

Лекция 3: Продолжение про F#

Комментарии по домашке

Начнём с небольшого фидбэка по домашним заданиям, вот наиболее частые проблемы в первой и второй домашке:

- Пользователям свойственно ошибаться, поэтому не следует ожидать, например, что факториал вызовут обязательно с положительным параметром. Программа, по крайней мере, не должна падать с каким-нибудь системным исключением. Есть функция `failwith`, которая бросает исключение с сообщением об ошибке, или `raise`, которая бросает объект-исключение произвольного класса. А типы-исключения описываются вот так: `exception MyException of string`. В идеале программа не должна вообще падать, а должна дать исправить ошибку, но это уж как пойдёт, всё не предугадаешь.
- Ещё бывает так, что входные данные корректны, но программа их всё равно не ожидает, потому что её автор делает какие-то неявные предположения на ввод. Надо уметь анализировать задачу на предмет внезапных входных данных, и система типов функциональных языков может в этом помочь — просто смотрите на тип того, что получилось, и думайте, можете ли вы обрабатывать все значения этого типа.
- С чем ещё может помочь система типов — заставить вызывающего ваш код (например, вас самих) подумать, все ли случаи вы учли. Например, в задаче про поиск числа в списке все по привычке возвращали `-1`, что могло привести к тому, что вызывающий не подумал бы о том, что значения в списке может не быть, и обратился бы к списку по индексу `-1`. Тип `option` специально придуман, чтобы так не могло получиться, и тут, если мы возвращаем `option`, надо было бы явно разобрать два случая — когда значение в списке есть и когда его нет. И если этого не сделать, компилятор скажет, что вы неправы. Вообще, общее правило состоит в том, что некорректное состояние системы должно быть невыразимо в системе типов, то есть если индекс — то его либо нет, либо он неотрицательный.
- Ещё многие несколько раз подряд делали одно и то же вычисление, что плохо, потому что F# не Haskell и ленивые вычисления прямо так в нём не работают.
- Красивый способ порезать список на два для `mergesort-a`:

```
let rec split ls left right =  
    match ls with  
    | [] -> (left, right)
```

```
| [a] -> (a::left, right)
| a::b::tail -> split tail (a::left) (b::right)
```

1. 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 перевычислять 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.

2. 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.

3. 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
```

4. 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 1000000 []

```

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 посчитать 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 поддеревя. 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`.