Funktionale Programmierung

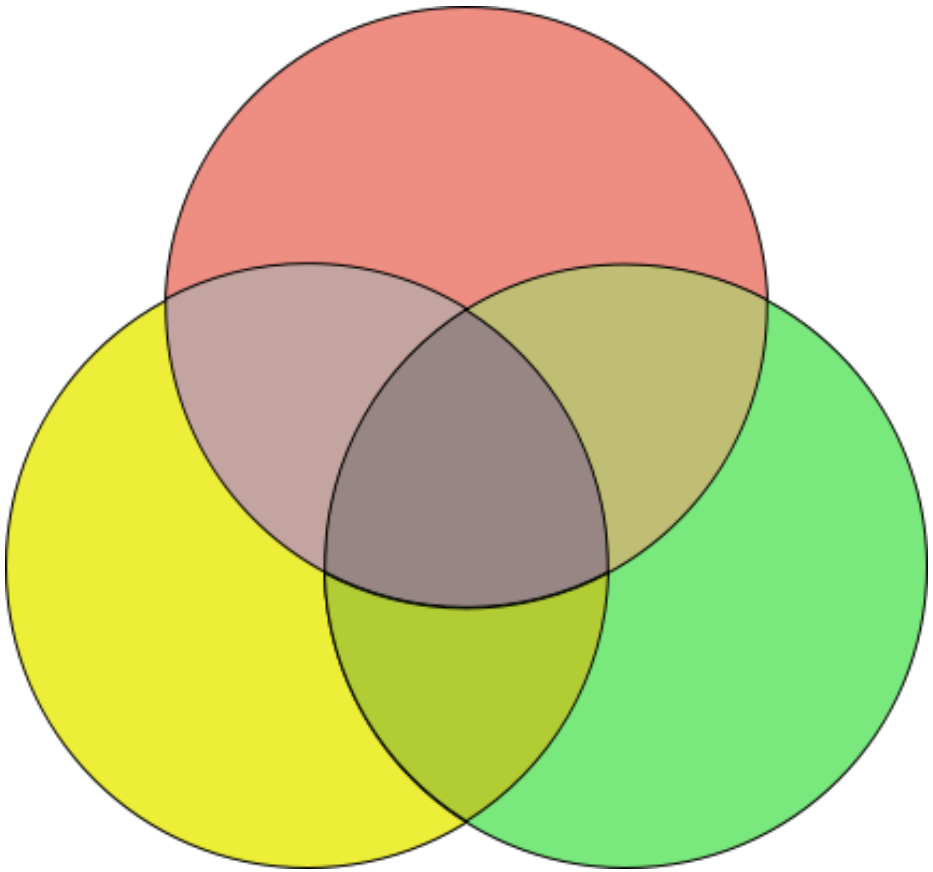
Funktionen und Typen I

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

- Grundlegendes
  - Was sind Typen und wie werden sie notiert
  - Werte und Variablen
  - Referenzielle Transparenz
- Algebraische Typen
  - Summen (Unions): Listen, rekursive Summen, Options
  - Produkte (Products): Tupel, Records



Darüber was genau ein Typ sei gibt es viele[1] Meinungen.

[1] Offenbar mindestens fünf

Darüber was genau ein Typ sei gibt es viele[1] Meinungen.

> *D. L. Parnas, J. E. Shore and David Weiss identified five definitions of a "type" that were used - sometimes implicitly - in the literature:. . .*
> *Wikipedia*

[1] Offenbar mindestens fünf

Einige Meinungen sind etwas ausführlicher:

Einige Meinungen sind etwas ausführlicher:

> *A type system is a tractable syntactic method of proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute.*
> *Benjamin Pierce, Types and Programming Languages*

Einige Meinungen sind etwas ausführlicher:

> *A type system is a tractable syntactic method of proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute.*
> *Benjamin Pierce, Types and Programming Languages*

...andere einfacher:

Einige Meinungen sind etwas ausführlicher:

> *A type system is a tractable syntactic method of proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute.*
> *Benjamin Pierce, Types and Programming Languages*

...andere einfacher:

> *Typen = Mengen.*
> *D. Flumini* 😎

Einige Meinungen sind etwas ausführlicher:

> *A type system is a tractable syntactic method of proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute.*
> *Benjamin Pierce, Types and Programming Languages*

...andere einfacher:

> *Typen $\simeq$ Mengen. Anstatt $x \in t$ schreibt man $x : t$.*

Die Objekte eines funktionalen Programmes, die keine Typen sind[2], entsprechen in der Mengenanalogie den Elementen und werden Werte (values) genannt.

