

Функциональное программирование

Егор Суворов

Курс «Парадигмы и языки программирования», подгруппа 3

Понедельник, 20 ноября 2017 года

План занятия

- 1 Функциональное программирование
 - Примеры на Python
 - Императивное программирование
 - Жизнь без переменных
- 2 Интересный Haskell
 - Рекурсия
 - Функции высшего порядка
 - Статический полиморфизм функций
 - Ленивые вычисления
 - Резюме
 - Грабли и плюшки
 - Отладка
- 3 Бонус
- 4 Ссылки

- 1 Функциональное программирование
 - Примеры на Python
 - Императивное программирование
 - Жизнь без переменных
- 2 Интересный Haskell
 - Рекурсия
 - Функции высшего порядка
 - Статический полиморфизм функций
 - Ленивые вычисления
 - Резюме
 - Грабли и плюшки
 - Отладка
- 3 Бонус
- 4 Ссылки

Императивное программирование

Руководствуемся следующим принципом:

Определение

Алгоритм — набор инструкций, описывающих порядок действий исполнителя для достижения некоторого результата

- В императивном стиле код программы описывает *как* надо достигать результата.
- Есть постоянно изменяющееся состояние программы (например, переменные).
- Хорошо ложится на «железо»: оно действительно меняет состояние памяти.

Что такое «действие»?

Можно работать на разных уровнях абстракции:

- Команда процессора: «возьми значение из ячейки памяти с таким номером»
- Вычисление арифметического выражения
- Конструкции управления `if/for/while`
- Вызовы функций

Функциональное программирование

Определение

Алгоритм — это *чистая* математическая функция от нескольких аргументов.

- Как решение задачи на курсе алгоритмов.
- Основное свойство — *чистота* функций; результат вычисления зависит только от аргументов функции.
- Ни слова про состояние программы или порядок вычислений.
- Остальные свойства выводятся из чистоты:
 - Порядок вычислений неважен.
 - Состояние программы не может влиять на работу функций.
 - Например, функция не может читать из глобальных переменных.

Побочные эффекты

- То, что происходит в функции помимо вычисления значения, называется *побочным эффектом*.
- Результаты всех функций зависят только от аргументов \iff у функций нет побочных эффектов.
- В «идеально чистых» языках запрещены не только функции с побочными эффектами, но любые побочные эффекты вообще.
- В частности, запрещено иметь изменяемое состояние — переменные.
- Никаких гонок данных.
- Непонятно, как тогда делать ввод-вывод — это точно побочный эффект.
- Идеально чистые языки бесполезны, все в той или иной степени «загрязнены».

Задача о рюкзаке, императивное решение

```
int maxSumCost = 0;
void solve(int i, int weight, int cost) {
    maxSumCost = max(maxSumCost, cost); // Не чистая
    if (i == n) return;
    solve(i + 1, weight, cost);
    if (weight >= ws[i]) {
        solve(i + 1, weight - ws[i], cost + cs[i]);
    }
}
```


Задача о рюкзаке, чистая функция-1

```
int solve(int i, int weight, int cost) {
    if (i == n) return cost;
    int ans = solve(i + 1, weight, cost);
    if (weight >= ws[i]) {
        ans = max(ans, solve(i + 1,
                             weight - ws[i],
                             cost + cs[i]));
    }
    return ans;
}
```

Задача о рюкзаке, чистая функция-2

```
int solve(int i, int weight, int cost) {
    if (i == n) return cost;
    if (weight < ws[i]) return solve(i + 1, weight, cost);
    return max(
        solve(i + 1, weight, cost),
        solve(i + 1, weight - ws[i], cost + cs[i])
    );
}
```

Задача о рюкзаке, чистая функция-3

```
int solve(int i, int weight) {
    if (i == n) return 0;
    if (weight < ws[i]) return solve(i + 1, weight);
    return max(
        solve(i + 1, weight),
        solve(i + 1, weight - ws[i]) + cs[i]
    );
}
```

Императивная работа с массивом

```
lst = [5, 10, 15, 10, 20]
partialSums = []
smallPrefix = []
s = 0
for x in lst:
    if x >= 10:
        s += x
        partialSums.append(s)
    if x <= 10:
        smallPrefix.append(x)

for i in range(len(lst)):
    lst[i] *= 2
```

Функциональная работа с массивом

```
import itertools
lst = [5, 10, 15, 10, 20]
partialSums = list(itertools.accumulate(
    # filter - "функция высшего порядка"
    filter(lambda x: x >= 10, lst)
))
smallPrefix = list(
    # takewhile- "функция высшего порядка"
    itertools.takewhile(
        lambda x: x <= 10, lst)
)
lst = [x * 2 for x in lst]
# lst = map(lambda x: x * 2, lst)
```

Haskell

- Haskell — чистый функциональный язык программирования.
- Компилируемый, статически типизирован, *очень* мощная система типов.
- Есть огромное число библиотек.
- Очень чистый по сравнению с остальными языками вроде OCaml.
- Помимо функциональной чистоты имеет огромное количество интересных особенностей.
- На нём действительно можно писать код.
- Стандартный компилятор — GHC (Glasgow Haskell Compiler).
- Рекомендуется использовать в составе [Haskell Platform](#).

Демонстрация

- 1 Демо: арифметика, простые типы, сравнения, списки, работа со списками, list comprehension.
- 2 Упражнение: как найти все Пифагоровы тройки ($x^2 + y^2 = z^2$) при $1 \leq x, y, z \leq 10$?
- 3 Интерпретатор ghci не поддерживает многострочные определения функций.
- 4 Поэтому с некоторого момента лучше набирать код в файле, подгружая его в интерпретатор:
 - `:load file.hs (:l file.hs)` компилирует `file.hs` и подгружает определения в интерпретатор.
 - `:reload` перекомпилирует и переподключит все файлы.
- 5 Демо: функции, комментарии, определения функций «и», «или», «сумма двух чисел».

Жизнь без циклов

Упражнения на Python:

- Как посчитать сумму чисел в списке, если нет переменных и циклов?

Жизнь без циклов

Упражнения на Python:

- Как посчитать сумму чисел в списке, если нет переменных и циклов?
- Функция `sum`.

Жизнь без циклов

Упражнения на Python:

- Как посчитать сумму чисел в списке, если нет переменных и циклов?
- Функция `sum`.
- Как посчитать сумму квадратов чисел?

