

Funktionale Programmierung in F# (1)

Erste Schritte in F#

Göran Kirchner¹

2021-02-26

¹e_kirchnerg@doz.hwr-berlin.de

Organisatorisches

- Termine
 - [26.02, 12.03, 26.03, 09.04, 23.04]
- Bewertung
 - .50: Hausaufgaben (10)
 - .25: Programmieraufgabe (in der vorvorletzten Einheit)
 - .25: Test (multiple choice, in der letzten Einheit)

Was ist F#

F# ist eine funktionale Programmiersprache, die das Schreiben von korrektem und verwaltbarem Code erleichtert. In F# können Typen und Funktionen geschrieben werden, deren Typen automatisch generalisiert werden. Dies ermöglicht es, sich auf die Problemdomäne zu konzentrieren und die Daten zu bearbeiten, statt sich um die Details der Programmierung zu kümmern.

F# verfügt über zahlreiche Features, einschließlich:

- Leichtgewichtige Syntax
- Standardmäßig unveränderliche Werte
- Typrückschluss und automatische Generalisierung
- Funktionen erster Klasse
- Musterabgleich
- Asynchrone Programmierung

So sieht F# aus

```
open System // Gets access to functionality in System namespace.

// Defines a function that takes a name and produces a greeting.
let getGreeting name =
    sprintf "Hello, %s! Isn't F# great?" name

[<EntryPoint>]
let main args =
    // Defines a list of names
    let names = [ "Don"; "Julia"; "Xi" ]
    // Prints a greeting for each name!
    names
    |> List.map getGreeting
    |> List.iter (fun greeting -> printfn "%s" greeting)

0
```

Programm

- ➊ Einführung & Erste Schritte
- ➋ Grundlagen & ROP (Railway Oriented Programming)
- ➌ Grundlagen & DDD (Domain Driven Design)
- ➍ Property-Based Testing
- ➎ Parser Combinators

Einführung & Erste Schritte (1)

- Werkzeuge (VS Code, git, dotnet)
- Datentypen
- Collections
- Pattern Matching
- Funktionen

Grundlagen & ROP - Railway Oriented Programming (2)

- Fehlende Daten (option)
- Umgang mit Fehlern
- Railway Oriented Programming
- (Asynchron & Parallel)

Grundlagen & DDD (Domain Driven Design) (3)

- Immutability
- Algebraische Datentypen, Record Types (Wiederholung)
- Typen-getriebene Programmierung (Type Driven Design)
- (Klassen)
- [Programmieraufgabe!]

Property-Based Testing (4)

- Testing
- Property-Based Testing
- FParsec
- Diamond-Kata?
- (Performance)

Parser Combinators (5)

- Parser Combinator verstehen
- FParsec
- [Test!]

Benötigte Software

- .Net Core SDK 5.0
- git
- Visual Studio Code
 - ionide.io
- exercism.io
- (github.com)

Übung 1

- Installation überprüfen
 - dotnet core
 - git
 - code (VS Code)
 - exercism

Lösungen 1

```
. /usr/local/opt/asdf/asdf.sh  
dotnet --version
```

5.0.101

```
git --version
```

git version 2.24.3 (Apple Git-128)

```
code --version
```

1.53.2
622cb03f7e070a9670c94bae1a45d78d7181fbd4
x64

```
exercism version
```

exercism version 3.0.13

Übung 2

- Erste Schritte in F# mit der .NET Core-CLI
- Erstelle ein F# Projekt
 - Untersuche das Ergebnis
 - Starte das Programm

Lösung 2

```
dotnet new sln -o FSNetCore
cd FSNetCore

dotnet new classlib -lang "F#" -o src/Library
dotnet add src/Library/Library.fsproj package Newtonsoft.Json
dotnet sln add src/Library/Library.fsproj

dotnet new console -lang "F#" -o src/App
dotnet add src/App/App.fsproj reference \
    src/Library/Library.fsproj
dotnet sln add src/App/App.fsproj

cd src/App
dotnet run Hello World
```

git

- git

```
git init
git clone '<repo>'
git status
git add .
git commit -m 'good message'
git log
```

- github.com
 - getting-started-with-github/create-a-repo
- gitignore.io
- How to Write a Git Commit Message

Übung 3

- Initialisiere ein git-Repo (`course.2021.hwr.fun`)
 - (pull von github)
 - Committe eine Änderung
 - Betrachte die Historie
 - (push nach github)

exercism

- exercism.io/about
- Konfiguration

```
exercism configure --token=99ada440-..-0be7ce2cfe40  
exercism configure --workspace=/foo
```

