

# Grundlagen der Programmierung 2 A (Listen)

## Haskell: Listen

Prof. Dr. Manfred Schmidt-Schauß

Sommersemester 2018

Listen modellieren Folgen von gleichartigen, gleichgetypten Objekten.

Ausdruck im Programm	Erklärung

Listen modellieren Folgen von gleichartigen, gleichgetypten Objekten.

Ausdruck im Programm	Erklärung
<code>[0,1,2,3,4,5,6,7,8]</code>	Typ: <code>[Integer]</code> ; d.h. Liste von Integer.

Listen modellieren Folgen von gleichartigen, gleichgetypten Objekten.

Ausdruck im Programm	Erklärung
<code>[0,1,2,3,4,5,6,7,8]</code>	Typ: <code>[Integer]</code> ; d.h. Liste von Integer.
<code>[]</code>	leere Liste, (Nil)

Listen modellieren Folgen von gleichartigen, gleichgetypten Objekten.

Ausdruck im Programm	Erklärung
<code>[0,1,2,3,4,5,6,7,8]</code>	Typ: <code>[Integer]</code> ; d.h. Liste von Integer.
<code>[]</code>	leere Liste, (Nil)
<code>['a', 'b', 'c']</code>	Typ: <code>[Char]</code> ;

Listen modellieren **Folgen** von **gleichartigen, gleichgetypten** Objekten.

Ausdruck im Programm	Erklärung
<code>[0,1,2,3,4,5,6,7,8]</code>	Typ: <code>[Integer]</code> ; d.h. Liste von Integer.
<code>[]</code>	leere Liste, (Nil)
<code>['a', 'b', 'c']</code>	Typ: <code>[Char]</code> ;
<code>[[], [0], [1,2]]</code>	Liste von Listen; Typ <code>[[Integer]]</code> , d.h. eine Liste von Listen von Integer-Objekten.

Listen modellieren **Folgen** von **gleichartigen, gleichgetypten** Objekten.

Ausdruck im Programm	Erklärung
<code>[0,1,2,3,4,5,6,7,8]</code>	Typ: <code>[Integer]</code> ; d.h. Liste von Integer.
<code>[]</code>	leere Liste, (Nil)
<code>['a', 'b', 'c']</code>	Typ: <code>[Char]</code> ;
<code>[[], [0], [1,2]]</code>	Liste von Listen; Typ <code>[[Integer]]</code> , d.h. eine Liste von Listen von Integer-Objekten.
<code>[1..]</code>	potentiell unendliche Liste der Zahlen 1,2,3,...; Typ: <code>[Integer]</code>

Vorteile der Listen im Vergleich mit z.B Arrays:

- Haskell-Listen sind seiteneffektfrei;  
Programmierung mit Arrays ist eher imperativ
- Potentiell unendliche Listen sind möglich
- Programmlogik mit Listen ist einfacher: Es gibt mehr korrekte Programmtransformationen



Vorteile der Listen im Vergleich mit z.B Arrays:

- Haskell-Listen sind seiteneffektfrei;  
Programmierung mit Arrays ist eher imperativ
- Potentiell unendliche Listen sind möglich
- Programmlogik mit Listen ist einfacher: Es gibt mehr korrekte Programmtransformationen
- **Weniger ist Mehr**

zwei Schreibweisen für Listen:

<code>[0,1,2]</code>	<code>(0 : (1 : (2 : [])))</code>
schöne Darstellung Druckbild einer Liste	interne Darstellung mit zweistelligem Infix-Listen-Konstruktor „:“ und dem Konstruktor <code>[]</code>

zwei Schreibweisen für Listen:

<code>[0,1,2]</code>	<code>(0 : (1 : (2 : [])))</code>
schöne Darstellung Druckbild einer Liste	interne Darstellung mit zweistelligem Infix-Listen-Konstruktor „:“ und dem Konstruktor <code>[]</code>

Eingebaute, listenerzeugende Funktionen:

Ausdruck im Programm	Erklärung
<code>[n..]</code>	erzeugt die Liste der Zahlen ab $n$ .
<code>[n..m]</code>	erzeugt die Liste von $n$ bis $m$
<code>[1..10]</code>	ergibt <code>[1,2,3,4,5,6,7,8,9,10]</code>
<code>[n,m..k]</code>	erzeugt die Liste von $n$ bis $k$ mit Schritten $m - n$

- Listendarstellung
- Listenerzeugung
- Listen-Druckbild

Listen als interne Datenstrukturen sind aufgebaut mittels zwei Konstruktoren:

- [] Konstante für die leere Liste
  - :
- Zweistelliger Infix-Konstruktor

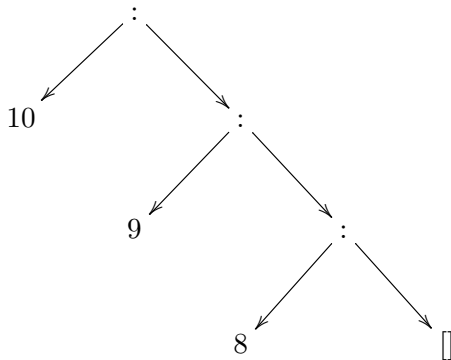
Listen als interne Datenstrukturen sind aufgebaut mittels zwei Konstruktoren:

`[]`      Konstante für die leere Liste  
`:`        Zweistelliger Infix-Konstruktor

`a : b`    Linkes Argument a:    erstes Element der Liste  
          Rechtes Argument b: Restliste

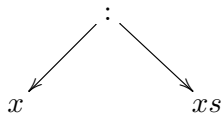
Beispiel für Haskell's Listenerzeugung:

<code>8 : []</code>	Liste <code>[8]</code> mit dem Element 8
<code>9 : (8 : [])</code>	Liste <code>[9,8]</code> mit zwei Elementen 9,8
<code>10 : (9 : (8 : []))</code>	Liste <code>[10,9,8]</code> mit drei Elementen

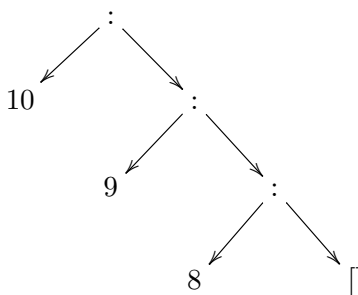


wird gedruckt als: `[10, 9, 8]`

Pattern

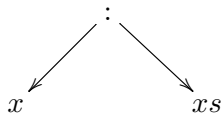


Liste

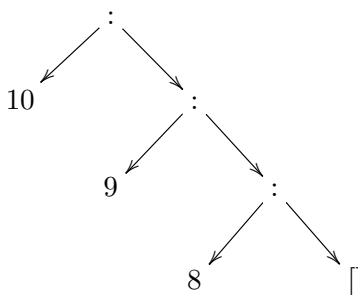




Pattern



Liste





Ergebnis des Pattern-Match:  $x \mapsto 10$ ,  $xs \mapsto [9, 8]$


## Definitionen


```
head (x:xs) = x    -- extrahiert das erste Element  
tail (x:xs) = xs   -- extrahiert die Restliste
```