Жизнь без циклов

Упражнения на Python:

- Как посчитать сумму чисел в списке, если нет переменных и циклов?
- Функция `sum`.
- Как посчитать сумму квадратов чисел?
- Определить лямбда-функцию для возведения в квадрат и применить `map` с `sum`.

Жизнь без циклов

Упражнения на Python:

- Как посчитать сумму чисел в списке, если нет переменных и циклов?
- Функция `sum`.
- Как посчитать сумму квадратов чисел?
- Определить лямбда-функцию для возведения в квадрат и применить `map` с `sum`.
- Какие вообще операции со списками мы ещё не умеем делать без циклов?

Жизнь без циклов

Упражнения на Python:

- Как посчитать сумму чисел в списке, если нет переменных и циклов?
- Функция `sum`.
- Как посчитать сумму квадратов чисел?
- Определить лямбда-функцию для возведения в квадрат и применить `map` с `sum`.
- Какие вообще операции со списками мы ещё не умеем делать без циклов?
- В которых элементы влияют друг на друга.

Демо: то же самое на Haskell.

Не самый хороший императивный код

```
s = "hello good world"
a = 0
b = 0
flag = False
for c in s + " ":
    if c == " ":
        if flag:
            a += 1
        flag = False
    else:
        b += 1
        flag = True
print(b / a)
```

И что тут происходит?

Моя реакция на такой код



На самом деле

Это вычисление средней длины слова в строке.

Чем плохо?

- 11 строк на такое простое действие.
- Есть сложный инвариант.
- Тестировать сложно: надо подбирать специальную строку, на которой отличается средняя длина слов (а не просто воспроизводится баг).

Функциональное решение

- Как записать то же самое функционально на Python?

Функциональное решение

- Как записать то же самое функционально на Python?
- Предполагаем, что есть функции `avg`, `map`, `split`.

Функциональное решение

- Как записать то же самое функционально на Python?
- Предполагаем, что есть функции `avg`, `map`, `split`.
- `avg(map(len, "hello world".split()))`

Функциональное решение

- Как записать то же самое функционально на Python?
- Предполагаем, что есть функции `avg`, `map`, `split`.
- `avg(map(len, "hello world".split()))`
- Тут что-то сложное делает только функция `split`.
- Обычно `split()` пишут «императивно», а всю последующую обработку — «функционально».
- Таким образом императивная сложность изолирована, все кусочки можно тестировать по отдельности.
- Опять ввели абстракцию: три встроенные функции.
- Оказывается, что «необходимых кирпичиков» не так много.

Моя реакция на функциональный код



Рандом в функциональном стиле

```
int getRandomNumber()  
{  
    return 4; // chosen by fair dice roll.  
             // guaranteed to be random.  
}
```

- 1 Функциональное программирование
 - Примеры на Python
 - Императивное программирование
 - Жизнь без переменных
- 2 Интересный Haskell
 - Рекурсия
 - Функции высшего порядка
 - Статический полиморфизм функций
 - Ленивые вычисления
 - Резюме
 - Грабли и плюшки
 - Отладка
- 3 Бонус
- 4 Ссылки

Упражнения на рекурсию

Факториал:

```
fac 0 = 1
```

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

Степень двойки:

Упражнения на рекурсию

Факториал:

```
fac 0 = 1
```

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

Степень двойки:

```
powerOfTwo 0 = 1
```

```
powerOfTwo n = 2 *
```

```
powerOfTwo (n - 1)
```

Числа Фибоначчи:

Упражнения на рекурсию

Факториал:

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

Степень двойки:

```
powerOfTwo 0 = 1
powerOfTwo n = 2 *
    powerOfTwo (n - 1)
```

Числа Фибоначчи:

```
fib n = fib' 0 1 n
fib' f1 f2 0 = f1
fib' f1 f2 n = fib' f2 (f1 + f2) (n - 1)
```

- Функции могут содержать апостроф
- Функции всегда должны начинаться с маленькой буквы.
- Мы увидели *pattern matching*:
 - Вместо if пишем «шаблон» для аргумента функции.
 - Шаблонов может быть несколько, они проверяются сверху вниз.
 - Приближает нас к математической нотации, избавляет от if'ов.
 - Очень популярно в функциональных языках.

If не нужен

```
-- If
fac n = if n == 0
        then 1
        else n * fac (n - 1)

-- Pattern matching
fac 0 = 1
fac n = n * fac (n - 1)

-- If
facTwo n = if n <= 1
            then 1
            else n * facTwo (n - 2)

-- Guards
facTwo n | n <= 1      = 1
         | otherwise = n * fac (n - 2)
```

Жизнь без циклов

Как теперь написать функцию `sum`?

```
sum' (x:xs) = x + sum' xs
```

```
sum' _ = 0
```

- Иногда можно делать сложный `pattern matching` вроде `x:xs` (список, первый элемент которого — `x`, а хвост — `xs`).
- Шаблоны проверяются сверху вниз, выбирается первый подходящий.
- Вместо имени переменной можно написать `_`.

Неявный инвариант

Если в императивной программе есть цикл — то можно вынести одну итерацию этого цикла в функцию и сделать рекурсию вместо цикла:

Python

Haskell

```
def sum(xs):  
    a = 0  
    for x in xs:  
        a += x  
    return a
```

```
sum' xs = sum'' 0 xs  
sum'' a [] = a  
sum'' a (x:xs) = sum'' (a + x) xs
```

Неявный инвариант

Если в императивной программе есть цикл — то можно вынести одну итерацию этого цикла в функцию и сделать рекурсию вместо цикла:

Python

```
def sum(xs):  
    a = 0  
    for x in xs:  
        a += x  
    return a
```

Haskell

```
sum' xs = sum'' 0 xs  
sum'' a [] = a  
sum'' a (x:xs) = sum'' (a + x) xs
```

Или так:

```
sum' xs = sum'' 0 xs  
  where  
    sum'' a [] = a  
    sum'' a (x:xs) = ...
```

Ещё пример

```
def foo(xs):  
    res = []  
    if not xs:  
        return  
    res.append(xs[0])  
    for x in xs[1:]:  
        if res[-1] != x:  
            res.append(x)  
    return res
```

Что это?

