CS212. Парадигмы и технологии программирования: часть 1, функциональное программирование

Лекция 1. Введение в функциональное программирование на языке Haskell

---

В. Н. Брагилевский

14 февраля 2019 г.

Направление «Фундаментальная информатика и информационные технологии»
Институт математики, механики и компьютерных наук имени И. И. Воровича
Южный федеральный университет

Функциональные языки программирования

- Lisp и его наследники (Common Lisp, Scheme, Clojure)
- Haskell
- ML и его диалекты (Standard ML, OCaml)
- F#
- Scala
- Erlang
- ...

Важнейшие черты функционального стиля

- Всякое вычисление трактуется как вычисление значения математической функции.
- Отсутствует изменяемое состояние (нет оператора присваивания, переменных, циклов).
- Имеется богатый инструментарий для работы с функциями (различные способы определения функций, функции высших порядков).

Язык Haskell и среда программирования

Язык программирования Haskell

Определение из википедии

Haskell — это чисто функциональный язык общего назначения с нестрогой семантикой, сильной статической типизацией и выводом типов.

Haskell Curry (1900–1982)

Создатели языка (1990)

Simon Peyton Jones, Lennart Augustsson, Dave Barton, Brian Boutel, Warren Burton, Joseph Fasel, Kevin Hammond, Ralf Hinze, Paul Hudak, John Hughes, Thomas Johnsson, Mark Jones, John Launchbury, Erik Meijer, John Peterson, Alastair Reid, Colin Runciman, Philip Wadler.

Стандарты и реализации

Версии и стандарты

- Haskell 1.0, 1.1, 1.2, 1.3, 1.4 (1990 — 1997)
- Haskell 98
- Haskell 2010
- Haskell 2020 (планируется)

Реализации

- **GHC (Glasgow Haskell Compiler)** + расширения
- UHC (Utrecht Haskell Compiler)
- YHC (York Haskell Compiler)
- LHC (LLVM Haskell Compiler)
- Hugs (только интерпретатор)

Haskell Platform: `http://www.haskell.org/platform/`

- Компилятор GHC и командный интерпретатор GHCi (8.2.2).
- Cabal — система сборки.
- Stack — система разработки проектов на Haskell.
- Haddock — система документирования.
- Средства профилирования кода.
- Избранные библиотеки.

Пример 1: сумма чисел от 1 до n

Сумма чисел

Постановка задачи

Дано число n. Вычислить сумму:

$$1 + 2 + \cdots + n.$$

Идеи решения

- Решение 1 (рекурсивное): сумма(n) = n + сумма($n - 1$).
- Решение 2 (итерационное): заводим аккумулятор с нулевым значением и последовательно прибавляем к нему числа от 1 до n.

Решение 1

Код

```
sumN 0 = 0
sumN n = n + sumN (n-1)
```

Вычисление

```
sumN 5 = 5 + sumN 4
       = 5 + (4 + sumN 3)
       = 5 + (4 + (3 + sumN 2))
       = 5 + (4 + (3 + (2 + sumN 1)))
       = 5 + (4 + (3 + (2 + (1 + sumN 0))))
       = 5 + (4 + (3 + (2 + (1 + 0))))
       = 5 + (4 + (3 + (2 +
       = 5 + (4 + (3 + 3))
       = 5 + (4 + 6)
       = 5 + 10 = 15
```

NB!

Рекурсивный вычислительный процесс.

Решение 2

Код

```
sumN' 0 = 0
sumN' n = go 0 n
  where
    go s 0 = s
    go s i = go (s+i) (i-1)
```

Вычисление

```
sumN' 5 = go 0  5
        = go 5  4
        = go 9  3
        = go 12 2
        = go 14 1
        = go 15 0 = 15
```

NB!

Рекурсия в коде превратилась в итерационный вычислительный процесс.

Что там с типами?

```
ghci> :type sumN
sumN :: (Eq a, Num a) => a -> a
ghci> :type sumN'
sumN' :: (Eq a, Num a) => a -> a
```

Коротко о типах

Типы: основные понятия

- Любое выражение имеет тип (Int, Integer, Double, Bool, Char).
- В языке Haskell используется сильная статическая типизация (strong static typing) с выводом типов (type inference).
- Для справки:
  - Паскаль, Java — сильная статическая;
  - C — слабая статическая;
  - PHP, Javascript — слабая динамическая;
  - Python, Ruby — сильная динамическая.
- Замечание: любая простая классификация условна.

