Ликбез по F#

Юрий Литвинов

14.07.2021г

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)
```

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 (+)
```

F#

- Типизированный функциональный язык для платформы .NET
- НЕ чисто функциональный (можно императивный стиль и ООП)
- Первый раз представлен публике в 2005 г., актуальная версия — 5.0 (10 ноября 2020 года)
- Создавался под влиянием OCaml (практически диалект OCaml под .NET)
- Использует .NET CLI
- Компилируемый и интерпретируемый
- Иногда используется в промышленности, в отличие от многих чисто функциональных языков

Что скачать и поставить

- Под Windows — Visual Studio, Rider
- Под Linux — Rider, Visual Studio Code + Ionide
- Прямо в браузере: https://dotnetfiddle.net/

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 того, что справа от $=$ вместо того, что слева, в выражение после in

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*
- Нет возможности вернуться из функции раньше её конца

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
```

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

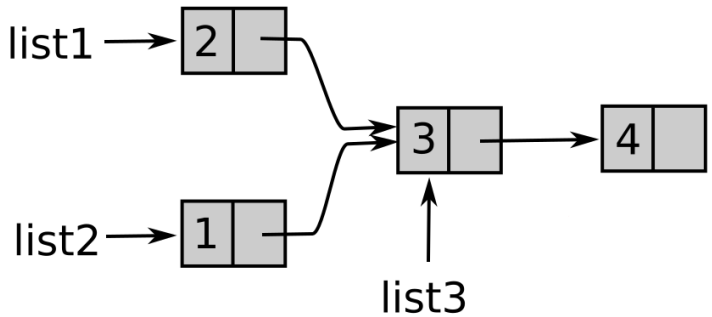
Синтаксис	Описание	Пример
$[]$	Пустой список	$[]$
$[expr; ...; expr]$	Список с элементами	$[1; 2; 3]$
expr :: *list*	cons, добавление в голову	$1 :: [2; 3]$
$[expr .. expr]$	Промежуток целых чисел	$[1..10]$
$[for\ x\ in\ list \rightarrow expr]$	Генерированный список	$[for\ x\ in\ 1..99 \rightarrow x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

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	*List.length* $[1;2;3]$	3
List.nth	n-ый элемент списка	*List.nth* $[1;2;3]$ 1	2
List.init	Генерирует список	*List.init* 3($fun\ i \rightarrow i * i$)	$[0;1;4]$
List.head	Голова списка	*List.head* $[1;2;3]$	1
List.tail	Хвост списка	*List.tail* $[1;2;3]$	$[2;3]$
List.map	Применяет функцию ко всем элементам	*List.map* ($fun\ i \rightarrow i * i$) $[1;2;3]$	$[1;4;9]$
List.filter	Отбирает нужные элементы	*List.filter* ($fun\ x \rightarrow x$ % $2 <> 0$) $[1;2;3]$	$[1;3]$
List.fold	”Свёртка”	*List.fold* ($fun\ x\ acc \rightarrow acc * x$) 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)
```

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
$(pat, \ldots, pat)$	Кортеж	$(1, 2, (\text{“}3\text{“}, x))$
$[pat; \ldots; pat]$	Список	$[x; y; 3]$
pat :: *pat*	cons	*h* :: *t*
pat | *pat*	”Или”	$[x] \mid [\text{“}X\text{“} ; x]$
pat & *pat*	”И”	$[p]\&[(x, y)]$
pat as id	Именованный шаблон	$[x]$ *as inp*
id	Переменная	*x*
_	Wildcard (что угодно)	_
литерал	Константа	239, *DayOfWeek.Monday*
:? *type*	Проверка на тип	:? *string*

Последовательности

Ленивый тип данных

```
seq {0 .. 2}
seq {1I .. 1000000000000I}
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	$\#seq <' a> \to \#seq <' a> \to seq <' a>$
Seq.concat	$\#seq < \#seq <' a >> \to seq <' a>$
Seq.choose	$('a \to' b\ option) \to \#seq <' a> \to seq <' b>$
Seq.empty	$seq <' a>$
Seq.map	$('a \to' b) \to \#seq <' a> \to \#seq <' b>$
Seq.filter	$('a \to bool) \to \#seq <' a> \to seq <' a>$
Seq.fold	$('s \to' a \to' s) \to' s \to seq <' a> \to' s$
Seq.initInfinite	$(int \to' a) \to seq <' a>$

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
```

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)))
```

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 resultOfRecusion = factorial (x - 1)
        let result = x * resultOfRecusion
        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)
    mapTR f (List.rev list) []
```

