Haskell

- Лаконичность
- Строгая типизация
- Модульность
- Функции — объекты первого порядка
- Отсутствие побочных эффектов (взрыв)
- Ленивые вычисления

Элементарщина

```
> 2 + 2
4
> 6 / 5
1.2
> 6 / -5
Precedence parsing error
        cannot mix `/' [infixl 7] and prefix `-'
[infixl 6] in the same infix expression
> 6 / (-5)
-1.2
   ---
> True || False
True
> True && False
False
> not False
True
> 2 * 2 == 4
True
> 2 * 2 /= 5
True
> 2 == “два” ?
```

Это вам не js и не python

<interactive>:56:1:

No instance for (Num [Char])

arising from the literal `2'

Possible fix: add an instance declaration for (Num [Char])

In the first argument of `(==)', namely `2'

In the expression: 2 == "sad"

In an equation for `it': it = 2 == "sad"

Типы (:set +t)

Всегда с большой буквы

- Integer, Int
- Float, Double
- Bool (True, False) (алгебраический тип)
- Char

Типы (:set +t)

```
> :set +t
> 42
42
it :: Integer

> 3.1415

> True

> :t 42
42 :: Num a => a (параметрический, специальный (ad hoc) полиморфизм)

42 + 1.2
it :: Double

> :t 42 + 8.2
42 + 8.2 :: Fractional a => a
```

Функции

```
> succ 9
10
> max 2.5 5.6
5.6
> 2 + max 2.5 5.6 + min 3.4 (succ 4)
11.0
> 2 + (max 2.5 5.6) + (min 3.4 (succ 4))
```

Ассоциативность слева,справа

```
2 ** 2 ** 3 / (2 ** 2) ** 3/  2 ** (2 ** 3)
2 - 2 - 2  ?
```

Напишите выражение, высчитывающее минимальное число среди 10, 2, 5, -1, 99

Опять факториал

lab1.hs

```
fac 0 = 1
fac n = n * fac (n-1)
```

```
> ghci
> :load ~lab1.hs
> fac 6
720
> let a = 20
> fac a
2432902008176640000
```

Опять факториал

Определение функции похоже на математическое определение факториала

- Функциональное программирование имеет очень четкую математическую основу
- Рассуждение о программах: доказательство корректности, …

Определение последовательности действий – рекурсивно

- При умелом программировании не ведет к падению эффективности (компилятор сводит к итерации)

Отсутствует оператор присваивания

- Определения функций, (=) имеет другую семантику – связывание имен
- Будучи один раз связанным, имя не может менять свое значение (в рамках области видимости)
- Следствие: нет побочных эффектов
- Раз в императивной программе 90% - это операторы присваивания, то функциональные программы на 90% короче!

Простые функции

```
succ' n = n + 1

> succ' 9
10

square x = x * x

> square 4
16

squareIfLess5 = if x < 5 then square x else x

> squareIfLess5 3
9
> squareIfLess5 7
7
```

Работа со строками

```
-- строка – это список символов

> "Марьванна"
"Марьванна"
it :: [Char]

> 'М':"арьванна"
> ['М','а','р','ь','в','а','н','н','а']
-- про списки ниже!

hello s = "Hello, " ++ s ++ "!"
> hello "Марьванна"
"Hello, \1052\1072\1088\1100\1074\1072\1085\1085\1072!"
```

Типы функций

```
> :t hello
hello :: [Char] -> [Char]
```

```
> :t fac
fac :: (Eq a, Num a) => a -> a
```

ограничение - контекст

```
> :t max
max :: Ord a => a -> a -> a
```

```
> :t squareIfLess5
squareIfLess5 :: (Num a, Ord a) => a -> a
```

Префикс, суффикс, инфикс

```
(sumWithSquareOf) x y = x + square y

> 3 `sumWithSquareOf` 5
28
> (sumWithSquareOf) 3 5
28

import Prelude hiding ((+))
x + y = x - y
> 8 + 3
5

> (-) 23 13
10
> :t (-)
(-) :: Num a => a -> a -> a

Пригодится для ФВП
```

Списки

```
> let lostNumbers = [4,8,15,16,23,42]

> let cards = [3,7,12]

> let mix = lostNumbers ++ cards

> mix
[4,8,15,16,23,42,3,7,12]

> let lostNumbers = [13,666,1488]

> mix ?

> [1, "Гарри Поттер", 16.5] ?
```

Списки

```
> []
it :: [a]

> [1, 13, 666]
it :: [Integer]

> [[1, 2], [3, 4]]
it :: [[Integer]]

> [[1, 2], [3, 4]] ++ [[5, 6]]
it :: [[Integer]]

> [[], []]
it :: [[a]]

> [1, [2,3,4]]
Без вопросов, ничего дельного
```

Оператор пары

```
> 3 : mix
[3,4,8,15,16,23,42,3,7,12]

> [3] : [1, 3, 4]
фигвам
> [3] : [[5, 6]]
it :: [[Integer]]

> mix!!5 -- получение элемента
42

> 13 : [] ?
```

Mix самостоятельно, м?

