

Typ-basiertes Programmieren und Schließen in Funktionalen Sprachen

Jun.-Prof. Janis Voigtländer

Institut für Informatik III

Universität Bonn

WS 2009/10

Organisation

- Wahlpflichtvorlesung im Diplomstudiengang, 4 LP

Organisation

- Wahlpflichtvorlesung im Diplomstudiengang, 4 LP
- Voraussetzung: z.B. Vorlesung „Deskriptive Programmierung“

Organisation

- Wahlpflichtvorlesung im Diplomstudiengang, 4 LP
- Voraussetzung: z.B. Vorlesung „Deskriptive Programmierung“
- Prüfung schriftlich oder mündlich

Organisation

- Wahlpflichtvorlesung im Diplomstudiengang, 4 LP
- Voraussetzung: z.B. Vorlesung „Deskriptive Programmierung“
- Prüfung schriftlich oder mündlich
- Vorlesung donnerstags, 13:15–14:45, in Raum A207:

Organisation

- Wahlpflichtvorlesung im Diplomstudiengang, 4 LP
- Voraussetzung: z.B. Vorlesung „Deskriptive Programmierung“
- Prüfung schriftlich oder mündlich
- Vorlesung donnerstags, 13:15–14:45, in Raum A207:
 - ▶ 13 oder 14 Termine, davon 9 vor Weihnachten

Organisation

- Wahlpflichtvorlesung im Diplomstudiengang, 4 LP
- Voraussetzung: z.B. Vorlesung „Deskriptive Programmierung“
- Prüfung schriftlich oder mündlich
- Vorlesung donnerstags, 13:15–14:45, in Raum A207:
 - ▶ 13 oder 14 Termine, davon 9 vor Weihnachten
- Übung: vierzehntägig,
Vorschlag: dienstags, 9:15–10:45, in Raum A121

Organisation

- Wahlpflichtvorlesung im Diplomstudiengang, 4 LP
- Voraussetzung: z.B. Vorlesung „Deskriptive Programmierung“
- Prüfung schriftlich oder mündlich
- Vorlesung donnerstags, 13:15–14:45, in Raum A207:
 - ▶ 13 oder 14 Termine, davon 9 vor Weihnachten
- Übung: vierzehntägig,
Vorschlag: dienstags, 9:15–10:45, in Raum A121
- Webseite: in Vorbereitung

Organisation

- Wahlpflichtvorlesung im Diplomstudiengang, 4 LP
- Voraussetzung: z.B. Vorlesung „Deskriptive Programmierung“
- Prüfung schriftlich oder mündlich
- Vorlesung donnerstags, 13:15–14:45, in Raum A207:
 - ▶ 13 oder 14 Termine, davon 9 vor Weihnachten
- Übung: vierzehntägig,
Vorschlag: dienstags, 9:15–10:45, in Raum A121
- Webseite: in Vorbereitung
- Material:

Organisation

- Wahlpflichtvorlesung im Diplomstudiengang, 4 LP
- Voraussetzung: z.B. Vorlesung „Deskriptive Programmierung“
- Prüfung schriftlich oder mündlich
- Vorlesung donnerstags, 13:15–14:45, in Raum A207:
 - ▶ 13 oder 14 Termine, davon 9 vor Weihnachten
- Übung: vierzehntägig,
Vorschlag: dienstags, 9:15–10:45, in Raum A121
- Webseite: in Vorbereitung
- Material:
 - ▶ Folien

Organisation

- Wahlpflichtvorlesung im Diplomstudiengang, 4 LP
- Voraussetzung: z.B. Vorlesung „Deskriptive Programmierung“
- Prüfung schriftlich oder mündlich
- Vorlesung donnerstags, 13:15–14:45, in Raum A207:
 - ▶ 13 oder 14 Termine, davon 9 vor Weihnachten
- Übung: vierzehntägig,
Vorschlag: dienstags, 9:15–10:45, in Raum A121
- Webseite: in Vorbereitung
- Material:
 - ▶ Folien
 - ▶ Skript „Types for Programming and Reasoning“