## Auswertungen

```
Prelude> head []   
?????
```

```
Prelude> head [1]   
?????
```

```
Prelude> tail []   
?????
```

```
Prelude> tail [1]   
?????
```

```
length :: [a] -> Int
length []      = 0
length (x:xs) = 1 + (length xs)
```

**Auswertung** bei bereits ausgewerteter Liste  
length (10:(9:(8:[])))

```
length :: [a] -> Int
length []      = 0
length (x:xs) = 1 + (length xs)
```

**Auswertung** bei bereits ausgewerteter Liste  
`length (10:(9:(8:[])))`

```
length :: [a] -> Int
length []      = 0
length (x:xs) = 1 + (length xs)
```

**Auswertung** bei bereits ausgewerteter Liste

length (10:(9:(8:[])))      Zweiter Fall; [10/x, (9:(8:[]))/xs]  
1+ (length (9:(8:[])))

```
length :: [a] -> Int
length []      = 0
length (x:xs) = 1 + (length xs)
```

**Auswertung** bei bereits ausgewerteter Liste

length (10:(9:(8:[])))      Zweiter Fall; [10/x, (9:(8:[]))/xs]  
1+ (length (9:(8:[])))

```
length :: [a] -> Int
length []      = 0
length (x:xs) = 1 + (length xs)
```

**Auswertung** bei bereits ausgewerteter Liste

$\text{length } (10:(9:(8:[])))$	Zweiter Fall; $[10/x, (9:(8:[]))/xs]$
$1 + (\text{length } (9:(8:[])))$	Zweiter Fall; $[9/x, (8:[])/xs]$
$1 + (1 + (\text{length } (8:[])))$	

```
length :: [a] -> Int
length []      = 0
length (x:xs) = 1 + (length xs)
```

**Auswertung** bei bereits ausgewerteter Liste

<code>length (10:(9:(8:[])))</code>	Zweiter Fall; $[10/x, (9:(8:[]))/xs]$
<code>1+ (length (9:(8:[])))</code>	Zweiter Fall; $[9/x, (8:[])/xs]$
<code>1+(1+ (<b>length</b> (<b>8:[]</b>)))</code>	



```
length :: [a] -> Int
length []      = 0
length (x:xs) = 1 + (length xs)
```

**Auswertung** bei bereits ausgewerteter Liste

$\text{length } (10:(9:(8:[])))$	Zweiter Fall; $[10/x, (9:(8:[]))/xs]$
$1 + (\text{length } (9:(8:[])))$	Zweiter Fall; $[9/x, (8:[])/xs]$
$1 + (1 + (\text{length } (8:[])))$	Zweiter Fall; $[8/x, ([])/xs]$
$1 + (1 + (1 + (\text{length } [])))$	

```
length :: [a] -> Int
length []      = 0
length (x:xs) = 1 + (length xs)
```

**Auswertung** bei bereits ausgewerteter Liste

<code>length (10:(9:(8:[])))</code>	Zweiter Fall; <code>[10/x, (9:(8:[]))/xs]</code>
<code>1+ (length (9:(8:[])))</code>	Zweiter Fall; <code>[9/x, (8:[])/xs]</code>
<code>1+(1+ (length (8:[])))</code>	Zweiter Fall; <code>[8/x, ([])/xs]</code>
<code>1+(1+ (1+ (length [])))</code>	

```
length :: [a] -> Int
length []      = 0
length (x:xs) = 1 + (length xs)
```

**Auswertung** bei bereits ausgewerteter Liste

<code>length (10:(9:(8:[])))</code>	Zweiter Fall; <code>[10/x, (9:(8:[]))/xs]</code>
<code>1+ (length (9:(8:[])))</code>	Zweiter Fall; <code>[9/x, (8:[])/xs]</code>
<code>1+(1+ (length (8:[])))</code>	Zweiter Fall; <code>[8/x, ([])/xs]</code>
<code>1+(1+ (1+ (length [])))</code>	Erster Fall
<code>1+(1+ (1+ (0)))</code>	

```
length :: [a] -> Int
length []      = 0
length (x:xs) = 1 + (length xs)
```

**Auswertung** bei bereits ausgewerteter Liste

length (10:(9:(8:[])))	Zweiter Fall; [10/x, (9:(8:[]))/xs]
1+ (length (9:(8:[])))	Zweiter Fall; [9/x, (8:[])/xs]
1+(1+ (length (8:[])))	Zweiter Fall; [8/x, ([])/xs]
1+(1+ (1+ (length [])))	Erster Fall
1+(1+ (1+ (0)))	3 Additionen

```
length :: [a] -> Int
length []      = 0
length (x:xs) = 1 + (length xs)
```

**Auswertung** bei bereits ausgewerteter Liste

length (10:(9:(8:[])))	Zweiter Fall; [10/x, (9:(8:[]))/xs]
1+ (length (9:(8:[])))	Zweiter Fall; [9/x, (8:[])/xs]
1+(1+ (length (8:[])))	Zweiter Fall; [8/x, ([])/xs]
1+(1+ (1+ (length [])))	Erster Fall
1+(1+ (1+ (0)))	3 Additionen
3	

```
map :: (a -> b) -> [a] -> [b]
map f []                = []
map f (x:xs)            = (f x) : (map f xs)
```

map definiert durch eine Fallunterscheidung.

[] und (x:xs) links von „=" sind **Muster**(Pattern)

Z.B.   Muster                   (x:xs)  
      und Argument           (s:t)

```
map :: (a -> b) -> [a] -> [b]
map f []                = []
map f (x:xs)            = (f x) : (map f xs)
```

map definiert durch eine Fallunterscheidung.

[] und (x:xs) links von „=" sind **Muster**(Pattern)

Z.B. Muster (x:xs)  
und Argument (s:t)  
ergibt die Ersetzung: [s/x, t/xs]

```
map :: (a -> b) -> [a] -> [b]
map f []                = []
map f (x:xs)            = (f x) : (map f xs)
```

map definiert durch eine Fallunterscheidung.

