

Продолжение про F#

Юрий Литвинов

27.02.2018г

Комментарии по домашке

- ▶ Обработка ошибочного пользовательского ввода
- ▶ Обработка всех возможных входных данных
- ▶ Если элемента в списке не нашлось, надо возвращать не -1, а None
 - ▶ Некорректное состояние должно быть невыразимо
- ▶ Стараться не делать лишних вычислений, язык не ленивый
- ▶ Как порезать список на два:

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

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) }
```

Обход папок через yield

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

Ленивое чтение из файла

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

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
```

```
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
```

Замена цикла рекурсией

Императивное 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)
```


Замена цикла рекурсией

Рекурсивное разложение на множители

```
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, проблема

Императивный вариант

```
open System.Collections.Generic
```

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

Хвостовая рекурсия, проблема

Рекурсивный вариант, казалось бы

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

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)  
  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)))
```

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
```