

# Лекция 2: Продолжение про F#

## 1. Юнит-тестирование в F#

Начнём с юнит-тестирования, штуки, хоть и не связанной с функциональным программированием напрямую, но имеющей в F# несколько приятных функциональных особенностей. Кстати, теперь юнит-тесты в домашке обязательны.

Поскольку F# компилируется в байт-коды .NET, то нет никаких причин не работать стандартным библиотекам модульного тестирования, которые работают и в C#, например, NUnit, Microsoft Testing Framework и т.д. Ими вполне можно пользоваться, но есть более красивые с функциональной точки зрения обёртки над ними, которые позволят писать тесты в более F#-овом стиле, прежде всего FsUnit, стандарт де-факто в модульном тестировании для F#. Есть и библиотеки, специфичные для F#: FsCheck и Unquote, например (на самом деле, не совсем для F#, FsCheck портирована с Haskell, но в C# точно нет ничего такого — хотя никто не мешает тесты для программ на C# писать на F#-е, естественно, пользуясь его тулами).

Вот небольшой пример теста с FsUnit-ом:

```
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
```

Тут показан, кстати, способ записи идентификаторов через двойные апострофы — вся строка внутри считается именем, даже если там пробелы, ключевые слова и прочая жесь. Сам тест помечен атрибутом [<Test>], и вообще в F# атрибуты пишутся в скобках вида [<>] (потому что просто квадратные скобки, как в C#, заняты под списки). *should equal* — это что-то вроде *Assert.Equals*, только круче.

Круче тем, что условие, которое надо проверить, на самом деле может быть довольно сложным, не просто равенство-истинность-не null, как принято в C#. Условие записывает-

ся в виде так называемых *матчеров*. Для тех, кто в курсе, такой же подход используется в библиотеке Hamcrest/НHamcrest. Вот примеры, из которых по идее всё должно стать понятно:

```
1 |> should equal 1
1 |> should not' (equal 2)
10.1 |> should (equalWithin 0.1) 10.11
"ships" |> should startWith "sh"
"ships" |> should not' (endsWith "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)
```

Есть некоторые заморочки с should throw и shouldFail, они хотят функцию без аргументов, возвращающую ничего, поэтому при вызове настоящий код обёрнут в лямбду.

Вот ещё примеры матчеров:

```
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 (lessThanOrEqual 10.1)

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

Choice<int, string>.Choice10f2(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)
```

Более интересная, хотя и реже используемая штука — это FsCheck. FsCheck берёт функцию, рефлексией понимает, что функция принимает, и генерирует случайные значения её параметрам, после чего сколько-то раз её вызывает. Функция должна возвращать булево значение. Если функция вернёт false, то FsCheck скажет, что тест провален, и ещё и покажет минимальный контрпример, на котором, собственно, получилось плохо. Например:

**open** FsCheck

```
let revRevIsOrig (xs:list<int>) = List.rev(List.rev xs) = xs
```

```
Check.Quick revRevIsOrig
// Ok, passed 100 tests.
```

```
let revIsOrig (xs:list<int>) = List.rev xs = xs
Check.Quick revIsOrig
// Falsifiable, after 2 tests (2 shrinks) (StdGen (338235241,296278002)):
// Original:
// [3; 0]
// Shrunk:
// [1; 0]
```

Замечу, что FsCheck прекрасно работает в паре с FsUnit (бросает исключение, которое FsUnit ловит и помечает тест как проваленный), так что писать часть тестов руками, а часть отдавать FsCheck не зазорно и повышает качество кода.

Ещё одна интересная библиотека — это Unquote. Вообще, она просто интерпретатор F#-а, использующий встроенный в язык механизм Quotation-ов, дающий возможность в программе получить дерево разбора куска кода на F#, потом обходить его как обычное дерево и делать с ним что угодно. Unquote хороша тем, что умеет не просто сказать, прошёл тест или нет, но и если не прошёл, показать последовательность преобразований, которая привела к ошибке. Вот пример:

```
[<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
```

Эти “<@” и “@>” и есть границы quotation-а, всё, что внутри, не вычисляется, а превращается в дерево разбора, которое отдаётся функции test из Unquote.

