Funktionale Programmierung

Funktionen und Typen III

Programm

Übersicht

Woche	Thema	Praktika
8	Organisatorisches, historisches, Begriff der funktionalen Programmierung, Einführung F#	1
9	Funktionen und Typen I: Werte, Unions (Listen), Produkte	2
10	Funktionen und Typen II: Options, Funktionstyp, “partial application”, Currying	2,3
11	Funktionen und Typen III: Kombinatoren und höhere Funktionen	3
12	Rekursion I: Formen der Rekursion	4
13	Rekursion II: Fixpunkte	4,5
14	Rekursion III: Rekursion und Compiler Optimierungen	5

Programm

Detail

- Höhere Funktionen
  - Höhere Funktionen als Abstraktion
  - Einführende Beispiele
  - Parser-Kombinatoren (als gemeinsames Tutorial)


#!/bin/
map
filter
fold

Höhere Funktionen

(Keine) Definition

Eine Funktion höherer Ordnung[1] (engl. higher-order function manchmal auch Kombinator[2] oder Funktional) ist eine Funktion, die Funktionen als Argumente erhält oder Funktionen als Ergebnis liefert.

[1] Einige Autoren verlangen, dass eine höhere Funktion mindestens eine Funktion als Argument nimmt und/oder zurückgibt.

[2] Der Term "Kombinator" wird in der Literatur nicht einheitlich benutzt.

Was fällt Ihnen auf?

```
let rec sum = function
    | [] -> 0
    | x::xs -> x + (sum xs)
let rec prod = function
    | [] -> 1
    | x::xs -> x * (prod xs)
let rec sumOfSquares = function
    | [] -> 0
    | x::xs -> x*x + (sumOfSquares xs)
let rec prodOfEvens = function
    | [] -> 1
    | x::xs when x%2=0 -> x * (prodOfEvens xs)
    | x::xs -> prodOfEvens xs
```

Anstatt oft denselben Code zu schreiben, schreiben wir den Code einmal "parametrisch" (als höhere Funktion) und wenden diesen oft an:

```
let rec fold folder state = function
    | [] -> state
    | x::xs -> fold folder (folder state x) xs

let sum' = fold (+) 0

let prod' = fold (*) 1

let sumOfSquares' = fold (fun x y -> x + y * y) 0

let prodOfEvens' = fold (fun x y -> ...) 1
```

Die Möglichkeit Funktionen höherer Ordnung zu deklarieren und daraus via Komposition und partieller Anwendung weitere Funktionen "zusammenzubauen", konstituiert ein mächtiges Mittel zur Abstraktion.

Weil Sie "theoretisch" höhere Funktionen bereits aus dem letzten Kapitel kennen[3], wollen wir im Folgenden einige Beispiele betrachten, die das genannte Abstraktionsverhalten aufzeigen.

[3] Weil diese nur ein Spezialfall von "normalen" Funktionen darstellen.

Als erstes einfaches Beispiel betrachten wir eine rudimentäre “logger” Funktion:

```
let logger f x =
    printfn "computing %A..." x
    f x
```

Die Funktion `logger` kann etwa wie folgt angewendet werden:

```
>let loggedMap f x = List.map (logger f) x
loggedMap (fun x -> x*x) [1..4];;

computing @ 1...
computing @ 2...
computing @ 3...
computing @ 4...
```

Die Funktion `iter` nimmt als Argumente eine Zahl `n` und eine Funktion `f`, die Rückgabe ist eine Funktion, die `f` genau `n`-mal auf ein gegebenes Argument anwendet:

```
let rec iter n f x =
    if n < 1 then x
    else iter (n-1) f (f x)
```

Wie wir den Iterator einsetzen können, sehen wir am nächsten Beispiel.

