

# Вычислительные выражения в F#

## Computation Expressions, Workflows

Юрий Литвинов  
y.litvinov@spbu.ru

12.03.2026

# Что это и зачем нужно

- ▶ Механизм управления процессом вычислений
  - ▶ Обобщённые функции
- ▶ В функциональных языках — единственный способ определить порядок вычислений
- ▶ Зачастую — нетривиальным образом (Async)
- ▶ Способ не писать кучу вспомогательного кода (сродни аспектно-ориентированному программированию)
- ▶ В теории ФП они называются монадами
- ▶ На самом деле, синтаксический сахар

# Пример

Классический пример с делением на 0

Сопротивление сети из параллельных резисторов:

$$1/R = 1/R_1 + 1/R_2 + 1/R_3$$

$R_1$ ,  $R_2$  и  $R_3$  могут быть 0. Что делать?

- ▶ Бросать исключение — плохо
- ▶ Использовать option — много работы, но попробуем

# Реализация вручную

divide

```
let divide x y =  
  match y with  
  | 0.0 -> None  
  | _ -> Some (x / y)
```

# Реализация вручную

## Само вычисление

```
let resistance r1 r2 r3 =  
  let r1' = divide 1.0 r1  
  match r1' with  
  | None -> None  
  | Some x -> let r2' = divide 1.0 r2  
              match r2' with  
              | None -> None  
              | Some y -> let r3' = divide 1.0 r3  
                          match r3' with  
                          | None -> None  
                          | Some z -> let r = divide 1.0 (x + y + z)  
                                      r
```

## То же самое, через Workflow Builder

```
let resistance r1 r2 r3 =  
  maybe {  
    let! r1' = divide 1.0 r1  
    let! r2' = divide 1.0 r2  
    let! r3' = divide 1.0 r3  
    let! r = divide 1.0 (r1' + r2' + r3')  
    return r  
  }
```

## seq — это тоже Computation Expression

```
let daysOfTheYear =  
  seq {  
    let months =  
      ["Jan"; "Feb"; "Mar"; "Apr"; "May"; "Jun";  
       "Jul"; "Aug"; "Sep"; "Oct"; "Nov"; "Dec"]  
    let daysInMonth month =  
      match month with  
      | "Feb" -> 28  
      | "Apr" | "Jun" | "Sep" | "Nov" -> 30  
      | _ -> 31  
    for month in months do  
      for day = 1 to daysInMonth month do  
        yield (month, day)  
      }  
  }
```

## Ещё один пример

```
let debug x = printfn "value is %A" x
```

```
let withDebug =
```

```
    let a = 1
```

```
    debug a
```

```
    let b = 2
```

```
    debug b
```

```
    let c = a + b
```

```
    debug c
```

```
    c
```



## То же самое с Workflow

```
let withDebug = debugFlow {  
    let! a = 1  
    let! b = 2  
    let! c = a + b  
    return c  
}
```

# Как это сделать

let, «многословный» синтаксис

**let**  $x = \text{something}$

равносильно

**let**  $x = \text{something}$  **in** [ выражение с  $x$  ]

например,

**let**  $x = 1$  **in**

**let**  $y = 2$  **in**

**let**  $z = x + y$  **in**

$z$

## let и лямбды

**fun** x -> [ выражение с x ]

или

something |> (**fun** x -> [ выражение с x ])

и обращаем внимание, что:

**let** x = someExpression **in** [ выражение с x ]

someExpression |> (**fun** x -> [ выражение с x ])

# let и CPS

```
let x = 1 in  
  let y = 2 in  
    let z = x + y in  
      z
```

```
1 |> (fun x ->  
  2 |> (fun y ->  
    x + y |> (fun z ->  
      z))))
```

## Можно обобщить до выполнения произвольного действия при вызове

```
let pipeInto expr f =  
  expr |> f  
pipeInto (1, fun x ->  
  pipeInto (2, fun y ->  
    pipeInto (x + y, fun z ->  
      z)))
```

# Зачем

```
let pipeInto (expr, f) =  
  printfn "expression is %A" expr  
  expr |> f  
  
pipeInto (1, fun x ->  
  pipeInto (2, fun y ->  
    pipeInto (x + y, fun z ->  
      z)))
```

## То же самое с Workflow

```
type DebugBuilder() =  
  member this.Bind(x, f) =  
    debug x  
    f x  
  member this.Return(x) = x  
  
let debugFlow = DebugBuilder ()  
  
let withDebug = debugFlow {  
  let! a = 1  
  let! b = 2  
  let! c = a + b  
  return c  
}
```

