# Лекция 1: F#, значения, типы, функции, типы функций, коллекции, map & fold

F# — язык программирования от Microsoft Research, вобравший в себя лучшее из миров функционального, императивного и объектно-ориентированного программирования.

*Императивное программирование* происходит от идеи под названием «*машина Тьюринга*» *(Turing machine)*: мы можем представить вычисление как последовательность команд, изменяющих некое *состояние*, хранимое в памяти.

*Функциональное программирование* происходит от *лямбда-исчисления* *(lambda calculus)*, изобретённого Алонзо Чёрчем: мы можем представить вычисление как задание и применение функций.

Функциональное программирование избегает изменения однажды заданных значений, поэтому в нём **нет** *оператора присваивания*.

## Значения

В F# можно записать различные *значения (value)*, такие как строки, числа, символы, … Каждое значение в F# имеет *тип (type)*: т.е. принадлежит к некому, наперёд заданному *множеству* возможных значений типа и допускает применение определённых операций к себе.

Типы в большинстве случаев выводятся автоматически, это называется *выведением типов (type inference)*.

Значениям можно давать имена с помощью оператора let:

let a = 300 //целое число, тип: integer

let b = 500.5 //вещественное число, тип: float

let str = "This is BlackJack!" //строка, тип: string

let delimiter = ' ' //символ, тип: char

Выведенный тип можно увидеть, если навести мышку на название значения или отправить написанное в *интерпретатор* с помощью комбинации клавиш **Alt+Enter**.

F# является *строго типизированным (strongly typed)* языком, т.е. компилятор не позволяет выполнять операции на некорректных типах и не производит приведения (т.е. изменения) типов без прямого указания программиста:

0.5 + 5 //ошибка

Для приведения типов существует набор функций, с помощью которых программист может конкретизировать свои намерения:

(int 0.5) + 5

0.5 + (float 5)

int (0.5 + (float 5))

## Функции

Функции являются такими же значениями, как и значения других типов. Мы можем легко задать *анонимную функцию (lambda expression, anonymous function)* с помощью ключевого слова fun:

(fun x -> x \* 2) //функция, тип: int -> int

Мы так же легко можем дать ей имя:

let f = (fun x -> x \* 2)

А можно записать это по-другому:

let f x = x \* 2

Естественно, функции были бы бесполезны, если бы мы не могли применять их к неким значениям, чтобы получить результат:

f 3

//равносильно:

(fun x -> x \* 2) 3 //x теперь равен 3

3 \* 2 // = 6

Функции можно передавать как аргументы других функций, например, следующая функция применяет переданную ей функцию f к некому x три раза подряд:

let g f x = f (f (f x))

g (fun x -> x \* 2) 5 // = 40

## Частичное применение и типы функций

Тип функции g можно записать как:

Что это значит? Это значит, что функция g принимает два аргумента: функцию f, которая принимает какое-то значение любого типа и возвращает результат того же типа, и некое значение этого же типа x.

1) Как вы думаете, почему F# решил, что тип на выходе функции f должен быть тот же, что и на входе? Почему тип в данном случае некорректен?

Большинство функций в F# не простые, а *каррированные* (*currying*, в честь математика Хаскелла Карри). Это не значит, что кто-то добавил в них индийскую приправу ☺, а значит то, что если функция принимает n аргументов, то на самом деле она принимает только один аргумент и возвращает функцию, которая принимает n-1 аргументов, в свою очередь принимающую только один аргумент и возвращающую функцию, которая принимает n-2 аргументов, и так далее.

Например, функция, складывающая два числа, может быть записана так:

let add x y = x + y

А её тип так:

Таким образом, мы можем передать ей только один из двух аргументов (это называется *частичным применением, partial application*) и получить функцию, в теле которой все упоминания первого аргумента будут механически заменены на то значение, которое мы передали:

let addFive = add 5 // = 5 + y

addFive 3 // = 5 + 3 = 8

Тип функции addFive получается как бы сокращением типа функции add:

2) Как вы думаете, что делает функция doSomething, заданная следующим образом:

let doSomething = (fun x y z -> x \* y - z) 3 4

?

## Коллекции

Иногда нам хочется работать не с отдельными значениями, а с целыми наборами значений. Для этого у нас есть специальные типы, т.н. типы коллекций. Мы рассмотрим два из них: списки и массивы.

### Списки

Список — это упорядоченный набор значений какого-то типа. В F# для записи типа списка используется слово list, предваряемое типом элементов списка. Например, int list —список целых, string list — список строк, int list list — список списков целых.

Чтобы задать список, мы можем просто перечислить его элементы, разделяя их точкой с запятой и заключив всё это в прямоугольные скобки:

[ 1; 2; 3 ] //int list, список целых

[ "slow"; "poke"; "F#"; "cool" ] //string list, список строк

[ [1;2;3]; [4;5;6]; [7;8;9] ] //int int list, список списков целых

Но так не задать большие списки! На помощь приходят lists comprehensions, особый синтаксис языка для создания списков. Например, мы можем просто записать диапазон значений, которые должны войти в список:

[ 1..100 ] //список целых от 1 до 100 включительно

//список вещественных чисел от 5.0 до 10.0 с шагом 0.5

[ 5.0 .. 0.5 .. 10.0 ]

Или же, мы можем записать некоторое выражение, которое пройдётся по такому диапазону, произведёт какие-то преобразования с каждым элементом и сгенерирует новый список:

//список квадратов чисел от 0 до 5:

//[0; 1; 4; 9; 16; 25]

[ for x in 0..5 -> x \* x ]

Список можно представить как штуку, состоящую из головы (первого элемента списка) и хвоста (всех элементов списка, кроме первого). Чтобы отделить голову от хвоста, мы можем воспользоваться сопоставлением с образцом (pattern matching):

let x::xs = [1; 2; 3]

//x теперь равен 1, а xs = [2; 3]

Таким образом, на самом деле запись списков через точку с запятой — это просто сокращение, внутри они выглядят так:

// [] — пустой список

1 :: (2 :: (3 :: []))

А магические два двоеточия — на самом деле функция (СЮРПРИЗ) с типом .

