

# Введение в язык Haskell

---

Михаил Беляев

October 5, 2016

- *Тьюринг-полный* язык программирования
- Основан на идеях диалектов LISP и ML
- Чисто функциональный язык программирования
- Ленивая стратегия выполнения
- Мощный компилятор и вывод типов

***Неформальный способ определения***  
*Работа с бесконечной памятью + в*  
*бесконечном времени*

***Неформальный способ определения***  
*Работа с бесконечной памятью + в*  
*бесконечном времени*

Примеры тьюринг-полных концепций?

## **Тьюринг-полная концепция**

*Любой известный вам язык программирования*

*— C, C++, Java, etc.*

- *Память бесконечная?*
- *Время бесконечное?*

## ***Не тьюринг-полная концепция***

*Конечный автомат (автомат разбора, автомат Мура, автомат Мили)*

- Память бесконечная?*
- Время бесконечное?*

- Тьюринг-полный язык программирования
- Основан на идеях диалектов *LISP* и *ML*
- Чисто функциональный язык программирования
- Ленивая стратегия выполнения
- Мощный компилятор и вывод типов

См. предыдущую лекцию



- Тьюринг-полный язык программирования
- Основан на идеях диалектов LISP и ML
- *Чисто функциональный язык программирования*
- Ленивая стратегия выполнения
- Мощный компилятор и вывод типов

# Чистая функция

- Функция и в программном, и в математическом смысле
- Отсутствие побочных эффектов
  - Вывод в файл или консоль
  - Сетевое соединение
  - Системный вызов
  - etc.

# Чистая функция

- Функция и в программном, и в математическом смысле
- Отсутствие побочных эффектов
  - Вывод в файл или консоль
  - Сетевое соединение
  - Системный вызов
  - etc.

А имеет ли вообще смысл программа без побочных эффектов?

Изменение состояния (глобальных) данных— это тоже побочный эффект.

⇒ *Нет глобальных переменных*

Изменение состояния (глобальных) данных— это тоже побочный эффект.

⇒ *Нет глобальных переменных*

Фактически, нет и локальных переменных.

Изменение состояния (глобальных) данных— это тоже побочный эффект.

⇒ *Нет глобальных переменных*

Фактически, нет и локальных переменных.

Переменных вообще нет.

Можно ли программировать без переменных?

*Более обще*

*Нет изменяемого состояния (mutable state).*

*Более обще*

*Нет изменяемого состояния (mutable state).*

Можно ли программировать без изменяемого состояния?



- Функции являются объектами первого порядка.
- Функции могут принимать и возвращать функции.

# Функциональный подход

- Функции являются объектами первого порядка.
- Функции могут принимать и возвращать функции.

Что из этого доступно в C++, Java?

## Анонимные функции (лямбда-выражения)

- Похожи на выражения в лямбда-исчислении
- Имеются во многих языках, в том числе Python, C++11, C#4, Java8
- Вместо имени функции ставится  $\lambda$

# Анонимные функции (лямбда-выражения)

- Похожи на выражения в лямбда-исчислении
- Имеются во многих языках, в том числе Python, C++11, C#4, Java8
- Вместо имени функции ставится  $\lambda$

$\lambda x \rightarrow x + 1$

## Каррирование (currying)

- Изобретение приписывается Хаскеллу Карри (Haskell Curry)
- Выражение функций от  $n$  параметров как функций от одного параметра

```
foo a b c = a + b * c
```

```
foo a = \ b -> \ c -> a + b * c
```

- Тьюринг-полный язык программирования
- Основан на идеях диалектов LISP и ML
- Чисто функциональный язык программирования
- **Ленивая стратегия выполнения**
- Мощный компилятор и вывод типов

$f(g(), h())$

Strict (eager) evaluation:

1. Вычисляем  $_1 = g()$
2. Вычисляем  $_2 = h()$
3. Вычисляем  $f(_1, _2)$

Lazy evaluation:

Вычисляем  $f(_1, _2)$ , если где-то внутри используются значения  $_1$  или  $_2$ , вычисляем их

## Ленивая стратегия выполнения: пример

C++: язык со строгим выполнением

```
int f(int p1, int p2) {  
    return -p1;  
}  
int g() {  
    return 2;  
}  
int h() {  
    throw SomeNastyError(":-P");  
}  
  
f(g(),h()); // throws exception
```



## Ленивая стратегия выполнения: пример

Haskell: язык с ленивым выполнением

```
f :: Int -> Int -> Int
```

```
f p1 p2 = -p1
```

```
g :: Int
```

```
g = 2
```

```
h :: Int
```

```
h = error ":-P"
```

```
f g h ==> -2
```

# Ленивая стратегия выполнения: новая ли это концепция?

В C++ и Java достаточно примеров ленивого выполнения:

- `if` ленивый по определению
  - Если бы это было не так, необходимо было бы:
    - Вычислить ветвь для `true`
    - Вычислить ветвь для `false`
    - Выбрать из двух **результатов** нужный . . .

