

Функциональное программирование на языке F#

Введение

Юрий Литвинов

19.02.2019г

О чём этот курс

- ▶ Теория и практика функционального программирования
 - ▶ λ -исчисление
 - ▶ Базовые принципы ФП (программирование без состояний, функции высших порядков, каррирование и т.д.)
 - ▶ Типы в функциональном программировании (немутабельные коллекции, генерики, автообобщение и т.д.)
 - ▶ Паттерны функционального программирования (CPS, монады, point-free)
- ▶ Программирование на F#
 - ▶ ООП в F#
 - ▶ Асинхронное и многопоточное программирование в F#
 - ▶ Может, анализ данных и машинное обучение

Императивное программирование

Программа как последовательность **операторов**, изменяющих **состояние** вычислителя.

Для конечных программ есть **начальное состояние**, **конечное состояние** и последовательность переходов:

$$\sigma = \sigma_1 \rightarrow \sigma_2 \rightarrow \dots \rightarrow \sigma_n = \sigma'$$

Основные понятия:

- ▶ Переменная
- ▶ Присваивание
- ▶ Поток управления
 - ▶ Последовательное исполнение
 - ▶ Ветвления
 - ▶ Циклы

Функциональное программирование

Программа как вычисление значения **выражения** в математическом смысле на некоторых входных данных.

$$\sigma' = f(\sigma)$$

- ▶ Нет состояния \Rightarrow нет переменных
- ▶ Нет переменных \Rightarrow нет циклов
- ▶ Нет явной спецификации потока управления

Порядок вычислений не важен, потому что нет состояния, результат вычисления зависит только от входных данных.

Сравним

C++

```
int factorial(int n) {  
    int result = 1;  
    for (int i = 1; i <= n; ++i) {  
        result *= i;  
    }  
    return result;  
}
```

F#

```
let rec factorial x =  
    if x = 1 then 1 else x * factorial (x - 1)
```

Как с ЭТИМ ЖИТЬ

- ▶ Состояние и переменные «эмулируются» параметрами функций
- ▶ Циклы «эмулируются» рекурсией
- ▶ Последовательность вычислений — рекурсия + параметры

F#

```
let rec sumFirst3 ls acc i =  
    if i = 3 then  
        acc  
    else  
        sumFirst3  
            (List.tail ls)  
            (acc + ls.Head)  
            (i + 1)
```

Зачем

- ▶ Строгая математическая основа
- ▶ Семантика программ более естественна
 - ▶ Применима математическая интуиция
- ▶ Программы проще для анализа
 - ▶ Автоматический вывод типов
 - ▶ Оптимизации
- ▶ Более декларативно
 - ▶ Ленивость
 - ▶ Распараллеливание
- ▶ Модульность и переиспользуемость
- ▶ Программы более выразительны

Пример: функции высших порядков

F#

```
let sumFirst3 ls =  
    Seq.fold  
        (fun x acc -> acc + x)  
        0  
        (Seq.take 3 ls)
```

F#

```
let sumFirst3 ls = ls |> Seq.take 3 |> Seq.fold (+) 0
```

F#

```
let sumFirst3 = Seq.take 3 >> Seq.fold (+) 0
```


Ещё пример

Возвести в квадрат и сложить все чётные числа в списке

F#

let calculate =

Seq.filter (fun x -> x % 2 = 0)

>> Seq.map (fun x -> x * x)

>> Seq.reduce (+)

Почему тогда все не пишут функционально

- ▶ Чистые функции не могут оказывать влияние на внешний мир. Ввод-вывод, работа с данными, вообще выполнение каких-либо действий не укладывается в функциональную модель.
- ▶ Сложно анализировать производительность, иногда функциональные программы проигрывают в производительности императивным. «Железо», грубо говоря, представляет собой реализацию машины Тьюринга, тогда как функциональные программы определяются над λ -исчислением.
- ▶ Требуется математический склад ума и вообще желание думать.

Задачи

1. Посчитать факториал
2. Посчитать числа Фибоначчи (за линейное время)
3. Реализовать функцию обращения списка (за линейное время)
4. Реализовать функцию, которая принимает на вход n и m и возвращает список из элементов $[2^n; 2^{n+1}; \dots; 2^{n+m}]$