Лекция 7. Введение в Haskell

Функциональное и логическое программирование (осень 2025)

Николай Кудасов

13 октября 2025

институт ispring

Содержание

Введение в Haskell

Вкратце о Haskell

1. Статичеки типизированный язык программирования
2. Принадлежит семейству языков ML (включает OCaml, Standard ML, F#)
3. Имеет очень большую глубину в плане концепций, мы остановимся на следующих
   - безопасность за счёт системы типов
   - алгебраические типы данных и тотальное сопоставление с образцом
   - параметрический полиморфизм
   - нестрогий (ленивый) порядок вычислений и цельное программирование
   - работа с монадическими эффектами

Вкратце о Haskell

1. Статичеки типизированный язык программирования
2. Принадлежит семейству языков ML (включает OCaml, Standard ML, F#)
3. Имеет очень большую глубину в плане концепций, мы остановимся на следующих
   - безопасность за счёт системы типов
   - алгебраические типы данных и тотальное сопоставление с образцом
   - параметрический полиморфизм
   - нестрогий (ленивый) порядок вычислений и цельное программирование
   - работа с монадическими эффектами

Вкратце о Haskell

1. Статичеки типизированный язык программирования
2. Принадлежит семейству языков ML (включает OCaml, Standard ML, F#)
3. Имеет очень большую глубину в плане концепций, мы остановимся на следующих
   - безопасность за счёт системы типов
   - алгебраические типы данных и тотальное сопоставление с образцом
   - параметрический полиморфизм
   - нестрогий (ленивый) порядок вычислений и цельное программирование
   - работа с монадическими эффектами

Пример программы на Haskell

1. Программа — набор *объявлений*.
2. `main` — точка входа в программу.
3. Тела функций и объявлений — *выражения*.
4. Программы с эффектами — *чистые значения*.

```haskell
fivePrimes = take 5 primes

primes = filterPrime [2..]
  where
    filterPrime (p:xs) =
      p : filterPrime
        [ x | x <- xs
        , x `mod` p /= 0 ]

printMany [] = return ()
printMany (x:xs) =
  print x >> printMany xs

main = do
  print fivePrimes
  printMany fivePrimes
```

Объявления

```
1 z = 2 + 2   -- $\text{это объявление, а не присваивание!}
2 z = 5
```

При попытке переопределить переменную, вы получите примерно следующее сообщение от компилятора:

```
<interactive>:3:1: error:
    Multiple declarations of ‘z’
    Declared at: <interactive>:2:1
                 <interactive>:3:1
```

Локальные объявления

```haskell
range = maxValue - minValue
  where
    maxValue = maximum values
    minValue = minimum values

values = [1, 3, 4, 2, 5]
```

Локальные объявления расположены в блоке `where`. Порядок объявлений при этом не имеет значения.

Сигнатуры типов

Haskell не требует явно прописывать типы выражений и объявлений, из-за мощного алгоритма вывода типов:

```
x = length "Hello" + 1
```

Тем не менее, стандартное соглашение заключается в том, что сигнатуры прописываются явно для всех объявлений на верхнем уровне:

```
1 x :: Int
2 x = 2 + 2
```

`x :: Int` читается как *“x имеет тип Int”*.

Сигнатуры типов для локальных объявлений

```haskell
range :: Int
range = maxValue - minValue
  where
    maxValue = maximum values
    minValue = minimum values

values :: [Int]
values = [1, 3, 4, 2, 5]
```

```haskell
range :: Int
range = maxValue - minValue
  where
    maxValue, minValue :: Int
    maxValue = maximum values
    minValue = minimum values

values :: [Int]
values = [1, 3, 4, 2, 5]
```

Типы локальных объявлений обычно очевидны из контекста, поэтому редко выписываются явно.

Стандартные типы в Haskell

Числовые типы (`Int`)

```haskell
n :: Int
n = 42

minInt, maxInt :: Int
minInt = -9223372036854775808
maxInt =  9223372036854775807
```

`Int` — целое число, соответствующее машинному слову. Работает эффективно и достаточен для большинства задач.

Числовые типы (Integer)

```haskell
bigNumber :: Integer
bigNumber = 3^(4^5)
```

```
37339184874102004353295975418486658822540977678373400775063693172207904061726525122999368893880397722046876506543147515810872705459216085858135133698280918731419174859426258093880701995195640428557181804104668128879740292551766801234061729839657473161915238672304623512593489605859058828465479354050593620237654780744273058214452705898875625145281779341335214192074462302751872918543286237573706398548531947641692626381997288700690701389925652429719852769874927419627681106070233371035648 1
```

`Integer` — число произвольного размера (длинная арифметика). Также очень эффективен.

