

## Продолжение про F#

Юрий Литвинов  
y.litvinov@spbu.ru

12.02.2026

# Юнит-тестирование в F#

- ▶ Работают все дотнетовские библиотеки (NUnit, MsTest и т.д.)
- ▶ Есть обёртки, делающие код тестов более «функциональным» (FsUnit)
- ▶ Есть чисто F#-овские штуки: FsCheck, Unquote
  - ▶ на самом деле, не совсем F#-овские, но в C# такого нет
  - ▶ на самом деле, есть, это называется property-based testing и считается передовой техникой тестирования, в F# было всегда

## FsUnit, пример

```
module ``Project Euler - Problem 1`` =
```

```
  open NUnit.Framework
```

```
  open FsUnit
```

```
let GetSumOfMultiplesOf3And5 max =
```

```
  seq{3 .. max - 1}
```

```
  |> Seq.fold(fun acc number ->
```

```
    (if (number % 3 = 0 || number % 5 = 0) then
```

```
      acc + number else acc)) 0
```

```
[<Test>]
```

```
let ``Sum of multiples of 3 and 5 to 10 should return 23`` () =
```

```
  GetSumOfMultiplesOf3And5(10) |> should equal 23
```

## FsUnit, матчеры

```
1 |> should equal 1
1 |> should not' (equal 2)
10.1 |> should (equalWithin 0.1) 10.11
"ships" |> should startWith "sh"
"ships" |> should not' (endWith "ss")
"ships" |> should haveSubstring "hip"
[1] |> should contain 1
[] |> should not' (contain 1)
anArray |> should haveLength 4

(fun () -> failwith "BOOM!") |> ignore
    |> should throw typeof<System.Exception>

shouldFail (fun () -> 5/0 |> ignore)
```

## FsUnit, ещё матчеры

**true** |> should be True

**false** |> should **not**' (be True)

**""** |> should be EmptyString

**null** |> should be Null

anObj |> should **not**' (be sameAs otherObj)

11 |> should be (greaterThan 10)

10.0 |> should be (lessThanOrEqualTo 10.1)

0.0 |> should be ofExactType<**float**>

1 |> should **not**' (be ofExactType<**obj**>)

## FsUnit, и ещё матчеры

Choice<int, **string**>.Choice1Of2(42) |> should be (choice 1)

"test" |> should be instanceOfType<**string**>

"test" |> should **not**' (be instanceOfType<int>)

2.0 |> should **not**' (be NaN)

[1; 2; 3] |> should be unique

[1; 2; 3] |> should be ascending

[1; 3; 2] |> should **not**' (be ascending)

[3; 2; 1] |> should be descending

[3; 1; 2] |> should **not**' (be descending)

# Значения в модулях не инициализируются!

Как внезапно прострелить себе ногу

Main.fs:

```
let value = [1]
```

Test.fs:

```
[<Test>]  
let Test () =  
    Assert.AreEqual([1], Main.value)
```

Вывод:

Failed Test1 [24 ms]

Error Message:

Expected: < 1 >

But was: null

# FsCheck

**open** FsCheck

**let** revRevsOrig (xs:**list**<int>) = **List**.rev(**List**.rev xs) = xs

**Check**.Quick revRevsOrig

*// Ok, passed 100 tests.*

**let** revsOrig (xs:**list**<int>) = **List**.rev xs = xs

**Check**.Quick revsOrig

*// Falsifiable, after 2 tests (2 shrinks) (StdGen (338235241,296278002)):*

*// Original:*

*// [3; 0]*

*// Shrunk:*

*// [1; 0]*

Для интеграции с FsUnit используйте **Check.QuickThrowOnFailure**

# Unquote

Вообще интерпретатор F#-а, очень полезный для тестирования:

```
[<Test>]
```

```
let ``Unquote demo`` () =
```

```
    test <@ ([3; 2; 1; 0] |> List.map ((+) 1)) = [1 + 3..1 + 0] @>
```

```
// ([3; 2; 1; 0] |> List.map ((+) 1)) = [1 + 3..1 + 0]
```

```
// [4; 3; 2; 1] = [4..1]
```

```
// [4; 3; 2; 1] = []
```

```
// false
```