- Download und Submit

```
exercism download --exercise=hello-world --track=fsharp  
exercism submit /path/to/file [/path/to/file2 ...]
```

Übung 4

- Konfiguriere exercism.io
- Lade die Hello-World-Aufgabe von exercism.io runter
 - Lass alle Tests laufen
 - Löse die Aufgabe
 - Submitte das Ergebnis
- commit & push

Lösung 4

```
exercism download --exercise=hello-world --track=fsharp
```

```
module HelloWorld =  
let hello: string = "Hello, World!"
```

Hello, World!

```
dotnet test  
exercism submit HelloWorld.fs
```

Pause

A language that doesn't affect the way you think about programming, is not worth knowing.

– Alan Perlis

Elementare Datentypen

```
let s = "hello" // string
let i = 42      // int
let f = 3.141   // float
let b = true    // bool
let l = [1;2;3] // list
printfn "%s, %i, %f, %g, %b, %A" s i f f b l
```

hello, 42, 3.141000, 3.141, true, [1; 2; 3]

val s : string = "hello"

val i : int = 42

val f : float = 3.141

val b : bool = true

val l : int list = [1; 2; 3]

Tuple

```
let t1 = (1, 2)
let t2 = ("one", "two", "three")
let t3 = (10, 10.0, "ten")
printfn "%A, %A %A" t1 t2 t3
```

```
(1, 2), ("one", "two", "three") (10, 10.0, "ten")
val t1 : int * int = (1, 2)
val t2 : string * string * string = ("one", "two", "three")
val t3 : int * float * string = (10, 10.0, "ten")
```

Discriminated Union

```
type Suit =  
    | Hearts  
    | Clubs  
    | Diamonds  
    | Spades  
type Rank =  
    | Value of int  
    | Ace  
    | King  
    | Queen  
    | Jack  
static member GetAllRanks() =  
    [ yield Ace  
      for i in 2 .. 10 do yield Value i  
      yield Jack; yield Queen; yield King ]
```


Record Types

- Gleichheit bei gleichen Werten
 - bei Klassen: Gleichheit bei gleicher Referenz!
- Können rekursiv sein

```
type Card = { Suit: Suit; Rank: Rank }  
  
/// This computes a list representing all the cards in the deck.  
let fullDeck =  
    [ for suit in [Hearts; Diamonds; Clubs; Spades] do  
        for rank in Rank.GetAllRanks() do  
            yield {Suit=suit; Rank=rank} ]  
fullDeck  
|> Seq.length
```

52

Lists, Array, Seq

```
let list1 = [ 1; 2; 3 ]
let list2 = [ for i in 1 .. 8 -> i*i ]
let list3 = []
let list4 = 100 :: list2
let list5 = list1 @ list2
let list6 = [1 .. 10]
let array1 = [| 1; 2; 3 |]
let seq1 = seq {1 .. 3}
printfn "%A, %A, %A" list1 array1 seq1
```











```
[1; 2; 3], [|1; 2; 3|], seq [1; 2; 3]
val list1 : int list = [1; 2; 3]
val list2 : int list = [1; 4; 9; 16; 25; 36; 49; 64]
val list3 : 'a list
val list4 : int list = [100; 1; 4; 9; 16; 25; 36; 49; 64]
val list5 : int list = [1; 2; 3; 1; 4; 9; 16; 25; 36; 49; 64]
val list6 : int list = [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]
val array1 : int [] = [|1; 2; 3|]
val seq1 : seq<int>
```

Unendliche Sequenzen :)

```
/// A random-number generator
let rand = System.Random() ;;
/// An infinite sequence of numbers
let randomNumbers = seq { while true do yield rand.Next(100000) }
/// The first 10 random numbers, sorted
let firstTenRandomNumbers =
    randomNumbers
    |> Seq.take 10
    |> Seq.sort
    |> Seq.toList
firstTenRandomNumbers
```

```
val randomNumbers : seq<int>
val firstTenRandomNumbers : int list =
    [3491; 24135; 35535; 45170; 48461; 59206; 82373; 85577; 87806; 98255]
```

Collection Functions

	Begin with	End up with	Functions
	Many	Equally Many	map, mapi, sort, sortBy, rev
	Many	Fewer	filter, choose, distinct, take, truncate, tail, sub
	Many	One	length, fold, reduce, average, head, sum, max, maxBy, min, minBy, find, pick
	Many	Boolean	exists, forall, isEmpty
	Nothing	Many	init, create, unfold
	Many	Nothing (except side-effects)	iter, iteri
	Many of Many	Many	concat, collect
	Many	Groupings	groupBy
	2 of Many	Many	append, zip
	Many	2 of Many	partition

