

# Функциональное программирование на языке F#

Юрий Литвинов  
y.litvinov@spbu.ru

11.02.2026

# О чём этот курс

- ▶ Теория и практика функционального программирования
  - ▶  $\lambda$ -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#
  - ▶ Компиляторы

# Отчётность

- ▶ Домашка (много несложной)
- ▶ Одна контрольная в середине семестра
- ▶ Доклад
- ▶ Сдача и учёт через HwProj 2
  - ▶ <https://hwproj.ru/courses/50070>



# Критерии оценивания

- ▶ Баллы за домашние задачи, баллы за контрольную
- ▶ Общий балл за домашки:  $MAX(0, (n/N - 0.6)) * 2.5 * 100$
- ▶ Общий балл за контрольные:  $n/N * 100$
- ▶ Итоговая оценка: минимум из этих двух баллов
- ▶ Дедлайны по домашкам, -1 балл за каждую неделю после дедлайна
- ▶ Сгорает не более половины баллов
  - ▶ Таким образом, задачи на 1 балл не штрафуются вовсе
  - ▶ Но их сделать проще, чем не сделать

# Ориентировочно

- ▶ Всего около 32 баллов за д/з и 30 за к/р
  - ▶ А — 31 балла за домашки, 27 за контрольную
  - ▶ В — 30 баллов за домашки, 24 за контрольную
  - ▶ С — 29 баллов за домашки, 21 за контрольную
  - ▶ D — 28 баллов за домашки, 18 за контрольную
  - ▶ E — 26 баллов за домашки, 15 за контрольную
- 
- ▶ Разбалловка может измениться!
  - ▶ Нагрузка правда меньше, делайте хорошие практики

# 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
  - ▶ Строгая типизация
- ▶ Более декларативно
  - ▶ В теории — ленивость
  - ▶ Ещё более в теории — распараллеливание
- ▶ Модульность и переиспользуемость
- ▶ Программы более выразительны

## Пример: функции высших порядков

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

⇓

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

⇓

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

⇓

```
let sumFirst3 = Seq.take 3 >> Seq.sum
```

## Ещё пример

Возвести в квадрат и сложить все чётные числа в списке

```
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  $\lambda$ -исчислением.
- ▶ Требуется математический склад ума и вообще желание думать.

# F#

- ▶ Типизированный функциональный язык для платформы .NET
- ▶ НЕ чисто функциональный (можно императивный стиль и ООП)
- ▶ Первый раз представлен публике в 2005 г., актуальная версия — 10.0 (11 ноября 2025 года)
- ▶ Создавался под влиянием OCaml (практически диалект OCaml под .NET, но со своей важной спецификой)
- ▶ Использует .NET CLI
- ▶ Компилируемый и интерпретируемый
- ▶ Иногда используется в промышленности, в отличие от многих чисто функциональных языков

## Что скачать и поставить

- ▶ Visual Studio Community
- ▶ .NET 10 + Visual Studio Code + Ionide
- ▶ Прямо в браузере: <https://dotnetfiddle.net/>
- ▶ Rider

# Пример программы

```
printfn "%s" "Hello, world!"
```



# let-определение

```
let x = 1
```

```
let x = 2
```

```
printfn "%d" x
```

можно читать как

```
let x = 1 in let x = 2 in printfn "%d" x
```

и понимать как подстановку  $\lambda$ -терма (про которые через одну пару)

# let-определение, функции

```
let powerOfFour x =  
  let xSquared = x * x  
  xSquared * xSquared
```

- ▶ Позиционный синтаксис
  - ▶ Отступы строго пробелами
  - ▶ Не надо ";"
- ▶ Нет особых синтаксических различий между переменной и функцией
- ▶ Не надо писать типы
- ▶ Не надо писать *return*

## Вложенные let-определения

```
let powerOfFourPlusTwoTimesSix n =  
  let n3 =  
    let n1 = n * n  
    let n2 = n1 * n1  
    n2 + 2  
  let n4 = n3 * 6  
  n4
```

- ▶  $n3$  — не функция!
- ▶ Компилятор отличает значения и функции по наличию аргументов
- ▶ Значение вычисляется, когда до *let* «доходит управление», функция — когда её вызовут. Хотя, конечно, функция — тоже значение.

# Типы

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

## F# Interactive

```
val f : x:int -> int
```

Каждое значение имеет тип, известный во время компиляции

# Элементарные типы

- ▶ *int*
- ▶ *double*
- ▶ *bool*
- ▶ *string*
- ▶ ... (.NET)
- ▶ *unit* — тип из одного значения, (). Аналог `void`.

## Кортежи (tuples)