[] und (x:xs) links von „=“ sind **Muster**(Pattern)

Z.B. Muster (x:xs)  
und Argument (s:t)  
ergibt die Ersetzung: [s/x, t/xs]

map **wendet** eine Funktion f auf alle Elemente einer Liste an  
und **konstruiert** die Liste der Ergebnisse.



```
map f []           = []  
map f (x:xs)       = (f x) : (map f xs)
```

Auswertung von `map quadrat (1:(2:[]))`:

Bei **vollständiger Auswertung der Ergebnisliste** durch den **Interpreter**:

```
map quadrat (1:(2:[]))
```

```
map f []           = []  
map f (x:xs)      = (f x) : (map f xs)
```

Auswertung von `map quadrat (1:(2:[]))`:

Bei **vollständiger Auswertung der Ergebnisliste** durch den **Interpreter**:

```
map quadrat (1:(2:[]))      [quadrat/f, 1/x, (2:[])/xs]  
(quadrat 1) : (map quadrat(2:[]))
```

```
map f []           = []  
map f (x:xs)      = (f x) : (map f xs)
```

Auswertung von `map quadrat (1:(2:[]))`:

Bei **vollständiger Auswertung der Ergebnisliste** durch den  
**Interpreter**:

```
map quadrat (1:(2:[]))           [quadrat/f, 1/x, (2:[])/xs]  
(quadrat 1) : (map quadrat(2:[]))
```

```
map f []           = []  
map f (x:xs)      = (f x) : (map f xs)
```

Auswertung von `map quadrat (1:(2:[]))`:

Bei **vollständiger Auswertung der Ergebnisliste** durch den **Interpreter**:

```
map quadrat (1:(2:[]))      [quadrat/f, 1/x, (2:[])/xs]  
(quadrat 1) : (map quadrat(2:[]))  bei vollst. Auswertung:  
1*1 : map quadrat (2:[])
```

```
map f []           = []  
map f (x:xs)      = (f x) : (map f xs)
```

Auswertung von `map quadrat (1:(2:[]))`:

Bei **vollständiger Auswertung der Ergebnisliste** durch den  
**Interpreter**:

```
map quadrat (1:(2:[]))      [quadrat/f, 1/x, (2:[])/xs]  
(quadrat 1) : (map quadrat(2:[]))  bei vollst. Auswertung:  
1*1 : map quadrat (2:[])
```

```
map f []           = []  
map f (x:xs)       = (f x) : (map f xs)
```

Auswertung von `map quadrat (1:(2:[]))`:

Bei **vollständiger Auswertung der Ergebnisliste** durch den **Interpreter**:

```
map quadrat (1:(2:[]))    [quadrat/f, 1/x, (2:[])/xs]  
(quadrat 1) : (map quadrat(2:[]))  bei vollst. Auswertung:  
1*1 : map quadrat (2:[])  
1 : map quadrat (2:[])
```

```
map f []           = []  
map f (x:xs)      = (f x) : (map f xs)
```

Auswertung von `map quadrat (1:(2:[]))`:

Bei **vollständiger Auswertung der Ergebnisliste** durch den **Interpreter**:

```
map quadrat (1:(2:[]))      [quadrat/f, 1/x, (2:[])/xs]  
(quadrat 1) : (map quadrat(2:[]))  bei vollst. Auswertung:  
1*1 : map quadrat (2:[])  
1 : map quadrat (2:[])
```

```
map f []           = []  
map f (x:xs)      = (f x) : (map f xs)
```

Auswertung von `map quadrat (1:(2:[]))`:

Bei **vollständiger Auswertung der Ergebnisliste** durch den **Interpreter**:

<code>map quadrat (1:(2:[]))</code>	<code>[quadrat/f, 1/x, (2:[])/xs]</code>
<code>(quadrat 1) : (map quadrat(2:[]))</code>	bei vollst. Auswertung:
<code>1*1 : map quadrat (2:[])</code>	
<code>1 : map quadrat (2:[])</code>	Zweite Gleichung
<code>1 : (quadrat 2 : map quadrat [])</code>	



```
map f []           = []  
map f (x:xs)       = (f x) : (map f xs)
```

Auswertung von `map quadrat (1:(2:[]))`:

Bei **vollständiger Auswertung der Ergebnisliste** durch den **Interpreter**:

<code>map quadrat (1:(2:[]))</code>	<code>[quadrat/f, 1/x, (2:[])/xs]</code>
<code>(quadrat 1) : (map quadrat(2:[]))</code>	bei vollst. Auswertung:
<code>1*1 : map quadrat (2:[])</code>	
<code>1 : map quadrat (2:[])</code>	Zweite Gleichung
<code>1 : (quadrat 2 : map quadrat [])</code>	

```
map f []           = []  
map f (x:xs)      = (f x) : (map f xs)
```

Auswertung von `map quadrat (1:(2:[]))`:

Bei **vollständiger Auswertung der Ergebnisliste** durch den **Interpreter**:

```
map quadrat (1:(2:[]))      [quadrat/f, 1/x, (2:[])/xs]  
(quadrat 1) : (map quadrat(2:[]))  bei vollst. Auswertung:  
1*1 : map quadrat (2:[])  
1 : map quadrat (2:[])  
1 : (quadrat 2 : map quadrat [])  
1 : (2*2 : map quadrat [])
```

Zweite Gleichung

```
map f []           = []  
map f (x:xs)      = (f x) : (map f xs)
```

Auswertung von `map quadrat (1:(2:[]))`:

Bei **vollständiger Auswertung der Ergebnisliste** durch den **Interpreter**:

<code>map quadrat (1:(2:[]))</code>	<code>[quadrat/f, 1/x, (2:[])/xs]</code>
<code>(quadrat 1) : (map quadrat(2:[]))</code>	bei vollst. Auswertung:
<code>1*1 : map quadrat (2:[])</code>	
<code>1 : map quadrat (2:[])</code>	Zweite Gleichung
<code>1 : (quadrat 2 : map quadrat [])</code>	
<code>1 : (2*2 : map quadrat [])</code>	

```
map f []           = []  
map f (x:xs)      = (f x) : (map f xs)
```

Auswertung von `map quadrat (1:(2:[]))`:

Bei **vollständiger Auswertung der Ergebnisliste** durch den **Interpreter**:

<code>map quadrat (1:(2:[]))</code>	<code>[quadrat/f, 1/x, (2:[])/xs]</code>
<code>(quadrat 1) : (map quadrat(2:[]))</code>	bei vollst. Auswertung:
<code>1*1 : map quadrat (2:[])</code>	
<code>1 : map quadrat (2:[])</code>	Zweite Gleichung
<code>1 : (quadrat 2 : map quadrat [])</code>	
<code>1 : (2*2 : map quadrat [])</code>	
<code>1 : (4 : <b>map quadrat []</b>)</code>	

```
map f []           = []  
map f (x:xs)      = (f x) : (map f xs)
```

Auswertung von `map quadrat (1:(2:[]))`:

Bei **vollständiger Auswertung der Ergebnisliste** durch den **Interpreter**:

<code>map quadrat (1:(2:[]))</code>	<code>[quadrat/f, 1/x, (2:[])/xs]</code>
<code>(quadrat 1) : (map quadrat(2:[]))</code>	bei vollst. Auswertung:
<code>1*1 : map quadrat (2:[])</code>	
<code>1 : map quadrat (2:[])</code>	Zweite Gleichung
<code>1 : (quadrat 2 : map quadrat [])</code>	
<code>1 : (2*2 : map quadrat [])</code>	
<code>1 : (4 : map quadrat [])</code>	Erste Gleichung
<code>1 : (4 : [])</code>	<code>= [1,4]</code>

```
istLeer []      = True
istLeer (x:xs) = False

zahlenAb n      = n: zahlenAb (n+1)
```