Mapping (1)

```
let primeCubes = List.map (fun n -> n * n * n) [2;3;5;7;11;13;17]
primeCubes
```

```
val primeCubes : int list = [8; 27; 125; 343; 1331; 2197; 4913]
```

```
open System.IO
open System.Net
/// Get the contents of the URL via a web request
let http (url: string) =
    let req = WebRequest.Create(url)
    let resp = req.GetResponse()
    let stream = resp.GetResponseStream()
    let reader = new StreamReader(stream)
    let html = reader.ReadToEnd()
    resp.Close()
    html
```

```
val http : url:string -> string
```

Mapping (2)

```
let sites = ["http://www.bing.com"; "http://www.google.com"]
let fetch url = (url, http url)
let ps = List.map fetch sites
let ls = List.map (fun (_,p) -> String.length p) ps
printfn "%A" ls
```

[73266; 47957]

```
val sites : string list = ["http://www.bing.com"; "http://www.google.com"]
val fetch : url:string -> string * string
val ps : (string * string) list =
  [("http://www.bing.com",
    "<!doctype html><html lang="de" dir="ltr"><head><meta name="th"+[73205 char
    ("http://www.google.com",
    "<!doctype html><html itemscope="" itemType="http://schema.org"+[47896 char
val ls : int list = [73266; 47957]
```

Folding

```
let fold1 = List.fold (fun acc x -> acc + x) 0 [1..10]
let fold2 = [1..100] |> List.fold (+) 0
let fold3 = (0, [1..1000]) ||> List.fold (+)
printfn "%i, %i, %i" fold1 fold2 fold3
```

55, 5050, 500500

val fold1 : int = 55

val fold2 : int = 5050

val fold3 : int = 500500

Pause

Computers are useless. They can only give you answers.

– Pablo Picasso

Basics

```
let matchInt i =  
    match i with  
    | 1 -> printfn "One"  
    | 2 -> printfn "Two"  
    | _ -> printfn "Other" // "_" is a wildcard  
  
matchInt 1  
matchInt 2  
matchInt 77
```

One

Two

Other

```
val matchInt : i:int -> unit
```

When Guards

```
let caseSwitch input =  
    match input with  
    | 1 -> printfn "One"  
    | 2 -> printfn "A couple"  
    | x when x < 12 -> printfn "Less than a dozen"  
    | x when x = 12 -> printfn "A dozen"  
    | _ -> printfn "More than a dozen"  
  
caseSwitch 2  
caseSwitch 5  
caseSwitch 12  
caseSwitch 18
```

A couple

Less than a dozen

A dozen

More than a dozen

```
val caseSwitch : input:int -> unit
```

Matching Tuples (1)

```
let extremes (s : seq<'a>) =  
    s |> Seq.min,  
    s |> Seq.max  
  
let l, h = [1; 2; 9; 3; -1] |> extremes  
(l,h)
```

```
val extremes : s:seq<'a> -> 'a * 'a when 'a : comparison  
val l : int = -1  
val h : int = 9
```

Matching Tuples (2)

```
open System
let tryParseInt (s:string) =
    match System.Int32.TryParse(s) with
    | true, i -> Some i
    | false, _ -> None

let a = "30" |> tryParseInt // Some 30
let b = "3X" |> tryParseInt // None
(a,b)
```

```
val tryParseInt : s:string -> int option
val a : int option = Some 30
val b : int option = None
```

Matching Records

```
type Track = { Title : string; Artist : string } ;;
let songs = [ { Title = "Summertime"; Artist = "Ray Barretto" };
               { Title = "La clave, maraca y guiro";
                 Artist = "Chico Alvarez" };
               { Title = "Summertime";
                 Artist = "DJ Jazzy Jeff & The Fresh Prince" } ] ;;
let dist =
  songs
  |> Seq.map (fun s -> match s with | {Title = title} -> title)
  |> Seq.distinct |> Seq.toList
dist
```

```
val dist : string list = ["Summertime"; "La clave, maraca y guiro"]
```

Matching Lists

```
let caseList l =  
    match l with  
    | [] -> printfn "An empty pond"  
    | [fish] -> printfn "A pond with one fish only: %s" fish  
    | head::tail -> printfn "A pond with one fish: \  
        %s (and %i more fish)" head (tail |> List.length)  
  
caseList []  
caseList ["One fish"]  
caseList ["One fish"; "Two fish"; "Red fish" ]  
caseList ["One fish"; "Two fish"; "Red fish"; "Blue fish" ]
```