## Более сложный пример, с делением

pipeInto, которая потом будет Bind

```
let pipeInto (expr, f) =  
  match expr with  
  | None ->  
    None  
  | Some x ->  
    x |> f
```



## Более сложный пример, с делением

Сам процесс

```
let resistance r1 r2 r3 =  
  let a = divide 1.0 r1  
  pipeInto (a, fun a' ->  
    let b = divide 1.0 r2  
    pipeInto (b, fun b' ->  
      let c = divide 1.0 r3  
      pipeInto (c, fun c' ->  
        let r = divide 1.0 (a + b + c)  
        pipeInto (r, fun r' ->  
          Some r  
        ))))
```

## Уберём временные let-ы

```
let resistance r1 r2 r3 =  
  pipeInto (divide 1.0 r1, fun a ->  
    pipeInto (divide 1.0 r2, fun b ->  
      pipeInto (divide 1.0 r3, fun c ->  
        pipeInto (divide 1.0 (a + b + c), fun r ->  
          Some r  
        )))
```

## И отформатируем

```
let resistance r1 r2 r3 =  
  pipeInto (divide 1.0 r1, fun a ->  
    pipeInto (divide 1.0 r2, fun b ->  
      pipeInto (divide 1.0 r3, fun c ->  
        pipeInto (divide 1.0 (a + b + c), fun r ->  
          Some r  
        ))))
```

## Сравним с оригиналом

```
let resistance r1 r2 r3 =  
  maybe {  
    let! r1' = divide 1.0 r1  
    let! r2' = divide 1.0 r2  
    let! r3' = divide 1.0 r3  
    let! r = divide 1.0 (r1' + r2' + r3')  
    return r  
  }
```

# WorkflowBuilder

- ▶ Bind создаёт цепочку continuation passing style-функций, возможно, с побочными эффектами
- ▶ Есть тип-обёртка (или монадический тип), в котором хранится состояние вычисления
  - ▶ Или, более функционально, которое представляет действие при вычислении
- ▶ let! вызывает Bind, return — Return, Bind принимает обёрнутое значение и функцию-continuation, return по необёрнутому значению делает обёрнутое
  - ▶ На самом деле, это просто композиция функций, но хитрая, потому что монадический тип
- ▶ WorkflowBuilder — это просто класс, в котором должны лежать методы с нужными сигнатурами, сам workflow — объект этого класса
  - ▶ Обычно он один, но в теории ничто не мешает хранить в нём побочные эффекты
  - ▶ Не путайте WorkflowBuilder с монадическим типом, это другое

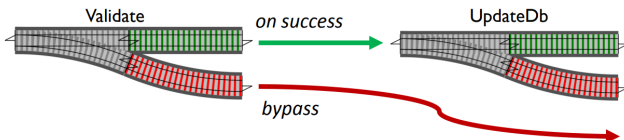
# Railway-oriented programming

Монадический тип может управлять вычислением

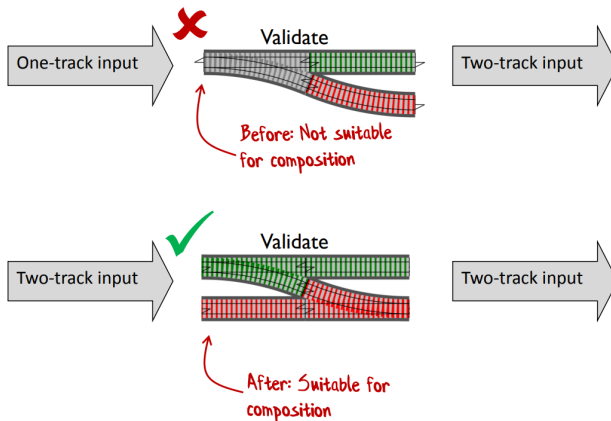


```

let validateInput input =
  if input.name = "" then
    Failure "Name must not be blank"
  else if input.email = "" then
    Failure "Email must not be blank"
  else
    Success input // happy path
  
```



# Что на самом деле делает Bind



© [https://github.com/swlaschin/RailwayOrientedProgramming/blob/master/Railway\\_Oriented\\_Programming\\_Slideshare.pdf](https://github.com/swlaschin/RailwayOrientedProgramming/blob/master/Railway_Oriented_Programming_Slideshare.pdf)

## Подробнее про Bind

- ▶  $\text{Bind} : M<'T> * ('T \rightarrow M<'U>) \rightarrow M<'U>$
- ▶  $\text{Return} : 'T \rightarrow M<'T>$