И конечно, библиотека для создания мок-объектов. В случае F#-а это Foq, тоже использующая quotation-ы, но на сей раз для генерации моков. Вот пример:

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

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

Этот мок имитирует мутабельный список, который на запрос Contains всегда возвращает true. Собственно, тоже хорошо работает с FsUnit и NUnit, можно использовать, чтобы красиво генерить тестовые заглушки и уменьшать зависимость теста от других кусков программы.

## 2. Каррирование

Для функционального программирования очень важно такое соображение, что функция от  $n$  аргументов может быть представлена как функция от одного аргумента, возвращающая функцию от  $n - 1$  аргумента — это называется **каррирование**. Каррирование поддержано и синтаксически (кстати, поэтому в списке аргументов функций в F# нет никаких скобок), и системой типов, что позволяет не передавать один из аргументов в функцию и получать функцию, которая хочет ещё аргументов. Зачем — частичное применение, например:

```
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)
```

Можно это повызывать из F# Interactive:

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

Практически полезно это не только тем, чтобы производить впечатление на коллег, но и для функций высших порядков. Например, как посчитать количество элементов сразу у пяти списков:

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

List.length тут функция, принимающая один аргумент, который ей передаёт List.map, когда идёт по списку. Посмотрим, что будет, если функция над списком принимает два аргумента, например, тот же map. Положим, мы хотим возвести элементы всех пяти списков в квадрат:

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

Тут уже каррирование работает во всей красе, внутренний `List.map` возвращает функцию от одного аргумента типа “список”, которую внешний `List.map` применяет к каждому элементу списка списков. Обратите внимание, что большинство функций из стандартной библиотеки принимают список последним аргументом, как раз для того, чтобы каррировать было удобнее.

## 2.1. Операторы конвейера и композиции

Каррирование работает ещё лучше, если использовать оператор конвейера, или `pipe forward`:

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

Он просто принимает аргумент и функцию, и вызывает функцию, передавая ей аргумент. Зачем — чтобы записывать вычисление в виде последовательно применяемых преобразований, вот так:

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

Согласитесь, это менее противостоестественно, чем как бы это выглядело на C# или Java:

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

Есть ещё оператор композиции, который почти как `pipe forward`, но позволяет аргумент функции в явном виде не писать:

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

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

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

Его можно понимать как математическую композицию — принимает две функции и возвращает функцию, которая сначала применяет первую функцию, затем вторую. А можно понимать как штуку, которая собирает конвейер, по которому потом пойдут данные. Сравните:

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

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

Есть ещё обратный конвейер и обратная композиция:

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

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

Зачем, ведь обратный конвейер вообще ничего не делает? На самом деле, чтобы воспользоваться приоритетом операций и не ставить скобки там, где иначе пришлось бы их ставить:

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

Сравните с

```
printfn "Result = %d" (factorial 5)
```

### 3. Использование библиотек .NET

Как уже не раз говорилось, классы из .NET работают в F# “даром”, кроме того, F# имеет ряд синтаксических особенностей, которые делают работу даже более приятной, чем ожидалось. Пример, небольшой текстовый редактор на Windows Forms, который грузит HTML-документ с удалённого адреса:

```
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
```

*open* — это что-то вроде *using*, импорт имён из пространства имён. *new Form* — обычный вызов обычного конструктора, только в его параметры передаются не только параметры конструктора, но и значения *public*-свойств, которые мы хотим установить изначально. “<-” означает оператор присваивания (.NET-классы же мутабельны, их свойства можно менять, несмотря на то, что ссылка на объект не мутабельна). Функция *ignore* определена так:

```
let ignore x = ()
```

Нужна она только для того, чтобы компилятор не выдавал предупреждение о неиспользованном значении, которое вернул метод *form.ShowDialog*.

### 4. Сопоставление шаблонов

Тут уже несколько раз встречался оператор *match*, настала пора рассказать про него подробнее. Это почти как *switch* в других языках, но немного круче, потому что позволяет

тут же декомпозировать сложную структуру данных, сопоставляя переменную с шаблоном. Опять-таки, в C# с версии 7.1 появилось что-то такое, но в силу ограниченности шаблонов в C#, не такое выразительное. В F# же можно писать, например, так:

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

Подчёркивание означает “что угодно”, и обратите внимание на два последних случая, в первом из них подчёркивание относится к первому элементу пары, во втором — ко всей паре сразу.