Числовые типы (`Double`)

```haskell
phi :: Double
phi = 1.61803

fourteen :: Double
fourteen = 14
```

`Double` — число с плавающей точкой с двойной точностью. Тоже эффективен.

Литералы

```haskell
intFourteen :: Int
intFourteen = 14

integerFourteen :: Integer
integerFourteen = 14

doubleFourteen :: Double
doubleFourteen = 14
```

Целые литералы — полиморфны. Конкретный тип определяется из контекста.

Операции (+), (*), (-)

```haskell
exampleInt :: Int
exampleInt = 10 + 2 * 37 - 50

exampleInteger :: Integer
exampleInteger = 10 + 2 * 37 - 50

exampleDouble :: Double
exampleDouble = 1 + 2 * 3.7 - 5
-- 3.4000000000000004
```

Сложение, умножение и вычитание работает для любых числовых типов.

Важно: типы аргументов и результата должны быть одинаковыми!

Операции над числами с плавающей точкой

```haskell
one :: Double
one = (sin x)^2 + (cos x)^2
  where
    x = log (sqrt 123)
-- 0.9999999999999999
```

Операции для чисел с плавающей точкой: `sin`, `cos`, `sqrt`, `log`, и т.д.

Операция `(/)` работает для “дробных” чисел (не только для чисел с плавающей точкой).

Целочисленное деление

```haskell
twelve :: Int
twelve = mod 98374 34

thirteen :: Integer
thirteen = 1 + (div 47591 3746)
```

```haskell
twelve :: Int
twelve = 98374 `mod` 34

thirteen :: Integer
thirteen = 1 + 47591 `div` 3746
```

`div` и `mod` соответствуют целочисленному делению и взятию остатка, соответственно.

Обратный апостроф (`) превращает имя функции в инфиксный оператор.

Haskell и приведение типов

```haskell
n :: Int
n = 42

phi :: Double
phi = 1.61803

bad = n + phi
```

При попытке сложить два числа разных типов, Haskell выкинет ошибку проверки типов (во время компиляции):

```
<interactive>:20:11: error:
    • Couldn't match expected type ‘Int’ with actual type ‘Double’
    • In the second argument of ‘(+)’, namely ‘phi’
      In the expression: n + phi
      In an equation for ‘bad’: bad = n + phi
```

Явное приведение типов (fromIntegral)

```haskell
n :: Int
n = 42

phi :: Double
phi = 1.61803

good1 = fromIntegral n + phi
```

`fromIntegral` приводит целое к любому числовому типу.

Явное приведение типов (`fromIntegral` и округление)

```haskell
n :: Int
n = 42

phi :: Double
phi = 1.61803

good1 = fromIntegral n + phi

good2 = n + floor   phi -- округление вниз
good3 = n + ceiling phi -- округление вверх
good4 = n + round   phi -- округление к ближайшему целому
```

`fromIntegral` приводит целое к любому числовому типу.

Также возможно округление вещественного числа.

Объявление функций

При объявлении функций, формальные аргументы вводятся слева от знака равенства (=):

```
1 square x = x * x
2
3 midpoint x y = (x + y) / 2
```

Сигнатуры типов для функций и несколько уравнений

Функции можно определять через несколько уравнений:

```haskell
square :: Double -> Double
square x = x * x

distance :: Double -> Double -> Double
distance x y = abs (x - y)

factorial :: Integer -> Integer
factorial 0 = 1                          -- если аргумент равен 0
factorial n = n * factorial (n - 1)      -- иначе
```

Важно: порядок уравнений в определении функций имеет значение!

Тип Bool

Булевы значения — это либо `True`, либо `False`:

```
otherwise :: Bool
otherwise = True

isSeven :: Int -> Bool
isSeven 7 = True
isSeven _ = False
```

Охранные выражения

Охранные выражения позволяют разбить уравнение на два или более случаев:

```
1  maxF :: Double -> String
2  maxF w
3    | x <= z && y <= z  = "sin"
4    | x <= y            = "cos"
5    | otherwise         = "id"
6    where
7      x = sin w
8      y = cos w
9      z = w
```

Локальные определения — общие для всех случаев в таком составном уравнении.

