Funktionale Programmierung in F# (3)

Grundlagen & Funktionales Design

Göran Kirchner[1]

2025-04-02

[1] e_kirchnerg@doz.hwr-berlin.de

Programm

- Hausaufgaben (5..8/10)
  - ☒ Queen Attack
  - ☒ RaindropsO
  - ☒ Gigaseconds
  - ☒ Bank Account
- Vertiefung Railway-Oriented Programming
- Prinzipien des funktionalen Designs
- Refactoring (Übung)

Queen Attack

```
open System
let create (row, col) = row >= 0 && row < 8 && col >= 0 && col < 8
let canAttack (queen1: int * int) (queen2: int * int) =
    let (r1, c1) = queen1
    let (r2, c2) = queen2
    Math.Abs(r1 - r2) = Math.Abs(c1 - c2) || r1 = r2 || c1 = c2
let whiteQueen1, blackQueen1 = (2, 2), (1, 1)
let test1 = canAttack blackQueen1 whiteQueen1
let whiteQueen2, blackQueen2 = (2, 4), (6, 6)
let test2 = canAttack blackQueen2 whiteQueen2
```

```
val create: row: int * col: int -> bool
val canAttack: int * int -> int * int -> bool
val whiteQueen1: int * int = (2, 2)
val blackQueen1: int * int = (1, 1)
val test1: bool = true
val whiteQueen2: int * int = (2, 4)
val blackQueen2: int * int = (6, 6)
val test2: bool = false
```

Raindrops

```
let rules =
    [ 3, "Pling"
      5, "Plang"
      7, "Plong" ]
let convert (number: int): string =
    let divBy n d = n % d = 0
    rules
    |> List.filter (fst >> divBy number)
    |> List.map snd
    |> String.concat ""
    |> function
       | "" -> string number
       | s -> s
let test = convert 105
```

```
val rules: (int * string) list = [(3, "Pling"); (5, "Plang"); (7, "Plong")]
val convert: number: int -> string
val test: string = "PlingPlangPlong"
```

Gigasecond

```
let add (beginDate : System.DateTime) = beginDate.AddSeconds 1e9
let test = add (DateTime(2015, 1, 24, 22, 0, 0)) = (DateTime(2046, 10,
↪  2, 23, 46, 40))
```

```
val add: beginDate: DateTime -> DateTime
val test: bool = true
```

Bank Account (1)

```
type OpenAccount =
    { mutable Balance: decimal }
type Account =
    | Closed
    | Open of OpenAccount
let mkBankAccount() = Closed
let openAccount account =
    match account with
    | Closed -> Open { Balance = 0.0m }
    | Open _ -> failwith "Account is already open"
```

```
type OpenAccount =
  { mutable Balance: decimal }
type Account =
  | Closed
  | Open of OpenAccount
val mkBankAccount: unit -> Account
val openAccount: account: Account -> Account
```

Bank Account (2)

```
let closeAccount account =
    match account with
    | Open _ -> Closed
    | Closed -> failwith "Account is already closed"
let getBalance account =
    match account with
    | Open openAccount -> Some openAccount.Balance
    | Closed -> None
let updateBalance change account =
    match account with
    | Open openAccount ->
        lock (openAccount) (fun _ ->
            openAccount.Balance <- openAccount.Balance + change
            Open openAccount)
    | Closed -> failwith "Account is closed"
```

Bank Account (3)

```
let account = mkBankAccount() |> openAccount
let updateAccountAsync =
    async { account |> updateBalance 1.0m |> ignore }
let ``updated from multiple threads`` =
    updateAccountAsync
        |> List.replicate 1000
        |> Async.Parallel
        |> Async.RunSynchronously
        |> ignore
let test1 = getBalance account = (Some 1000.0m)
```

Übung 1

- Implementiere einen Workflow (`validateInput`).

```
type Input = {Name : string; Email : string }
let checkNameNotBlank input =
  if input.Name = "" then
     Error "Name must not be blank"
  else Ok input
let checkName50 input =
  if input.Name.Length > 50 then
     Error "Name must not be longer than 50 chars"
  else Ok input
let checkEmailNotBlank input =
  if input.Email = "" then
     Error "Email must not be blank"
  else Ok input
```