**Auswertung** (mit Listenerzeuger als Argument)

```
istLeer []      = True
istLeer (x:xs) = False

zahlenAb n      = n: zahlenAb (n+1)
```

**Auswertung** (mit Listenerzeuger als Argument)  
`istLeer [1..]`

```
istLeer []      = True
istLeer (x:xs) = False

zahlenAb n      = n: zahlenAb (n+1)
```

**Auswertung** (mit Listenerzeuger als Argument)

istLeer [1..]                      **verwende zahlenAb**



```
istLeer []      = True
istLeer (x:xs) = False

zahlenAb n      = n: zahlenAb (n+1)
```

**Auswertung** (mit Listenerzeuger als Argument)

```
istLeer [1..]           verwende zahlenAb
istLeer (zahlenAb 1)
```

```
istLeer []      = True
istLeer (x:xs) = False

zahlenAb n      = n: zahlenAb (n+1)
```

**Auswertung** (mit Listenerzeuger als Argument)

```
istLeer [1..]           verwende zahlenAb
istLeer (zahlenAb 1)
istLeer (1: zahlenAb (1+1))
```

```
istLeer []      = True
istLeer (x:xs) = False

zahlenAb n      = n: zahlenAb (n+1)
```

**Auswertung** (mit Listenerzeuger als Argument)



```
istLeer [1..]           verende zahlenAb
istLeer (zahlenAb 1)
istLeer (1: zahlenAb (1+1)) Zweite Gleichung von istLeer
```

```
istLeer []      = True
istLeer (x:xs) = False

zahlenAb n      = n: zahlenAb (n+1)
```


**Auswertung** (mit Listenerzeuger als Argument)

```
istLeer [1..]           verwerde zahlenAb
istLeer (zahlenAb 1)
istLeer (1: zahlenAb (1+1))  Zweite Gleichung von istLeer
False
```


```
*Main> map quadrat [1..10]   
[1,4,9,16,25,36,49,64,81,100]  
*Main> map quadrat [1..]   
[1,4,9,16,25,36,49,64,81,100,121, ....]
```

Der Listenerzeuger `[1..]` erzeugt soviel von der potentiell unendlichen Liste `[1,2,3,4,5,...]` wie für das Ergebnis benötigt wird.

```
mapQuadrat xs = map quadrat  xs
```

```
*Main> :t mapQuadrat   
mapQuadrat :: forall a. (Num a) => [a] -> [a]
```

```
mapLength xs = map length  xs
```


```
*Main> :t mapLength   
mapLength :: forall a. [[a]] -> [Int]
```

Die folgende Funktion hängt zwei Listen zusammen  
(genauer: sie konstruiert die Resultat-Liste)


```
append :: [a] -> [a] -> [a]
append [] ys      = ys
append (x:xs) ys  = x : (append xs ys)
```

Haskell-Operator für append: ++ (Infix-Operator)


Haskell-Schreibweise: [1,2,3] ++ [4,5,6,7]  
ergibt [1,2,3,4,5,6,7]

```
*Main> [] ++ [3,4,5] 
```

```
[3,4,5]
```

```
*Main> [0,1,2] ++ [] 
```

```
[0,1,2]
```

```
*Main> [0,1,2] ++ [3,4,5] 
```

```
[0,1,2,3,4,5]
```

```
*Main> [0..10000] ++ [10001..20000] == [0..20000]
```





```
True
```



## Filtern von Elementen aus einer Liste:



```
filter :: (a -> Bool) -> [a] -> [a]
filter f []                = []
filter f (x:xs)            = if (f x) then  x : filter f xs
                           else filter f xs
```

## Beispiele:

```
*Main> filter (< 5) [1..10] 
[1,2,3,4]
*Main> filter primzahlq [2..] 
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,
 67,71,73,79,83,89,97,101,103,107,109,113,127,131,137,
 139,149,151,157,163,167,173,179,181,191,193,197,199,211,
```

Die ersten  $n$  Elemente der Liste  $xs$ :

```
take :: Int -> [a] -> [a]
take 0 _      = []
take n []     = []
take n (x:xs) = x : (take (n-1) xs)
```

```
*Main> take 10 [20..40] 
[20,21,22,23,24,25,26,27,28,29]
*Main> take 10 [20,23..] 
[20,23,26,29,32,35,38,41,44,47]
```

Auswertung von  $f\ s_1 \dots s_n$   
wenn  $f$  mittels **Pattern (Muster)** definiert ist,  
innerhalb einer Fallunterscheidung:

Erster Schritt:  
die **Argumente** soweit auswerten,  
bis die Fallunterscheidung **möglich** ist.  
(von links nach rechts)



Zweiter Schritt:  
Definitionseinsetzung

`elimdub` eliminiert doppelte benachbarte Vorkommen von Elementen aus Listen:

```
elimdub []          = []  
elimdub [x]         = [x]  
elimdub (x:(y:r)) = if x == y then  elimdub (y:r)  
                  else x : elimdub (y:r)
```

Beachte das Pattern `(x:(y:r))`

`nub` im Modul `Data.List` eliminiert **alle** Doppelten:

```
elimdub [1,2,3,3,2,1]   
[1,2,3,2,1]  
nub [1,2,3,3,2,1]   
[1,2,3]
```

## Beispiel:

- Tupel: mehrere Ausdrücke werden zusammengefasst.
- $(1, 2, ['a', 'b'])$  oder  $(1 + 2, 3 * 4)$
- Es gibt Tupel mit 2,3,.. Komponenten.

## Beispiel:

- Tupel: mehrere Ausdrücke werden zusammengefasst.
- $(1, 2, [a', b'])$  oder  $(1 + 2, 3 * 4)$
- Es gibt Tupel mit 2,3,.. Komponenten.

## Verwendung in Funktionen:

- dritteKomponente  $(x,y,z) = z$
- addierePaar  $(x1,x2) (y1,y2) = (x1+y1,x2+y2)$

## Beispiel:

- Tupel: mehrere Ausdrücke werden zusammengefasst.
- $(1, 2, [a', b'])$  oder  $(1 + 2, 3 * 4)$
- Es gibt Tupel mit 2,3,.. Komponenten.