Шаблонные типы

```
type 'a list = ...
type list<'a> = ...
List.map : ('a -> 'b) -> 'a list -> 'b list
let map<'a,'b> : ('a -> 'b) -> 'a list -> 'b list =
    List.map

let rec map (f : 'a -> 'b) (l : 'a list) =
    match l with
    | h :: t -> (f h) :: (map f t)
    | [] -> []
```

Автоматическое обобщение

```
let getFirst (a, b, c) = a
let mapPair f g (x, y) = (f x, g y)
```

F# Interactive

```
val getFirst: 'a * 'b * 'c -> 'a
val mapPair : ('a -> 'b) -> ('c -> 'd)
    -> ('a * 'c) -> ('b * 'd)
```

Автоматический вывод типов

- Алгоритм Хиндли-Милнера
- На самом деле, «алгоритм *W* Дамаса-Милнера» над системой типов Хиндли-Милнера
  - Одно из типизированных λ-исчислений
  - Используется далеко не только в F#
- Построение системы уравнений над типами с учётом ограничений
  - литералы, функции и другие виды «взаимодействия значений», явные ограничения на типы, аннотации типов
- Решение методом унификации
  - Множество выражений и «переменных типа», им соответствующих
  - Постепенное уточнение этого множества
  - Если остались переменные типа, обобщение
  - Алгоритм глобальный!

Однако не всё так просто

```
List.map (fun x -> x.Length) ["hello"; "world"]
```

— не скомпилируется, в момент вызова x неизвестно, что x строка

```
["hello"; "world"] |> List.map (fun x -> x.Length)
```

— скомпилируется

```
List.map (fun (x: string) -> x.Length) ["hello"; "world"]
```

— или так

Методы у типов

```
type Vector = {x : float; y : float} with
    member v.Length = sqrt(v.x * v.x + v.y * v.y)

let vector = {x = 1.0; y = 1.0}
let length = vector.Length

type Vector with
    member v.Scale k = {x = v.x * k; y = v.y * k}

let scaled = vector.Scale 2.0
```

Статические методы

```
type Vector = {x : float; y : float} with
    static member Create x y = {x = x; y = y}

let vector = Vector.Create 1.0 1.0

type System.Int32 with
    static member IsEven x = x % 2 = 0

printfn "%b" <| System.Int32.IsEven 10
```

Методы и каррирование

```
open Operators

type Vector = {x : float; y : float} with
    static member Create x y = {x = x; y = y}

let transform v rotate scale =
    let r = System.Math.PI * rotate / 180.0
    { x = scale * v.x * cos r - scale * v.y * sin r;
      y = scale * v.x * sin r + scale * v.y * cos r }

type Vector with
    member v.Transform = transform v

printfn "%A" <| (Vector.Create 1.0 1.0).Transform 45.0 2.0
```

Каррирование против кортежей

```
type Vector with
    member v.TupledTransform (r, s) = transform v r s
    member v.CurriedTransform r s = transform v r s

let v = Vector.Create 1.0 1.0
printfn "%A" <| v.TupledTransform (45.0, 2.0)
printfn "%A" <| v.CurriedTransform 45.0 2.0
```

Кортежи: перегрузка

```
type Vector with
    member v.TupledTransform (r, s) =
      transform v r s
    member v.TupledTransform r =
      transform v r 1.0

let v = Vector.Create 1.0 1.0
printfn "%A" <| v.TupledTransform (45.0, 2.0)
printfn "%A" <| v.TupledTransform (90.0)
```

Кортежи против каррирования

За:

- Можно вызывать из .NET-кода
- Опциональные и именованные аргументы, перегрузки

Против:

- Не поддерживают частичное применение
- Не дружат с функциями высших порядков

Классы, основной конструктор

```
type Vector(x, y) =
    member v.Length = x * x + y * y |> sqrt

printfn "%A" <| Vector (1.0, 1.0)
```

F# Interactive

```
FSI_0003+Vector
type Vector =
  class
    new : x:float * y:float -> Vector
    member Length : float
  end
val it : unit = ()
```

Методы и свойства

```
type Vector(x : float, y : float) =
  member v.Scale s = Vector(x * s, y * s)
  member v.X = x
  member v.Y = y
```

F# Interactive

```
type Vector =
 class
  new : x:float * y:float -> Vector
  member Scale : s:float -> Vector
  member X : float
  member Y : float
 end
```

Private-поля и private-методы

```
type Vector(x : float, y : float) =
    let mutable mX = x
    let mutable mY = y
    let lengthSqr = mX * mX + mY * mY
    member v.Length = sqrt lengthSqr
    member v.X = mX
    member v.Y = mY
    member v.SetX x = mX <- x
    member v.SetY y = mY <- y
```