Ещё пример

```
def foo(xs):  
    res = []  
    if not xs:  
        return  
    res.append(xs[0])  
    for x in xs[1:]:  
        if res[-1] != x:  
            res.append(x)  
    return res
```

Что это? Удаление одинаковых значений, идущих подряд:

[10, 10, 20, 10] -> [10, 20, 10]

Упражнение

- Простая версия: написать функцию `prod`, считающую произведение элементов в списке.
- Сложная версия: напишите удаление одинаковых значений из списка.
- Возьмите код на Python в качестве основы.
- Преобразуйте цикл `for` в рекурсию.

Упражнение

- Простая версия: написать функцию `prod`, считающую произведение элементов в списке.
- Сложная версия: напишите удаление одинаковых значений из списка.
- Возьмите код на Python в качестве основы.
- Преобразуйте цикл `for` в рекурсию.
- Инвариант — последний добавленный элемент.

Упражнение

- Простая версия: написать функцию `prod`, считающую произведение элементов в списке.
- Сложная версия: напишите удаление одинаковых значений из списка.
- Возьмите код на Python в качестве основы.
- Преобразуйте цикл `for` в рекурсию.
- Инвариант — последний добавленный элемент.

```
uniq [] = []
uniq (x:xs) = x:(uniq' x xs)
  where
    uniq' _ [] = []
    uniq' last (x:xs)
      | last == x = xs'
      | otherwise = x:xs'
    where xs' = uniq' x xs
```

И где золотые горы?



Мы пока написали на функциональном языке программу в императивном стиле. Но хотя бы работает.

Золотые горы

```
uniq (x:xs) = x:(uniq (dropWhile (== x) xs))  
uniq _ = []
```

- В функциональном стиле протаскивать состояние довольно неудобно.
- Это нормально, так и надо — всё можно написать, комбинируя функции, язык нас просто к этому *легонько* подталкивает.
- Язык предоставляет много функций для работы *и со списками тоже*, пользуемся!
- Мы снова поднялись на уровень абстракции:
 - Мы умеем комбинировать функции.
 - У нас есть набор *функций высшего порядка* (они принимают другие функции в качестве аргументов): `map`, `dropWhile`, `filter`...
 - Этим функций высшего порядка не слишком много, но хватает для жизни.

Играем в игру

Задача: пусть есть массив a ; длины n и массив индексов p . Мы хотим положить в a ; значение a_{p_i} .

Пример

i	0	1	2	3	4
a_i	10	11	10	12	9
p_i	2	0	0	4	4
a_{p_i}	10	10	10	9	9

Есть ли проблемы?

Решение

```
for i in range(len(a)):
    a[i] = a[p[i]]
```

Играем в игру

Задача: пусть есть массив a ; длины n и массив индексов p . Мы хотим положить в a значение a_{p_i} .

Пример

i	0	1	2	3	4
a_i	10	11	10	12	9
p_i	2	0	0	4	4
a_{p_i}	10	10	10	9	9

Решение

```
for i in range(len(a)):
    a[i] = a[p[i]]
```

Есть ли проблемы?

- Да, потому что массив меняется в процессе.
- Если $p_i < i$, то справа может получиться неверное значение.
- В примере нам повезло, потому что $a_{p_2} = a_0 = a_{p_0} = a_2$.
- Другой классический пример: merge sort.

Мораль: наличие состояния может не только быть неочевидно и вредить, но только на каких-то примерах.

Функциональное решение

- Как ни странно, есть оператор получения элемента списка по номеру:

```
[10, 11, 12, 13, 14] !! 2  -- 12
```

- Получаем решение:

```
permute a p = map (a !!) p
```

- Одна строка.
- Но для чтения надо знать, что такое `map`
- Также надо знать частичное применение оператора `!!`.


Резюме

- Программа на функциональных языках представляет собой комбинацию каких-то стандартных функций в нужном порядке.
- Обычно читается хорошо, если знать названия этих функций.
- Называется очень похоже в разных языках.
- Код на функциональных языках получается довольно короткий и плотный.
- Написать нечитаемо тоже можно:

```
filter (\x -> 1 == sum (map (\y -> 1 - min 1 (abs (x - y))) xs)) xs
```
- Не забываем про декларативный list comprehension и введение вспомогательных функций при помощи where!

Приятные бонусы

- Язык заставляет вас понимать инварианты.
- Практически не используются ветвления и циклы.
- Проще доказывать корректность.
- Сложнее допустить скрытый баг.
- Активно используются абстракции, в том числе из математики («моноид» легко может всплыть).
- ФП легко встраивается в императивные языки и получаем лучшее двух миров.




Твиттер спинного мозга 24 авг в 12:00


Для преданных пользователей языка Хаскель также будут созданы языки Коргель и Коллель.


[Андрей Заболотский](#)


Мне нравится 11


4 комментария

 **Соня Ебинова**
слышно ли что-то насчёт Пуделя?
24 авг в 12:26 [Ответить](#) 8

 **Sonofabitch Desert** ответил Соне
Соня, Пуделяль
24 авг в 12:40 [Ответить](#)

 **Yuriy Shilov**
кобол+хаскель=кобель
24 авг в 15:23 [Ответить](#) 3

 **Вера Проваторова**
На коргеле будут короткие операторы
25 авг в 10:25 [Ответить](#)

 Написать комментарий...

- 1 Функциональное программирование
 - Примеры на Python
 - Императивное программирование
 - Жизнь без переменных
- 2 **Интересный Haskell**
 - Рекурсия
 - **Функции высшего порядка**
 - Статический полиморфизм функций
 - Ленивые вычисления
 - Резюме
 - Грабли и плюшки
 - Отладка
- 3 Бонус
- 4 Ссылки

Напоминание

- Функция является *функцией высшего порядка*, если она в качестве одного из аргументов принимает другую функцию.
- Пример: `map`. Он первым параметром принимает функцию, которая преобразует элементы списка.
- Пример: `dropWhile`. Первый аргумент умеет по элементу сообщать, надо остановиться или нет.
- Пример: `filter cond list`. Оставляет в списке только элементы, удовлетворяющие условию.
- Функции высшего порядка являются основными кирпичиками в функциональном программировании.

Ещё один паттерн

```
sum (x:xs) = x + sum xs  
sum _     = 0
```

```
prod (x:xs) = x * prod xs  
prod _     = 1
```

```
max (x:xs) = max x (max xs)  
max x     = -1
```

```
concat (x:xs) = x ++ (concat xs)  
concat _     = ""
```

Что общего?