[2] Je nach Sprache können auch weitere Konstrukte, z.B. höhere Typen ("kinds") und Funktoren, beschrieben werden.

In der funktionalen Programmierung haben “Variablen”, und damit Wertzuweisungen eine **fundamental** andere Bedeutung als in der imperativen Programmierung.

In der funktionalen Programmierung haben “Variablen”, und damit Wertzuweisungen eine **fundamental** andere Bedeutung als in der imperativen Programmierung.

Die Zuweisung `x = 3` im Vergleich:

In der funktionalen Programmierung haben "Variablen", und damit Wertzuweisungen eine **fundamental** andere Bedeutung als in der imperativen Programmierung.

Die Zuweisung `x = 3` im Vergleich:

- Funktional: Der "Name" `x` benennt (in seinem Kontext), unabhängig von der Zeit, den **Wert** 3.

In der funktionalen Programmierung haben "Variablen", und damit Wertzuweisungen eine **fundamental** andere Bedeutung als in der imperativen Programmierung.

Die Zuweisung `x = 3` im Vergleich:

- Funktional: Der "Name" `x` benennt (in seinem Kontext), unabhängig von der Zeit, den **Wert** 3.
- Imperativ: Der "Name" `x` benennt einen **Ort** (Speicherbereich). Sein Wert ändert sich mit der Zeit, je nachdem was in besagtem Speicherbereich steht.

Konsequenzen:

- Die Wert-Variable-Relation ist im funktionalen Paradigma zeitunabhängig.

Konsequenzen:

- Die Wert-Variable-Relation ist im funktionalen Paradigma zeitunabhängig.
- Variablen im funktionalen Paradigma entsprechen eher “Konstanten” als Variablen $\rightarrow$ Stichwort “immutability”.

Neben dem Verzicht Variablen zu verändern, wird in der funktionalen Programmierung darauf geachtet auch andere Nebeneffekte möglichst zu vermeiden oder mindestens zu isolieren.

Neben dem Verzicht Variablen zu verändern, wird in der funktionalen Programmierung darauf geachtet auch andere Nebeneffekte möglichst zu vermeiden oder mindestens zu isolieren.

"Interne" Nebeneffekte ändern den Zustand des Programms und kommen in rein funktionalen Sprachen nicht vor.

```
let f x = y <- x; x
```

Neben dem Verzicht Variablen zu verändern, wird in der funktionalen Programmierung darauf geachtet auch andere Nebeneffekte möglichst zu vermeiden oder mindestens zu isolieren.

"Interne" Nebeneffekte ändern den Zustand des Programms und kommen in rein funktionalen Sprachen nicht vor.

```
let f x = y <- x; x
```

"Externe" Nebeneffekte verändern den Zustand des Kontextes (Aussenwelt) in den das Programm eingebettet ist und werden mit verschiedenen Techniken vom Rest des Programmes isoliert.

```
let f x = launchMissile(); 0
```

Eigenschaften von Nebeneffektfreiem (reinem) funktionalen Code:

[3] Unter berücksichtigung ihres Kontextes!

Eigenschaften von Nebeneffektfreiem (reinem) funktionalen Code:

- Eine "Variable" kann in einem Ausdruck immer[3] durch ihren Wert ersetz werden. Der Wert (Bedeutung) des betreffenden Ausdrucks (Programmes) verändert sich dadurch nicht.

[3] Unter berücksichtigung ihres Kontextes!

Eigenschaften von Nebeneffektfreiem (reinem) funktionalen Code:

- Eine "Variable" kann in einem Ausdruck immer[3] durch ihren Wert ersetz werden. Der Wert (Bedeutung) des betreffenden Ausdrucks (Programmes) verändert sich dadurch nicht.
- Der Wert eines Ausdrucks hängt nur von den Werten seiner Teilausdrücke ab.

[3] Unter berücksichtigung ihres Kontextes!

Eigenschaften von Nebeneffektfreiem (reinem) funktionalen Code:

- Eine "Variable" kann in einem Ausdruck immer[3] durch ihren Wert ersetz werden. Der Wert (Bedeutung) des betreffenden Ausdrucks (Programmes) verändert sich dadurch nicht.
- Der Wert eines Ausdrucks hängt nur von den Werten seiner Teilausdrücke ab.
- Die Evaluation von Ausdrücken ist unabhängig von Reihenfolge, in der seine Teilausdrücke evaluiert werden (parallele Auswertung!).