**let!**  $x = 1$  **in**  $x * 2$

`builder.Bind(1, (fun x -> x * 2))`



# Как в Haskell

```
let (>>=) m f = pipeInto(m, f)
```

```
let workflow =  
  1 >>= (+) 2 >>= (*) 42 >>= id
```

## Option.bind и maybe

```
module Option =  
  let bind f m =  
    match m with  
    | None ->  
      None  
    | Some x ->  
      x |> f  
  
type MaybeBuilder() =  
  member this.Bind(m, f) = Option.bind f m  
  member this.Return(x) = Some x
```

# Содержимое типа-обёртки может иметь разный тип

Пример, серия запросов к БД

```
type DbResult<'a> =  
  | Success of 'a  
  | Error of string
```

```
type CustomerId = CustomerId of string
```

```
type OrderId = OrderId of int
```

```
type ProductId = ProductId of string
```

## Пример, запросы

```
let getCustomerId name =
```

```
  if (name = "")
```

```
  then Error "getCustomerId failed"
```

```
  else Success (CustomerId "Cust42")
```

```
let getLastOrderForCustomer (CustomerId custId) =
```

```
  if (custId = "")
```

```
  then Error "getLastOrderForCustomer failed"
```

```
  else Success (OrderId 123)
```

```
let getLastProductForOrder (OrderId orderId) =
```

```
  if (orderId = 0)
```

```
  then Error "getLastProductForOrder failed"
```

```
  else Success (ProductId "Product456")
```

## Общение с БД вручную

```
let product =  
  let r1 = getCustomerId "Alice"  
  match r1 with  
  | Error e -> Error e  
  | Success custId ->  
    let r2 = getLastOrderForCustomer custId  
    match r2 with  
    | Error e -> Error e  
    | Success orderId ->  
      let r3 = getLastProductForOrder orderId  
      match r3 with  
      | Error e -> Error e  
      | Success productId ->  
        printfn "Product is %A" productId  
        r3
```

# Builder

```
type DbResultBuilder() =
```

```
member this.Bind(m, f) =  
    match m with  
    | Error e -> Error e  
    | Success a ->  
        printfn "Successful: %A" a  
        f a
```

```
member this.Return(x) =  
    Success x
```

```
let dbresult = new DbResultBuilder()
```

# Workflow

```
let product =  
  dbresult {  
    let! custId = getCustomerId "Alice"  
    let! orderId = getLastOrderForCustomer custId  
    let! productId = getLastProductForOrder orderId  
    printfn "Product is %A" productId  
    return productId  
  }  
printfn "%A" product
```

# Композиция Workflow-ов

```
let subworkflow1 = myworkflow { return 42 }
```

```
let subworkflow2 = myworkflow { return 43 }
```

```
let aWrappedValue =
```

```
  myworkflow {
```

```
    let! unwrappedValue1 = subworkflow1
```

```
    let! unwrappedValue2 = subworkflow2
```

```
    return unwrappedValue1 + unwrappedValue2
```

```
  }
```



## Вложенные Workflow-ы

```
let aWrappedValue =  
  myworkflow {  
    let! unwrappedValue1 = myworkflow {  
      let! x = myworkflow { return 1 }  
      return x  
    }  
    let! unwrappedValue2 = myworkflow {  
      let! y = myworkflow { return 2 }  
      return y  
    }  
    return unwrappedValue1 + unwrappedValue2  
  }
```

# ReturnFrom

```
type MaybeBuilder() =  
  member this.Bind(m, f) = Option.bind f m  
  member this.Return(x) =  
    printfn "Wrapping a raw value into an option"  
    Some x  
  member this.ReturnFrom(m) =  
    printfn "Returning an option directly"  
    m  
  
let maybe = new MaybeBuilder()
```

# Пример

```
maybe { return 1 }
```

```
maybe { return! (Some 2) }
```

# Зачем это

```
maybe {  
  let! x = divide 24 3  
  let! y = divide x 2  
  return y  
}
```

```
maybe {  
  let! x = divide 24 3  
  return! divide x 2  
}
```

# Первый и второй законы монад

- Bind и Return должны быть взаимно обратны

```
myworkflow {  
  let originalUnwrapped = something  
  let wrapped = myworkflow { return originalUnwrapped }  
  let! newUnwrapped = wrapped  
  assertEquals newUnwrapped originalUnwrapped  
}  
  
myworkflow {  
  let originalWrapped = something  
  let newWrapped = myworkflow {  
    let! unwrapped = originalWrapped  
    return unwrapped  
  }  
  assertEquals newWrapped originalWrapped  
}
```