Или вот так (пример с одного из предыдущих слайдов):

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

Тут мы смотрим, если список пустой, то его длина 0, если нет, то в нём есть голова - cons - хвост, и его длина — это длина его хвоста плюс 1.

Есть ещё ключевое слово `when`, позволяющее указать в `match` произвольное булево условие:

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

Это довольно странно делать `match`-ем, но в каком-то смысле получилось красивее, чем последовательность `if`-ов.

Однако важно понимать, что F# не пытается в `match` считать переменные с одинаковыми именами одной переменной (как делает, например, Prolog), потому что это привело бы к резкому увеличению трудоёмкости `match`-а (до экспоненциальной, как в Prolog). Например, вот так работать не будет:

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

А вот так будет:

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

Какие шаблоны ещё можно использовать в `match` (и, кстати, в `let`):

Синтаксис	Описание	Пример
$(pat, \dots, pat)$	Кортеж	$(1, 2, ("3", x))$
$[pat; \dots; pat]$	Список	$[x; y; 3]$
$pat :: pat$	<code>cons</code>	$h :: t$
$pat \mid pat$	“Или”	$[x] \mid ["X"; x]$
$pat \ \& \ pat$	“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	$:? \ string$

## 5. Последовательности

Последовательности — один из самых частоиспользуемых ленивых типов данных, в стандартной библиотеке он называется `seq`, но на самом деле это не более чем синоним к `IEnumerable<T>`. То есть последовательность — это просто штука, по которой можно ходить итератором. Таким образом, на самом деле и строки, и списки, и массивы, и обычные дотнетовские списки, и практически всё остальное реализует `seq`. А это значит, что все операции `seq` применимы и к ним ко всем.

“Просто последовательность” в F# описывается так:

```
seq {0 .. 2}
seq {1I .. 10000000000000I}
```

`seq` — это просто класс, возвращающий эnumератор, поэтому она, собственно, и ленивая. Хранить миллиард чисел в памяти было бы очень грустно, поэтому их никто и не хранит, просто есть эnumератор, который по текущему числу может вернуть следующее и знает, когда остановиться. Поэтому последовательности могут быть хоть бесконечными. Но зато чтобы вернуться к предыдущему элементу, последовательность надо перевычислить с самого начала.

Вот небольшой пример того, как с последовательностями работать:

```
open System.IO
let rec allFiles dir =
    Seq.append
        (dir |> Directory.GetFiles)
        (dir |> Directory.GetDirectories
            |> Seq.map allFiles
            |> Seq.concat)
```

Тут мы обходим файловую систему, собирая список файлов в текущей папке и всех её подпапках. Обратите внимание на использование операций модуля `Seq`, тут используется тот же паттерн, что и в списках — структура данных, которая ничего не умеет, и статический класс, который умеет всё, но ничего не хранит. Вот какие ещё методы бывают:



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	$( int \rightarrow 'a ) \rightarrow seq <'a>$

Как обычно, тут приводится только небольшая часть методов, наиболее интересных по тем или иным причинам. Традиционные map-filter-fold, два сорта операций конкатенации (append принимает два аргумента, concat — последовательность из последовательностей, которые надо склеить), choose — это map + filter. empty — это просто пустая последовательность, initInfinite принимает функцию-генератор, которая по данному индексу должна вернуть элемент, и возвращает бесконечную последовательность. Есть ещё Seq.unfold, которая тоже так умеет, но несколько более умно (она как fold, только наоборот, что, впрочем, понятно по её названию).

Кстати, решётки перед типами параметров означают “тип и все, кто от него наследуется”. Например, append может принять список и массив, но возвращает он именно последовательность.

Для последовательностей работает тот же синтаксис генерации, что и для списков, так что можно писать вот так:

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

Стрелочку надо читать как yield и обозначает она то же, что yield return в C# — когда дошли до этого места, вернуть значение, а потом, когда нас попросят следующее, продолжить с этого же места и считать, будто мы не прерывались. Вот ещё пример, генерим координаты белых (или чёрных) клеток шахматной доски:

```
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) }
```

Можно переписать через yield предыдущий пример с обходом файловой системы:

```
let rec allFiles dir =
    seq { for file in Directory.GetFiles(dir) -> file
        for subdir in Directory.GetDirectories dir ->>
            (allFiles subdir) }
```

Здесь используется новая конструкция — “-»” или yield! (читается как yield-bang) — если yield добавляет к результату один элемент, то yield! добавляет все элементы последовательности, один за одним (то есть это равносильно циклу for с yield-ом).

