

Введение в функциональное программирование

1. Основы Haskell

- Чисто функциональный: нет понятия состояния, каждая переменная — константа.
- Сильно типизированный: каждое значение имеет постоянный тип.
- Декларативный: программа состоит из ряда деклараций типа и определений значений, а не команд.
- Скомпилированный: каждая программа должна быть скомпилирована в бинарный файл перед исполнением.
- Ленивый: значение вычисляется только тогда, когда оно становится необходимо.

Пример императивного кода:

```
#include <stdio.h>
void main() {
    int a;
    scanf("%d", &a);
    if (a == 0) {
        printf("zero\n");
    } else {
        printf("not zero\n");
    }
}
```

Пример декларативного кода:

```
sayZero :: Int -> IO ()
sayZero 0 = putStrLn "zero"
sayZero _ = putStrLn "not zero"

main :: IO ()
main = getLine >= (sayZero . read)
```

Разбор примера выше:

- `()` — это тип данных в Haskell, обозначающий «ничего». Кортеж с 0 элементов.
- `IO` — это модификатор типов. Если `a` — это произвольный тип, то `IO a` — тип, который «производит какие-то операции ввода/вывода, а потом возвращает объект типа `a`». `putStrLn "zero"` имеет тип `IO ()`. `readLine` имеет тип `IO String`.
- `::` — декларация типа. `sayZero` — это функция из `Int` в `IO ()`. `main` — это объект типа `IO ()`.
- Нотация определения функции.
- `read` — это функция, которая парсит значения из строк.

```
read :: (Read a) => String -> a

read "20" :: Int
read "20" :: Float
```

Это пример полиморфизма в Haskell — все типы статичны, но одна и та же функция может принимать и возвращать сразу несколько типов.

- `Read` — это класс (как интерфейс в Java). Разные типы могут быть или не быть элементами класса. `Read` включает те типы, которые можно «распарсить».
- `(.)` — это композиция функций.

```
read :: String -> Int
sayZero :: Int -> IO ()

(sayZero . read) :: String -> IO ()
```

- `>=` — оператор применения монадической функции. (...proceeds to explain monads in Haskell...)

2. Первые функции, if-утверждения, рекурсия

```
f :: Int -> Int -- (optional)
f n = n*2
```

```
g :: Bool -> Int -> Int -- (Bool -> (Int -> Int))
g switch n = if switch then n*2 else n*3 -- (if [bool] then [type] else [same type])

(g True) = f -- currying
```

В Haskell **нет** циклов, но есть *рекурсия*:

```
// example in c
int sum (int n) {
    int res = 0;
    for (int i = 0; i <= n; i++) {
        res += i;
    }
    return res;
}
```

```
-- example in haskell
sum :: Int -> Int
sum 0 = 0
sum n = n + sum (n-1)
```

Упражнения:

-