Monady w F#

Wzorzec projektowy “Iterator” jest abstrakcją …

```
// type seq<'T> = IEnumerable<'T>

let numbers2: seq<int> = [| 1; 2; 3; 4 |] // List<T>, T[], Dictionary<T,K>, ...


// 1.
// petla "foreach"

for number in numbers2 do
    printfn "item: %d" number



// 2.
// funkcje dzialajace na sekwencjach (map, filter, fold, ...)

let evenNumbers1 = numbers2 |> Seq.filter (fun x -> x % 2 = 0)
let evenNumbers2 = numbers2.Where(fun x -> x % 2 = 0) // LINQ
```

seq<T>

```
// 'a -> seq<'a>
let returnS value = seq { yield value }

returnS 1 // -> seq [1]
Seq.singleton 1

// ('a -> 'b) -> seq<'a> -> seq<'b>
let mapS f items =
    seq {
        for item in items do
            yield f item
    }

mapS (fun x -> x * 10) [ 2; 3 ] // -> seq [20; 30]
Seq.map (fun x -> x * 10) [ 2; 3 ]
[ 2; 3 ].Select(fun x -> x * 10)


// ('a -> seq<'b>) -> seq<'a> -> seq<'b>
let bindS f items =
    seq {
        for item in items do
            yield! f item
    }

bindS (fun x -> Seq.replicate x x) [ 2; 3 ] // -> seq [2; 2; 3; 3; 3;]
Seq.collect (fun x -> Seq.replicate x x) [ 2; 3 ]
[ 2; 3 ].SelectMany(fun x -> Seq.replicate x x)
```

Array<T>

```
// 'a -> Array<'a>
let returnA value = [| value |]

returnA 1 // -> [|1|]

// ('a -> 'b) -> Array<'a> -> Array<'b>
let mapA f items =
    let len = Array.length items
    let result = Array.zeroCreate len
    for i = 0 to len - 1 do
        result.[i] <- f items.[i]
    result

mapA (fun x -> x * 10) [| 2; 3 |] // -> [|20; 30|]
Array.map (fun x -> x * 10) [| 2; 3 |]
System.Array.ConvertAll([| 1; 2; 3; 4 |], (fun x -> x * 10))

// ('a -> Array<'b>) -> Array<'a> -> Array<'b>
let bindA f items =
    let len, arrays =
        items
        |> Array.fold
            (fun (len, arrays) item ->
                let array = f item
                len + Array.length array, (len, array) :: arrays)
            (0, [])
    let result = Array.zeroCreate len
    for index, array in arrays do
        Array.blit array 0 result index array.Length
    result

bindA (fun x -> Array.replicate x x) [| 2; 3 |] // -> [|2; 2; 3; 3; 3|]
Array.collect (fun x -> Array.replicate x x) [| 2; 3 |]
```

Task<T>

```
// 'a -> Task<'a>
let returnT value = Task.FromResult value

returnT 1 // -> Task<int> { Result = 1}



// ('a -> 'b) -> Task<'a> -> Task<'b>
let mapT<'a, 'r> (f: 'a -> 'r) (task: Task<'a>) =
     task.ContinueWith(fun (t: Task<'a>) -> f t.Result)

let parseIntAsync str =
     Task.Delay(1000).ContinueWith(fun _ -> Int32.Parse str) // string -> Task<int>

mapT (fun x -> x * 10) (parseIntAsync "6") // -> Task<int> { Result = 60}



// ('a -> Task<'b>) -> Task<'a> -> Task<'b>
let bindT<'a, 'r> (f: 'a -> Task<'r>) (task: Task<'a>) =
     task.ContinueWith(fun (t: Task<'a>) -> f t.Result).Unwrap()

bindT (fun x -> String.replicate x "6" |> parseIntAsync) (parseIntAsync "3")// -> Task<int> { Result = 666}
```