Мутабельные свойства

```
type Vector(x, y) =
    let mutable mX = x
    let mutable mY = y
    member v.X
        with get () = mX
        and set x = mX <- x
    member v.Y
        with get () = mY
        and set y = mY <- y
```

Автоматические свойства

```
type Vector(x, y) =
    member val X = x with get,set
    member val Y = y with get,set

let v = Vector(1.0, 1.0)
v.X <- 2.0
```

Вернёмся к конструкторам

Дополнительное поведение

```
type Vector(x : float, y : float) =
    let length () = x * x + y * y |> sqrt
    do
        printfn "Vector (%f, %f), length = %f"
            x y <| length ()
        printfn "Have a nice day"
    let mutable x = x
    let mutable y = y

let v = Vector(1.0, 1.0)
```

let-функции и методы

```
type Vector(x : float, y : float) =
    let length () = x * x + y * y |> sqrt
    let normalize () = Vector(x / length(), y / length())
    member this.Normalize = normalize
    member this.X = x
    member this.Y = y

let v = Vector(2.0, 2.0)
let v' = v.Normalize ()
```

Рекурсивные методы

```
type Math() =
    member this.Fibonacci x =
        match x with
        | 0 | 1 -> 1
        | _ -> this.Fibonacci (x - 1)
               + this.Fibonacci (x - 2)

let math = new Math()
printfn "%i" <| math.Fibonacci 10
```

Много конструкторов

```
type Vector(x : float, y : float) =
    member this.X = x
    member this.Y = y
    new () =
        printfn "Constructor with no parameters"
        Vector(0.0, 0.0)

let v = Vector(2.0, 2.0)
let v' = Vector()
```

Модификаторы видимости

```
type Example() =
    let mutable privateValue = 42

    member this.PublicValue = 1
    member private this.PrivateValue = 2
    member internal this.InternalValue = 3

    member this.PrivateSetProperty
        with get () =
            privateValue
        and private set(value) =
            privateValue <- value
```

Наследование

```
type Shape() =
    class
    end

type Circle(r) =
    inherit Shape()
    member this.R = r
```

Абстрактные классы

```
[<AbstractClass>]
type Shape() =
    abstract member Draw : unit -> unit
    abstract member Name : string

type Circle(r) =
    inherit Shape()
    member this.R = r
    override this.Draw () =
        printfn "Drawing circle"
    override this.Name = "Circle"
```

Реализация по умолчанию

```
type Shape() =
    abstract member Draw : unit -> unit
    abstract member Name : string
    default this.Draw () =
        printfn "Drawing shape"
    default this.Name =
        "Shape"
```

Вызов метода родителя

```
type Shape() =
    abstract member Draw : unit -> unit
    abstract member Name : string
    default this.Draw () = printfn "Drawing shape"
    default this.Name = "Shape"

type Circle(r) =
    inherit Shape()
    member this.R = r
    override this.Draw () =
        base.Draw ()
        printfn "Drawing circle"
    override this.Name = "Circle"
```

Интерфейсы

```
type Shape =
    abstract member Draw : unit -> unit
    abstract member Name : string

type Circle(r) =
    member this.R = r
    interface Shape with
        member this.Draw () =
            printfn "Drawing circle"
        member this.Name = "Circle"
```

Явное приведение типов

```
let c = Circle 10
c.Draw ()  // Ошибка
(c :> Shape).Draw ()  // Ок

let draw (s : Shape) = s.Draw ()

draw c  // Ок
```

Наследование интерфейсов

```
type IEnumerable<'a> =
    abstract GetEnumerator : unit -> IEnumerator<'a>

type ICollection<'a> =
    inherit IEnumerable<'a>
    abstract Count : int
    abstract IsReadOnly : bool
    abstract Add : 'a -> unit
    abstract Clear : unit -> unit
    abstract Contains : 'a -> bool
    abstract CopyTo : 'a[] * int -> unit
    abstract Remove : 'a -> unit
```

Модули

```
type Vector =
    { x : float; y : float }

module VectorOps =
    let length v = sqrt(v.x * v.x + v.y * v.y)
    let scale k v = { x = k * v.x; y = k * v.y }
    let shiftX x v = { v with x = v.x + x }
    let shiftY y v = { v with y = v.y + y }
    let shiftXY (x, y) v = { x = v.x + x; y = v.y + y }
    let zero = { x = 0.0; y = 0.0 }
    let constX dx = { x = dx; y = 0.0 }
    let constY dy = { x = 0.0; y = dy }
```

Пространства имён

```
namespace Vectors

type Vector =
    { x : float; y : float }

module VectorOps =
    let length v = sqrt(v.x * v.x + v.y * v.y)
```