[3] Unter berücksichtigung ihres Kontextes!

Eigenschaften von Nebeneffektfreiem (reinem) funktionalen Code:

- Eine "Variable" kann in einem Ausdruck immer[3] durch ihren Wert ersetz werden. Der Wert (Bedeutung) des betreffenden Ausdrucks (Programmes) verändert sich dadurch nicht.
- Der Wert eines Ausdrucks hängt nur von den Werten seiner Teilausdrücke ab.
- Die Evaluation von Ausdrücken ist unabhängig von Reihenfolge, in der seine Teilausdrücke evaluiert werden (parallele Auswertung!).

[3] Unter berücksichtigung ihres Kontextes!

Eigenschaften von Nebeneffektfreiem (reinem) funktionalen Code:

- Eine "Variable" kann in einem Ausdruck immer[3] durch ihren Wert ersetz werden. Der Wert (Bedeutung) des betreffenden Ausdrucks (Programmes) verändert sich dadurch nicht.
- Der Wert eines Ausdrucks hängt nur von den Werten seiner Teilausdrücke ab.
- Die Evaluation von Ausdrücken ist unabhängig von Reihenfolge, in der seine Teilausdrücke evaluiert werden (parallele Auswertung!).

Diese Eigenschaften werden unter dem Begriff der referenziellen Transparenz zusammengefasst.

[3] Unter berücksichtigung ihres Kontextes!

Imperativer Code ist (im allgemeinen) nicht referenziell transparent:

```
let mutable x = 5
for i in 1..10 do
    x <- x + 1
    printfn "%i" x
```

```
let mutable x = 5
for i in 1..10 do
    x <- 5 + 1
    printfn "%i" x
```

Vorteile, die sich aus referenzieller Transparenz ergeben:

Vorteile, die sich aus referenzieller Transparenz ergeben:

- Einfachere Programmverifikation (weniger und einfachere/explizitere Abhängigkeiten).

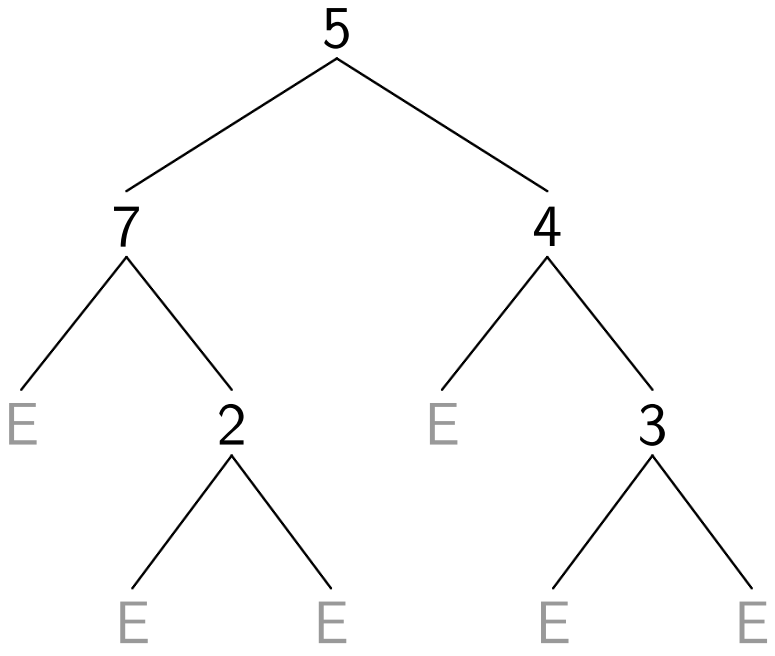
Vorteile, die sich aus referenzieller Transparenz ergeben:

- Einfachere Programmverifikation (weniger und einfachere/explizitere Abhängigkeiten).
- Erleichtert die Beweisführung, z.B. dass gewisse Optimierungen die Bedeutung eines transformierten Programmes nicht verändern.

Vorteile, die sich aus referenzieller Transparenz ergeben:

- Einfachere Programmverifikation (weniger und einfachere/explizitere Abhängigkeiten).
- Erleichtert die Beweisführung, z.B. dass gewisse Optimierungen die Bedeutung eines transformierten Programmes nicht verändern.
- Erleichtert es Programme zu verstehen und zu entwerfen (“equational reasoning”).