## Verwendung in Funktionen:

- dritteKomponente  $(x,y,z) = z$
- addierePaar  $(x1,x2) (y1,y2) = (x1+y1,x2+y2)$

## Typen:

- dritteKomponente::  $(a, b, c) \rightarrow c$
- addierePaar ::  $\text{Num } a, \text{Num } b \implies (a, b) \rightarrow (a, b) \rightarrow (a, b)$

Listen (bzw. Listenargumente) nennt man:  
**einfach ausgewertet:**

wenn Listen-Fallunterscheidung möglich ist,  
d.h. [] oder von der Form  $s : t$

**vollständig ausgewertet:**

wenn Liste endlich ist und  
der Tail, Tail-von Tail, ... alle usgewertet sind  
und alle Elemente ebenfalls vollständig ausgewertet sind,



Bei Verwendung von Listenargumenten:  
Die folgenden Begriffe sind **unverändert**:

- linear rekursiv,
- end-rekursiv (= tail-recursive)
- Baum-rekursiv
- geschachtelt Baum-rekursiv

(Bei applikativer Reihenfolge der Auswertung)

**iterativ** muss genauer erklärt werden.

Ein iterativer Auswertungsprozess liegt bei einer rekursiven Funktion  $f$ , vor wenn:

$$\begin{aligned} & (f \ a_1 \ \dots \ a_n) \\ \xrightarrow{*} & (f \ a'_1 \ \dots \ a'_n) \\ \xrightarrow{*} & (f \ a_1^{(2)} \ \dots \ a_n^{(2)}) \\ \xrightarrow{*} & (f \ a_1^{(3)} \ \dots \ a_n^{(3)}) \\ \xrightarrow{*} & \dots\dots\dots \\ \xrightarrow{*} & (f \ a_1^{(m)} \ \dots \ a_n^{(m)}) \quad \xrightarrow{*} \dots \end{aligned}$$

und alle  $a_i^{(j)}$  sind **Basiswerte** oder  
**vollständig ausgewertete, endliche Listen**  
(bei applikativer Reihenfolge der Auswertung.)

$f_{iter}$  ist **iterative Version** von  $f$

**Wenn:**  $f$  und  $f_{iter}$  das gleiche berechnen  
und  $f_{iter}$  einen iterativen Prozess erzeugt  
(unter applikativer R.)

für **alle** Basiswerte und  
**alle** komplett ausgewerteten endlichen Listen als Eingaben

```
length_lin xs          = length_linr 0 xs
length_linr s []       = s
length_linr s (x:xs) = length_linr (s+1) xs
```

nicht-iterative Version: (auch nicht endrekursiv)

```
length []      = 0
length (x:xs) = 1 + length xs
```

```
length (9:(8:(7:(6:... (1:[])))))
```

```
length (9:(8:(7:(6:... (1:[]))))  
1+(length (8:(7:(6:... (1:[]))))
```

```
length (9:(8:(7:(6:... (1:[]))))  
1+(length (8:(7:(6:... (1:[]))))  
1+(1+(length (7:(6:... (1:[]))))
```

```
length (9:(8:(7:(6:... (1:[]))))  
1+(length (8:(7:(6:... (1:[]))))  
1+(1+(length (7:(6:... (1:[]))))  
1+(1+(1+(length (6:... (1:[]))))
```



```
length (9:(8:(7:(6:... (1:[]))))  
1+(length (8:(7:(6:... (1:[]))))  
1+(1+(length (7:(6:... (1:[]))))  
1+(1+(1+(length (6:... (1:[]))))  
.....
```

```
length (9:(8:(7:(6:... (1:[]))))
1+(length (8:(7:(6:... (1:[]))))
1+(1+(length (7:(6:... (1:[]))))
1+(1+(1+(length (6:... (1:[]))))
.....
(1+(1+(1+(1+(1+(1+(1+(1+(1+0))))))))))
```

```
length (9:(8:(7:(6:... (1:[]))))  
1+(length (8:(7:(6:... (1:[]))))  
1+(1+(length (7:(6:... (1:[]))))  
1+(1+(1+(length (6:... (1:[]))))  
.....  
(1+(1+(1+(1+(1+(1+(1+(1+(1+0))))))))))  
.....
```

```
length (9:(8:(7:(6:... (1:[]))))  
1+(length (8:(7:(6:... (1:[]))))  
1+(1+(length (7:(6:... (1:[]))))  
1+(1+(1+(length (6:... (1:[]))))  
.....  
(1+(1+(1+(1+(1+(1+(1+(1+(1+0))))))))))  
.....  
9
```

Beachte: wir benutzen hier die applikative Reihenfolge der Auswertung

Beachte: wir benutzen hier die applikative Reihenfolge der Auswertung

```
length_lin (9:(8:(7:(6:... (1:[]))))))
```

Beachte: wir benutzen hier die applikative Reihenfolge der Auswertung

```
length_lin (9:(8:(7:(6:... (1:[])))))
```

```
length_linr 0 (9:(8:(7:(6:... (1:[])))))
```

Beachte: wir benutzen hier die applikative Reihenfolge der Auswertung

```
length_lin (9:(8:(7:(6:... (1:[]))))
```

```
length_linr 0 (9:(8:(7:(6:... (1:[]))))
```

```
length_linr 1 (8:(7:(6:... (1:[]))))
```



Beachte: wir benutzen hier die applikative Reihenfolge der Auswertung

```
length_lin (9:(8:(7:(6:... (1:[]))))
```

```
length_linr 0 (9:(8:(7:(6:... (1:[]))))
```

```
length_linr 1 (8:(7:(6:... (1:[]))))
```

```
length_linr 2 (7:(6:... (1:[])))
```

Beachte: wir benutzen hier die applikative Reihenfolge der Auswertung

```
length_lin (9:(8:(7:(6:... (1: [])))))  
length_linr 0 (9:(8:(7:(6:... (1: [])))))  
length_linr 1 (8:(7:(6:... (1: [])))))  
length_linr 2 (7:(6:... (1: [])))  
length_linr 3 (6:... (1: []))
```

Beachte: wir benutzen hier die applikative Reihenfolge der Auswertung

```
length_lin (9:(8:(7:(6:... (1: []))))))  
length_linr 0 (9:(8:(7:(6:... (1: []))))))  
length_linr 1 (8:(7:(6:... (1: []))))  
length_linr 2 (7:(6:... (1: [])))  
length_linr 3 (6:... (1: []))  
.....
```

Beachte: wir benutzen hier die applikative Reihenfolge der Auswertung

```
length_lin (9:(8:(7:(6:... (1: [])))))  
length_linr 0 (9:(8:(7:(6:... (1: [])))))  
length_linr 1 (8:(7:(6:... (1: [])))))  
length_linr 2 (7:(6:... (1: [])))  
length_linr 3 (6:... (1: []))  
.....  
length_linr 9 []
```

Beachte: wir benutzen hier die applikative Reihenfolge der Auswertung

```
length_lin (9:(8:(7:(6:... (1: [])))))  
length_linr 0 (9:(8:(7:(6:... (1: [])))))  
length_linr 1 (8:(7:(6:... (1: [])))))  
length_linr 2 (7:(6:... (1: [])))  
length_linr 3 (6:... (1: []))  
.....  
length_linr 9 []  
9
```

## Allgemeine Funktionen (Methoden):

**foldl** und **foldr** Links-Faltung und Rechts-Faltung

Die 3 Argumente sind:

- eine zweistellige Operation,
- ein Anfangselement (Einheitselement) und
- die Liste.