Определение типа в GHCi

```
ghci> :t 'a'
'a' :: Char
ghci> :t True
True :: Bool
ghci> :t 1729
1729 :: Num a => a
ghci> :t 3.1416
3.1416 :: Fractional a => a
ghci> :t max
max :: Ord a => a -> a -> a
ghci> :t succ
succ :: Enum a => a -> a
```

Типы и классы типов

- Значения имеют тип.
- Тип определяет множество значений.
- Класс типов определяет набор функций, которые можно вызывать для значений.
- Типы могут принадлежать классам типов (иметь экземпляр класса).

Некоторые классы типов

- `Eq` — проверка на равенство (==) и неравенство (/=)
- `Ord` — упорядочение (меньше, больше)
- `Enum` — перечисление (следующий, предыдущий)
- `Num` — общие числовые операции (сложение, вычитание)
- `Integral` — целочисленные операции и функции (div, mod)
- `Fractional` — операции и функции для дробных чисел (/)
- `Floating` — операции и функции для вещественных чисел (sin)
- `Show` — преобразование значения типа в строку (show)

Пример: классы, которым принадлежит тип Integer

`Eq`, `Ord`, `Num`, `Enum`, `Integral`, `Show`

Пример: класс типов Eq

```
ghci> :t (==)
(==) :: Eq a => a -> a -> Bool
ghci> :t (/=)
(/=) :: Eq a => a -> a -> Bool
ghci> 5 == 5
True
ghci> False == True
False
ghci> 'x' /= 'y'
True
ghci> :info Eq
...
```

Пример: класс типов Show

```
ghci> :t show
show :: Show a => a -> String
ghci> show 42
"42"
ghci> show pi
"3.141592653589793"
ghci> show True
"True"
```

Классы типов реализуют ограниченный полиморфизм:

```haskell
sayTwice :: Show a => a -> String
sayTwice w = show w ++ "-" ++ show w
```

Простейшие структуры данных: кортежи и списки

Кортеж

```
type SomeData = (Double, Integer, Bool)
```

Список

```
type Ages = [Integer]
```

- Гомогенные (список) и гетерогенные (кортеж) структуры данных.
- Списки имеют рекурсивное определение:
  список — это голова (элемент) и хвост (список),
  список может быть пустым ([]).

Пример 1: решение на списках

```haskell
sum' :: Num a => [a] -> a
sum' [] = 0
sum' (x:xs) = x + sum' xs

sumN'' n = sum' [1..n]
```

NB!

- Сопоставление с образцами для списка.
- Рекурсия по структуре списка.
- Генератор списка.

Пример: среднее арифметическое элементов списка

Постановка задачи

Дан список вещественных чисел. Найти их среднее арифметическое.

```haskell
mean :: [Double] -> Double
mean xs = sum' xs / length xs
```

Сообщение об ошибке

```
ex1.hs:5:11: error:
    • No instance for (Fractional Int) arising
      from a use of ‘/’
    • In the expression: sum' xs / length xs
      In an equation for ‘mean’:
        mean xs = sum' xs / length xs
```

```haskell
sum' :: Num a => [a] -> a
length :: [a] -> Int

mean :: [Double] -> Double
mean xs = sum' xs / length xs
```

Решение

```haskell
mean xs = sum' xs / fromIntegral (length xs)
```

```haskell
fromIntegral :: (Num b, Integral a) => a -> b
```

Пример: рекурсивное построение списка

Постановка задачи

Дан список целых чисел. Увеличить все его элементы на единицу.

```
incList :: [Integer] -> [Integer]
incList [] = []
incList (x:xs) = (x+1) : incList xs
```

NB!

- Сопоставление с образцами для списка.
- Рекурсия по структуре списка.
- Операция `(:)` — конструктор списка.

Пример 2: вычисление квадратного корня

Вычисление квадратного корня

Постановка задачи

Вычислить квадратный корень из заданного положительного вещественного числа.

Дано: $y > 0$. Найти $x (> 0) : \ x^2 = y$.