## Или то же самое на Haskell

```
return x >>= f == f x  
mv >>= return == mv
```

Или через монадную композицию ( $f \gg= g = \lambda x \rightarrow (f\ x \gg= g)$ ):

```
return >=> f == f  
f >=> return == f
```

В Haskell монады синтаксически приятнее

## Третий закон монад

► Ассоциативность композиции

```
let result1 = myworkflow {  
  let! x = originalWrapped  
  let! y = f x  
  return! g y  
}  
  
let result2 = myworkflow {  
  let! y = myworkflow {  
    let! x = originalWrapped  
    return! f x  
  }  
  return! g y  
}  
  
assertEqual result1 result2
```

## Или на Haskell

$$(mv \gg= f) \gg= g == mv \gg= (\backslash x \rightarrow (f\ x \gg= g))$$

Или через композицию:

$$(f \gg= g) \gg= h == f \gg= (g \gg= h)$$

Три закона монад обеспечивают адекватность их композиции.



## Какие ещё методы есть у WorkflowBuilder

Имя	Тип	Описание
Delay	$(\text{unit} \rightarrow M<'T>) \rightarrow M<'T>$	Превращает в функцию
Run	$M<'T> \rightarrow M<'T>$	Исполняет вычисление
Combine	$M<'T> * M<'T> \rightarrow M<'T>$	Последовательное исполнение
For	$\text{seq}<'T> * ('T \rightarrow M<'U>) \rightarrow M<'U>$	Цикл for
TryWith	$M<'T> * (\text{exn} \rightarrow M<'T>) \rightarrow M<'T>$	Блок try with
TryFinally	$M<'T> * (\text{unit} \rightarrow \text{unit}) \rightarrow M<'T>$	Блок finally
Using	$'T * ('T \rightarrow M<'U>) \rightarrow M<'U>$ when 'U :> IDisposable	use
While	$(\text{unit} \rightarrow \text{bool}) * M<'T> \rightarrow M<'T>$	Цикл while
Yield	$'T \rightarrow M<'T>$	yield или ->
YieldFrom	$M<'T> \rightarrow M<'T>$	yield! или ->>
Zero	$\text{unit} \rightarrow M<'T>$	Обёрнутое ()

## Подробнее про Run и Delay

Результат вычисления выражения:

```
builder.Run(builder.Delay(fun () -> {{ cexpr }}}))
```

# Пример, DSL для создания презентаций

*// Доменная модель*

```
type Slide = { Header: string }
```

```
type Deck = { Title: string; Slides: Slide list }
```

*// WorkflowBuilder*

```
type SlideBuilder() =  
  member inline _.Yield(()) = ()
```

*// Можно определять свои операции!*

*// ... но нужен yield*

```
[<CustomOperation("header")>]
```

```
member inline _.Header((), header: string) : Slide =  
  { Header = header }
```

```
let slide = SlideBuilder()
```

# Как использовать

```
slide {  
  header "Hello world!"  
}
```

# Сделаем генерацию Deck

```
[<RequireQualifiedAccess>]
```

```
type DeckProperty =  
  | Title of string
```

```
type DeckBuilder() =  
  member inline _.Yield(()) = ()  
  member inline _.Run(DeckProperty.Title title) = { Title = title; Slides =
```

```
[<CustomOperation("title")>]
```

```
member inline _.Title((), title: string) = DeckProperty.Title title
```

```
let deck = DeckBuilder()
```

## Сделаем генерацию слайдов (пока одного)

```
[<RequireQualifiedAccess>]
```

```
type DeckProperty = Title of string | Slide of Slide
```

```
type DeckBuilder() =
```

```
  member inline _.Yield(()) = ()
```

```
  member inline _.Yield(slide: Slide) = DeckProperty.Slide slide
```

```
  member inline _.Run(prop) =
```

```
    match prop with
```

```
    | DeckProperty.Title title -> { Title = title; Slides = [] }
```

```
    | DeckProperty.Slide slide -> { Title = ""; Slides = [ slide ] }
```

```
[<CustomOperation("title")>]
```

```
member inline _.Title((), title: string) = DeckProperty.Title title
```