**foldl**  $\otimes e [a_1, \dots, a_n]$  entspricht  
 $((\dots ((e \otimes a_1) \otimes a_2) \dots) \otimes a_n).$

**foldr**  $\otimes e [a_1, \dots, a_n]$  entspricht  $a_1 \otimes (a_2 \otimes (\dots (a_n \otimes e)))$

foldl (Linksfaltung)

foldr (Rechtsfaltung)

```
foldl          :: (a -> b -> a) -> a -> [b] -> a
foldl f z []   = z
foldl f z (x:xs) = foldl f (f z x) xs
```

```
foldr          :: (a -> b -> b) -> b -> [a] -> b
foldr f z []   = z
foldr f z (x:xs) = f x (foldr f z xs)
```

Summe bzw. Produkt einer Liste von Zahlen:

```
sum xs      = foldl (+) 0 xs
produkt xs  = foldl (*) 1 xs
concat xs   = foldr (++) [] xs
```

```
foldl (+) 0 [1,2,3,4]      ≡ (((0+1)+2)+3)+4)
foldr (++) [] [[0],[2,3],[5]] ≡ [0] ++ ([2,3] ++ ([5] ++ []))
```

Je nach Operator ergibt `foldl`, oder `foldr` eine schnellere Verarbeitung.



## Lokale Funktionsdefinitionen, anonyme Funktionen

### Lambda-Ausdruck

$$\backslash x_1 \dots x_n \rightarrow \langle \text{Ausdruck} \rangle$$

$x_1, x_2, \dots$  sind die formalen Parameter

### Beispiel

$$\text{quadrat} = \backslash x \rightarrow x * x$$

Der Lambdaausdruck kann wie eine Funktion verwendet werden

$\text{let } \{x_1 = s_1; \dots; x_n = s_n\} \text{ in } t$

$\{x_1 = s_1; \dots; x_n = s_n\}$  ist eine lokale Umgebung  
die Variablen  $x_i$  können in  $t$  vorkommen  
mit der Bedeutung: „Wert von  $s_i$ “

$t$  der eigentliche Ausdruck

```
let x1 = 5
    x2 = "abc"
    x3 = 7*x1
in (x1,x2,x3)
```

In Haskell: *rekursives let*.

D.h.  $x_i$  kann in jedem  $s_j$  vorkommen in  
 $\{x_1 = s_1; \dots; x_n = s_n\}$

**Beachte** im ghci-Interpreter: Spezielle Verwendung des `let`

Funktionen sind definierbar direkt in einem rekursiven let:

$$\text{let } \{f \ x_1 \ \dots \ x_n = s; \dots\} \text{ in } t$$

Zum Beispiel:

```
let {hochdrei x = x*x*x; a = 3} in hochdrei a
```

Neue Sichtweise:  
Jetzt betrachten wir Programme als  
statischen,  
strukturierten Text

## Statische Analysen:

Untersuche den **Programmtext** bzw. den **Syntaxbaum**.

Um Definitionen von lokalen Variablen (Namen) korrekt zu handhaben, braucht man neue Begriffe:

<b>Gültigkeitsbereich</b> einer Variablen $x$	Text-Fragment(e) des Programms in dem dieses $x$ gemeint ist.
<b>freie Variablen</b> eines Ausdrucks	Variablen, deren Bedeutung außerhalb des Ausdrucks festgelegt wird.
<b>gebundene Variablen</b> eines Ausdrucks	Variablen, deren Bedeutung innerhalb des Ausdrucks festgelegt wird.

**Problem:** Variablen können mit gleichem Namen, aber verschiedener Bedeutung (bzw. Verwendung / Intention) in einem Ausdruck vorkommen:

**Problem:** Variablen können mit gleichem Namen, aber verschiedener Bedeutung (bzw. Verwendung / Intention) in einem Ausdruck vorkommen:

- Lösung:
- Exakte Festlegung der Gültigkeitsbereiche für jedes syntaktische Konstrukt
  - Umbenennen von gebundenen Variablennamen, falls nötig



$\backslash x \rightarrow x * x$

Gültigkeitsbereich von  $x$ : der Ausdruck  $x * x$   
die Variable  $x$  ist gebunden von  $\backslash x$

$\backslash x \rightarrow x * x$

Gültigkeitsbereich von  $x$ : der Ausdruck  $x * x$   
die Variable  $x$  ist gebunden von  $\backslash x$

$x * x$

in diesem Ausdruck ist  $x$  frei

$\backslash x \rightarrow x * x$

Gültigkeitsbereich von  $x$ : der Ausdruck  $x * x$   
die Variable  $x$  ist gebunden von  $\backslash x$

$x * x$

in diesem Ausdruck ist  $x$  frei

$(\text{let } x = 1; y = 2$   
     $\text{in } x * y * z)$

$x$  und  $y$  sind gebunden,  
 $z$  ist frei

FV: ergibt Menge von Variablen-Namen.

- $FV(x) := \{x\}$  , wenn  $x$  ein Variablenname ist
- $FV((s\ t)) := FV(s) \cup FV(t)$
- $FV(\text{if } t_1 \text{ then } t_2 \text{ else } t_3) := FV(t_1) \cup FV(t_2) \cup FV(t_3)$
- $FV(\backslash x_1 \dots x_n \rightarrow t) := FV(t) \setminus \{x_1, \dots, x_n\}$
- $FV(\text{let } x_1 = s_1, \dots, x_n = s_n \text{ in } t)$   
 $:= (FV(t) \cup FV(s_1) \cup \dots \cup FV(s_n)) \setminus \{x_1, \dots, x_n\}$
- $FV(\text{let } f\ x_1 \dots x_n = s \text{ in } t)$   
 $:= FV(\text{let } f = \backslash x_1 \dots x_n \rightarrow s \text{ in } t)$

Beachte: FV ist eine Funktion auf dem Syntaxbaum;

$$\begin{aligned} FV(\backslash x \rightarrow (f \ x \ y)) &= FV(f \ x \ y) \setminus \{x\} \\ &= \dots \\ &= \{x, f, y\} \setminus \{x\} \\ &= \{f, y\} \end{aligned}$$

Entsprechend der  $FV$ -Definition:

- $GV(x) := \emptyset$
- $GV((s\ t)) := GV(s) \cup GV(t)$
- $GV(\text{if } t_1 \text{ then } t_2 \text{ else } t_3) := GV(t_1) \cup GV(t_2) \cup GV(t_3)$
- $GV(\backslash x_1 \dots x_n \rightarrow t) := GV(t) \cup \{x_1, \dots, x_n\}$
- $GV(\text{let } x_1 = s_1, \dots, x_n = s_n \text{ in } t)$   
 $\quad := (GV(t) \cup GV(s_1) \cup \dots \cup GV(s_n) \cup \{x_1, \dots, x_n\})$
- $GV(\text{let } f\ x_1 \dots x_n = s \text{ in } t)$   
 $\quad := GV(\text{let } f = \backslash x_1 \dots x_n \rightarrow s \text{ in } t)$   
 $\quad = \{f, x_1, \dots, x_n\} \cup GV(s) \cup GV(t)$

Auch hier **GV** ist eine Funktion auf dem Syntaxbaum;

$$\begin{aligned}GV(\backslash x \rightarrow (f\ x\ y)) &= GV(f\ x\ y) \cup \{x\} \\&= \dots \\&= \emptyset \cup \{x\} \\&= \{x\}\end{aligned}$$

$\text{let } x = s \text{ in } t$  die Vorkommen der freien Variablen  $x$  in  $s, t$  werden gebunden.