Идея решения (метод Ньютона)

Если x — некоторое приближение к $\sqrt{y}$, то более точное приближение можно вычислить по формуле:

$$x' = \frac{x + y/x}{2}$$

Можно выбрать произвольное начальное приближение, а затем улучшать его, пока оно не окажется достаточно хорошим.

Решение

```haskell
abs' a = if a > 0 then a else -a
sqr' a = a * a
average a b = (a + b) / 2
eps = 0.00000000001

sqrt' y = go 1
  where
    goodEnough x = abs' (sqr' x - y) < eps
    improve x = average x (y/x)
    go guess
        | goodEnough guess = guess
        | otherwise = go (improve guess)
```

Решение на Common Lisp

```
(defun abs1 (a) (if (< a 0) (- a) a))
(defun sqr (a) (* a a))
(defun average (a b) (/ (+ a b) 2))
(defconstant eps 0.00000000001)

(defun sqrt1 (y)
  (labels (
    (goodEnough (x) (< (abs1 (- (sqr x) y) ) eps))
    (improve (x) (average x (/ y x)))
    (go (guess) (
      if (goodEnough guess)
         guess
         (go (improve guess)))))
    (go 1)))
```

Решение на Scala (1)

```scala
def abs(a: Double): Double = {
  if (a > 0) a else -a
}
def sqr(a: Double): Double = {a * a}
def average(a: Double, b: Double): Double = {
  (a + b) / 2
}
def eps = 0.00000000001
```

Решение на Scala (2)

```
def sqrt1(y: Double): Double = {
  def goodEnough(x: Double): Boolean = {
    abs(sqr(x) - y) < eps
  }
  def improve(x: Double): Double = {
    average(x, y/x)
  }
  def sqrtIter(guess: Double): Double = {
    if (goodEnough(guess))
      guess
    else
      sqrtIter(improve(guess))
  }
  sqrtIter(1)
}
```

Пример 3: вычисление кубического корня

Вычисление кубического корня

Постановка задачи

Вычислить кубический корень из заданного положительного вещественного числа.

Дано: $y > 0$. Найти $x (> 0) : \ x^3 = y$.

Идея решения (метод Ньютона)

Если x — некоторое приближение к $\sqrt[3]{y}$, то более точное приближение можно вычислить по формуле:

$$x' = \frac{2x + y/x^2}{3}$$

Можно выбрать произвольное начальное приближение, а затем улучшать его, пока оно не окажется достаточно хорошим.

Идея

Вычисление квадратного корня

```
abs' a = if a > 0 then a else -a
sqr' a = a * a
average a b = (a + b) / 2
eps = 0.00000000001

sqrt' y = go 1
  where
    goodEnough x = abs' (sqr' x - y) < eps
    improve x = average x (y/x)
    go guess
        | goodEnough guess = guess
        | otherwise = go (improve guess)
```

NB!

Функцию improve и возведение в квадрат нужно параметризовать!

Функции высшего порядка

Функции высшего порядка — кто они?

Определение (Википедия)

Функция высшего порядка (higher-order function) — функция, принимающая функции в качестве аргументов или возвращающая функцию в качестве результата.

Пример

```
twice :: (a -> a) -> a -> a
twice f x = f (f x)
```

Пример вызова

```
ghci> twice sqrt 16
2.0
```

Функция max и частичное применение

```
ghci> :t max
max :: Ord a => a -> a -> a
ghci> max 4 5
5
ghci> (max 4) 5
5
```

- Вопрос: каков тип выражения max 4?

```
ghci> :t max 4
max 4 :: (Ord a, Num a) => a -> a
```

Каррированные функции и частичное применение

- Все функции в языке Haskell каррированы, т.е. они могут принимать только один параметр.
- Функции с несколькими параметрами при передаче им одного параметра возвращают новую функцию от меньшего числа параметров.
- Вызов функции с недостаточным числом параметров называется частичным применением (partial application).