# Foq

Ну и, конечно же, mock-объекты:

```
[<Test>]
```

```
let ``Foq demo`` () =
```

```
    let mock = Mock<System.Collections.Generic.IList<int>>>()  
        .Setup(fun x -> <@ x.Contains(any()) @>).Returns(true)  
        .Create()
```

```
mock.Contains 1 |> Assert.True
```

# Каррирование, частичное применение

```
let shift (dx, dy) (px, py) = (px + dx, py + dy)
let shiftRight = shift (1, 0)
let shiftUp = shift (0, 1)
let shiftLeft = shift (-1, 0)
let shiftDown = shift (0, -1)
```

## F# Interactive

```
> shiftDown (1, 1);;
val it : int * int = (1, 0)
```

## Зачем — функции высших порядков

```
let lists = [[1; 2]; [1]; [1; 2; 3]; [1; 2]; [1]]  
let lengths = List.map List.length lists
```

или

```
let lists = [[1; 2]; [1]; [1; 2; 3]; [1; 2]; [1]]  
let squares = List.map (List.map (fun x -> x * x)) lists
```

Функции стандартной библиотеки стараются принимать список последним, для каррирования

# Оператор | >

Pipe forward

```
let (|>) x f = f x
```

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

ВМЕСТО

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

# Оператор >>

Композиция

```
let (>>) f g x = g (f x)
```

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

```
let result = sumFirst3 [1; 2; 3; 4; 5]
```

# Операторы `<|` и `<<`

Pipe-backward и обратная композиция

```
let (<|) f x = f x
```

```
let (<<) f g x = f (g x)
```

Зачем? Чтобы не ставить скобки:

```
printfn "Result = %d" <| factorial 5
```

# Использование библиотек .NET

open System.Windows.Forms

```
let form = new Form(Visible = false, TopMost = true, Text = "Welcome to F#")
let textB = new RichTextBox(Dock = DockStyle.Fill, Text = "Some text")
form.Controls.Add(textB)
```

open System.IO

open System.Net

*/// Get the contents of the URL via a web request*

```
let http(url: string) =
    let req = System.Net.WebRequest.Create(url)
    let resp = req.GetResponse()
    let stream = resp.GetResponseStream()
    let reader = new StreamReader(stream)
    let html = reader.ReadToEnd()
    resp.Close()
    html
```

```
textB.Text <- http("http://www.google.com")
```

```
form.ShowDialog () |> ignore
```

# Сопоставление шаблонов

```
let urlFilter url agent =  
    match (url, agent) with  
    | "http://www.google.com", 99 -> true  
    | "http://www.yandex.ru" , _ -> false  
    | _, 86 -> true  
    | _ -> false
```

```
let sign x =  
    match x with  
    | _ when x < 0 -> -1  
    | _ when x > 0 -> 1  
    | _ -> 0
```

## F# — не Prolog

Не получится писать так:

```
let isSame pair =  
    match pair with  
    | (a, a) -> true  
    | _ -> false
```

Нужно так:

```
let isSame pair =  
    match pair with  
    | (a, b) when a = b -> true  
    | _ -> false
```

# Какие шаблоны бывают

Синтаксис	Описание	Пример
$(pat, \dots, pat)$	Кортеж	$(1, 2, ("3", x))$
$[pat; \dots; pat]$	Список	$[x; y; 3]$
$pat :: pat$	cons	$h :: t$
$pat \mid pat$	"Или"	$[x] \mid [{"X"}; x]$
$pat \& pat$	"И"	$[p] \& [(x, y)]$
$pat \text{ as } id$	Именованный шаблон	$[x] \text{ as } inp$
$id$	Переменная	$x$
$\_$	Wildcard (что угодно)	$\_$
литерал	Константа	239, <i>DayOfWeek.Monday</i>
$?: type$	Проверка на тип	$?: string$