$s, t$  ist der lexikalische Gültigkeitsbereich der Variablen  $x$

$f \ x \ y \ z = t$  die freien Variablen  $x, y, z$  in  $t$  werden gebunden.

$t$  ist der lexikalische Gültigkeitsbereich von  $x, y, z$ ;

auch tlw. von  $f$ .

$\backslash x \ y \ z \rightarrow t$   $t$  ist der lexikalische Gültigkeitsbereich von  $x, y, z$ .



Ausdruck  $t = \lambda x \rightarrow (x (\lambda x \rightarrow x * x))$

$x$  ist in  $t$  gebunden, aber in **zwei Bindungsbereichen**:

$\lambda x \rightarrow (x (\lambda x \rightarrow x * x))$

In  $(x (\lambda x \rightarrow x * x))$  kommt  $x$  **frei** und **gebunden** vor.

Umbenennen des gebundenen  $x$  in  $y$  ergibt:

$(x (\lambda y \rightarrow y * y))$

**Zwei** Bindungsbereiche für  $x$  in einem `let`-Ausdruck:

```
let  $x = 10$  in (let  $x = 100$  in ( $x+x$ )) +  $x$ 
```

Umbenennung ergibt:

```
let  $x1 = 10$  in (let  $x2 = 100$  in ( $x2+x2$ )) +  $x1$ 
```

Dieser Term wertet zu 210 aus.

Beispiel:

```
let  $x = (x*x)$  in ( $x+x$ )
```

Diese rekursiven Bindungen sind erlaubt.

**Zwei** Bindungsbereiche für  $x$  in einem `let`-Ausdruck:

```
let x = 10 in (let x = 100 in (x+x)) + x
```

Umbenennung ergibt:

```
let x1 = 10 in (let x2 = 100 in (x2+x2)) + x1
```

Dieser Term wertet zu 210 aus.

**Beispiel:**

```
let x = (x*x) in (x+x)
```

Diese rekursiven Bindungen sind erlaubt.

Aber führt zu Nichtterminierung des Haskell-Interpreters  
ohne Reduktionen auszuführen.

```
let    y = 20*z  
      x = 10+y  
      z = 15  
      in x
```

Wertet aus zu : 310.

```
let  {x = 1; y = 7}  
in (let  {y = 2; z = 4}  
    in  (let z = 5  
        in (x+y+z)))
```

x = 1; y = 7

y = 2; z = 4

z = 5

x+y+z

Vermeidung **redundanter Auswertungen** mit let

$$f(x,y) := x(1 + xy)^2 + y(1-y) + (1+xy)(1-y)$$

optimierbar durch Vermeidung von Doppelauswertungen:

Der zugehörige Ausdruck ist:

```
let a    = 1 + x*y
    b    = 1 - y
in  x*a*a + y*b + a*b
```

# Zusammengesetzte Daten

## Datenobjekte

## Tupel

## Verallgemeinerungen von Listen

Für Datentypen benötigt man:

Datenkonstruktor(en)

Datenselektor(en)

Beispiel

Paarkonstruktor  $s, t \longrightarrow (s, t)$

Paarselektoren  $\text{fst}, \text{snd}$

Eigenschaften:

$\text{fst}(s, t) = s$       und

$\text{snd}(s, t) = t.$



$n$ -Tupelkonstruktor  $t_1, \dots, t_n \longrightarrow (t_1, \dots, t_n)$

Tupelselektoren  $n$  Selektoren: pro Stelle des Konstruktors  
ein Selektor

$n$ -Tupel haben einen impliziten Konstruktor:

$(\underbrace{., \dots, .}_n)$

Muster (pattern) statt Selektoren.

Muster sind syntaktisch dort erlaubt, wo formale Parameter (Variablen) neu eingeführt werden:

- in Funktionsdefinitionen,
- in Lambda-Ausdrücken und
- in let-Ausdrücken.

Beispiel-Definitionen von Selektoren mittels Muster

```
fst (x,y) = x
snd (x,y) = y
selektiere_erstes_von_3 (x1,x2,x3) = x1
selektiere_zweites_von_3 (x1,x2,x3) = x2
selektiere_drittes_von_3 (x1,x2,x3) = x3
```

$(1, 1) \quad :: \quad (\text{Integer}, \text{Integer})$

$(1, (2, \text{True})) \quad :: \quad (\text{Integer}, (\text{Integer}, \text{Bool}))$

$(\underbrace{., \dots, .}_n) \quad :: \quad a_1 \rightarrow a_2 \rightarrow \dots \rightarrow a_n \rightarrow (a_1, a_2, \dots, a_n)$

`selektiere_drittes_von_3`  $:: (a_1, a_2, a_3) \rightarrow a_3$

**Benutzerdefinierte** Konstruktoren sind  
definierbar in Haskell mittels data-Anweisung

## Beispiel

```
data Punkt    = Punktkonstruktor Double Double
data Strecke = Streckenkonstruktor Punkt Punkt
```

Punkt, Strecke:	Typen
Punktkonstruktor Streckenkonstruktor	Konstruktoren
Double, Punkt (rechts)	Typen der Argumente

## Beispiel

```
data Punkt    = Punktkonstruktor    Double Double
               deriving (Show,Eq)

data Strecke = Streckenkonstruktor Punkt Punkt
               deriving (Show,Eq)
```

Grund für die Ergänzung

deriving (Show,Eq):

man kann dann Objekte drucken und mit == vergleichen.

streckenAnfang (**Streckenkonstruktor** x y) = x

- Nutzen der Muster:
- tiefes Selektieren
  - Ersatz für Selektoren
- Man braucht keine Funktionsnamen

Syntax der Muster:

$$\begin{aligned} \langle \text{Muster} \rangle &::= \langle \text{Variable} \rangle \mid (\langle \text{Muster} \rangle) \\ &\mid \langle \text{Konstruktor}_{(n)} \rangle \underbrace{\langle \text{Muster} \rangle \dots \langle \text{Muster} \rangle}_n \\ &\mid (\langle \text{Muster} \rangle, \dots, \langle \text{Muster} \rangle) \end{aligned}$$