Übung 1 (Lösung)

```
let validateInput input =
    input
    |> checkNameNotBlank
    |> Result.bind checkName50
    |> Result.bind checkEmailNotBlank

let goodInput = {Name="Max"; Email="x@example.com"}
let blankName = {Name=""; Email="x@example.com"}
let blankEmail = {Name="Nora"; Email=""}
[validateInput goodInput; validateInput blankName; validateInput
↪  blankEmail]
```

Übung 2

- Definiere einen *Custom Error Type*. Benutze diesen in den Validierungen.
- Übersetze die Fehlermeldungen (EN, FR, DE?).

```
type ErrorMessage =
  | ??   // name not blank
  | ?? of int  // name not longer than
  | ??   // email not longer than
let translateError_EN err =
  match err with
  | ?? -> "Name must not be blank"
  | ?? i -> sprintf "Name must not be longer than %i chars" i
  | ?? -> "Email must not be blank"
  | SmtpServerError msg -> sprintf "SmtpServerError [%s]" msg
```

Übung 2 (Lösung)

```
type ErrorMessage =
    | NameMustNotBeBlank
    | NameMustNotBeLongerThan of int
    | EmailMustNotBeBlank
    | SmtpServerError of string
let translateError_FR err =
    match err with
    | NameMustNotBeBlank -> "Nom ne doit pas être vide"
    | NameMustNotBeLongerThan i -> sprintf "Nom ne doit pas être plus
    ↪  long que %i caractères" i
    | EmailMustNotBeBlank -> "Email doit pas être vide"
    | SmtpServerError msg -> sprintf "SmtpServerError [%s]" msg
```

Funktionales Design

$\rightsquigarrow$ Functional Design Patterns

–Scott Wlashin: F# for Fun and Profit

Prinzipien (1)

- Funktionen sind Daten!
- überall Verkettung (Composition)
- überall Funktionen
- Typen sind keine Klassen
- Typen kann man ebenfalls verknüpfen (algebraische Datentypen)
- Typsignaturen lügen nicht!
- statische Typen zur Modellierung der Domäne (später mehr;)

Prinzipien (2)

- Parametrisiere alles!
- Typsignaturen sind "Interfaces"
- Partielle Anwendung ist "Dependency Injection"
- Monaden entsprechen dem Chaining of Continuations"
  - bind für Options
  - bind für Fehler
  - bind für Tasks
- map Funktionen
  - Nutze map Funktion von generische Typen!
  - wenn man einen generischen Typ definiert, dann auch eine map Funktion

Übung 3

- Typsignaturen
- Funktionen sind Daten

Übung 4 (Think of a Number)

```
let thinkOfANumber numberYouThoughtOf =
    let addOne x = x + 1
    let squareIt x = ??
    let subtractOne x = ??
    let divideByTheNumberYouFirstThoughtOf x = ??
    let subtractTheNumberYouFirstThoughtOf x = ??

    // define these functions
    // then combine them using piping

    numberYouThoughtOf
    |> ??
    |> ??
    |> ??
```

Übung 4 (Lösung)

```
let thinkOfANumber numberYouThoughtOf =
    let addOne x = x + 1
    let squareIt x = x * x
    let subtractOne x = x - 1
    let divideByTheNumberYouFirstThoughtOf x = x / numberYouThoughtOf
    let subtractTheNumberYouFirstThoughtOf x = x - numberYouThoughtOf
    numberYouThoughtOf
    |> addOne
    |> squareIt
    |> subtractOne
    |> divideByTheNumberYouFirstThoughtOf
    |> subtractTheNumberYouFirstThoughtOf
thinkOfANumber 42
```

```
val thinkOfANumber: numberYouThoughtOf: int -> int
val it: int = 2
```

Übung 5 (Decorator)

- Implementiere das Decorator-Emtwurfsmuster für `add1`.