Символы и строки

Строки — в двойных кавычках:

```
name :: String
name = "Иван Иванов"
```

Символы — в одинарных кавычках:

```
sigma :: Char
sigma = 'Σ'
```

Кортежи

Кортежи — это безымянные структуры с фиксированным количеством полей фиксированных типов:

```haskell
tuple1 :: (Int, String, Bool)
tuple1 = (123, "Привет", False)

tuple2 :: (Int, String, Bool)
tuple2 = (678, "Пока", True)

student1 = ("Иван Иванов", 4)
student2 = ("Пётр Петров", 5)

point3D = (1.0, -2.2, 5.3)
vector3D = (3.0, 0.0, 4.0)
```

Функции на кортежах

Чтобы определить функцию на кортежах, мы разбираем входной кортеж на компоненты через сопоставление с образцом:

```
1 norm :: (Double, Double, Double) -> Double
2 norm (x, y, z) = sqrt (x^2 + y^2 + z^2)
```

1. Функция `norm` принимает **один** аргумент!
2. **Сопоставление с образцом** разбирает структуру кортежа на 3 компоненты.
3. Переменные `x`, `y` и `z` **связываются** со значениями компонент.

Перерыв

Списки

Списки

Списки записываются в квадратных скобках, с элементами через запятую:

```haskell
list1 :: [Int]
list1 = [1, 2, 3]

list2 :: [String]
list2 = ["Привет", "мир"]

list3 :: [Bool]
list3 = [False, True]

list4 :: [(String, Int)]
list4 = [("Иван Иванов", 4), ("Пётр Петров", 5)]

list5 :: [[Int]]
list5 = [[1], [2, 3], []]
```

Однородные списки (пример 1 из 2)

Списки должны быть однородными (содержать элементы одного типа):

```
1 bad = [1, "два"]
```

```
<interactive>:21:8: error:
    • No instance for (Num [Char]) arising from the literal ‘1’
    • In the expression: 1
      In the expression: [1, "два"]
      In an equation for ‘bad’: bad = [1, "два"]
```

Ошибка говорит о том, что один из элементов списка является числом (литерал `1`), при этом элементы списка — строки (`String`, который синоним для списка `Char`). Но `String` не является числом (не входит в класс типов `Num`)!

Однородные списки (пример 1 из 2)

Списки должны быть однородными (содержать элементы одного типа):

```
1 bad = [False, "два"]
```

```
<interactive>:22:15: error:
    • Couldn't match expected type ‘Bool’ with actual type ‘[Char]’
    • In the expression: "два"
      In the expression: [False, "два"]
      In an equation for ‘bad’: bad = [False, "два"]
```

Ошибка говорит о том, что один из элементов списка является булевым значением, при этом есть элементы списка — строки (`String`, который синоним для списка `Char`). Но `Bool` и `[Char]` — разные типы!

Построение списков

Списки либо пустые (`[]`), либо непустые и состоят из головы и хвоста (`:`):

```haskell
list0 = []
list1 = 1 : list0     -- [1]
list2 = 2 : list1     -- [2, 1]
list3 = 3 : list2     -- [3, 2, 1]

list4 = 4 : (3 : (2 : (1 : []))) -- [4, 3, 2, 1]
```

Квадратные скобки — синтаксический сахар!

Сопоставление с образцом для списков

```haskell
prettyList :: [String] -> String
prettyList [] = "ничего"
prettyList [x] = x
prettyList [x, y] = x ++ " и " ++ y
prettyList (x:xs) = x ++ ", " ++ prettyList xs
prettyList (x:y:xs) = x ++ ", " ++ y ++ ", " ++ prettyList xs
```

```
>>> putStrLn ("Моя семья — это "
      ++ prettyList ["мама","папа","брат"] ++ "!")
Моя семья — это мама, папа и брат!
```

1. Последняя строка — мёртвый код.
2. Образец `[x]` — сахар для `(x : [])`.
3. Образец `[x, y]` — сахар для `(x : (y : []))`.
4. Образец `(x:y:xs)` — то же самое что `(x:(y:xs))`.