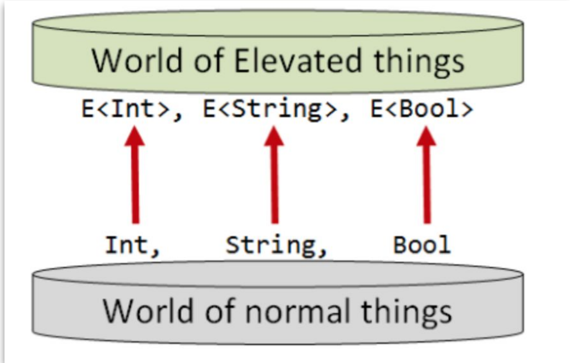
Option<T>

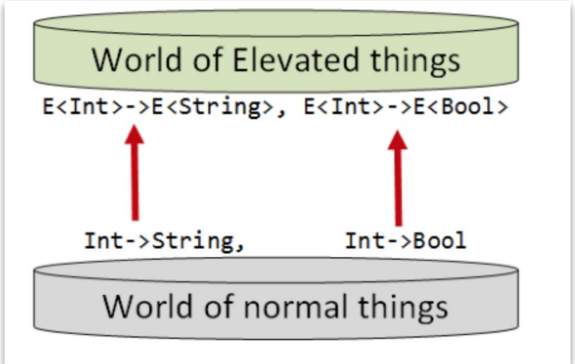
```
// 'a -> Option<'a>
let returnO value = Some value

returnO 1 // -> Some 1




// ('a -> 'b) -> Option<'a> -> Option<'b>
let mapO f option =
    match option with
    | Some value -> Some(f value)
    | None -> None

mapO (fun x -> x * 10) (tryParseInt "2") // -> Some 20
Option.map (fun x -> x * 10) (tryParseInt "2")




// ('a -> Option<'b>) -> Option<'a> -> Option<'b>
let bindO f option =
    match option with
    | Some value -> f value
    | None -> None

bindO (fun x -> String.replicate x "6" |> tryParseInt) (tryParseInt "3") // -> Some 666
Option.bind (fun x -> String.replicate x "6" |> tryParseInt) (tryParseInt "3")
```

funktor … aplikatywny funktor … monada

return:	T	->	E<T>		
map:	(T -> R)	->	E<T>	->	E<R>
bind:	(T -> E<R>)	->	E<T>	->	E<R>
apply:	E< T -> R >	->	E<T>	->	E<R>

- Funktor: map
- Aplikatywny funktor: return, apply
  - let map f e = apply (return f) e
- Monada: return, bind
  - let map f e = bind (fun x -> f x |> return) e
  - let apply f e = bind (fun x -> f |> map (fun ff -> ff x)) e





https://fsharpforfunandprofit.com/posts/elevated-world/

“Monadic computation” (Option)

```
// string -> string -> int
let parseTwoInts str1 str2 =
    let value1 = Int32.Parse str1
    let value2 = Int32.Parse str2
    value1 + value2

// string -> string -> option<int>
let tryParseTwoInts str1 str2 =
    match tryParseInt str1 with
    | None -> None
    | Some value1 ->
        match tryParseInt str2 with
        | None -> None
        | Some value2 -> Some(value1 + value2)

// string -> string -> option<int>
let tryParseTwoInts'' str1 str2 =
    tryParseInt str1
    |> bindO (fun value1 -> tryParseInt str2 |> bindO (fun value2 -> returnO (value1 + value2)))

// string -> string -> option<int>
let parseTwoInts''' str1 str2 =
    option {
        let! value1 = tryParseInt str1
        let! value2 = tryParseInt str2
        return value1 + value2
    }
```

“Monadic computation” (Task)

```
// string -> string -> Task<int>
let parseTwoIntsAsync str1 str2 =
    parseIntAsync str1
    |> bindT (fun value1 -> parseIntAsync str2 |> bindT (fun value2 -> returnT (value1 + value2)))


// string -> string -> Task<int>
let parseTwoIntsAsync' str1 str2 =
    task {
        let! value1 = parseIntAsync str1
        let! value2 = parseIntAsync str2
        return value1 + value2
    }

// async/await w C# “zostal skopiowany” z F# :)
```