Algebraische Typen



In der Mengenanalogie entspricht der Summentyp der Vereinigung von disjunkten Mengen[4]. In F# wird die Vereinigung von Typen durch "**Discriminated-Unions**" implementiert:

```
type <Name> =
    | <Union case 1>
    | <Union case 2>
    | ...
```

[4] Ein wesentlicher Unterschied besteht darin, dass den Elementen des Summentyps die Zugehörigkeit zum entsprechenden "Summanden" explizit mitgegeben wird.

Beispiel einer Discriminated-Union:

```
type Shape =
    | Rectangle of float*float
    | Square of float
    | Circle of float
```

Summentypen können auch rekursiv sein:

```
type 'a BinaryTree =
    | E
    | Node of 'a BinaryTree * 'a * 'a BinaryTree
```

Mit Pattern-Matches lassen sich Discriminated-Unions dekonstruieren:

Fläche einer Form:

```
let area = function
    | Rectangle (l,h) -> l * h
    | Square l -> l * l
    | Circle r -> System.Math.PI * r * r
```

Tiefe eines Baumes:

```
let rec depth = function
    | E -> 0
    | Node (left,x,right)
        -> 1 + max (depth left) (depth right)
```

Aufgabe

Definieren Sie einen Typ `Complex`, der aus den beiden Union-Cases `Polar` und `Cart` besteht. `Cart` und `Polar` sollen für komplexe Zahlen in Polar-, respektive kartesischen Koordinaten stehen. Definieren Sie weiter eine Funktion `multPolar`, die Komplexe Zahlen (in beliebiger Darstellung) miteinander multipliziert und das Resultat in Polarkoordinaten zurückgibt.

```
type Complex =
   | Polar of float*float
   | Cart of float*float

let rec mulPolar x y =
   let toPolar = function
      | Cart (r,i)
         -> let d = Math.Sqrt (r*r + i*i)
            let a = Math.Atan (i/r)
            Polar (d,a)
      | arg -> arg
   match  x,y with
   | Polar (d,a), Polar (d',a')
      -> Polar (d * d', a + a')
   |  _ -> mulPolar (toPolar x) (toPolar y)
```

Listen[5] sind als Summentyp definiert:

```
type 'a List =
    | E
    | Cons of 'a * 'a List
```

F# stellt, so wie viele funktionale Sprachen, für Listen einiges an "syntactic sugar" und eine umfangreiche Bibliothek zur Verfügung.

[5] Einfach verkettete Listen

In F# können Listen direkt mit der Syntax

```
[Element1; Element2;...]
```

definiert werden. Weiter wird der `Cons` Operator/Konstruktor in F# durch `::` zur Verfügung gestellt.

```
1::2::3::[] = [1;2;3].
```

Listen können mit der "List-Comprehension" Syntax angegeben werden:

```
[for i in <Bereich> do yield <Ausdruck>].
```

Bereiche können auch verschachtelt werden:

```
[for i in ..  do for j in ..  do yield ..].
```

Einfache Bereiche können auch "angedeutet" werden:

```
[2...3..10] = [2;5;8].
```

Aufgabe

Studieren Sie das List Modul (eine Auswahl davon finden Sie **hier**). Nutzen Sie Funktionen aus dem Modul um die Funktionen der zweiten Aufgabe des ersten Praktikums zu definieren. Nutzen Sie Pattern-Matchings (mit dem Cons-Operator) um die Funktion

$$x \mapsto \text{“Liste aller Primzahlen bis } n\text{”}$$

zu implementieren.

Aufgabe

Studieren Sie...

```
let primes n =
  let rec sieve = function
   | [] -> []
   | x::xs
        -> let filter y = y % x <> 0
           x::(sieve (List.filter filter xs))
  sieve [2..n]
```

In der Mengenanalogie entspricht der Produkttyp dem Kartesischen Produkt von Mengen. Die Elemente eines Produkttyps sind demnach Tupel. In F# werden Tupel direkt oder durch "Records" implementiert. Records sind Tupel, in denen die Einträge Labels (Namen) tragen.

```
type <Name> = {<lb> : <Typ>; <lb>:<Typ2>;... }
```

Beispiel eines Records:

```
type Comp = {Re: float; Im: float}
```

Auf Elemente eines Records kann mit der "Punktschreibweise" zugegriffen werden:

```
>let x = {Re = 0.0; Im = 1.0}
 x.Im;;

val x : Comp = {Re = 0.0;
                Im = 1.0;}
val it : float = 1.0
```