Organisation

- Wahlpflichtvorlesung im Diplomstudiengang, 4 LP
- Voraussetzung: z.B. Vorlesung „Deskriptive Programmierung“
- Prüfung schriftlich oder mündlich
- Vorlesung donnerstags, 13:15–14:45, in Raum A207:
 - ▶ 13 oder 14 Termine, davon 9 vor Weihnachten
- Übung: vierzehntägig,
Vorschlag: dienstags, 9:15–10:45, in Raum A121
- Webseite: in Vorbereitung
- Material:
 - ▶ Folien
 - ▶ Skript „Types for Programming and Reasoning“
 - ▶ Links und Referenzen

Motivation

Das Problem:

Heutige Software ist:

Motivation

Das Problem:

Heutige Software ist:

- fast überall

Motivation

Das Problem:

Heutige Software ist:

- fast überall
- oft sicherheitskritisch

Motivation

Das Problem:

Heutige Software ist:

- fast überall
- oft sicherheitskritisch
- teuer, wenn fehlerhaft

Motivation

Das Problem:

Heutige Software ist:

- fast überall
- oft sicherheitskritisch
- teuer, wenn fehlerhaft
- zunehmend komplex

Motivation

Das Problem:

Heutige Software ist:

- fast überall
- oft sicherheitskritisch
- teuer, wenn fehlerhaft
- zunehmend komplex
- verteilt im Einsatz

Motivation

Das Problem:

Heutige Software ist:

- fast überall
- oft sicherheitskritisch
- teuer, wenn fehlerhaft
- zunehmend komplex
- verteilt im Einsatz

Motivation

Das Problem:

Heutige Software ist:
● fast überall
● oft sicherheitskritisch
● teuer, wenn fehlerhaft
● zunehmend komplex
● verteilt im Einsatz

Motivation

Das Problem:

Heutige <ul style="list-style-type: none">• fast	Software überall	st:
<ul style="list-style-type: none">• oft• teuer	sicherheit er, wenn n	skritisch fehlerhaft
<ul style="list-style-type: none">• zunehmend• verteil	ehmend k eilt im E	omplex insatz

Formale und Informale Methoden

Ansätze zur Gewährleistung korrekter Software umfassen:

Formale und Informale Methoden

Ansätze zur Gewährleistung korrekter Software umfassen:

- Dokumentation

Formale und Informale Methoden

Ansätze zur Gewährleistung korrekter Software umfassen:

- Dokumentation
- formale Spezifikation

Formale und Informale Methoden

Ansätze zur Gewährleistung korrekter Software umfassen:

- Dokumentation
- formale Spezifikation
- Testen

Formale und Informale Methoden

Ansätze zur Gewährleistung korrekter Software umfassen:

- Dokumentation
- formale Spezifikation
- Testen
- Typsysteme

Formale und Informale Methoden

Ansätze zur Gewährleistung korrekter Software umfassen:

- Dokumentation
- formale Spezifikation
- Testen
- Typsysteme
- Laufzeittests

Formale und Informale Methoden

Ansätze zur Gewährleistung korrekter Software umfassen:

- Dokumentation
- formale Spezifikation
- Testen
- Typsysteme
- Laufzeittests
- formaler Beweis

Formale und Informale Methoden

Ansätze zur Gewährleistung korrekter Software umfassen:

- Dokumentation
- formale Spezifikation
- Testen
- **Typsysteme**
- Laufzeittests
- formaler Beweis

Typen

Heutige <ul style="list-style-type: none">• fast	Software überall	st:
<ul style="list-style-type: none">• oft• teuer	sicherheit er, wenn r	skritisch fehlerhaft
<ul style="list-style-type: none">• zun• verte	ehmend k eilt im E.	omplex insatz

Typen

Heutige <ul style="list-style-type: none">• fast	Software überall	st:
<ul style="list-style-type: none">• oft• teuer	sicherheit: er, wenn r	skritisch fehlerhaft
<ul style="list-style-type: none">• zun• verte	ehmend k eilt im E	omplex insatz

Typen

Heutige • fast	Software überall	st:
• oft • teuer	sicherheit: er, wenn r	skritisch fehlerhaft
• zun • verte	ehmend k eilt im E	omplex insatz

Typen dienen zur:

Typen

Heutige • fast	Software überall	st:
• oft • teuer	sicherheit: er, wenn r	skritisch fehlerhaft
• zun • verte	ehmend k eilt im E	omplex insatz

Typen dienen zur:

- Dokumentation

Typen

Heutige • fast	Software überall	st:
• oft • teuer	sicherheit: er, wenn r	skritisch fehlerhaft
• zun • verte	ehmend k eilt im E	omplex insatz

Typen dienen zur:

- Dokumentation
- Spezifikation

Typen

Heutige • fast	Software überall	st:
• oft • teuer	sicherheit: er, wenn r	skritisch fehlerhaft
• zun • verte	ehmend k eilt im E	omplex insatz

Typen dienen zur:

- Dokumentation
- Spezifikation
- Fehlervermeidung

Typen

Heutige • fast	Software überall	st:
• oft • teuer	sicherheit: er, wenn r	skritisch fehlerhaft
• zun • verte	ehmend k teilt im E	omplex insatz

Typen dienen zur:

- Dokumentation
- Spezifikation
- Fehlervermeidung
- Verhaltens-
einschränkung

Typen

Heutige • fast	Software überall	st:
• oft • teuer	sicherheit: er, wenn r	skritisch fehlerhaft
• zun • verte	ehmend k eilt im E	omplex insatz

Typen dienen zur:

- Dokumentation
- Spezifikation
- Fehlervermeidung
- Verhaltens-
einschränkung

Inhaltsüberblick

- Haskell

Inhaltsüberblick

- Haskell
- formales Schließen, gleichungsbasiert; Induktion

Inhaltsüberblick

- Haskell
- formales Schließen, gleichungsbasiert; Induktion
- Typen, Polymorphismus

Inhaltsüberblick

- Haskell
- formales Schließen, gleichungsbasiert; Induktion
- Typen, Polymorphismus
- aus Typen gewonnene Aussagen

Inhaltsüberblick

- Haskell
- formales Schließen, gleichungsbasiert; Induktion
- Typen, Polymorphismus
- aus Typen gewonnene Aussagen
- Programmtransformation

Inhaltsüberblick

- Haskell
- formales Schließen, gleichungsbasiert; Induktion
- Typen, Polymorphismus
- aus Typen gewonnene Aussagen
- Programmtransformation
- Korrektheit von Algorithmen

Inhaltsüberblick

- Haskell
- formales Schließen, gleichungsbasiert; Induktion
- Typen, Polymorphismus
- aus Typen gewonnene Aussagen
- Programmtransformation
- Korrektheit von Algorithmen
- formale Grundlagen, Erweiterungen

Nötige Vorkenntnisse

- Haskell in Grundzügen

Nötige Vorkenntnisse

- Haskell in Grundzügen
- mathematische Grundlagen, Schließen und Beweisen

Nötige Vorkenntnisse

- Haskell in Grundzügen
- mathematische Grundlagen, Schließen und Beweisen
- im Folgenden, eine Reihe von Folien der Vorlesung „Deskriptive Programmierung“

Nötige Vorkenntnisse

- Haskell in Grundzügen
- mathematische Grundlagen, Schließen und Beweisen
- im Folgenden, eine Reihe von Folien der Vorlesung „Deskriptive Programmierung“

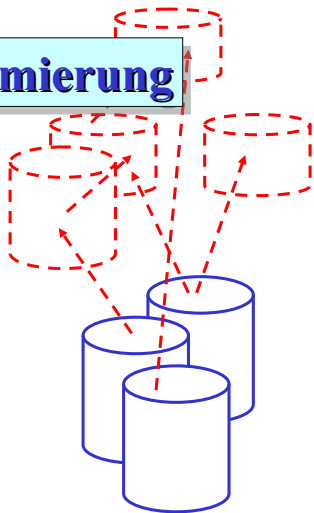
Bücher zur Auffrischung, zum Beispiel:

- Programming in Haskell, Graham Hutton
- Introduction to Functional Programming using Haskell, Richard Bird
- The Haskell School of Expression, Paul Hudak

Deskriptive Programmierung

SS 2009

Dr. Andreas Behrend
Institut für Informatik III
Universität Bonn



```
procedure quicksort(l,r : integer);
var x,i,j,tmp : integer;
begin
  if r>l then
    begin
      x := a[l]; i := l; j := r+1;
      repeat
        repeat i := i+1 until a[i]>=x;
        repeat j := j-1 until a[j]<=x;
        tmp := a[j] ; a[j] := a[i] ; a[i] := tmp;
      until j<=i;
      a[i] := a[j]; a[j] := a[l]; a[l] := tmp;
      quicksort(l,j-1) ;
      quicksort(j+1,r)
    end
  end.
```

```
quicksort [ ] = [ ]
quicksort (x:xs) =
  quicksort [n | n ← xs, n < x] ++
  [x] ++
  quicksort [n | n ← xs, n ≥ x]
```

deskriptiv (funktional: Haskell)

imperativ (PASCAL)