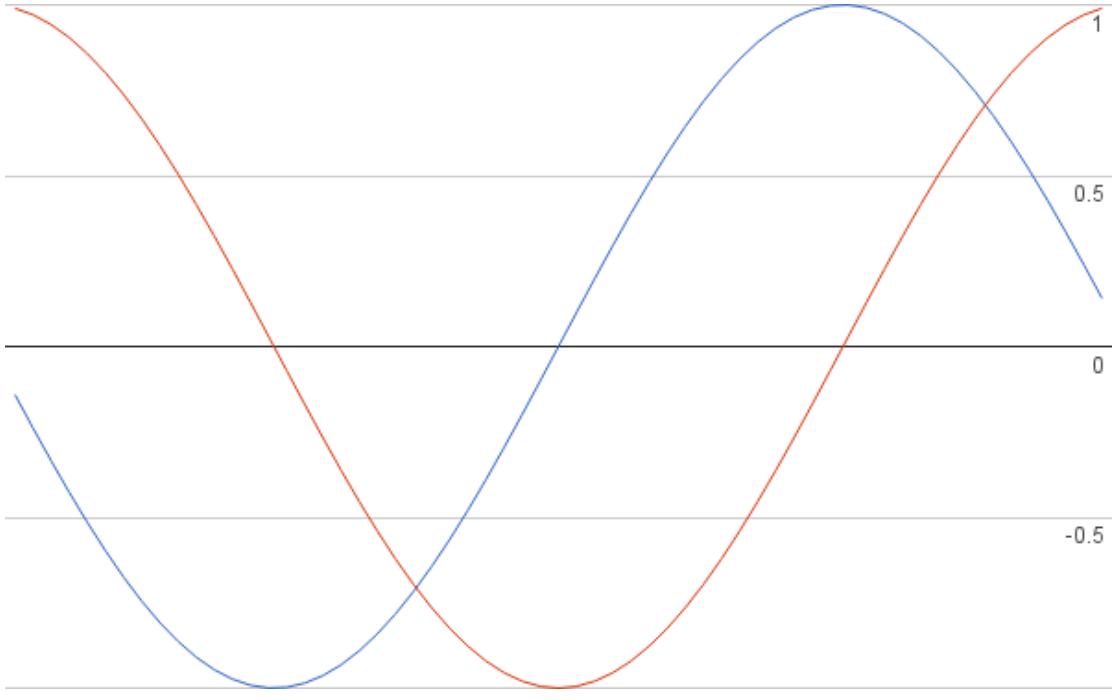
`D` akzeptiert als Argument eine Funktion `f: float -> float` und gibt die (numerische Approximation der) Ableitung von `f` als Rückgabe.

```
let D f x =
    let dx = 0.00000001 // kleine Zahl
    let dy = f (x+dx) - f x
    dy/dx
```

Höhere Funktionen

Differnezial-Operator

```
>plot [sin;D sin]
```



Zusammen mit der Funktion `iter` können wir nun auch den n-Fachen Ableitungsoperator implementieren:

```
let diff n = iter n D
```

Unter Memoisation versteht man Optimierungsverfahren, die Funktionsauswertung dadurch beschleunigen, dass Rückgabewerte zwischengespeichert anstatt neu berechnet werden. Die Funktion `memoize` stellt die einfachste Art eines "Memoisierungs-Kombinators" dar.

```
open System.Collections.Generic

let memoize f =
    let m = new Dictionary<'a,'b>()
    fun x ->
        match m.TryGetValue(x) with
        | (false,_) -> let y = f x in m.Add(x,y); y
        | (true, y) -> y
```

Die memoisierte Variante der Fibonacci Funktion zeigt das erwartete Laufzeitverhalten:

```
>#time
let mFibs = memoize fibs
mFibs 38;;
Real: 00:00:00.258, ...
val it : int = 39088169
>mFibs 38;;
Real: 00:00:00.000, ...
val it : int = 39088169
```

Andererseits könnte es besser sein:

```
let mFibs = memoize fibs
mFibs 38;;
Real: 00:00:00.258, ...
val it : int = 39088169
>mFibs 38;;
Real: 00:00:00.000, ...
val it : int = 39088169
> mFibs 37;;
Real: 00:00:00.152, ...
val it : int = 24157817
```

Wieso wurde `fib 37` nicht memoisiert, obwohl es beim Funktionsaufruf von `mFibs 38` zweifelsohne berechnet wurde? Wie können wir die Funktion `memoize` dazu bringen auch rekursive Aufrufe zu speichern?

Wir müssen dazu die Rekursion in der Fibonacci Funktion auflösen und in einer höheren Funktion "codieren":

```
let fibF fib x =
    if x < 2 then x
    else fib (x-1) + fib (x-2)
```

Derart "aufgelöst-rekursive" Funktionen können wir mit einem leicht modifizierten `memoize` behandeln:

```
let memoizeRec f =
  let m = new Dictionary<'a,'b>()
  let rec f' x =
    match m.TryGetValue(x) with
    |(false,_) -> let y = f f' x in m.Add(x,y); y
    |(true, y) -> y
  f'
```

```
>let mrFibs = memoizeRec fibF
mrFibs 38;;
Real: 00:00:00.000, ...
> mrFibs 37;;
Real: 00:00:00.000,
```

```
>let mrFibs = memoizeRec fibF
mrFibs 38;;
Real: 00:00:00.000, ...
> mrFibs 37;;
Real: 00:00:00.000,
```

Ok, das sieht zwar sehr gut aus, aber wieso wurde der Wert `fibRecM 30` scheinbar schon vor dem ersten Aufruf gecached?

Die memoisierte Funktion hat den Wert nicht gecached, sondern ist einfach grundsätzlich viel effizienter. Ansonsten verhält sie sich genau wie wir uns das gewünscht hätten: Alle Berechnungen die (auch rekursiv) bereits evaluiert wurden stehen bei späteren (auch impliziten rekursiven) Aufrufen zur Verfügung! 😎

```
>> mrFibs 10000;;
Real: 00:00:00.016,
> mrFibs 10100;;
Real: 00:00:00.000
```

Aufgabe

Passen Sie die `logger` Funktion so an, dass sie auch rekursive Aufrufe der übergebenen Funktion (als Funktional) “loggt”.