Фунции над списками

В большинстве случаев достаточно разобрать два случая:

```
f []     = -- что вернуть, если входной список пуст
f (x:xs) = -- что вернуть, если список состоит из головы и хвоста
```

Например, так можно вычислить сумму списка:

```
sum :: [Double] -> Double
sum []     = 0
sum (x:xs) = x + sum xs
```

Пользовательские типы в Haskell

Синонимы типов

Синоним типа — это просто другое имя для уже существующего типа:

```haskell
-- | Точка на плоскости.
type Point = (Double, Double)

-- | Температура в градусах Цельсия.
type Celsius = Double

-- | Температура в градусах Фаренгейта.
type Fahrenheit = Double

celsiusToFahrenheit :: Celsius -> Fahrenheit
celsiusToFahrenheit celsius = (celsius * 1.8) + 32
```

Типы-произведения

Типы-произведения — это по записи с безымянными полями (или именованные кортежи):

```haskell
-- | Двумерный вектор.
data Vector = MkVector Double Double
```

1. `Vector` — это новый тип.
2. `MkVector` — это **конструктор значений**, принимающий два аргумента типа `Double`.

```haskell
v :: Vector
v = MkVector 3 4

norm :: Vector -> Double
norm (MkVector x y) = sqrt (x^2 + y^2)
```

Пространства имён

Значение и типы живут в разных *пространствах имён*. Поэтому часто в качестве имени конструктора значений используют то же имя, что и имя типа:

```haskell
-- | Двумерный вектор.
data Vector = Vector Double Double

v :: Vector
v = Vector 3 4

norm :: Vector -> Double
norm (Vector x y) = sqrt (x^2 + y^2)
```

Типы-обёртки

Синонимы типов улучшают читаемость, но не безопасность. Чтобы случайно не перепутать, например, единицы измерения, полезно использовать типы-обёртки:

```haskell
newtype Celsius = Celsius Double
newtype Fahrenheit = Fahrenheit Double

celsiusToFahrenheit :: Celsius -> Fahrenheit
celsiusToFahrenheit celsius
  = (celsius * 1.8) + 32
```

Типы-обёртки

Синонимы типов улучшают читаемость, но не безопасность. Чтобы случайно не перепутать, например, единицы измерения, полезно использовать типы-обёртки:

```haskell
newtype Celsius = Celsius Double
newtype Fahrenheit = Fahrenheit Double

celsiusToFahrenheit :: Celsius -> Fahrenheit
celsiusToFahrenheit celsius
  = (celsius * 1.8) + 32
```

```
<interactive>:30:5: error:
    • Couldn't match expected type ‘Fahrenheit’
                  with actual type ‘Celsius’
    • In the expression: (celsius * 1.8) + 32
      In an equation for ‘celsiusToFahrenheit32
```

Типы-обёртки

Синонимы типов улучшают читаемость, но не безопасность. Чтобы случайно не перепутать, например, единицы измерения, полезно использовать типы-обёртки:

```haskell
newtype Celsius = Celsius Double
newtype Fahrenheit = Fahrenheit Double

celsiusToFahrenheit :: Celsius -> Fahrenheit
celsiusToFahrenheit celsius
  = Fahrenheit ((celsius * 1.8) + 32)
```

Типы-обёртки

Синонимы типов улучшают читаемость, но не безопасность. Чтобы случайно не перепутать, например, единицы измерения, полезно использовать типы-обёртки:

```haskell
newtype Celsius = Celsius Double
newtype Fahrenheit = Fahrenheit Double

celsiusToFahrenheit :: Celsius -> Fahrenheit
celsiusToFahrenheit celsius
  = Fahrenheit ((celsius * 1.8) + 32)
```

```
<interactive>:39:17: error:
    • Couldn't match expected type ‘Double’ with actual type ‘Celsius’
    • In the first argument of ‘Fahrenheit’, namely
        ‘((celsius * 1.8) + 32)'
      In the expression: Fahrenheit ((celsius * 1.8) + 32)
      In an equation for ‘celsiusToFahrenheit’:
          celsiusToFahrenheit celsius = Fahrenheit ((celsius * 1.8) + 32)
```

Типы-обёртки

Синонимы типов улучшают читаемость, но не безопасность. Чтобы случайно не перепутать, например, единицы измерения, полезно использовать типы-обёртки:

```haskell
newtype Celsius = Celsius Double
newtype Fahrenheit = Fahrenheit Double

celsiusToFahrenheit :: Celsius -> Fahrenheit
celsiusToFahrenheit (Celsius celsius)
  = Fahrenheit ((celsius * 1.8) + 32)
```

Спасибо за внимание!