Ещё один паттерн

```
sum (x:xs) = x + sum xs
```

```
sum _ = 0
```

```
prod (x:xs) = x * prod xs
```

```
prod _ = 1
```

```
max (x:xs) = max x (max xs)
```

```
max x = -1
```

```
concat (x:xs) = x ++ (concat xs)
```

```
concat _ = ""
```

Что общего?

- Все эти функции считают функцию от множества элементов.
- Для пересчёта требуется знать только текущее значение и очередной элемент.

Правая свёртка

```
foldr f a (x:xs) = f x (foldr f a xs)
```

```
foldr f a _ = a
```

```
sum    xs = foldr (+) 0    xs
```

```
prod   xs = foldr (*) 1    xs
```

```
max    xs = foldr max (-1) xs
```

```
concat xs = foldr (++) ""  xs
```

Ещё одна популярная функция высшего порядка.

Упражнение на понимание

```
foldr f a (x:xs) = f x (foldr f a xs)
```

```
foldr f a _ = a
```

А что такое `foldr (:) [4,5] xs`?

Упражнение на понимание

```
foldr f a (x:xs) = f x (foldr f a xs)
foldr f a _ = a
```

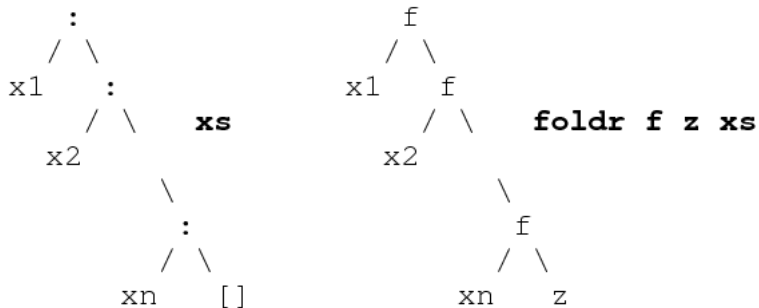
А что такое `foldr (:) [4,5] xs`?

```
foldr (:) [4,5] [1,2,3] =
1:(foldr (:) [4,5] [2,3]) =
1:(2:(foldr (:) [4,5] [3])) =
1:(2:(3:(foldr (:) [4,5] []))) =
1:(2:(3:[4,5])) =
[1,2,3,4,5]
```

Дописывание `[4,5]` в конец списка.

```
(++) a b = foldr (:) b a
```

Картинка



Бамбук растёт вправо, поэтому *правая* свёртка.

Упражнения

Как узнать, все ли элементы равны True?

Упражнения

Как узнать, все ли элементы равны True?

```
foldr (&&) True xs
```

Как узнать сумму квадратов чисел в массиве?

Упражнения

Как узнать, все ли элементы равны True?

```
foldr (&&) True xs
```

Как узнать сумму квадратов чисел в массиве?

```
foldr (+) 0 (map (^2) xs)
```

Как выразить map через foldr?

Упражнения

Как узнать, все ли элементы равны True?

```
foldr (&&) True xs
```

Как узнать сумму квадратов чисел в массиве?

```
foldr (+) 0 (map (^2) xs)
```

Как выразить map через foldr?

```
map f xs = foldr (\a x -> (f a):x) [] xs
```

Вывод: в теории почти всё есть foldr. На практике лучше использовать готовые функции.

- 1 Функциональное программирование
 - Примеры на Python
 - Императивное программирование
 - Жизнь без переменных
- 2 **Интересный Haskell**
 - Рекурсия
 - Функции высшего порядка
 - **Статический полиморфизм функций**
 - Ленивые вычисления
 - Резюме
 - Грабли и плюшки
 - Отладка
- 3 Бонус
- 4 Ссылки

- Мы нигде не указывали типы ни аргументов функций, ни возвращаемых значений.
- Свободно использовали функции для разных типов (вроде `map`).
- Если набрать `:t map` в GHCi, увидим её тип:

$$\underbrace{(a \rightarrow b)}_{\text{функция}} \rightarrow \underbrace{[a]}_{\text{исходный список}} \rightarrow \underbrace{[b]}_{\text{результат}} .$$

- Справа от последней `->` — возвращаемое значение, до этого — аргументы.
- Тут `a` и `b` — типовые переменные. На их месте может стоять любой тип.
- Естественным образом получаем, что `map` вообще всё равно, с какими списками работать.
- Haskell автоматически выводит наиболее общие типы для практически всех функций.
- Все проверки типов — *на этапе компиляции*.

Что могут делать функции

- Что вообще может делать *чистая* функция с типом `Bool -> Bool`?
- Их всего $2^2 = 4$ различных: всегда `True`, всегда `False`, отрицание, тождественная.
- А что может делать полиморфная функция с типом `a -> a`?

Что могут делать функции

- Что вообще может делать *чистая* функция с типом `Bool -> Bool`?
- Их всего $2^2 = 4$ различных: всегда `True`, всегда `False`, отрицание, тождественная.
- А что может делать полиморфная функция с типом `a -> a`?
- Только возвращать свой аргумент — она не имеет права ничего про него предполагать.

Что могут делать функции

- Что вообще может делать *чистая* функция с типом `Bool -> Bool`?
- Их всего $2^2 = 4$ различных: всегда `True`, всегда `False`, отрицание, тождественная.
- А что может делать полиморфная функция с типом `a -> a`?
- Только возвращать свой аргумент — она не имеет права ничего про него предполагать.
- А функции с типом `a -> b` не бывает — она в общем случае не может создать что-то типа `b`.
- Что может делать `a -> [a]`?

Что могут делать функции

- Что вообще может делать *чистая* функция с типом `Bool -> Bool`?
- Их всего $2^2 = 4$ различных: всегда `True`, всегда `False`, отрицание, тождественная.
- А что может делать полиморфная функция с типом `a -> a`?
- Только возвращать свой аргумент — она не имеет права ничего про него предполагать.
- А функции с типом `a -> b` не бывает — она в общем случае не может создать что-то типа `b`.
- Что может делать `a -> [a]`?
- Только создавать список из одинаковых элементов *фиксированной* длины, которая не зависит от аргумента.

Игра

Ваша задача — по типу функции угадать, что она делает.

Тип	Функция
$a \rightarrow a$	

Игра

Ваша задача — по типу функции угадать, что она делает.

Тип	Функция
$a \rightarrow a$	$\text{id } x = x$

Игра

Ваша задача — по типу функции угадать, что она делает.