Пример

```haskell
multThree :: Int -> Int -> Int -> Int
multThree x y z = x * y * z
```

```haskell
multThree :: Int -> (Int -> (Int -> Int))
```

```
ghci> let multTwoWithNine = multThree 9
ghci> multTwoWithNine 2 3
54
```

Композиция функций

Композиция функций в математике

$$(f \circ g)(x) = f(g(x))$$

Композиция функций в Haskell

```
ghci> (negate . abs) (-5)
-5
ghci> (negate . abs) 5
-5
ghci> negate (abs 5)
-5
ghci> :t (negate . abs)
(negate . abs) :: Num c => c -> c
```

Сечения арифметических операций

```
ghci> 3 + 5
8
ghci> (3+) 5
8
ghci> (+5) 3
8
ghci> :t (3+)
(3+) :: Num a => a -> a
ghci> :t (+5)
(+5) :: Num a => a -> a
```

Определение функции «на лету»

```
ghci> twice (+3) 1
7
```

Сечения арифметических операций: вычитание

```
ghci> :t (5-)
(5-) :: Num a => a -> a
ghci> :t (-5)
(-5) :: Num a => a
ghci> :t (subtract 5)
(subtract 5) :: Num a => a -> a
```

```
ghci> (subtract 5) 47
42
ghci> (47-) 5
42
```

Сечения функций

```
ghci> :t elem
elem :: Eq a => a -> [a] -> Bool
```

```
isUpperAlpha :: Char -> Bool
isUpperAlpha = (`elem` ['A'..'Z'])
```

```
ghci> isUpperAlpha 'X'
True
ghci> isUpperAlpha 'a'
False
```

Пример

Задача

Комиссия за выполнение платежа в банке «Край-Финанс» составляет 1% от суммы, причём минимальный размер комиссии — 30 рублей. Вычислить размер комиссии по заданной сумме платежа.

```
comission :: Double -> Double
comission = max 30 . (* 0.01)
```

```
ghci> comission 1970
30.0
ghci> comission 4570
45.7
```

Способы создания функций «на лету»

- Частичное применение
- Композиция
- Сечения
- ???

Анонимные функции

```
ghci> (\x -> x * x + x) 5
30
ghci> (\xs -> 2 * length xs) [1,3,5]
6
ghci> (\(x:xs) -> (x, length xs + 1)) [1,1,1,1]
(1,4)
ghci> (\x y -> x^2 + y^2) 2 3
13
ghci> (\x -> if odd x then 1 else 0) 5
1
```

```
ghci> :t (\x y -> x^2 + y^2)
(\x y -> x^2 + y^2) :: Num a => a -> a -> a
```

Происхождение идеи: λ-функции (Алонсо Чёрч)

- $\lambda x.x+1$ — это функция, прибавляющая единицу к своему аргументу ($f(x) = x+1$, $x \mapsto x+1$);
- $\lambda x.\lambda y.x+y$ — это функция, складывающая два своих аргумента ($g(x,y) = x+y$, $(x,y) \mapsto x+y$).

Композиция как ФВП

```haskell
(.) :: (b -> c) -> (a -> b) -> a -> c
f . g = \x -> f (g x)
```

Функция ($)

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

infixr 0 $
```

```
ghci> sin 4 + 5
4.243197504692072
ghci> sin (4 + 5)
0.4121184852417566
ghci> sin $ 4 + 5
0.4121184852417566
ghci> not $ odd $ 3 + 7
True
```

Композиция vs ($)

```haskell
f = ceiling . negate . tan . cos . max 50

f x = ceiling (negate (tan (cos (max 50 x))))

f x = ceiling $ negate $ tan $ cos $ max 50 x

f x = ceiling . negate . tan . cos . max 50 $ x
```

Пример

```
ghci> f 42
-1
```

Возвращение к примеру 3

```haskell
newton :: (Double -> Double -> Double) -- improve
          -> (Double -> Double) -- проверка
          -> Double -> Double -- аргумент и результат

newton improve check y = go 1
  where
    goodEnough x = abs' (check x - y) < eps
    go guess
        | goodEnough guess = guess
        | otherwise = go (improve guess y)

sqrt' = newton (\x y -> average x (y/x)) (^2)
cbrt = newton (\x y -> (2*x+y/(x^2))/3) (^3)
```

Итоги

- Язык Haskell.
- Функциональное программирование — без присваиваний и циклов.
- Рекурсивные и итерационные вычислительные процессы.
- Типы и классы типов.
- Простейшие структуры данных: списки и кортежи.
- Функции высших порядков и определение функций «на лету».