Работа со списком

Список – это

1). Конструктор списка (пустой список): []

2). Набор элементов одного типа

3). Оператор пары

1 : 2 : 3 : 4 : 5 : []

Всё остальное – синтаксический сахар

Работа со списком

```
> head [1..5] -- [1, 2, 3, 4, 5]
1
> head []
*** Exception: Prelude.head: empty list
> head [13] ?

> tail [1..5]
[2, 3, 4, 5]
> last [1..5]
5
> init [1..5]
[1, 2, 3, 4]
> length [2..6]
5
> null [1..5]
False
> null []
True
```

Работа со списком

```
null' x = if (x=[]) then True else False
```

или

```
null' [] = True
null' _ = False

>:t null'
null :: [a] -> Bool

> take 4 [1..10]
[1, 2, 3, 4]
> take 4 [1..]
[1, 2, 3, 4]
> [1..] ?
```

Работа со списком

```
> drop 4 [1..10]
[5, 6, 7, 8, 9, 10]
> take 4 (drop 7 "Hello, dude!")
"dude"
> drop 666 [1..10]
[]
> drop 4 [1..] ?

> minimum [10, 2, 5, -1, 99]
-1
> sum [1..10]
55
> product [1..10]
3628800
hmmm
```

Работа со списком

```
fac n = product [1..n]
> fac 6
720

> elem 4 [1..10]
True
> elem 666 [1..10]
False
> 'o' `elem` "balloooon"
True
```

Простая генерация

```
> [1,3..20]
[1,3,5,7,9,11,13,15,17,19]
> ['a'..'z']
"abcdefghijklmnopqrstuvwxyz"
> ['a'..] ?
> [0.1, 0.3 .. 1]
[0.1,0.3,0.5,0.7,0.8999999999999999,1.0999999999999999]
> take 11 (cycle "LOL ")
"LOL LOL LOL"
> take 10 (repeat 5)
[5,5,5,5,5,5,5,5,5,5]
> take 10 [5,5..]
[5,5,5,5,5,5,5,5,5,5]
> replicate 10 5
[5,5,5,5,5,5,5,5,5,5]
replicate' how what = take how (repeat what)
```

Генерация списков

```
> [ a*a | a <- [1..10] ]
[1,4,9,16,25,36,49,64,81,100]
> [ [a,b] | a <- [1..3], b <- [3..7] ]
[[1,3],[1,4],[1,5],[1,6],[1,7],[2,3],[2,4],[2,5],[2,6],
[2,7],[3,3],[3,4],[3,5],[3,6],[3,7]]
> [ [a,b] | a <- [1..3], b <- [3..7], 5 == a + b ]
[[1,4],[2,3]]
boomBangs xs = [ if x < 10 then "BOOM!" else "BANG!" | x <-
xs, odd x]
> boomBangs [7..13]
["BOOM!","BOOM!","BANG!","BANG!"]
-- лень
> let a = [1..100000000000]
-- и.. ничего
> let b = [ div 10 x | x <- [5,4..0] ]
> take 5 b
> take 6 b
```

Генерация списков

- Теорема ферма

  let rightTriangles = [[a,b,c] | c ← [1..9], b ← [1..c], a ← [1..b], a^2 + b^2 == c^2]

  let rightTriangles = [(a,b,c) | c ← [1..9], b ← [1..c], a ← [1..b], a^2 + b^2 == c^2]

Кортежи

```
> (1,2)
(1,2)
> ("Гарри Поттер", 13, 2.5)
("Гарри Поттер", 13, 2.5)
> fst (1,2)
1
> snd (1,2)
2
> fst (1,2,3) ?
> let (a,b,c) = (1,2,3) (сопоставление с образцом, 3с. ниже)
> b
2
> [("Гарри Поттер", 13, 2.5),("Колобок", 1, 55.5)]
```

Синонимы типов

```
> type String = [Char]

> type Name = String
> type Person = (Name, Integer, Float)
> let a = ("ГП", 12, 3.4)::Person
> :t a
a :: Person
> :t [a, ("ГГ", 19, 0.2)]
[a,("ГГ", 19, 0.2)] :: [Person]

Больше — создание новых типов
```

Задание

- Опишите функцию, которая для данного числа n возвращает список из всех строк длины n, состоящих из чисел 1,2,3. причем сумма двух соседних чисел не больше 4. Например, при n=2 функция должна вернуть список [[1,1], [1,2], [1,3], [2,1], [2,2], [3,1]].
- Опишите функцию, которая для данного числа n создает список из всех попарных сумм чисел от 1 до n. (Т.е. [1+1, 1+2, 1+3, ..., 1+n, 2+1, 2+2, ..., n+n] - всего n*n элементов)

Сопоставление с образцом

```
sayWhat :: Int -> String
sayWhat 1 = "место встречи изменить нельзя"
sayWhat 2 = "вторник четвёртая пара"
sayWhat 3 = "вторник пятая пара"
sayWhat 4 = "я ничего не знаю"
sayWhat 5 = "ненавижу хаскель, заберите меня отсюда"

> sayWhat 3

Чего не хватает?
Non-exhaustive patterns in function sayWhat
```