А вот как можно получить последовательность всех строчек в файле:

```
let reader =
    seq {
        use reader = new StreamReader(
            File.OpenRead("test.txt")
        )
        while not reader.EndOfStream do
            yield reader.ReadLine() }
```

Обратите внимание на `use`, это как `using` в C# — объявить переменную типа, реализующего интерфейс `IDisposable`, и вызвать его метод `Dispose` как только переменная выйдет из области видимости. Это очень похоже на идиому RAII из C++ и служит тем же целям — корректно завершить работу с ресурсом даже с учётом бросания исключений.

На самом деле, так читать из файла — плохая идея, потому что последовательности ленивые, следовательно, следующая строка будет считана только когда её попросят. При этом файл будет открыт, пока программа не дойдёт до самого конца файла. А вот если в конце сделать `Seq.toList`, это заставит файл считаться целиком, списки не ленивы и держат всё своё содержимое в памяти.

## 6. Записи

В F# есть аналог структур из C#, только тут они называются “записи” и немутабельны по умолчанию. И в отличие от C#, они по умолчанию ссылочные типы. Описание типа-записи выглядит вот так:

```
type Person =
    { Name: string;
      DateOfBirth: System.DateTime; }
```

Объект этого типа описывается, например, так:

```
{ Name = "Bill";
  DateOfBirth = new System.DateTime(1962, 09, 02) }
```

Или так:

```
{ new Person
  with Name = "Anna"
  and DateOfBirth = new System.DateTime(1968, 07, 23) }
```

Обратите внимание, что в первом случае тип записи явно не указывается, выводилка типов находит тип с подходящими полями и считает типом записи его.

Поскольку записи немутабельны, традиционный принцип работы “скопировать и поменять” без специальной поддержки со стороны языка был бы очень неудобен, пришлось бы вручную копировать кучу полей. Но синтаксис копирования с изменением есть, выглядит так:

```

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 }

```

Кстати, обратите внимание, что апостроф (одиночная кавычка) — вполне законная часть имени, может стоять на любой позиции, кроме первой.

## 7. Размеченные объединения

Размеченные объединения — это типичный для функциональных языков тип, который почему-то редко встречается в объектно-ориентированных языках. Ближе всего размеченное объединение к конструкции `union` в C++ или к вариантным записям Паскаля. Это тип, значение которого может быть либо чем-то, либо чем-то ещё (одного из нескольких “под-типов”). Например:

```

type Route = int
type Make = string
type Model = string

type Transport =
| Car of Make * Model
| Bicycle
| Bus of Route

let bus = Bus(420)

```

Переменная `bus` имеет тип `Transport`, при этом является альтернативой `Bus` и хранит в себе данные типа `Route` (который на самом деле синоним `int`). А можно было бы написать

```
let car = Car("SomeCar", "Luxury Sedan")
```

Тогда `car` тоже была бы типа `Transport`, но тем не менее, хранила бы в себе другие данные и помнила, что она машина. Кстати, эти `Car`, `Bicycle` и `Bus` часто называют дискриминаторами.

Некоторые типы данных, которые мы видели — это на самом деле размеченные объединения. Например, `option` — это либо `None`, либо `Some` что-то:

```

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

```

Или даже список — это либо пустой список, либо cons, содержащий в себе элемент и хвост списка. Да, размеченные объединения могут быть рекурсивны:

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

Вот так выглядит создание объектов 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
```

Видно, что дискриминатор, используемый для создания объекта размеченного объединения, ведёт себя как функция.

Размеченные объединения можно разбирать с помощью шаблонов, удобнее всего это делать в match-е, например, так:

```
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
```

Поэтому, кстати, размеченные объединения хороши для представления неоднородных иерархических данных, например, деревьев разбора. Поэтому F# и OCaml так любят компиляторщики. Вот пример, дерево разбора логического выражения и код, который его вычисляет:

