# Vortrag F# - Funktionale Programmierung

# Allgemeines

* Keine Folien
* Viel Code
* Zwischenfragen und Kommentare sind ausdrücklich erwünscht
* Keine reine F# Werbeveranstaltung
  + Allgemein funktionale Konzepte und Denkmuster
  + Anwendung teilweise in imperativen Sprachen möglich
* F# ist für .NET – kann im Prinzip alles was C# kann

# Anschreiben lassen:

int SumOfSquares(List<int> list){

}

# Was ist funktionale Programmierung?

„Functions are data“

Beweis?

Def: Eine Funktion *f* ordnet *jedem* [Element](http://de.wikipedia.org/wiki/Element_%28Mathematik%29) *x* einer [Definitionsmenge](http://de.wikipedia.org/wiki/Definitionsmenge) *D* *genau ein* Element *y* einer [Zielmenge](http://de.wikipedia.org/wiki/Zielmenge) *Z* zu.

f

D

Z

* Jede Funktion lässt sich über LookUp-Table ersetzen. (Wird z.B: für Performance getan)
* Jede Funktion hat einen Typ. SumOfSquares: List<int> -> int

Schreibweise:

f\colon\, D\to Z,\quad x\mapsto y.

# 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
* Printfn (Type safe – compiler trick)
* Immutable at default
  + X = X +1 ? (Mathematisches Schließen unmöglich)
  + Vs. X <- X + 1
* Loops
  + for I in 1..10 do
  + printfn “ I = %d” i

for I in [“A”; “b”; “c”] do

* + printfn „i = %s“ i
  + Schachteln
* **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 // no-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