# Последовательности

## Ленивый тип данных

```
seq {0 .. 2}
```

```
seq {11 .. 1000000000000001}
```

```
open System.IO
```

```
let rec allFiles dir =
```

```
    Seq.append
```

```
    (dir |> Directory.GetFiles)
```

```
    (dir |> Directory.GetDirectories
```

```
        |> Seq.map allFiles
```

```
        |> Seq.concat)
```

# Типичные операции с последовательностями

Операция	Тип
Seq.append	$\#seq <' a > \rightarrow \#seq <' a > \rightarrow seq <' a >$
Seq.concat	$\#seq < \#seq <' a > > \rightarrow seq <' a >$
Seq.choose	$('a \rightarrow ' b \text{ option}) \rightarrow \#seq <' a > \rightarrow seq <' b >$
Seq.empty	$seq <' a >$
Seq.map	$('a \rightarrow ' b) \rightarrow \#seq <' a > \rightarrow \#seq <' b >$
Seq.filter	$('a \rightarrow bool) \rightarrow \#seq <' a > \rightarrow seq <' a >$
Seq.fold	$('s \rightarrow ' a \rightarrow ' s) \rightarrow ' s \rightarrow seq <' a > \rightarrow ' s$
Seq.initInfinite	$(int \rightarrow ' a) \rightarrow seq <' a >$

# Записи

```
type Person =  
    { Name: string  
      DateOfBirth: System.DateTime }  
  
{ Name = "Bill"  
  DateOfBirth = new System.DateTime(1962, 09, 02) }  
  
{ new Person  
  with Name = "Anna"  
  and DateOfBirth = new System.DateTime(1968, 07, 23) }
```

# Деконструкция

```
let person = { Name = "Anna"  
               DateOfBirth = new System.DateTime(1968, 07, 23) }
```

```
let { Name = name; DateOfBirth = date } = person
```

```
// деконструкция в параметре функции f  
let f { Name = name; DateOfBirth = date } = ..
```

## Анонимные записи

```
let person = { | Name = "Anna"; DateOfBirth = DateTime(1968, 07, 23) | }
```

- ▶ Могут возвращаться из функций (в отличие от анонимных объектов в C#)
- ▶ Имеют структурное равенство и сравнение

```
{ | a = 2 | } > { | a = 1 | } // true
```
- ▶ Не могут участвовать в сопоставлении с шаблоном

# Размеченные объединения

Discriminated unions

```
type Route = int
```

```
type Make = string
```

```
type Model = string
```

```
type Transport =
```

```
  | Car of Make * Model
```

```
  | Bicycle
```

```
  | Bus of Route
```

```
let bus = Bus(420)
```

# Известные примеры

```
type 'a option =  
    | None  
    | Some of 'a
```

```
type 'a list =  
    | ([])  
    | (::) of 'a * 'a list
```

# Использование размеченных объединений

```
type IntOrBool = I of int | B of bool
```

```
let i = I 99
```

```
let b = B true
```

```
type C = Circle of int | Rectangle of int * int
```

```
[1..10]
```

```
|> List.map Circle
```

```
[1..10]
```

```
|> List.zip [21..30]
```

```
|> List.map Rectangle
```

## Использование в match

```
type Tree<'a> =  
    | Tree of 'a * Tree<'a> * Tree<'a>  
    | Tip of 'a
```

```
let rec size tree =  
    match tree with  
    | Tree(_, l, r) -> 1 + size l + size r  
    | Tip _ -> 1
```

# Пример

Дерево разбора логического выражения

```
type Proposition =
```

```
| True
| And of Proposition * Proposition
| Or of Proposition * Proposition
| Not of Proposition
```

```
let rec eval (p: Proposition) =
```

```
  match p with
```

```
  | True -> true
  | And(p1, p2) -> eval p1 && eval p2
  | Or (p1, p2) -> eval p1 || eval p2
  | Not(p1) -> not (eval p1)
```

```
printfn "%A" <| eval (Or(True, And(True, Not True)))
```