# Как использовать

```
deck {  
  yield slide {  
    header "Hello world!"  
  }  
}
```

# Сделаем цепочку слайдов

```
type DeckBuilder() =
    (* ... *)
```

```
member inline _.Delay(f: unit -> DeckProperty list) = f()
```

```
member inline _.Delay(f: unit -> DeckProperty) = [f ()]
```

```
member inline _.Combine(newProp: DeckProperty,
    previousProps: DeckProperty list) =
    newProp :: previousProps
```

```
member inline x.Run(props: DeckProperty list) =
    props
    |> List.fold
        (fun deck prop ->
            match prop with
            | DeckProperty.Title title -> { deck with Title = title }
            | DeckProperty.Slide slide -> { deck with Slides = slide :: deck.Slides })
        { Title = ""; Slides = [] }
```

```
member inline x.Run(prop: DeckProperty) = x.Run([prop])
```



## Теперь можно так

```
deck {  
  title "Testing Deck with title"  
  
  slide {  
    header "This works"  
  }  
  
  slide {  
    header "...and also this!"  
  }  
  
  slide {  
    header "Much wow!"  
  }  
}
```

Но надо ещё For

# Итого

```
[<RequireQualifiedAccess>]
```

```
type DeckProperty =
```

```
  | Title of string
```

```
  | Slide of Slide
```

```
type DeckBuilder() =
```

```
  member inline _.Yield(()) = ()
```

```
  member inline _.Yield(slide: Slide) = DeckProperty.Slide slide
```

```
  member inline _.Delay(f: unit -> DeckProperty list) = f ()
```

```
  member inline _.Delay(f: unit -> DeckProperty) = [ f () ]
```

```
  member inline _.Combine(newProp: DeckProperty, previousProps: DeckProperty list) = newProp :: previousProps
```

```
  member inline x.For(prop: DeckProperty, f: unit -> DeckProperty list) = x.Combine(prop, f ())
```

```
  member inline x.For(prop: DeckProperty, f: unit -> DeckProperty) = [prop; f()]
```

```
  member inline x.Run(props: DeckProperty list) =
```

```
    props
```

```
    |> List.fold
```

```
      (fun deck prop ->
```

```
        match prop with
```

```
        | DeckProperty.Title title -> { deck with Title = title }
```

```
        | DeckProperty.Slide slide -> { deck with Slides = deck.Slides @ [ slide ] }
```

```
      { Title = "", Slides = [] }
```

```
  member inline x.Run(prop: DeckProperty) = x.Run([ prop ])
```

```
[<CustomOperation("title")>]
```

```
  member inline _.Title((), title: string) = DeckProperty.Title title
```

# Моноиды

## Немного алгебры

Множество с бинарной операцией

- ▶ Замкнутость относительно операции
- ▶ Ассоциативность
- ▶ Наличие нейтрального элемента

Например,  $[a] @ [b] = [a; b]$

## Пример

```
type OrderLine = {Quantity : int; Total : float}
```

```
let orderLines = [  
  {Quantity = 2; Total = 19.98};  
  {Quantity = 1; Total = 1.99};  
  {Quantity = 2; Total = 3.98}; ]
```

```
let addLine line1 line2 =  
  {Quantity = line1.Quantity + line2.Quantity;  
   Total = line1.Total + line2.Total}
```

```
orderLines |> List.reduce addLine
```

# Эндоморфизмы

Эндоморфизм — функция, у которой тип входного значения совпадает с типом выходного

Множество функций + композиция — моноид, если функции — эндоморфизмы

## Пример

```
let plus1 x = x + 1
```

```
let times2 x = x * 2
```

```
let subtract42 x = x - 42
```

```
let functions = [  
  plus1;  
  times2;  
  subtract42 ]
```

```
let newFunction = functions |> List.reduce (>>)
```

```
printfn "%d" <| newFunction 20
```

# Не только эндоморфизмы могут образовать моноид

```
type Predicate<'A> = 'A -> bool
```

```
let predAnd p1 p2 x =  
  if p1 x  
  then p2 x  
  else false
```

```
let predicates = [isMoreThan10Chars; isMixedCase;  
  isNotDictionaryWord]
```

```
let combinePredicates = predicates |> List.reduce predAnd
```

# Полезные ссылки

Откуда взяты примеры

- ▶ <https://fsharpforfunandprofit.com/series/computation-expressions.html> — описание Workflow-ов в F# без использования слова «монада»
- ▶ <https://fsharpforfunandprofit.com/fppatterns/> — отличная презентация про ФП вообще, включая Railroad programming и монады
- ▶ <https://habr.com/ru/articles/127556/> — перевод статьи с простым объяснением монад в Haskell
- ▶ <https://sleepyfran.github.io/blog/posts/fsharp/ce-in-fsharp/> — пример с DSL на Workflow-ax