## Ленивая стратегия выполнения: новая ли это концепция?

В C++ и Java достаточно примеров ленивого выполнения:

- Логические операции `&&` и `||`

```
if(a == null
    && a.length > 1
    && a[1] == 40) {
// everything is ok
}
if(a == null
    || a.length <= 1
    || a[1] == 40) {
// everything is bad as hell
}
```

[https://en.wikipedia.org/wiki/Evaluation\\_strategy](https://en.wikipedia.org/wiki/Evaluation_strategy)

- Eager evaluation:
  - Call-by-value  
Вычисляем аргумент до вызова функции и передаём туда его значение
  - Call-by-reference  
Вычисляем аргумент до вызова функции и передаём в функцию ссылку на него
  - etc.

[https://en.wikipedia.org/wiki/Evaluation\\_strategy](https://en.wikipedia.org/wiki/Evaluation_strategy)

- Lazy evaluation:
  - Call-by-name  
Вычисляем аргумент по мере необходимости каждый раз, когда он нужен
  - Call-by-need  
Вычисляем аргумент в какой-то момент времени до того как он понадобится, и запоминаем его

- Тьюринг-полный язык программирования
- Основан на идеях диалектов LISP и ML
- Чисто функциональный язык программирования
- Ленивая стратегия выполнения
- Мощный **компилятор** и **вывод типов**

# Средства разработки для языка Haskell

- Компилятор GHC, текущая версия — 7.0.3
  - Поддержка большинства существующих расширений
  - Входит в т.н. Haskell Platform, <http://www.haskell.org>
- Система пакетов
  - Единая база пакетов — Hackage
  - Управление (установка и обновление) пакетов утилитой Cabal
- IDE: IntelliJ for Haskell, EclipseFP, Leksah

## Запуск в режиме интерпретатора: GHCi

```
[root@vpupkin ~]$ ghci
GHCi, version 7.10.1: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Loading package ffi-1.0 ... linking ... done.
Prelude> 2 + 2
4
Prelude> 2 ** 2
4.0
Prelude>
```



- Позволяет не писать тип выражения
- Иногда типы функций могут быть очень сложными, в этом случае может помочь интерпретатор:

```
Prelude> let a = 2*2
Prelude> :t a
a :: Integer
```

## Обобщённые типы

Квантор общности —  $\forall$

Пример — функция `id`, имеющая тип `forall a. a -> a`,  
сокращённо просто `a -> a`

## Обобщённые типы

Квантор общности —  $\forall$

Пример — функция `id`, имеющая тип `forall a. a -> a`, сокращённо просто `a -> a`

```
Prelude> let iden a = a
```

## Обобщённые типы

Квантор общности —  $\forall$

Пример — функция `id`, имеющая тип `forall a. a -> a`, сокращённо просто `a -> a`

```
Prelude> let iden a = a
```

```
Prelude> let iden a = a
```

```
Prelude> :t iden
```

```
iden :: t -> t
```

```
Prelude> let a = 2*2
```

```
Prelude> :t iden a
```

```
iden a :: Integer
```

## Каррирование в интерпретаторе

```
Prelude> let foo x y = x + y
Prelude> :t foo
foo :: Num a => a -> a -> a
Prelude> let bar = \ x -> \ y -> x + y
Prelude> :t bar
bar :: Num a => a -> a -> a
Prelude> foo 2 3
5
Prelude> bar 2 3
5
```

# Hello world!

- Ввод-вывод в Haskell связан с побочными эффектами
- Побочные эффекты «спрятаны» под монадой IO
- Монады мы будем изучать сильно позже

# Hello world!

- Ввод-вывод в Haskell связан с побочными эффектами
- Побочные эффекты «спрятаны» под монадой IO
- Монады мы будем изучать сильно позже
  
- На данный момент можно обойтись функцией `interact`:

```
program :: String -> String  
program input = "Hello world"
```

```
main = interact program
```

Поскольку нет последовательности высказываний, if это тоже выражение, имеющее значение

Синтаксис: `if x then y else z`



# Факториал в Haskell

```
fact i = if i <= 1 then 1 else i * fact (i - 1)
```

В языке Haskell *строгая статическая* система типов:

- У всего есть тип, известный в процессе компиляции
- Нет неявных приведений типов

```
double foo(double x);
```

```
int y = 52;  
foo(y);
```

В языке Haskell *строгая статическая* система типов:

- У всего есть тип, известный в процессе компиляции
- Нет перегруженных функций

```
double bar(double x);  
double bar(const char* str);  
double bar(int x);
```

```
int y = 52;  
bar(y);
```

# Строгая типизация

```
Prelude> let foo :: Double -> Double; foo x = x + 3.14
```

```
Prelude> let y :: Integer; y = 42
```

```
Prelude> foo y
```

```
<interactive>:4:5:
```

```
    Couldn't match expected type 'Double' with actual type 'Integer'
```

```
    In the first argument of 'foo', namely 'y'
```

```
    In the expression: foo y
```

Демо?