Сопоставление с образцом

```
sayWhat :: Int -> String  -- кстати, тип можно не писать!
sayWhat 1 = "место встречи изменить нельзя"
sayWhat 2 = "вторник четвёртая пара"
sayWhat 3 = "вторник пятая пара"
sayWhat 4 = "я ничего не знаю"
sayWhat 5 = "ненавижу хаскель, заберите меня отсюда"
sayWhat _ = "кто здесь?"

> sayWhat 3
```

Сопоставление с образцом

```
sayWhat :: Int -> String
sayWhat 1 = "место встречи изменить нельзя"
sayWhat 2 = "вторник четвёртая пара"
sayWhat _ = "кто здесь?"
sayWhat 3 = "вторник пятая пара"
sayWhat 4 = "я ничего не знаю"
sayWhat 5 = "ненавижу хаскель, заберите меня отсюда"

> sayWhat 3
> sayWhat 1

Pattern match(es) are overlapped
```

Сопоставление кортежей

```
opinion :: (Int, String) ->String
opinion man =
  "Я, " ++ snd man ++ " считаю \"" ++ sayWhat (fst man) ++
"\""
-- обратите внимание на скобки

> opinion (5, "Пупкин Васёк")

opinion :: (Int, String) ->String
opinion (num, name) =
  "Я, " ++ name ++ " считаю \"" ++ sayWhat num ++ "\""
```

show read

```
> show 1
"1"
> show (2, "Пупкин Васёк")
"(2, \"Пупкин Васёк\")"

> read "True" || False
True
```

Охранные выражения

* похожи на switch/case
* слоник

```
bmiTell :: (RealFloat a) => a -> a -> String
bmiTell weight height
    | weight / height ^ 2 <= 18.5 = "You're underweight, you emo, you!"
    | weight / height ^ 2 <= 25.0 = "You're supposedly normal. Pffft, I bet you're
ugly!"
    | weight / height ^ 2 <= 30.0 = "You're fat! Lose some weight, fatty!"
    | otherwise                   = "You're a whale, congratulations!"
```

where

```
bmiTell :: (RealFloat a) => a -> a -> String
bmiTell weight height
    | bmi <= 18.5 = "You're underweight, you emo, you!"
    | bmi <= 25.0 = "You're supposedly normal. Pffft, I bet you're
ugly!"
    | bmi <= 30.0 = "You're fat! Lose some weight, fatty!"
    | otherwise   = "You're a whale, congratulations!"
    where bmi = weight / height ^ 2
```

where

```
bmiTell :: (RealFloat a) => a -> a -> String
bmiTell weight height
    | bmi <= skinny = "You're underweight, you emo, you!"
    | bmi <= normal = "You're supposedly normal.Pffft, I bet you're
ugly!"
    | bmi <= fat    = "You're fat! Lose some weight, fatty!"
    | otherwise     = "You're a whale, congratulations!"
    where bmi = weight / height ^ 2
          skinny = 18.5
          normal = 25.0
          fat = 30.0
```

where

```
bmiTell :: (RealFloat a) => a -> a -> String
bmiTell weight height
    | bmi <= skinny = "You're underweight, you emo, you!"
    | bmi <= normal = "You're supposedly normal.Pffft, I bet you're
ugly!"
    | bmi <= fat    = "You're fat! Lose some weight, fatty!"
    | otherwise     = "You're a whale, congratulations!"
    where bmi = weight / height ^ 2
      (skinny, normal, fat) = (18.5, 25.0, 30.0)
```

let

```
> [let square x = x * x in (square 5, square 3, square 2)]

> let square x = x * x in map square [5,3,2]

> (let a = 100; b = 200; c = 300 in a*b*c, let foo="Hey ";
bar = "there!" in foo ++ bar)

some =
  let opinion man = "Я, " ++ snd man ++ " считаю \"" ++ sayWhat (fst man) ++
"\""
      dude = (5, "Пупкин Васёк")
  in opinion dude
```

case

```
sayWhat :: Int -> String
sayWhat pos = "Моё мнение — " ++
  case pos of 1 -> "оставить как есть"
              2 -> "втоник четвёртая пара"
              3 -> "пятница, никогда"
              4 -> "я ничего не знаю"
              5 -> "ненавижу хаскель, заберите меня отсюда"
              _ -> "кто здесь?"
```

Рекурсивные структуры данных

Списки

Списки

```
isEmpty :: [a] -> Boolean
isEmpty [] = True
isEmpty _ = False

tell :: (Show a) => [a] -> String
tell [] = "Список пуст"
tell (x:[]) = "В списке только один элемент: " ++ show x
tell (x:y:[]) = "Два элемента: " ++ show x ++ " and " ++
show y
tell (x:y:_) = "Много. Первые 2: " ++ show x ++ " and " ++
show y
```

Списки

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

max' :: (Ord a) => [a] -> a
max' [] = error "У пустого списка нет максимального эл-та!"
max' [x] = x
max' (x:xs) = max x (max' xs)

Типы можно не писать, но это дурной тон

Самостоятельно length, с указанием типа
```