst = [1, 2, 3, 4]
lst = [1..4]
```

Кортежи в Haskell

```haskell
pair :: a -> b -> (a, b)
pair x y = (x, y)
triple :: a -> b -> c -> (a, b, c)
triple x y z = (x, y, z)
```

Другие встроенные типы

- `()` (произносится «unit») — пустой тип
- Можно рассматривать как альтернативу `void`
- Имеет одно значение — `()`

```
nothing :: ()
nothing = ()
```

Синтаксис определения своих типов

- Псевдонимы типов

```
type IntList = [Int]
```

- Строка — это просто список символов

```
type String = [Char]
foo :: [Char]
foo = "hello"
```

Синтаксис определения своих типов

- Типы-данные: типы-суммы (sum types)

```
data Term = IntConstant Int
          | Variable String
          | BinaryTerm Term Term
```

Sum types: конструкторы

- Каждый конструктор является функцией:

```
Prelude> data Term =
    IntConstant Int
  | Variable String
  | BinaryTerm Term Term
Prelude> let ic = IntConstant 2
Prelude> :t ic
ic :: Term
Prelude> let vc = Variable "x"
Prelude> :t vc
vc :: Term
Prelude> let bc = BinaryTerm ic vc
Prelude> :t bc
bc :: Term
```

Синтаксис определения своих типов

- Типы-данные: записи (records)

```
data Coordinates = Coordinates{ x :: Int, y :: Int}
```

Синтаксис определения своих типов

- Комбинирование вышеприведённого:

```
data Term = IntConstant{ intValue :: Int }
          | Variable{ varName :: String }
          | BinaryTerm{ lhv :: Term, rhv :: Term }
```

Записи: расширенное понимание

Каждое имя поля в типе `T` с типом `X` генерирует функцию-геттер с типом `T -> X`

```
Prelude> data Coordinates = Coordinates{ x :: Int, y :: Int}
Prelude> :t x
x :: Coordinates -> Int
Prelude> :t y
y :: Coordinates -> Int
Prelude> let coords = Coordinates 2 3
Prelude> x coords
2
Prelide> y coords
3
```

Newtype

Если ваш тип содержит одно поле, то можно вместо ключевого слова `data` использовать слово `newtype`

```haskell
newtype TimeData = TimeData{ ms :: Integer }
```

`newtype` это новый тип (не псевдоним), но компилятор представляет его также, как и тип, который в нём содержится

Параметризованные типы

Типы могут иметь типы-параметры

```
type List a = [a]
data Tree a = Leaf a | Node (Tree a) (Tree a)
leaf42 :: Tree Int
leaf42 = Leaf 42
```

Некоторые типы из стандартной библиотеки: Bool

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

Некоторые типы из стандартной библиотеки: Maybe

```haskell
data Maybe a = Just a | Nothing
maybeInt1 :: Maybe Int
maybeInt1 = Just 42
maybeInt2 :: Maybe Int
maybeInt2 = Nothing
```

Некоторые типы из стандартной библиотеки: Either

```haskell
data Either a b = Left a | Right b
eitherStringOrInt1 :: Either String Integer
eitherStringOrInt1 = Left "Hello"
eitherStringOrInt2 :: Either String Integer
eitherStringOrInt2 = Right 1337
```

Встроенные типы можно было бы реализовать через data

```haskell
data List a = Nil | Cons a (List a)
data Pair a b = Pair a b
data Unit = Unit
```

Написание функций над типами данных

Сравнение с образцом (более подробно — в следующей лекции)

```
data Tree a = Leaf a | Node (Tree a) (Tree a)

contains (Leaf x)            y = (x == y)
contains (Node left right) y = contains left y || contains r
```

Написание функций над типами данных

Сравнение с образцом (более подробно — в следующей лекции)

```
listContains [] x = False
listContains (h: t) x = if h == x
                        then True
                        else listContains t x
listEmpty [] = True
listEmpty _  = False
```

Написание функций над типами данных

Сравнение с образцом можно использовать внутри `let` и `where`

```
pair a b = (a,b)
flip pr = let (k, h) = pr in (h, k)
```