# Vortrag F# - Funktionale Programmierung

# Anschreiben lassen:

int SumOfSquares(List<int> list){

}

# Sytaxeinführung in F# Interactive 🡺 Let-Binding

* White space sensitiv
  + Let a = 1
  + Let b = 1.
  + Let result = a + b
  + Let add x y = x + y
  + Nur explizite Casts!
* Tupel
* Listen
  + Let l = 1 :: [2; 3]
* Type inference
* Immutable at default
  + X = X +1 ? (Mathematisches Schließen unmöglich)
  + Vs. X <- X + 1
* **1. Drittel der Sprache geschafft**

Let sum1 list =

let mutable I = 0

for x in list do

I <- I + x

i

// Without mutable state

let rec sum2 list =

match list with

| x::rest -> x + sum2 rest // not tail recursive  
 | [] -> 0

## Nettes Pattern

/// reveres a list

let rec rev1 list =

match list with

| x::rest -> rev1 rest @ [x] // not tail-recursive

| [] -> []

/// counts the even-valued numbers

let rec countEven1 list =

match list with

| x::rest ->

if x % 2 = 0 then

1 + countEven1 rest // not tail-recursive

else

countEven1 rest

| [] -> 0

# Achtung! Pattern nicht gut!

Printfn „%d“ (sum2 [1..100]) // Gaus-aufgabe

printfn „%d“ (sum2 [1..100000]) // Stack overflow

/// calculates the sum of the list – tail-recursive

let rec sum3 acc list =

match list with

| x::rest -> sum3 (acc + x) rest

| [] -> acc

## Verallgemeinern!

/// “Folds a list”

let rec fold f acc l =

match l with

| x::rest -> fold f (f x acc) rest

| [] -> acc

// using fold

let add x y = x + y

let sum4 list = fold add 0 list

printfn „Sum4: %d“ (sum3 list1)

// Using anonymous lambda expression

let sum5 list = fold (fun a b -> a + b) 0 list

printfn “Sum5: %d” (sum5 list1)

let rev2 list = fold (fun a b -> b @ [a]) [] list

printfn “Rev2: %A” (rev2 list1)

// built-in

let sum6 l = List.fold (+) 0 l

let rev3 l = List.rev l

# Zusammenfassung fold:

* Erste Higher-order function gefunden
* = LINQ “Aggregate”
* Wichtig in Zusammenhang mit map
* Keyword **fun** kennengelernt 🡺 2 Drittel der Sprache geschafft

# |> + „partial application“ / Currying

* let add x y = x + y
* let addT (x,y) = x + y
* let add1 y = add 1 y
* Pipeline operator |>

let fac x = List.fold (\*) 1I [1I..x]

let sum =

[1I..3000I]

|> List.map fac

|> List.fold (+) 0I

printfn "Sum of Factorials: %A" sum

**TaskManager**

let sum =

[1I..3000I]

|> adapt

|> pMap fac

|> pFold (+) 0I

printfn "Sum of Factorials: %A" sum

### ^^ GOOGLE MAPREDUCE

# SAMPLE Open Closed

# Concurrency

# DSLs

FAKE

module Yahtzee.Specs

open NaturalSpec

open Model

let placed\_on category list =

printMethod category

calcValue category list

// Ones, Twos, Threes, Fours, Fives, Sixes:

// The player scores the sum of the dice that reads one, two, three, four, five or six, respectively.

// For example, 1, 1, 2, 4, 4 placed on "fours" gives 8 points.

[<Scenario>]

let ``Given 1, 1, 2, 4, 4 placed on "fours" gives 8 points.`` () =

Given (1, 1, 2, 4, 4)

|> When placed\_on Fours

|> It should equal 8

|> Verify

[<Scenario>]

let ``Given 1, 1, 6, 4, 6 placed on "sixes" gives 12 points.`` () =

Given (1, 1, 6, 4, 6)

|> When placed\_on Sixes

|> It should equal 12

|> Verify

// Pair: The player scores the sum of the two highest matching dice.

// For example, 3, 3, 3, 4, 4 placed on "pair" gives 8.