```
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)))

```

Кстати, обратите внимание, что кортеж в записи типа обозначается “\*”, а размеченное объединение — “|”. Это неспроста, кортеж и запись (которая по сути тоже кортеж, но с именованными полями) — это декартово произведение множества значений входящих в них типов, а размеченное объединение — это просто объединение множеств значений (например, размеченное объединение `int` и `bool` может принимать в качестве значений целые числа и `true` с `false`), отсюда использование чего-то, похожего на оператор “или” в записи. Кортеж — это произведение типов, а размеченное объединение — сумма типов.

Впрочем, кортежи хорошо работают вместе с размеченными объединениями, возможны взаимно ссылающиеся друг на друга типы, например, вот такое описание графа:

```

type node =
    { Name : string;
      Links : link list }
and link =
    | Dangling
    | Link of node

```

## 8. Паттерны функционального программирования

Далее речь пойдёт о некоторых базовых приёмах, типичных для функциональных программ, которые чем-то похожи на низкоуровневые паттерны ООП. Вообще, в функциональном программировании паттерны не так распространены, как в объектно-ориентированном, возможно потому, что в функциональных программах такие конструкции выражаются гораздо естественнее. Тем не менее, есть некоторые общеизвестные приёмы, которые мы тут обсудим.

Во-первых, как сделать цикл, если использовать циклы нельзя — заменить на рекурсию, передавая счётчик цикла и текущее состояние вычисления как параметр в рекурсивный вызов. Рассмотрим нерекурсивную программу, например, разложение на множители:

```

let factorizeImperative n =
    let mutable primefactor1 = 1
    let mutable primefactor2 = n
    let mutable i = 2
    let mutable fin = false
    while (i < n && not fin) do
        if (n % i = 0) then
            primefactor1 <- i

```

```

    primefactor2 <- n / i
    fin <- true
    i <- i + 1
    if (primefactor1 = 1) then None
    else Some (primefactor1, primefactor2)

```

Кстати, это вполне валидная программа на F#, но выглядит она так, будто написана на C#-е. mutable — это прямая противоположность const или readonly в C#, разрешает переменной менять значение, “<-” — оператор присваивания.

Как бы это можно было переписать рекурсивно:

```

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

```

Здесь как раз и используется типичный для ФП приём — завести вложенную функцию, которая бы принимала ещё один параметр (n попадает ей в замыкание, так что тоже доступен), и в этот параметр и передавать счётчик цикла. А дальше всё как обычно. В более сложных случаях состояние цикла может быть не только счётчиком, но и чем угодно, хоть здоровенной структурой.

Однако не всё так просто. Рассмотрим задачу создания списка чисел от 0 до 100000, и её очевидное императивное решение:

```

open System.Collections.Generic

```

```

let createMutableList () =
    let l = new List<int>()
    for i = 0 to 100000 do
        l.Add(i)
    l

```

Воспользуемся приобретёнными знаниями и перепишем её рекурсивно:

```

let createImmutableList () =
    let rec createList i max =
        if i = max then
            []
        else
            i :: createList (i + 1) max
    createList 0 100000

```

Запустим и с удивлением обнаружим, что она упала с переполнением стека. В общем-то, ничего удивительного, у нас 100000 рекурсивных вызовов тут, но если у нас в языке есть только рекурсия, то, кажется, у нас проблемы — списки больше примерно 5000 элементов оказываются для нас вообще недоступны.

Разумеется, это не так. Посмотрим на проблему более внимательно, на примере типичной реализации факториала:

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

Это на самом деле то же самое, что и

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

— после рекурсивного вызова его результат ещё и умножается на значение  $x$ , которое, как параметр, лежит на стеке вызовов.

А если переписать факториал вот так:

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

Теперь умножение выполняется ДО рекурсивного вызова, при вычислении аргумента, и рекурсивный вызов — это последний оператор функции. Тогда на самом деле кадр стека не нужен, потому что после рекурсивного вызова ничего уже не происходит, поэтому хранить параметры и локальные переменные не надо, да и адрес возврата нам неинтересен. Достаточно просто передать управление на начало функции, поменяв её параметры. Собственно, компилятор F# (как и любого другого нормального функционального языка) так и делает, вот что получится при декомпиляции написанного выше кода в C#:

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