F# Computation Expression

```fsharp
let parseTwoInts str1 str2 =
    option {
        let! value1 = tryParseInt str1
        let! value2 = tryParseInt str2
        return value1 + value2
    }

let parseTwoInts str1 str2 =
    option.Bind(
        (tryParseInt str1),
        (fun value1 -> option.Bind(tryParseInt str2, (fun value2 -> option.Return(value1 + value2))))
    )
```

```fsharp
// https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/computation-expressions

type OptionBuilder() =
    member this.Return(value) = return0 value
    member this.Bind(monad, binder) = bind0 binder monad

let option = OptionBuilder()
```

```fsharp
type DelayedO<'T> = unit -> Option<'T>

type OptionBuilder'() =
    member this.Yield(value) = returnO value
    member this.YieldFrom(value) = value
    member this.Return(value) = returnO value
    member this.ReturnFrom(value) = value
    member this.Zero() = returnO Unchecked.defaultof<_>
    member this.Delay(f: DelayedO<_>) = f
    member this.Run(delayed: DelayedO<_>) = delayed ()
    member this.Bind(monad, binder) = bindO binder monad
    member this.Combine(monad1, monad2: DelayedO<_>) = monad1 |> bindO (fun _ -> this.Run(monad2))

    member this.TryFinally(body: DelayedO<_>, finallyBody) =
        try
            this.Run(body)
        finally
            finallyBody ()

    member this.TryWith(body: DelayedO<_>, catchBody) =
        try
            this.Run(body)
        with
        | e -> catchBody e

    member this.Using(res: #IDisposable, body) =
        this.TryFinally(this.Delay(fun _ -> body res), (fun () -> if not (isNull (box res)) then res.Dispose()))

    member this.While(guard, body: DelayedO<_>) =
        if guard () then this.Run(body) |> bindO (fun _ -> this.While(guard, body)) else this.Zero()

    member this.For(sequence: seq<_>, body) =
        this.Using(
            sequence.GetEnumerator(),
            (fun iterator -> this.While((fun () -> iterator.MoveNext()), this.Delay(fun _ -> body iterator.Current)))
        )

// Creating a New Type of Computation Expression
// https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/computation-expressions#creating-a-new-type-of-computation-expression
```

Co dają nam abstrakcje?

“option/task { ... }” dla Monad jest ...

... tym czym ...

“foreach(var item in items) { ... }” jest dla wzorca iteratora (usprawnieniami w składni języka programowania)

Monadic Functions (mapM)

```
// ('a -> 'b ) -> seq<'a> -> seq<'b>
[ "1"; "2"; "3" ] |> Seq.map tryParseInt // -> [Some 1; Some 2; Some 3]


// Option<T>
// ('a -> Option<'b> ) -> seq<'a> -> Option<seq<'b>>
[ "1"; "2"; "3" ] |> mapMO tryParseInt // -> Some [1; 2; 3]
[ "1"; "2"; "3a" ] |> mapMO tryParseInt // -> None


// Task<T>
// ('a -> Task<'b> ) -> seq<'a> -> Task<seq<'b>>
[ "1"; "2"; "3" ] |> mapMT parseIntAsync // -> Task<seq<int>> { Result = [1; 2; 3] }



let mapM bindM returnM mapM f items =
    items |> foldM bindM returnM (fun a c -> f c |> mapM (fun v -> Seq.append a [ v ])) Seq.empty

let mapMO f items = mapM bindO returnO mapO f items
let mapMT f items = mapM bindT returnT mapT f items
```