[<Scenario>]

let ``Given 3, 3, 3, 4, 4 placed on "pair" gives 8.`` () =

Given (3, 3, 3, 4, 4)

|> When placed\_on Pair

|> It should equal 8

|> Verify

[<Scenario>]

let ``Given 5, 3, 5, 4, 4 placed on "pair" gives 10.`` () =

Given (5, 3, 5, 4, 4)

|> When placed\_on Pair

|> It should equal 10

|> Verify

[<Scenario>]

let ``Given 5, 3, 2, 4, 1 placed on "pair" gives 0.`` () =

Given (5, 3, 2, 4, 1)

|> When placed\_on Pair

|> It should equal 0

|> Verify

// Two pairs: If there are two pairs of dice with the same number,

// the player scores the sum of these dice. If not, the player scores 0.

// For example, 1, 1, 2, 3, 3 placed on "two pairs" gives 8.

[<Scenario>]

let ``Given 1, 1, 2, 3, 3 placed on "two pair" gives 8.`` () =

Given (1, 1, 2, 3, 3)

|> When placed\_on TwoPair

|> It should equal 8

|> Verify

[<Scenario>]

let ``Given 1, 6, 6, 3, 3 placed on "two pair" gives 18.`` () =

Given (1, 6, 6, 3, 3)

|> When placed\_on TwoPair

|> It should equal 18

|> Verify

[<Scenario>]

let ``Given 1, 1, 2, 4, 3 placed on "two pair" gives 0.`` () =

Given (1, 1, 2, 4, 3)

|> When placed\_on TwoPair

|> It should equal 0

|> Verify

// Three of a kind: If there are three dice with the same number,

// the player scores the sum of these dice. Otherwise, the player scores 0.

// For example, 3, 3, 3, 4, 5 places on "three of a kind" gives 9.

[<Scenario>]

let ``Given 3, 3, 3, 4, 5 placed on "three of a kind" gives 9.`` () =

Given (3, 3, 3, 4, 5)

|> When placed\_on ThreeOfAKind

|> It should equal 9

|> Verify

[<Scenario>]

let ``Given 3, 4, 3, 4, 5 placed on "three of a kind" gives 0.`` () =

Given (3, 4, 3, 4, 5)

|> When placed\_on ThreeOfAKind

|> It should equal 0

|> Verify

// Four of a kind: If there are four dice with the same number,

// the player scores the sum of these dice. Otherwise, the player scores 0.

// For example, 2, 2, 2, 2, 5 places on "four of a kind" gives 8.

[<Scenario>]

let ``Given 2, 2, 2, 2, 5 placed on "four of a kind" gives 8.`` () =

Given (2, 2, 2, 2, 5)

|> When placed\_on FourOfAKind

|> It should equal 8

|> Verify

[<Scenario>]

let ``Given 2, 6, 2, 2, 5 placed on "four of a kind" gives 0.`` () =

Given (2, 6, 2, 2, 5)

|> When placed\_on FourOfAKind

|> It should equal 0

|> Verify

// Small straight: If the dice read 1,2,3,4,5, the player scores 15 (the sum of all the dice),

// otherwise 0.

[<Scenario>]

let ``Given 1,2,3,4,5 placed on "Small Straight" gives 15.`` () =

Given (1,2,3,4,5)

|> When placed\_on SmallStraight

|> It should equal 15

|> Verify

[<Scenario>]

let ``Given 1,2,5,4,3 placed on "Small Straight" gives 15.`` () =

Given (1,2,5,4,3)

|> When placed\_on SmallStraight

|> It should equal 15

|> Verify

[<Scenario>]

let ``Given 1,2,6,4,3 placed on "Small Straight" gives 0.`` () =

Given (1,2,6,4,3)

|> When placed\_on SmallStraight

|> It should equal 0

|> Verify

// Large straight: If the dice read 2,3,4,5,6, the player scores 20 (the sum of all the dice),

// otherwise 0.

[<Scenario>]

let ``Given 2,3,4,5,6 placed on "Large Straight" gives 20.`` () =

Given (2,3,4,5,6)

|> When placed\_on LargeStraight

|> It should equal 20

|> Verify

[<Scenario>]

let ``Given 6,2,5,4,3 placed on "Large Straight" gives 20.`` () =

Given (6,2,5,4,3)

|> When placed\_on LargeStraight

|> It should equal 20

|> Verify

[<Scenario>]

let ``Given 1,2,6,4,3 placed on "Large Straight" gives 0.`` () =

Given (1,2,6,4,3)

|> When placed\_on LargeStraight

|> It should equal 0

|> Verify

// Full house: If the dice are two of a kind and three of a kind,

// the player scores the sum of all the dice.