### Массивы

Массивы — это тоже упорядоченные коллекции каких-то значений. Их можно задавать так же как списки, но нужно писать не просто прямоугольные скобки, а прямоугольные скобки с вертикальной чертой:

//массив кубов чисел от 1 до 1000000

[| for x in 1..1000000 -> x \* x \* x |]

Массивы нельзя так просто разделить на голову и хвост, зато к каждому элементу массива можно обратиться по его номеру (индексу), причём первый элемент массива имеет индекс 0:

let arr = [| 1..10 |]

arr.[0] //1

arr.[2] //3

arr.[9] //10

arr.[10] //ошибка, элементы кончились!

Тип массива записывается с помощью слова array или квадратных скобочек без вертикальных черт, например int array или int [] — тип массива целых чисел.

## Map

Уметь задавать коллекции — хорошо, уметь работать с ними — лучше! К счастью, F# содержит множество функций для этого. Функции для работы со списками живут в *модуле* List, а для работы с массивами — в модуле Array.

Одна из двух самых классных функций в этих модулях — map. Она умеет применять какую-то другую, переданную ей как аргумент, функцию ко всем элементам коллекции. Например, так мы можем разделить все элементы списка на два:

List.map (fun x -> x / 2.0) [ 1. .. 100. ]

А вот так можно превратить все элементы массива в строки:

Array.map string [| 1..100 |]

Мы можем сами создать функцию map c помощью рекурсии:

let rec map f list =

match list with //pattern matching

| [] -> [] //если список пустой, то возвращаем пустой список

| x::xs -> (f x)::(map f xs)

## Fold

Тем не менее, map не может решить всех проблем человечества — она должна обработать каждый элемент списка и вернуть список той же длины, а нам иногда хочется получить какое-то общее для всего списка значение, проанализировав все его элементы. Это можно сделать с помощью функции fold.

Fold принимает 3 аргумента: некую функцию f, некое начальное значение аккумулятора и список, а возвращает конечное значение аккумулятора. f же умеет из текущего значения аккумулятора и текущего элемента списка получать новое значение аккумулятора.

Вначале fold берёт переданное ей начальное значение аккумулятора, первый элемент списка, применяет f к ним, повторяет всё это для нового (полученного из f) значения аккумулятора и следующего элемента списка и так далее, пока список не закончится. В конце она просто возвращает последнее значение аккумулятора.

Вот так можно сложить все элементы списка:

List.fold (fun acc elem -> acc + elem) 0 [1..100]

//или короче:

List.fold (+) 0 [1..100]

Давайте попробуем сами написать fold. Вот что у нас может получиться:

//самодельный fold для списка

let rec fold f init list = //rec потому что функция рекурсивная

match list with //pattern matching

| [] -> init //если пустой список — возвращаем аккумулятор

| x::xs -> fold f (f init x) xs //иначе применяем f к голове и тек. аккумулятору и продолжаем для хвоста

Другой пример применения fold’a — мы можем найти минимальное в массиве число:

Array.fold (fun acc elem -> if acc > elem then elem else acc) Int32.MaxValue [| 3; -6; 5 |]

## |> и <|

Иногда удобнее писать не f x y z, а, например f x y <| z или z |> f x y. Операторы <| и |> просто меняют местами функцию и аргумент и позволяют избавиться от лишних скобок.

//todo: git, tryfsharp, книги, встречи в 5 в среды, индивидуальные задания, контакты

# Задания к лекции 1

Для каждого из вас я сделал отдельный файл, названный по вашему имени. Вы можете писать код в нём и коммитить изменения на github.

В этих файлах так же содержится вспомогательный код, который поможет вам начать.

## Для Ирины

1. Создать массив с 10000 последовательных целых чисел, начиная с 5.
2. Дать ему имя
3. Посчитать сумму этих чисел с помощью fold и с помощью Array.sum
4. Найти сумму всех чисел, делящихся нацело на 3 или 5 и меньших 1000.

## Для Камилы

1. Создать список с чётными числами от 1 до 200
2. Дать ему имя
3. Посчитать сумму этих чисел с помощью fold и с помощью List.sum
4. Построить список из первых 10000 чисел Фибоначчи. На первых двух позициях должны стоять 1, на следующих за ними — суммы чисел в двух предыдущих позициях. Например, вот начало такого списка: [1, 1, 2, 3, 5, 8, …]

## Для Сергея

1. Напиши функцию, которая умеет находить [моду](http://goo.gl/bsrrn) массива чисел.

## Для Николая

1. Напиши функцию, которая принимает некую функцию f и список, и возвращает список, который состоит только из элементов первого списка, для которых f вернула true.