Monadic Functions (filterM)

```
// Option<T>
// ('a -> Option<bool> ) -> seq<'a> -> Option<seq<'a>>
[ "1"; "2"; "3"; "4" ]
|> filterMO (fun x -> tryParseInt x |> mapO (fun i -> i % 2 = 0)) // -> Some [2; 4]


// Task<T>
// ('a -> Task<bool> ) -> seq<'a> -> Task<seq<'b>>
[ "1"; "2"; "3"; "4" ]
|> filterMT (fun x -> parseIntAsync x |> mapT (fun i -> i % 2 = 0)) // -> Task<seq<int>> {Result = [2; 4]}



let filterM bindM returnM mapM f items =
    items
    |> foldM bindM returnM (fun a c -> f c |> mapM (fun v -> if v then Seq.append a [ c ] else a)) Seq.empty

let filterMO f items = filterM bindO returnO mapO f items
let filterMT f items = filterM bindT returnT mapT f items
```

Monadic Functions (foldM)

```
// Option<T>
// ('a -> 'b -> Option<'a> ) -> 'a -> seq<'b> -> Option<'a>
[ "1"; "2"; "3" ]
|> foldMO (fun a c -> tryParseInt c |> mapO (fun v -> a + v)) 0 // -> Some 6

[ "1"; "2"; "3a" ]
|> foldMO (fun a c -> tryParseInt c |> mapO (fun v -> a + v)) 0 // -> None


// Task<T>
// ('a -> 'b -> Task<'a>) -> 'a ->  seq<'b> -> Task<'a>
[ "1"; "2"; "3" ]
|> foldMT (fun a c -> parseIntAsync c |> mapT (fun v -> a + v)) 0 // -> Task<int> { Result= 6 }
```

```
let foldM bindM returnM f seed items =
    items |> Seq.fold (fun state item -> state |> bindM (fun v -> f v item)) (returnM seed)

let foldMO f seed items = foldM bindO returnO f seed items
let foldMT f seed items = foldM bindT returnT f seed items
```

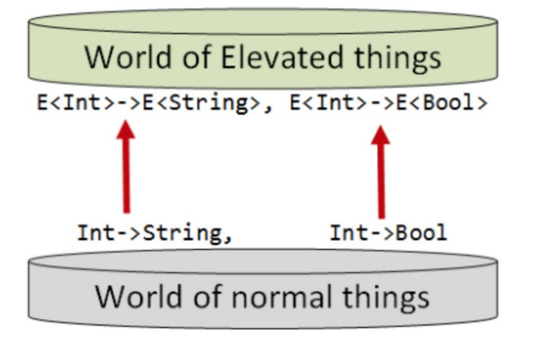
Co dają nam abstrakcje ?

funkcje mapM/filterM/folderM/… dla Monad są ...

... tym czym ...

funkcje filter/map/fold/… są dla wzorca iteratora
(funkcjami pomocniczymi bazującymi na abstrakcji IEnumerable<T>)

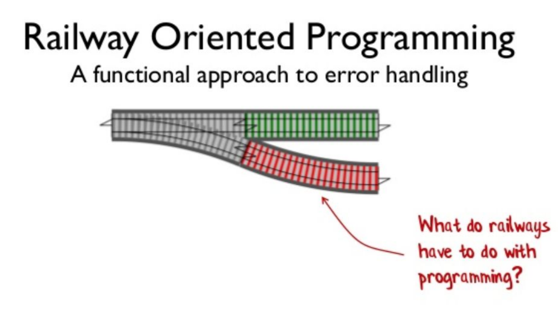
Nowa interpretacja funkcji map, bind, apply




```
// "liftery"
// return:      T               ->  E<T>
// map:         (T -> R)        ->  E<T> -> E<R>
// bind:        (T -> E<R>)     ->  E<T> -> E<R>
// apply:       E<T -> R>       ->  E<T> -> E<R>
```

```
// string |> (string -> int) |> (int -> int) |> (string -> int) |> ....
"1" |> parseInt |> increment |> string |> duplicateStr |> parseInt // -> 22
```

```
"1"
|> tryParseInt              // string -> Option<int>
|> mapO increment           // Option<int> -> Option<int>
|> mapO string              // Option<int> -> Option<string>
|> mapO duplicateStr        // Option<string> -> Option<string>
|> bindO tryParseInt        // Option<string> -> Option<int>
```