```

    acc *= x;
    x--;
}
}

```

Видим, что рекурсия заменилась просто циклом. Это возможно только тогда, когда рекурсивный вызов строго последнее действие, совершаемое функцией. Проблема F# в том, что компилятор не подсказывает, что получилась или не получилась хвостовая рекурсия, поэтому надо самим аккуратно следить. Собственно, вот создавалка списка из предыдущего примера, переписанная с помощью хвостовой рекурсии, она уже реально работает:

```

let createImmutableList () =
    let rec createList i max result =
        if i = max then
            List.rev result
        else
            createList (i + 1) max (i :: result)
    createList 0 100000 []

```

Чтобы аккуратно следить было не слишком больно, придумали паттерн “аккумулятор”, который как раз и заключается в том, чтобы накапливать результат в одном из параметров, а потом вернуть результат целиком. Примеры выше были на самом деле примерами применения этого паттерна, вот ещё один пример, функция map:

```

let rec map f list =
    match list with
    | [] -> []
    | hd :: tl -> (f hd) :: (map f tl)

```

— так плохо,

```

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) []

```

— так хорошо. Идея, как видим, тут точно такая же, как и в предыдущих примерах. Код получается страшнее, требуется вложенная функция, чтобы не показывать параметр-аккумулятор пользователю, зато рекурсивная программа работает так же, как обычная императивная программа с циклами.

Можно пойти дальше и вспомнить, что у нас язык-то функциональный, поэтому в качестве аккумулятора может выступать функция, которая постепенно строится, чтобы в конце рекурсии посчитаться и вернуть значение (или сделать то, что нужно). Например, распечатать список в обратном порядке:



```

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!")

```

Здесь при первом вызове printListRevTR передаётся весь список и функция, печатающая “Done!”, затем происходит рекурсивный вызов, где в printListRevTR передаётся хвост списка и функция, печатающая голову, затем вызывающая функцию, печатающую “Done!”, и т.д. В самом конце это всё вызывается, печатается последний элемент, предпоследний, ..., первый и “Done!”.

Обратите внимание, что вызов функции-продолжения стоит последним, поэтому это тоже хвостовая рекурсия, кадра стека для каждого вызова не создаётся (хоть это и, вообще говоря, разные функции). Компилятор раскрывает их в последовательное выполнение.

Собственно, стиль, при котором в функцию передаётся функция, которая должна быть выполнена после того, как первая функция закончит работу, называется Continuation Passing Style, а сама передаваемая функция — Continuation, продолжение. Стиль прижился в асинхронном программировании, где встречается даже в программах на C# или Java — например, в функцию, выполняющую запрос к серверу, передаются функции, которые вызываются, если от сервера пришёл ответ или произошла сетевая ошибка.

Несколько хуже обстоят дела с обходом ветвящихся структур данных, например, двоичного дерева. При использовании обычного паттерна “аккумулятор” нам бы пришлось в конце делать два рекурсивных вызова, для обхода левого и правого поддерева. Тогда они не могли бы быть хвостовой рекурсией, хотя бы один честно нуждался бы в кадре стека для хранения адреса возврата. Поэтому в таких случаях используют Continuation Passing Style, формируя функцию, которая последовательно вызовет себя для всех узлов дерева и при этом будет хвосторекурсивной. Вот пример:

```

type Tree<'a> =
  | Node of 'a * Tree<'a> * Tree<'a>
  | Empty

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)))

```

```

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

```

Размеченное объединение `ContinuationStep` служит тут для хранения состояния вычисления — оно может быть либо закончено, либо посещением узла с необходимостью вызвать функцию-продолжение (функцию типа `unit -> ContinuationStep<'a>`). Обход дерева выполняет функция `linearize`, которое, если дошла до листа, вызывает продолжение, а если посещает узел, возвращает `Step` со значением в этом узле и продолжением, которое вызовет `linearize` для левого поддерева, сказав ему (вот где хитрость), что потом в качестве продолжения надо вызвать `linearize` для правого поддерева. ну а потом функция `iter` получает последовательность `step`-ов и для каждого `step`-а вызывает функцию `f` и продолжение, чтобы получить следующий `step` и обработать его. То есть обход дерева не только хвосторекурсивный, но ещё и ленивый, `linearize` вызывается только когда надо получить следующий `step`.