Pause

> If we'd asked the customers what they wanted, they would have said "faster horses".
>
> – Henry Ford

Tree Building (Übung)

```
exercism download --exercise=tree-building --track=fsharp
```

Tree Building (Imperativ)

```
let buildTree records =
    let records' = List.sortBy (fun x -> x.RecordId) records
    if List.isEmpty records' then failwith "Empty input"
    else
        let root = records'.[0]
        if (root.ParentId = 0 |> not) then
            failwith "Root node is invalid"
        else
            if (root.RecordId = 0 |> not) then failwith "Root node is
            ↪  invalid"
            else
                let mutable prev = -1
                let mutable leafs = []
                for r in records' do
                    if (r.RecordId <> 0 && (r.ParentId > r.RecordId ||
                    ↪   r.ParentId = r.RecordId)) then
                        failwith "Nodes with invalid parents"
                    else
                        if r.RecordId <> prev + 1 then
                            failwith "Non-continuous list"
                        else
                            prev <- r.RecordId
```

Tree Building (Funktional)

```
let buildTree records =
    records
    |> List.sortBy (fun r -> r.RecordId)
    |> validate
    |> List.tail
    |> List.groupBy (fun r -> r.ParentId)
    |> Map.ofList
    |> makeTree 0

let rec makeTree id map =
    match map |> Map.tryFind id with
    | None -> Leaf id
    | Some list -> Branch (id,
        list |> List.map (fun r -> makeTree r.RecordId map))
```

Tree Building (Error Handling)

```
let validate records =
    match records with
    | [] -> failwith "Input must be non-empty"
    | x :: _ when x.RecordId <> 0 ->
        failwith "Root must have id 0"
    | x :: _ when x.ParentId <> 0 ->
        failwith "Root node must have parent id 0"
    | _ :: xs when xs |> List.exists (fun r -> r.RecordId < r.ParentId)
    ↪   ->
        failwith "ParentId should be less than RecordId"
    | _ :: xs when xs |> List.exists (fun r -> r.RecordId = r.ParentId)
    ↪   ->
        failwith "ParentId cannot be the RecordId except for the root
        ↪   node."
    | rs when (rs |> List.map (fun r -> r.RecordId) |> List.max) >
    ↪   (List.length rs - 1) ->
        failwith "Ids must be continuous"
    | _ -> records
```

Tree Building (Benchmarking)

- BenchmarkDotNet

```
dotnet run -c release
```

```
sed -n 381,391p $benchmarks
```

BenchmarkDotNet=v0.12.1, OS=macOS 13.7.4 (22H420) [Darwin 22.6.0] Intel Core i7-7920HQ CPU 3.10GHz (Kaby Lake), 1 CPU, 8 logical and 4 physical cores .NET Core SDK=9.0.200 [Host] : .NET Core 9.0.2 (CoreCLR 9.0.225.6610, CoreFX 9.0.225.6610), X64 RyuJIT DEBUG DefaultJob : .NET Core 9.0.2 (CoreCLR 9.0.225.6610, CoreFX 9.0.225.6610), X64 RyuJIT

Method	Mean	Error	StdDev	Ratio	RatioSD	Gen 0	Gen 1	Gen 2	Allocated
Baseline	6.308 s	0.1256 s	0.2328 s	1.00	0.00	3.3188	-	-	13.56 KB
Mine	3.567 s	0.0703 s	0.1250 s	0.57	0.03	1.8196	-	-	7.45 KB

Zusammenfassung

- funktionaler Umgang mit Fehlern (ROP)
- funktionales Design
- funktionales Refactoring

Links

- oodesign.com
- fsharp.org
- docs.microsoft.com/../dotnet/fsharp
- F# weekly
- fsharpforfunandprofit.com
- github.com/../awesome-fsharp

Hausaufgabe (Erinnerung)

- exercism.io (bis 07.04)
  - ☐ Accumulate
  - ☐ Space Age
- exercism.io (bis 07.04)
  - ☐ Poker (Programmieraufgabe)

Termine

- ☒ 12.03 11:30 - 15:45
- ☒ 26.03 11:30 - 15:45 (online)
- ☒ 02.04 8:45 - 11:45 (online)
- ☐ 04.04 13:00 - 17:15 (online?!)
- ☐ 09.04 11:30 - 15:45