## Взаимосвязанные типы

```
type Node =  
  { Name : string;  
    Links : Link list }  
and Link =  
  | Dangling  
  | Link of Node
```

## Одноэлементные объединения, без

```
type CustomerId = int // синоним типа
type OrderId = int // ещё один синоним типа

let printOrderId (orderId: OrderId) =
    printfn "The orderId is %i" orderId

let customerId = 1
printOrderId customerId // Печалааль
```

## Одноэлементные объединения, с

**type CustomerId** = CustomerId **of** int *// размеченное объединение*

**type OrderId** = OrderId **of** int *// ещё одно*

**let** printOrderId (OrderId orderId) = *// деконструкция в параметре*  
printfn "The orderId is %i" orderId

**let** customerId = CustomerId 1  
printOrderId customerId *// Ошибка компиляции*

## Факториал без хвостовой рекурсии

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

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

## Факториал с хвостовой рекурсией

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

## После декомпиляции в C#

```
C#  
  
public static int tailRecursiveFactorial(int x, int acc)  
{  
    while (true)  
    {  
        if (x <= 1)  
        {  
            return acc;  
        }  
        acc *= x;  
        x--;  
    }  
}
```

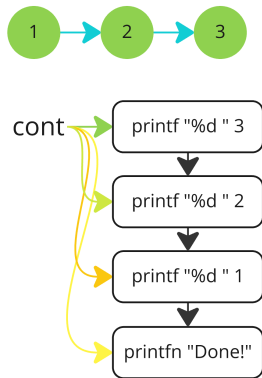
## Паттерн “Аккумулятор”

```
let rec map f list =  
  match list with  
  | [] -> []  
  | hd :: tl -> (f hd) :: (map f tl)  
  
let map f list =  
  let rec mapTR f list acc =  
    match list with  
    | [] -> acc  
    | hd :: tl -> mapTR f tl (f hd :: acc)  
  in mapTR f (List.rev list) []
```

# Continuation Passing Style

Аккумулятор — функция

```
let printListRev list =
  let rec printListRevTR list cont =
    match list with
    | [] -> cont ()
    | hd :: tl ->
      printListRevTR tl (fun () ->
        printf "%d " hd; cont () )
  printListRevTR list (fun () -> printfn "Done!")
```

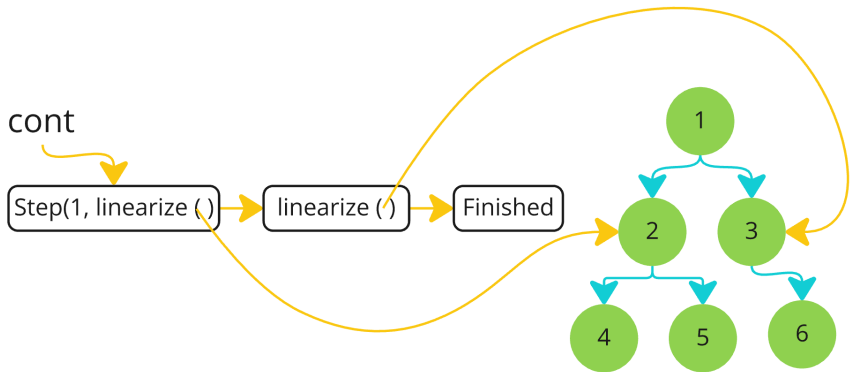


# Когда всё не так просто

```
type ContinuationStep<'a> =
    | Finished
    | Step of 'a * (unit -> ContinuationStep<'a>)
```

```
let rec linearize binTree cont =
    match binTree with
    | Empty -> cont()
    | Node(x, l, r) ->
        Step(x, (fun () -> linearize l (fun () ->
            linearize r cont)))
```

# Что происходит



## Собственно, обход

```
let iter f binTree =
  let steps = linearize binTree (fun () -> Finished)
```

```
  let rec processSteps step =
    match step with
    | Finished -> ()
    | Step(x, getNext) ->
        f x
        processSteps (getNext())
```

```
processSteps steps
```

# Пример шага обхода