// For example, 1,1,2,2,2 placed on "full house" gives 8. 4,4,4,4,4 is not "full house".

[<Scenario>]

let ``Given 1,1,2,2,2 placed on "full house" gives 8.`` () =

Given (1,1,2,2,2)

|> When placed\_on FullHouse

|> It should equal 8

|> Verify

[<Scenario>]

let ``Given 4,4,4,4,4 placed on "full house" gives 0.`` () =

Given (4,4,4,4,4)

|> When placed\_on FullHouse

|> It should equal 0

|> Verify

[<Scenario>]

let ``Given 1,1,2,3,2 placed on "full house" gives 0.`` () =

Given (1,1,2,3,2)

|> When placed\_on FullHouse

|> It should equal 0

|> Verify

// Yahtzee: If all dice are the have the same number, the player scores 50 points, otherwise 0.

[<Example(1)>]

[<Example(2)>]

[<Example(3)>]

[<Example(4)>]

[<Example(5)>]

[<Example(6)>]

let ``Given n,n,n,n,n placed on "Yahtzee" gives 50.`` n =

Given (n,n,n,n,n)

|> When placed\_on Yahtzee

|> It should equal 50

|> Verify

[<Scenario>]

let ``Given 1,1,1,2,1 placed on "Yahtzee" gives 0.`` () =

Given (1,1,1,2,1)

|> When placed\_on Yahtzee

|> It should equal 0

|> Verify

// Chance: The player gets the sum of all dice, no matter what they read.

[<Scenario>]

let ``Given 1,1,1,2,1 placed on "Chance" gives 6.`` () =

Given (1,1,1,2,1)

|> When placed\_on Chance

|> It should equal 6

|> Verify

[<Scenario>]

let ``Given 1,6,1,2,1 placed on "Chance" gives 11.`` () =

Given (1,6,1,2,1)

|> When placed\_on Chance

|> It should equal 11

|> Verify

module Yahtzee.Model

type Roll = int \* int \* int \* int \* int

type Category =

| Ones

| Twos

| Threes

| Fours

| Fives

| Sixes

| Pair

| TwoPair

| ThreeOfAKind

| FourOfAKind

| SmallStraight

| LargeStraight

| FullHouse

| Yahtzee

| Chance

let toList (a,b,c,d,e) = [a;b;c;d;e]

let sumNumber number =

Seq.filter ((=) number)

>> Seq.sum

let sumAsTuple value list number =

let numberCount = list |> Seq.filter ((=) number) |> Seq.length

if numberCount >= value then value \* number else 0

let allNumbers = [1..6]

let allPairs =

[for i in allNumbers do

for j in allNumbers -> i,j]

let takeBest = Seq.max

let takeBestTuple value list =

allNumbers

|> Seq.map (sumAsTuple value list)

|> takeBest

let takeBestCombo value1 value2 list =

allPairs

|> Seq.filter (fun (a,b) -> a <> b)

|> Seq.map (fun (a,b) ->

let a' = sumAsTuple value1 list a

let b' = sumAsTuple value2 list b

if a' = 0 || b' = 0 then 0 else a' + b')

|> takeBest

let calcValue category roll =

let list = toList roll

match category with

| Ones -> sumNumber 1 list

| Twos -> sumNumber 2 list

| Threes -> sumNumber 3 list

| Fours -> sumNumber 4 list

| Fives -> sumNumber 5 list

| Sixes -> sumNumber 6 list

| Pair -> takeBestTuple 2 list

| TwoPair -> takeBestCombo 2 2 list

| ThreeOfAKind -> takeBestTuple 3 list

| FourOfAKind -> takeBestTuple 4 list

| SmallStraight ->

match list |> List.sort with

| [1;2;3;4;5] -> 15

| \_ -> 0

| LargeStraight ->

match list |> List.sort with

| [2;3;4;5;6] -> 20

| \_ -> 0

| FullHouse -> takeBestCombo 2 3 list

| Yahtzee ->

let a,b,c,d,e = roll

if a = b && a = c && a = d && a = e then 50 else 0

| Chance -> List.sum list