An empty pond

A pond with one fish only: One fish

A pond with one fish: One fish (and 2 more fish)

A pond with one fish: One fish (and 3 more fish)

val caseList : l:string list -> unit

Active Patterns (|?|)

```
let (|Even|Odd|) input = if input % 2 = 0 then Even else Odd
```

```
val ( |Even|Odd| ) : input:int -> Choice<unit,unit>
```

```
let TestNumber input =  
    match input with  
    | Even -> printfn "%d is even" input  
    | Odd -> printfn "%d is odd" input
```

```
TestNumber 7  
TestNumber 8  
TestNumber 9
```

7 is odd

8 is even

9 is odd

```
val TestNumber : input:int -> unit
```

Pause

Any fool can write code that a computer can understand. Good programmers write code that humans can understand.

– Martin Fowler

Basics

```
let squareIt1 n = n * n
let squareIt2 = fun n -> n * n
let r1 = squareIt1 8
let r2 = squareIt2 9

let listOfFunctions = [squareIt1; squareIt2]
for fn in listOfFunctions do
    let result = fn 100
    printfn "If 100 is the input, the output is %i" result
```

If 100 is the input, the output is 10000

If 100 is the input, the output is 10000

```
val squareIt1 : n:int -> int
```

```
val squareIt2 : n:int -> int
```

```
val r1 : int = 64
```

```
val r2 : int = 81
```

```
val listOfFunctions : (int -> int) list =
```

```
  [<fun:listOfFunctions@449-2>; <fun:listOfFunctions@449-3>]
```

Rekursion

```
/// Computes the greatest common factor of two integers.  
///  
/// Since all of the recursive calls are tail calls,  
/// the compiler will turn the function into a loop,  
/// which improves performance and reduces memory consumption.  
let rec gcf a b =  
    match a with  
    | 0 -> b  
    | a when a < b -> gcf a (b - a)  
    | _ -> gcf (a - b) b  
  
printfn "The Greatest Common Factor of 300 \  
        and 620 is %d" (gcf 300 620)
```

The Greatest Common Factor of 300 and 620 is 20
val gcf : a:int -> b:int -> int

Partielle Anwendung

```
let add1 = (+) 1
let r1 = add1 2    // result => 3
let multiplyBy2 = (*) 2
let r2 = multiplyBy2 3    // result => 6
let equals3 = (=) 3
let r3 = equals3 3    // result => true

printfn "%i, %i, %b" r1 r2 r3
```

```
3, 6, true
val add1 : (int -> int)
val r1 : int = 3
val multiplyBy2 : (int -> int)
val r2 : int = 6
val equals3 : (int -> bool)
val r3 : bool = true
```

Composition

```
let negate x = -1 * x
let square x = x*x
let print x = printfn "The number is: %d" x

let snp x = print (negate (square x))
let ``sqr, neg, and print`` x = x |> square |> negate |> print
let snp' = square >> negate >> print

snp 9, ``sqr, neg, and print`` 10, snp' 11
```

```
The number is: -81
The number is: -100
The number is: -121
val negate : x:int -> int
val square : x:int -> int
val print : x:int -> unit
val snp : x:int -> unit
val ( sqr, neg, and print ) : x:int -> unit
val snp' : (int -> unit)
```

Zusammenfassung

- Wichtige Werkzeuge (git, dotnet, code)
- Elementare Syntax
- Funktionen, Pattern Matching, Discriminated Unions (DU)
- Tuple, Record, List, Array, Seq
- Was ist Funktionale Programmierung?

Links

- fsharp.org
- docs.microsoft.com/..../dotnet/fsharp
- [wikipedia.org/..../F Sharp](https://wikipedia.org/..../F%20Sharp) (programming language)
- [wikipedia.org/..../List of programming languages by type#Functional languages](https://wikipedia.org/..../List%20of%20programming%20languages%20by%20type#Functional%20languages)
- [F# weekly](#)
- fsharpforfunandprofit.com
- tutorialspoint.com/fsharp
- rosettacode.org

Hausaufgabe

- ☐ [git Tutorial](#)
- ☐ [Tour durch F#](#)
 - [exercism.io](#) (E-Mail bis 05.03, mit *account!*)
 - ☐ Two-Fer
 - ☐ Leap
 - ☐ Isogram
 - ☐ Sum Of Multiples