Тип	Функция
$a \rightarrow a$	$\text{id } x = x$
$a \rightarrow b \rightarrow a$	

Игра

Ваша задача — по типу функции угадать, что она делает.

Тип	Функция
<code>a -> a</code>	<code>id x = x</code>
<code>a -> b -> a</code>	<code>fst x y = x</code>

Игра

Ваша задача — по типу функции угадать, что она делает.

Тип	Функция
<code>a -> a</code>	<code>id x = x</code>
<code>a -> b -> a</code>	<code>fst x y = x</code>
<code>a -> b -> b</code>	

Игра

Ваша задача — по типу функции угадать, что она делает.

Тип	Функция
<code>a -> a</code>	<code>id x = x</code>
<code>a -> b -> a</code>	<code>fst x y = x</code>
<code>a -> b -> b</code>	<code>snd x y = y</code>

Игра

Ваша задача — по типу функции угадать, что она делает.

Тип	Функция
<code>a -> a</code>	<code>id x = x</code>
<code>a -> b -> a</code>	<code>fst x y = x</code>
<code>a -> b -> b</code>	<code>snd x y = y</code>
<code>(a -> b) -> a -> b</code>	

Игра

Ваша задача — по типу функции угадать, что она делает.

Тип	Функция
$a \rightarrow a$	<code>id x = x</code>
$a \rightarrow b \rightarrow a$	<code>fst x y = x</code>
$a \rightarrow b \rightarrow b$	<code>snd x y = y</code>
$(a \rightarrow b) \rightarrow a \rightarrow b$	<code>apply f x = f x</code>

Игра

Ваша задача — по типу функции угадать, что она делает.

Тип	Функция
<code>a -> a</code>	<code>id x = x</code>
<code>a -> b -> a</code>	<code>fst x y = x</code>
<code>a -> b -> b</code>	<code>snd x y = y</code>
<code>(a -> b) -> a -> b</code>	<code>apply f x = f x</code>
<code>[a] -> a</code>	

Игра

Ваша задача — по типу функции угадать, что она делает.

Тип	Функция
<code>a -> a</code>	<code>id x = x</code>
<code>a -> b -> a</code>	<code>fst x y = x</code>
<code>a -> b -> b</code>	<code>snd x y = y</code>
<code>(a -> b) -> a -> b</code>	<code>apply f x = f x</code>
<code>[a] -> a</code>	<code>get xs = xs !! c</code>

Игра

Ваша задача — по типу функции угадать, что она делает.

Тип	Функция
<code>a -> a</code>	<code>id x = x</code>
<code>a -> b -> a</code>	<code>fst x y = x</code>
<code>a -> b -> b</code>	<code>snd x y = y</code>
<code>(a -> b) -> a -> b</code>	<code>apply f x = f x</code>
<code>[a] -> a</code>	<code>get xs = xs !! c</code>
<code>(a -> Bool) -> [a] -> [a]</code>	

Игра

Ваша задача — по типу функции угадать, что она делает.

Тип	Функция
<code>a -> a</code>	<code>id x = x</code>
<code>a -> b -> a</code>	<code>fst x y = x</code>
<code>a -> b -> b</code>	<code>snd x y = y</code>
<code>(a -> b) -> a -> b</code>	<code>apply f x = f x</code>
<code>[a] -> a</code>	<code>get xs = xs !! c</code>
<code>(a -> Bool) -> [a] -> [a]</code>	<code>filter</code>

Игра

Ваша задача — по типу функции угадать, что она делает.

Тип	Функция
<code>a -> a</code>	<code>id x = x</code>
<code>a -> b -> a</code>	<code>fst x y = x</code>
<code>a -> b -> b</code>	<code>snd x y = y</code>
<code>(a -> b) -> a -> b</code>	<code>apply f x = f x</code>
<code>[a] -> a</code>	<code>get xs = xs !! c</code>
<code>(a -> Bool) -> [a] -> [a]</code>	<code>filter</code>
<code>(a -> Bool) -> [a] -> [a]</code>	

Игра

Ваша задача — по типу функции угадать, что она делает.

Тип	Функция
<code>a -> a</code>	<code>id x = x</code>
<code>a -> b -> a</code>	<code>fst x y = x</code>
<code>a -> b -> b</code>	<code>snd x y = y</code>
<code>(a -> b) -> a -> b</code>	<code>apply f x = f x</code>
<code>[a] -> a</code>	<code>get xs = xs !! c</code>
<code>(a -> Bool) -> [a] -> [a]</code>	<code>filter</code>
<code>(a -> Bool) -> [a] -> [a]</code>	<code>dropWhile</code>

Игра

Ваша задача — по типу функции угадать, что она делает.

Тип	Функция
<code>a -> a</code>	<code>id x = x</code>
<code>a -> b -> a</code>	<code>fst x y = x</code>
<code>a -> b -> b</code>	<code>snd x y = y</code>
<code>(a -> b) -> a -> b</code>	<code>apply f x = f x</code>
<code>[a] -> a</code>	<code>get xs = xs !! c</code>
<code>(a -> Bool) -> [a] -> [a]</code>	<code>filter</code>
<code>(a -> Bool) -> [a] -> [a]</code>	<code>dropWhile</code>
<code>(a -> Int) -> [a] -> Int</code>	

Игра

Ваша задача — по типу функции угадать, что она делает.

Тип	Функция
<code>a -> a</code>	<code>id x = x</code>
<code>a -> b -> a</code>	<code>fst x y = x</code>
<code>a -> b -> b</code>	<code>snd x y = y</code>
<code>(a -> b) -> a -> b</code>	<code>apply f x = f x</code>
<code>[a] -> a</code>	<code>get xs = xs !! c</code>
<code>(a -> Bool) -> [a] -> [a]</code>	<code>filter</code>
<code>(a -> Bool) -> [a] -> [a]</code>	<code>dropWhile</code>
<code>(a -> Int) -> [a] -> Int</code>	<code>sum (map f xs)</code>

Игра

Ваша задача — по типу функции угадать, что она делает.

Тип	Функция
<code>a -> a</code>	<code>id x = x</code>
<code>a -> b -> a</code>	<code>fst x y = x</code>
<code>a -> b -> b</code>	<code>snd x y = y</code>
<code>(a -> b) -> a -> b</code>	<code>apply f x = f x</code>
<code>[a] -> a</code>	<code>get xs = xs !! c</code>
<code>(a -> Bool) -> [a] -> [a]</code>	<code>filter</code>
<code>(a -> Bool) -> [a] -> [a]</code>	<code>dropWhile</code>
<code>(a -> Int) -> [a] -> Int</code>	<code>sum (map f xs)</code>
<code>(a -> b -> b) -> b -> [a] -> b</code>	

Игра

Ваша задача — по типу функции угадать, что она делает.