```
let site1 = ("scholar.google.com", 10)
```

```
let site2 = ("citeseerx.ist.psu.edu", 5)
```

```
let site3 = ("scopus.com", 4)
```

```
let sites = (site1, site2, site3)
```

```
let url, relevance = site1
```

```
let site1, site2, site3 = sites
```

# Value Tuples

```
let origin = struct (0, 0)
```

```
let displace struct (x, y) struct (dx, dy)  
  = struct (x + dx, y + dy)
```

```
displace origin struct (1, 1)
```

# Лямбды

```
let primes = [2; 3; 5; 7]
let primeCubes = List.map (fun n -> n * n * n) primes
```

## F# Interactive

```
> primeCubes;;
val it : int list = [8; 27; 125; 343]
```

```
let f = fun x -> x * x
let n = f 4
```



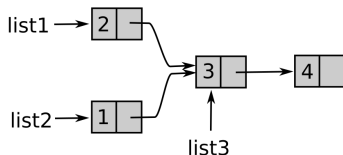
# Списки

Синтаксис	Описание	Пример
<code>[]</code>	Пустой список	<code>[]</code>
<code>[<i>expr</i>; ...; <i>expr</i>]</code>	Список с элементами	<code>[1; 2; 3]</code>
<code><i>expr</i> :: <i>list</i></code>	cons, добавление в голову	<code>1 :: [2; 3]</code>
<code>[<i>expr</i> .. <i>expr</i>]</code>	Промежуток целых чисел	<code>[1..10]</code>
<code>[<i>for x in list</i> → <i>expr</i>]</code>	Генерированный список	<code>[<i>for x in</i> 1..99 → <i>x</i> * <i>x</i>]</code>
<code><i>list</i> @ <i>list</i></code>	Конкатенация	<code>[1; 2] @ [3; 4]</code>

## Примеры работы со списками

```
let oddPrimes = [3; 5; 7; 11]
let morePrimes = [13; 17]
let primes = 2 :: (oddPrimes @ morePrimes)
let printFirst primes =
    match primes with
    | h :: t -> printfn "First prime in the list is %d" h
    | [] -> printfn "No primes found in the list"
```

# Устройство списков



```
let list3 = [3; 4]  
let list1 = 2 :: list3  
let list2 = 1 :: list3
```

- ▶ Списки немутабельны
- ▶ Cons-ячейки, указывающие друг на друга
- ▶ cons за константное время, @ — за линейное

# Операции над списками

Модуль Microsoft.FSharp.Collections.List

Функция	Описание	Пример	Результат
List.length	Длина списка	<i>List.length</i> [1; 2; 3]	3
List.nth	n-ый элемент списка	<i>List.nth</i> [1; 2; 3] 1	2
List.init	Генерирует список	<i>List.init</i> 3 ( <i>fun i</i> → <i>i * i</i> )	[0; 1; 4]
List.head	Голова списка	<i>List.head</i> [1; 2; 3]	1
List.tail	Хвост списка	<i>List.tail</i> [1; 2; 3]	[2; 3]
List.map	Применяет функцию ко всем элементам	<i>List.map</i> ( <i>fun i</i> → <i>i * i</i> ) [1; 2; 3]	[1; 4; 9]
List.filter	Отбирает нужные элементы	<i>List.filter</i> ( <i>fun x</i> → <i>x % 2 &lt;&gt; 0</i> ) [1; 2; 3]	[1; 3]
List.fold	"Свёртка"	<i>List.fold</i> ( <i>fun x acc</i> → <i>acc * x</i> ) 1 [1; 2; 3]	6

# Тип Option

Либо *Some* что-то, либо *None*, представляет возможное отсутствие значения.

```
let people = [ ("Adam", None); ("Eve" , None);
  ("Cain", Some("Adam","Eve"));
  ("Abel", Some("Adam","Eve")) ]
```

```
let showParents (name, parents) =
  match parents with
  | Some(dad, mum) ->
    printfn "%s, father %s, mother %s" name dad mum
  | None -> printfn "%s has no parents!" name
```

# Рекурсия

```
let rec length l =  
  match l with  
  | [] -> 0  
  | h :: t -> 1 + length t
```

```
let rec even n = (n = 0u) || odd(n - 1u)  
and odd n = (n <> 0u) && even(n - 1u)
```