https://fsharpforfunandprofit.com/rop

Funkcje przyjmujące wiele argumentów (Lift**M**, “**m**onadycznie”)

```
let add a b = a + b
add 1 2 // -> 3
(1, 2) ||> add // let (||>) (a1, a2) f = f a1 a2

// (int,int) ||> (int -> int -> int) |> (int -> int)
parsePoint "1,2" ||> add |> increment // -> 4

// liftMO2: ('a -> 'b -> 'c) -> Option<'a> -> Option<'b> -> Option<'c>
// liftMT2: ('a -> 'b -> 'c) -> Task<'a> -> Task<'b> -> Task<'c>


// (Option<int>,Option<int>) ||> (Option<int> -> Option<int> -> Option<int>) |> (Option<int> -> ...)
tryParsePoint "1,2" ||> liftMO2 add |> mapO increment // -> Some 4

// (Task<int>,Task<int>) ||> (Task<int> -> Task<int> -> Task<int>)
(parseIntAsync "1", parseIntAsync "2") ||> liftMT2 add // Task<int> {Result=3}



let liftM2 bindM mapM f m1 m2 = m1 |> bindM (fun v1 -> m2 |> mapM (fun v2 -> f v1 v2))
let liftMO2 f o1 o2 = liftM2 bindO mapO f o1 o2
let liftMT2 f t1 t2 = liftM2 bindT mapT f t1 t2


// ('a -> 'b -> 'c -> 'd) -> Option<'a> -> Option<'b> -> Option<'c> -> Option<'d>
let liftMO3 f m1 m2 m3 = m1 |> bindO (fun v1 -> m2 |> bindO (fun v2 -> m3 |> mapO (fun v3 -> f v1 v2 v3)))
let liftMT3 f m1 m2 m3 = m1 |> bindT (fun v1 -> m2 |> bindT (fun v2 -> m3 |> mapT (fun v3 -> f v1 v2 v3)))
```

Funkcje przyjmujące wiele argumentów (Lift**A**, “**a**plikatywnie”)

```
// liftM i liftA posiadaja identyczna sygnature i zwracany rezultat (ale inne dzialanie)
tryParsePoint "1,2" ||> liftMA2 add |> mapO increment // -> Some 4
(parseIntAsync "1", parseIntAsync "2") ||> liftMA2 add // -> Task<int> {Result=3}


// Option<('a -> 'b)> -> Option<'a> -> Option<'b>
let applyO f option =
    match f, option with
    | Some ff, Some value -> Some(ff value)
    | _ -> None

// Task<('a -> 'b)> -> Task<'a> -> Task<'b>
let applyT (f: Task<('a -> 'b)>) (task: Task<'a>) =
    Task.WhenAll([ f :> Task; task :> Task ]).ContinueWith(fun _ -> f.Result task.Result)


// funkcje pomocnicze, zamieniona kolejnosc parametrow
let applyO_ option f = applyO f option
let applyT_ option f = applyT f option

let liftAO2 f o1 o2 = o1 |> mapO f |> applyO_ o2
let liftAT2 f t1 t2 = t1 |> mapT f |> applyT_ t2
```