Тип	Функция
<code>a -> a</code>	<code>id x = x</code>
<code>a -> b -> a</code>	<code>fst x y = x</code>
<code>a -> b -> b</code>	<code>snd x y = y</code>
<code>(a -> b) -> a -> b</code>	<code>apply f x = f x</code>
<code>[a] -> a</code>	<code>get xs = xs !! c</code>
<code>(a -> Bool) -> [a] -> [a]</code>	<code>filter</code>
<code>(a -> Bool) -> [a] -> [a]</code>	<code>dropWhile</code>
<code>(a -> Int) -> [a] -> Int</code>	<code>sum (map f xs)</code>
<code>(a -> b -> b) -> b -> [a] -> b</code>	<code>foldr</code>

Вывод типов

Ваша задача — по определению функции вывести наиболее общий тип.

Функция	Тип
<code>foo x y = x y</code>	

Вывод типов

Ваша задача — по определению функции вывести наиболее общий тип.

Функция	Тип
<code>foo x y = x y</code>	<code>(a -> b) -> a -> b</code>

Вывод типов

Ваша задача — по определению функции вывести наиболее общий тип.

Функция	Тип
<code>foo x y = x y</code>	$(a \rightarrow b) \rightarrow a \rightarrow b$
<code>foo x y z = x y z</code>	

Вывод типов

Ваша задача — по определению функции вывести наиболее общий тип.

Функция	Тип
<code>foo x y = x y</code>	<code>(a -> b) -> a -> b</code>
<code>foo x y z = x y z</code>	<code>(a -> b -> c) -> a -> b -> c</code>

Вывод типов

Ваша задача — по определению функции вывести наиболее общий тип.

Функция	Тип
<code>foo x y = x y</code>	<code>(a -> b) -> a -> b</code>
<code>foo x y z = x y z</code>	<code>(a -> b -> c) -> a -> b -> c</code>
<code>foo x y z = (x y) + (x z)</code>	

Вывод типов

Ваша задача — по определению функции вывести наиболее общий тип.

Функция	Тип
<code>foo x y = x y</code>	<code>(a -> b) -> a -> b</code>
<code>foo x y z = x y z</code>	<code>(a -> b -> c) -> a -> b -> c</code>
<code>foo x y z = (x y) + (x z)</code>	<code>(a -> Int) -> a -> a -> Int</code>

Вывод типов

Ваша задача — по определению функции вывести наиболее общий тип.

Функция	Тип
<code>foo x y = x y</code>	<code>(a -> b) -> a -> b</code>
<code>foo x y z = x y z</code>	<code>(a -> b -> c) -> a -> b -> c</code>
<code>foo x y z = (x y) + (x z)</code>	<code>(a -> Int) -> a -> a -> Int</code>
<code>foo x y = (x y) + y</code>	

Вывод типов

Ваша задача — по определению функции вывести наиболее общий тип.

Функция	Тип
<code>foo x y = x y</code>	<code>(a -> b) -> a -> b</code>
<code>foo x y z = x y z</code>	<code>(a -> b -> c) -> a -> b -> c</code>
<code>foo x y z = (x y) + (x z)</code>	<code>(a -> Int) -> a -> a -> Int</code>
<code>foo x y = (x y) + y</code>	<code>(Int -> Int) -> Int -> Int</code>

Вывод типов

Ваша задача — по определению функции вывести наиболее общий тип.

Функция	Тип
<code>foo x y = x y</code>	<code>(a -> b) -> a -> b</code>
<code>foo x y z = x y z</code>	<code>(a -> b -> c) -> a -> b -> c</code>
<code>foo x y z = (x y) + (x z)</code>	<code>(a -> Int) -> a -> a -> Int</code>
<code>foo x y = (x y) + y</code>	<code>(Int -> Int) -> Int -> Int</code>
<code>foo x y = (x y):y</code>	

Вывод типов

Ваша задача — по определению функции вывести наиболее общий тип.

Функция	Тип
<code>foo x y = x y</code>	<code>(a -> b) -> a -> b</code>
<code>foo x y z = x y z</code>	<code>(a -> b -> c) -> a -> b -> c</code>
<code>foo x y z = (x y) + (x z)</code>	<code>(a -> Int) -> a -> a -> Int</code>
<code>foo x y = (x y) + y</code>	<code>(Int -> Int) -> Int -> Int</code>
<code>foo x y = (x y):y</code>	<code>([a] -> a) -> [a] -> [a]</code>

Резюме

- Без полиморфизма функции высшего порядка были бы бесполезны.
- Часто по типу полиморфной функции можно догадаться, что она делает.
- Есть специальный поисковик [Hoogle](#), который ищет функции по их типу.
- Hoogle — полезная штука, если вам нужна какая-то «очевидно полезная» функция. Найдётся всё.

- 1 Функциональное программирование
 - Примеры на Python
 - Императивное программирование
 - Жизнь без переменных
- 2 Интересный Haskell
 - Рекурсия
 - Функции высшего порядка
 - Статический полиморфизм функций
 - **Ленивые вычисления**
 - Резюме
 - Грабли и плюшки
 - Отладка
- 3 Бонус
- 4 Ссылки

Идея

- В классических языках аргументы сначала вычисляются, потом передаются функции: в вызове `foo(bar())` сначала вычислится `bar()`, а результат передадут в `foo()`.
- В Haskell наоборот: если функции неважно, что именно ей передали в качестве аргумента — это нечто не будет вычислено.
- Другими словами, вычисления производят только от плохой жизни: если по-другому никак.
- Например, если надо аргумент сравнить с шаблоном.
- В функции `fst` второй аргумент вычисляться никогда не будет:
`fst x y = x`
- А в функции `if` будет вычислен только нужный аргумент:
`if' True x _ = x`
`if' False _ y = y`
- Приходится тащить вместо невычисленного выражения инструкцию, как его получить.

Бесконечные структуры

- Напоминание: `[1, 2, 3]` — это сахар для `1:2:3:[]`.

