

# Язык F#

## Введение

Юрий Литвинов

11

# F#

- ▶ Типизированный функциональный язык для платформы .NET
- ▶ НЕ чисто функциональный (можно императивный стиль и ООП)
- ▶ Первый раз представлен публике в 2005 г.
- ▶ Создавался под влиянием OCaml (практически диалект OCaml под .NET)
- ▶ Использует .NET CLI
- ▶ Компилируемый и интерпретируемый
- ▶ Используется в промышленности, в отличие от многих чисто функциональных языков

# Что скачать и поставить

- ▶ Под Windows — Visual Studio, из коробки
- ▶ Под Linux
  - ▶ Mono + MonoDevelop + F# Language Binding, из репозитория
  - ▶ .NET Core + Visual Studio Code
- ▶ Прямо в браузере: <http://www.tryfsharp.org/Learn>

# Пример программы

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

Сравните с

```
namespace HelloWorld
```

```
{
```

```
    class Program
```

```
    {
```

```
        static void Main(string[] args)
```

```
        {
```

```
            System.Console.WriteLine("Hello, world!");
```

```
        }
```

```
    }
```

```
}
```

## Ещё пример

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

# 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

# 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
```

```
    ((x, y): struct (int * int))
```

```
    ((dx, dy): struct (int * int))
```

```
=
```

```
struct (x + dx, y + dy)
```

# 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

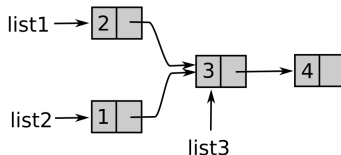
Синтаксис	Описание	Пример
<code>[]</code>	Пустой список	<code>[]</code>
<code>[<i>expr</i>; ...; <i>expr</i>]</code>	Список с элементами	<code>[1; 2; 3]</code>
<code><i>expr</i> :: <i>list</i></code>	<code>cons</code> , добавление в голову	<code>1 :: [2; 3]</code>
<code>[<i>expr</i> .. <i>expr</i>]</code>	Промежуток целых чисел	<code>[1..10]</code>
<code>[<i>for</i> <i>x</i> <i>in</i> <i>list</i> → <i>expr</i>]</code>	Генерированный список	<code>[<i>for</i> <i>x</i> <i>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>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>i</i> ) [1; 2; 3]	[1; 4; 9]
List.filter	Отбирает нужные элементы	<i>List.filter</i> ( <i>fun x</i> → <i>x</i> % 2 <> 0) [1; 2; 3]	[1; 3]
List.fold	"Свёртка"	<i>List.fold</i> ( <i>fun x acc</i> → <i>acc</i> * <i>x</i> ) 1 [1; 2; 3]	6
List.zip	Делает из двух списков список пар	<i>List.zip</i> [1; 2] [3; 4]	[(1, 3); (2, 4)]

# 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)
```

# 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 = " <| factorial 5
```

# 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)
```

# 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"	$[x] \mid ["X" \mid x]$
$pat \& pat$	"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 {1| .. 10000000000000|}
```

**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	$('a \rightarrow 'b\ option) \rightarrow \#seq <'a> \rightarrow seq <'b>$
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	$(int \rightarrow 'a) \rightarrow seq <'a>$

## Задание последовательностей

```
let squares = seq { for i in 0 .. 10 -> (i, i * i) }  
seq { for (i, isquared) in squares ->  
      (i, isquared, i * isquared) }
```

```
let checkerboardCoordinates n =  
  seq { for row in 1 .. n do  
    for col in 1 .. n do  
      if (row + col) % 2 = 0 then  
        yield (row, col) }
```

# 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) }
```

# Клонирование записей

```
type Car =  
  {  
    Make : string  
    Model : string  
    Year : int  
  }  
  
let thisYear's = { Make = "SomeCar";  
                  Model = "Luxury Sedan";  
                  Year = 2010 }  
let nextYear's = { thisYear's with Year = 2011 }
```

# Размеченные объединения

Discriminated unions

```
type Route = int
```

```
type Make = string
```

```
type Model = string
```

```
type Transport =
```

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

```
| Bicycle
```

```
| Bus of Route
```



# Известные примеры

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

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

# Использование размеченных объединений

```
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
```

# Замена цикла рекурсией

Рекурсивное разложение на множители

F#

```
let factorizeRecursive n =  
    let rec find i =  
        if i >= n then None  
        elif (n % i = 0) then Some(i, n / i)  
        else find (i + 1)  
    find 2
```

# Хвостовая рекурсия

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#

```
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)  
  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!")
```