Funkcje przyjmujące wiele argumentów (Lift**A**, “**a**plikatywnie”)

```
// Option<T>
// ('a -> 'b -> 'c) -> Option<'a> -> Option<'b> -> Option<'c>
// ('a -> 'b -> 'c -> 'd) -> Option<'a> -> Option<'b> -> Option<'c> -> Option<'d>
// ('a -> 'b -> 'c -> 'd -> 'e ) -> Option<'a> -> Option<'b> -> Option<'c> -> Option<'d> -> Option<'e>
let liftAO2 f o1 o2 = o1 |> mapO f |> applyO_ o2
let liftAO3 f o1 o2 o3 = o1 |> mapO f |> applyO_ o2 |> applyO_ o3
let liftAO4 f o1 o2 o3 o4 = o1 |> mapO f |> applyO_ o2 |> applyO_ o3|> applyO_ o4


// uogolniona implementacja dla dowolnego E<T>, dodatkowo liftA3 uzywa liftA2, ...
let liftA2 mapM applyM f m1 m2 = m1 |> mapM f |> applyM m2
let liftA3 mapM applyM1 applyM2 f m1 m2 m3 =
    liftA2 mapM applyM1 f m1 m2 |> applyM2 m3
let liftA4 mapM applyM1 applyM2 applyM3 f m1 m2 m3 m4 =
    liftA3 mapM applyM1 applyM2 f m1 m2 m3 |> applyM3 m4


// Option<T>
let liftAO2 f o1 o2 = liftA2 mapO applyO_ f o1 o2
let liftAO3 f o1 o2 o3 = liftA3 mapO applyO_ applyO_ f o1 o2 o3
let liftAO4 f o1 o2 o3 o4 = liftA4 mapO applyO_ applyO_ applyO_ f o1 o2 o3 o4
```

Przetwarzanie monadyczne vs aplikatywne

```
// zakladajac ze 'parseIntAsync' zwraca wynik po 1s
// obie ponizsze implementacje (liftM, liftA) zwroca wynik po 1s
(parseIntAsync "1", parseIntAsync "2") ||> liftMT2 add
(parseIntAsync "1", parseIntAsync "2") ||> liftAT2 add


// dlaczego?




// jak 'pod spodem' dzialaja funkcje liftM i liftA
let liftMT2 f t1 t2 = t1 |> bindT (fun v1 -> t2 |> mapT (fun v2 -> f v1 v2))

let liftMT2 f t1 t2 =
     t1.ContinueWith(fun tt1 -> t2.ContinueWith(fun tt2 -> f tt1.Result tt2.Result)).Unwrap()


let liftAT2 f t1 t2 = t1 |> mapT f |> applyT_ t2

let liftAT2 f t1 t2 =
     let ft = t1.ContinueWith(fun tt1 -> f tt1.Result)
     Task.WhenAll([ ft ; t2 ]).ContinueWith(fun _ -> ft.Result t2.Result)
```

Przetwarzanie monadyczne vs aplikatywne (Rx)

```
// implementacja funkcji dla typu IObservable<T>
let returnR value = Observable.Return(value)
let mapR f obs = Observable.Select(obs, (f: _ -> _))
let bindR f obs = Observable.SelectMany(obs, (f: _ -> IObservable<_>))
let applyR f obs = Observable.CombineLatest(f, obs, (fun f v -> f v))

let liftMR2 f t1 t2 = t1 |> bindR (fun v1 -> t2 |> mapR (fun v2 -> f v1 v2))
let applyR_ o f = applyR f o
let liftAR2 f t1 t2 = t1 |> mapR f |> applyR_ t2


// string -> IObservable<int>
let parseIntAsyncR str =
     Observable.Delay(Observable.Return(Int32.Parse str), TimeSpan.FromMilliseconds(1000))


// wykorzystujac funkcje liftM, wynik zostanie zwrocony po 2s
let res1 = (parseIntAsyncR "1", parseIntAsyncR "2") ||> liftMR2 add
res1.Subscribe(fun v -> printfn " %d" v) |> ignore

// wykorzystujac funkcje liftA, wynik zostanie zwrocony po 1s
let res2 = (parseIntAsyncR "1", parseIntAsyncR "2") ||> liftAR2 add
res2.Subscribe(fun v -> printfn " %d" v) |> ignore
```