- Попробуем завести бесконечный список из единиц:

```
ones = 1:1:1:1:1:1:...
```

- Так как бесконечную строчку написать не можем, придётся заметить рекурсию:

```
ones = 1:ones
```

- Этого уже хватит для вычисления, попробуйте `take 10 ones`.
- Как работает: хвост списка вычисляется только тогда, когда он реально нужен.
- Поэтому если никто не залез дальше второго элемента, то третий и последующие не будут вычислены, а будут просто храниться, как их можно в случае чего получить.

Подробный пример

```
take 0 _ = []
take n (x:xs) = x:(take (n - 1) xs)
ones = 1:ones
take 3 ones = take 3 (1:ones) = 1:(take (3-1) ones)
take 2 ones = take 2 (1:ones) = 1:(take (2-1) ones)
take 1 ones = take 1 (1:ones) = 1:(take (1-1) ones)
take 0 ones = []
take 3 ones = 1:1:1:[] = [1, 1, 1]
```

Пример похитрее

```
iota n = n:(iota (n + 1))  
take 3 (iota 5) = take 3 (5:iota 6) =  
5:(take 2 (iota 6)) = 5:(take 2 (6:iota 7)) =  
5:6:(take 1 (iota 7)) = 5:6:(take 1 (7:iota 8)) =  
5:6:7:(take 0 (iota 8)) = 5:6:7:[]  
iota n = [n, n + 1, n + 2, ...]
```

Практически так реализован синтаксический сахар [5..].

Числа Фибоначчи

Составляем уравнение на список из чисел Фибоначчи:

```
fib    = [1, 1, 2, 3, 5, 8, ...]
0:fib  = [0, 1, 1, 2, 3, 5, 8, ...]
zipWith (+) fib (0:fib) =
    [1, 2, 3, 5, 8, 13, ...]
1:zipWith (+) fib (0:fib) =
    [1, 1, 2, 3, 5, 8, ...]
fib = 1:zipWith (+) fib (0:fib)  -- Определение в Haskell
```


Вычисление Фибоначчи

```
fib = 1:zipWith (+) fib (0:fib)  -- Определение в Haskell
```

- Изначально знаем только `fib !! 0`.
- Когда надо вычислить `fib !! 1`, надо вычислить нулевой элемент от `zipWith`.
- А для этого надо знать нулевые элементы от `fib` и `0:fib`, оба знаем.
- Значит, их сумма и есть `fib !! 1`.
- Когда потребуется вычислить `fib !! 2`, надо будет вычислить первый элемент от `zipWith`.
- Для этого придётся раскрыть `fib` дважды (как в предыдущих пунктах), а `0:fib` — один раз.
- Раскрыть получится — это элементы мы уже считали. Значит, посчитаем их сумму и `fib !! 2`.
- И так до бесконечности.
- Время работы и память неизвестны :)

Особенности бесконечных списков

- У них нет конца.
- Если ваша функция пытается дойти до конца списка — у неё проблемы:

```
getFirstTen xs = take (min 10 (length xs)) xs
length xs     -- Не работает с бесконечными списками.
```

- Из-за ленивости операции над бесконечными списками реальны:

```
take 0 _      = [] -- Тут второй аргумент не вычисляется
take n (x:xs) = x:(take (n - 1) xs)
```

- Или даже так:

```
any' (True :_) = True
any' (_      :xs) = any' xs
```

- Функции в домашнем задании должны корректно обрабатывать те бесконечные списки, на которых поведение определено.
- Надо писать аккуратно.

Упражнение

- Напишите функцию `concat`, который приписывает один список к другому.
- Работает в точности как оператор `(++)`.
- Когда осмысленно получать на вход бесконечный список?

Упражнение

- Напишите функцию `concat`, который приписывает один список к другому.
- Работает в точности как оператор `(++)`.
- Когда осмысленно получать на вход бесконечный список?

```
concat (x:xs) ys = x:(concat xs ys)
```

```
concat _      ys = ys
```

- Если получаем бесконечный список как первый аргумент — второй не используется.
- Если получаем бесконечный список как второй аргумент — он дописывается в конец первого.
- В Haskell оператор `(++)` действительно работает за линию.

- 1 Функциональное программирование
 - Примеры на Python
 - Императивное программирование
 - Жизнь без переменных
- 2 **Интересный Haskell**
 - Рекурсия
 - Функции высшего порядка
 - Статический полиморфизм функций
 - Ленивые вычисления
 - **Резюме**
 - Грабли и плюшки
 - Отладка
- 3 Бонус
- 4 Ссылки

- Haskell строго типизирован и есть полиморфные функции, всё проверяется на этапе компиляции.
- Циклов нет, переменных нет, всё делается рекурсивно.
- `if`'ы не нужны — используем `pattern matching`, в крайнем случае — `guards`.
- На Haskell тоже можно как-то писать в императивном стиле.
- Лучше писать в функциональном стиле, комбинируя функции высшего порядка со своими.
- Самая мощная функция из известных нам сейчас — `foldr`.
- Вычисления очень ленивы: пока не потребуется сравнить с чем-то, вычисления не будет.
- Из-за этого возможны бесконечные структуры и разумные конечные операции с ними.
- Списки односвязные, из-за этого они бывают бесконечными.

- 1 Функциональное программирование
 - Примеры на Python
 - Императивное программирование
 - Жизнь без переменных
- 2 Интересный Haskell
 - Рекурсия
 - Функции высшего порядка
 - Статический полиморфизм функций
 - Ленивые вычисления
 - Резюме
 - **Грабли и плюшки**
 - Отладка
- 3 Бонус
- 4 Ссылки

Классы типов

- Иногда хочется функцию полиморфную, но с ограничением на тип.
- Например, `min` должен уметь сравнивать свои аргументы.
- Набор свойств, которыми должен обладать тип, называют *классом типа*.
- Подробнее разберём на следующем занятии.
- То, что идёт перед `=>` в типе в Haskell — это как раз ограничения на классы:

```
min :: Ord a => a -> a -> a
```

- `a` — любой тип, лежащий в классе `Ord`.
- Не путать с классами из ООП!
- Тут «класс» означает «множество», как в математике.

Очень строгая типизация

- Есть разные типы для вещественных и целых чисел.
- Оператор `==` между разными типами чисел не определён.
- Но тестом `2 == 2.0` это не поймать.
- Тип числовой константы определяется в момент компиляции: это будет либо вещественное число (класс `Fractional`), либо целое (класс `Integral`), либо любое (класс `Num`).
- Для конвертации между `Fractional` и `Integral` можно использовать `fromIntegral` и `round`.