**Bedingung:** in einem Muster darf keine Variable doppelt vorkommen

Anpassen des Objekts an das Muster  
gleichzeitige Selektion mittels impliziter let-Bindungen  
Tlw. vorher Auswertung des Objekts erforderlich

## Beispiele

Anpassen des Objekts an das Muster  
gleichzeitige Selektion mittels impliziter let-Bindungen  
Tlw. vorher Auswertung des Objekts erforderlich

## Beispiele

$(x, y, (u, v))$  anpassen an:  $(1, 2, (3, 4))$

ergibt: `let x = 1; y = 2; u = 3; v = 4 in ...`



Anpassen des Objekts an das Muster  
gleichzeitige Selektion mittels impliziter let-Bindungen  
Tlw. vorher Auswertung des Objekts erforderlich

## Beispiele

$(x,y,(u,v))$  anpassen an:  $(1,2,(3,4))$

**ergibt:** `let x = 1;y = 2;u = 3;v = 4 in ...`

$(x,y,(u,v))$  anpassen an:  $(1,2, True)$

**ergibt:** Fehler. Kann nicht vorkommen wegen Typcheck.

Anpassen des Objekts an das Muster  
gleichzeitige Selektion mittels impliziter let-Bindungen  
Tlw. vorher Auswertung des Objekts erforderlich

## Beispiele

`(x,y,(u,v))` anpassen an: `(1,2,(3,4))`

**ergibt:** `let x = 1;y = 2;u = 3;v = 4 in ...`

`(x,y,(u,v))` anpassen an: `(1,2,True)`

**ergibt:** Fehler. Kann nicht vorkommen wegen Typcheck.

`(x,y,u)` anpassen an: `(1,2,(4,5))`

**ergibt:** `let x = 1; y = 2;u = (4,5) in ...`

Anpassen des Objekts an das Muster  
gleichzeitige Selektion mittels impliziter let-Bindungen  
Tlw. vorher Auswertung des Objekts erforderlich

## Beispiele

$(x,y,(u,v))$  anpassen an:  $(1,2,(3,4))$

**ergibt:** `let x = 1; y = 2; u = 3; v = 4 in ...`

$(x,y,(u,v))$  anpassen an:  $(1,2, True)$

**ergibt:** Fehler. Kann nicht vorkommen wegen Typcheck.

$(x,y,u)$  anpassen an:  $(1,2,(4,5))$

**ergibt:** `let x = 1; y = 2; u = (4,5) in ...`

$(x,y,u)$  anpassen an:  $(1,2,4+5)$

**ergibt:** `let x = 1; y = 2; u = 4+5 in ...`

**Beispiel** Punkt, Strecke, Polygonzug

```
data Punkt a      = Punkt a a
data Strecke a    = Strecke (Punkt a) (Punkt a)
data Vektor a     = Vektor a a
data Polygon a    = Polygon [Punkt a]
```

Typ und Konstruktor können gleiche Namen haben.

Der Parameter  $a$  kann jeder Typ sein: z.B.:

Float,

Int,

aber auch  $[(Int, Char)]$

```
addiereVektoren::Num a => Vektor a -> Vektor a ->Vektor a
addiereVektoren (Vektor a1 a2) (Vektor b1 b2) =
    Vektor (a1 + b1) (a2 + b2)
skalarProdukt    (Vektor a1 a2) (Vektor b1 b2) =
    a1*b1 + a2*b2
streckenLaenge (Strecke (Punkt a1 a2) (Punkt b1 b2)) =
    sqrt (fromInteger ( (quadrat (a1-b1))
                        + (quadrat (a2-b2))))
```

**Summentypen:** diese haben mehr als einen Konstruktor

**Beispiele:** Bool mit True False

```
data Bool = True | False
```

**Aufzählungstyp:**

```
data Farben = Rot | Gruen | Blau | Weiss | Schwarz
```

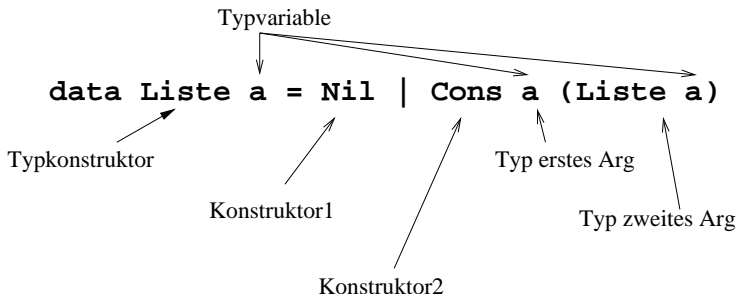
```
data Kontostand = Euro Integer | Dollar Integer  
                | Renminbi Integer | SFranken Integer
```

```
data Maybe a = Just a | Nothing
```

```
Just 1 : Nothing : []  
[Just 1,Nothing]
```



selbstdefinierte Listen: (sogar rekursiv definierter Typ)



Listen-Definition in Haskell:

```
data [a] = [] | a : [a]
```



## Syntax:

```
case <Ausdruck> of    {< Muster> -> <Ausdruck>;  
                      ...;  
                      < Muster> -> <Ausdruck>}
```

**Einschränkung:** nur einfache Muster:  $K x_1 \dots x_n$

**Kontextbedingung:** die Muster müssen vom Typ her passen.

## Beispiel:

```
und x y = case x of True -> y; False -> False
```

$$FV(\text{case } x \text{ of True } \rightarrow y; \text{ False } \rightarrow \text{False}) = \{x, y\}$$

$$FV(\text{case } x \text{ of (Punkt } u \text{ v) } \rightarrow u) = \{x\}$$

$$GV(\text{case } x \text{ of (Punkt } u \text{ v) } \rightarrow u) = \{u, v\}$$

(zusätzliche Auswertungsregel)

case-Reduktion

$$\frac{(\text{case } (c \ t_1 \dots t_n) \text{ of } \dots (c \ x_1 \dots x_n \rightarrow s) \dots)}{s[t_1/x_1, \dots, t_n/x_n]}$$

```
map f []           = []  
map f (x:xs)      = f x : map f xs
```

kann man auch so programmieren:

```
map f lst =  
  (case lst of [] -> []; (x:xs) -> f x : map f xs)
```

- Normale und verzögerte Reihenfolge der Auswertung werten **nicht** die Argumentausdrücke von Konstruktoren aus;  
erst wenn explizit angefordert (durch Pattern z.B.  
 $\Rightarrow$  Dadurch kann man (potentiell) unendliche Listen verarbeiten

## Beispiele:

<code>(1+2) : []</code>	wird nicht ausgewertet: ist ein Wert
<code>1:repeat 1</code>	wird nicht ausgewertet: ist ein Wert

**Tests:**

<code>seq (1:repeat 1) 7</code>
<code>seq (bot:repeat 1) 8</code>