Каррирование

- Функция от двух аргументов $f(x, y)$ — то же самое, что функция от одного аргумента $f'(x)$, которая возвращает функцию от одного аргумента $g_x(y) = f(x, y)$.
- Такой переход называется *каррированием*
- С точки зрения Haskell, функции типов $a \rightarrow b \rightarrow c$ и $a \rightarrow (b \rightarrow c)$ неотличимы.
- Все функции в Haskell — от одного аргумента, остальное — сахар.
- Из-за этого частичное применение и работает: $(3+)$ и $(+) 3$ равны.
- Можно явно написать функцию от двух аргументов:

```
s (a, b) = a + b
```

```
s' = uncurry (+)
```

Способы определения функций

- `addTen x = x + 10`
`addTenToAll x = map addTen x`
- `addTen = (+10)`
`addTenToAll = map f`

Композиция функций

Точка — оператор композиции функций:

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
```

-- *Приоритет низкий*

```
((+5) . (*3)) 10
```

Читать надо справа налево:

```
(concat . map words . lines) "1 2\n10 11\n"
```

Композиции функций в каком-то смысле лучше, чем свои вспомогательные функции или явные вызовы.

Унарный минус

- Единственный унарный оператор в Haskell.
- Захардкожен костылями.
- Ломает красоту, потому что (-3) надо интерпретировать как унарный минус, а не как оператор $-$ с зафиксированным операндом.
- А $((-)3)$ надо интерпретировать, как оператор $(-)$ с зафиксированным первым (левым) операндом.
- Поэтому зафиксировать правый аргумент у бинарного минуса никак нельзя, кроме лямбда-функций.

1 Функциональное программирование

- Примеры на Python
- Императивное программирование
- Жизнь без переменных

2 Интересный Haskell

- Рекурсия
- Функции высшего порядка
- Статический полиморфизм функций
- Ленивые вычисления
- Резюме
- Грабли и плюшки
- **Отладка**

3 Бонус

4 Ссылки

Как отлаживать

- Ваш лучший друг — чтение кода, выписывание типов, тестирование.
- Haskell обычно выдаёт точный символ, в котором произошла ошибка. Там и надо смотреть.
- Начинать лучше с самой верхней ошибки.
- Можно закомментировать кусок кода, а у оставшегося явно написать нужный тип:

```
map' :: (a -> b) -> [a] -> [b]
```

```
map' f a b = f a -- Ошибка компиляции
```

- Проверяйте код при помощи [hlint](#).
- Он заодно проверяет соответствие некоторому стилю.
- Не все рекомендации hlint жизненно необходимо выполнять, так как единого стиля нет.

Trace

```
import Debug.Trace
-- traceShow :: Show a => a -> b -> b
sum' xs = sum'' 0 xs
  where
    sum'' a [] = traceShow (a, [] :: [Int]) a
    sum'' a (x:xs) = traceShow (a, x:xs)
                      (sum'' (a + x) xs)
```

- Использовать только для отладки
- У `traceShow` два параметра — один она печатает, второй возвращает
- `traceShow` нарушает чистоту

HSpec

Можно и нужно писать тесты для функций:

```
import Test.Hspec -- cabal install hspec
main = hspec $ do
  describe "length function" $ do
    it "works on empty list" $ do
      (length []) 'shouldBe' 0
    it "works on single-item list" $ do
      (length [10]) 'shouldBe' 1
      (length ["foo"]) 'shouldBe' 1
```

- 1 Функциональное программирование
 - Примеры на Python
 - Императивное программирование
 - Жизнь без переменных
- 2 Интересный Haskell
 - Рекурсия
 - Функции высшего порядка
 - Статический полиморфизм функций
 - Ленивые вычисления
 - Резюме
 - Грабли и плюшки
 - Отладка
- 3 Бонус
- 4 Ссылки

Ввод-вывод

В домашке не потребуется

```
solve :: Int -> Int
solve = (+1)
mainPure :: [String] -> [String]
mainPure = map (show . solve . (read :: String -> Int))
main = interact (unlines . mainPure . lines)
```

- Функции `main` и `interact` — пока что магия
- Функция `show` преобразует почти любое значение в строку
- Функция `read`, если явно указать её тип, преобразует строку в значение
- Всё работает лениво и интерактивно

Особенности функционального стиля

Необязательно, но обычно:

- Алгоритм разбивается не на «шаги», а на мелкие функции с чёткими контрактами
- Вместо циклов — рекурсия или встроенные функции (коих много)
- Очень мощная система типов:
 - Функции обобщаются получше
 - Алгебраические типы данных, позволяют писать *везде определённые* функции
 - Вместо `if` — *pattern matching*
- Очень компактный код
- Нет никаких изменяемых переменных, захотели изменить объект — создали копию (*иммутабельность*)
- Частичное применение функций (или каррирование, из одного другое в каком-то смысле следует)

Мои наблюдения-1

- На функциональных языках обычно очень компактные программы и много синтаксического сахара.
- Обычно функциональные языки (в том числе Haskell) умеют очень сильно расширять свой синтаксис до неузнаваемости.
- Иногда всё это превращается в сахарную вату.
- Элементы ФП в разной степени поддерживаются в разных языках, в том числе в «императивных»: C++, Python, Java.
- Есть модные смеси императивного и функционального программирования вроде Scala или OCaml.
- Некоторые функциональные языки используются в реальной жизни: Erlang.
- Чисто функциональные программы может быть сложнее отлаживать, так как нет «состояния программы».

Мои наблюдения-2

- Многие идеи из ФП полезны и в повседневной жизни:
 - Неизменяемое состояние.
 - Функции высшего порядка (где есть поддержка в языке).
 - Чистые функции без побочных эффектов.
- Если язык поддерживают хотя бы `map`, лямбда-функции или `list comprehension`, на нём уже намного приятнее писать.
- Функциональные элементы могут сильно упростить код императивной программы без потери скорости.
- Дополнительных проблем эти элементы не вносят.
- Надо быть аккуратными и не мешать их с изменяемым состоянием.

- [Практическое введение в функциональное программирование](#) — идеология.
- learnyouahaskell.com, нас интересуют главы 2 и 4-6.
- Есть перевод на русский: «Изучай Haskell во имя добра!»
- [Проверяющая система с задачами](#).
- [Лекции и слайды на английском](#).
- [Руководство на русском](#) (пропустите раздел с модулями).