- **Prinzip** der FP:
 - **Spezifikation** = Folge von Funktionsdefinitionen
 - Funktionsdefinition = Folge von definierenden Gleichungen
 - **Operationalisierung** = stufenweise Reduktion von Ausdrücken auf Werte
- **Ausdrücke**:
 - Konstanten, Variablen
 - strukturierte Ausdrücke: **Listen**, **Tupel**
 - **Applikationen**
 - **list comprehensions**
- Systeme definierender **Gleichungen**:
 - Kopf, Rumpf (mit div. Beschränkungen)
 - (ggf.) mehrelementige Parameterlisten
 - **Wächter**
- **Reduktion**:
 - **pattern matching**
 - eindeutige Fallauswahl
 - **lazy evaluation**

- Listen:
 - Klammerdarstellung vs. Baumdarstellung (:)
 - pattern matching mit Listentermen
 - spez. Listenfunktionen (z.B. length, ++, !!)
 - **arithmetische Sequenzen**
 - **unendliche Listen**
 - **list comprehension**: Muster – Generator – Filter
- Typen:
 - **Datentypen** (f. Ausdrücke)
 - Basisdatentypen (Integer etc.)
 - strukturierte Typen (Listen, Tupel)
 - Datentypdeklarationen, Konstruktoren
 - **Funktionstypen** (f. Funktionen)
 - **Funktionsstypdeklarationen**: Funktionsname – Parametertyp – Resultattyp
 - **Curryfizierung**: mehrstellige F. als mehrstufige, einstellige F.
 - **Typinferenz**, Typprüfung, starke Typisierung
 - **Typvariablen**, **polymorphe** Typen
- Funktionen **höherer Ordnung**: F. als Parameter und/oder F. als Resultate

Typisierung und "type checking"

- In einem "zulässigen" Haskell-Programm hat jeder Ausdruck **genau einen** Datentyp, der bereits direkt nach der Eingabe des Programm bestimmbar ist:

Haskell ist eine stark getypte Sprache.

- Jedes Haskell-System überprüft jedes Programm und jede Applikation auf **korrekte Verwendung** von Typen:

Typprüfung

(engl.: "type checking")

- Treten falsch getypte Eingaben auf, wird ein **Typfehler** gemeldet und die entsprechende Applikation nicht ausgeführt.

Starke Typisierung fördert die Korrektheit der Programme.

Die Typisierung stellt eine **echte** Steigerung der Ausdrucksmächtigkeit des λ -Kalküls dar und erlaubt

(2) Programmierfehler zu vermeiden

Klassifikation von Typsystemen:

monomorph : Jeder Bezeichner hat genau einen Typ.
polymorph : Ein Bezeichner kann mehrere Typen haben.
statisch : Typkorrektheit wird zur Übersetzungszeit überprüft.
dynamisch : Typkorrektheit wird zur Laufzeit überprüft.

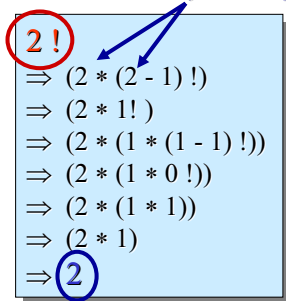
	statisch	dynamisch
monomorph	Pascal	
polymorph	ML, Haskell C++, Java	Lisp, Smalltalk

Spezifikation („Programm“) \equiv
Funktionsdefinition

$$n! = \begin{cases} 1 & \text{falls } n = 0 \\ n * (n-1)! & \text{falls } n > 0 \end{cases}$$

Bsp. Fakultätsfunktion

vordefinierte Operatoren



Eingabe: auszuwertender Term



(wiederholte) Funktionsanwendung



Ausgabe: resultierender Funktionswert

Deklaration von Funktionen

```
min3 :: (Int,Int,Int) -> Int
```

```
min3 (x,y,z) = if x<y then (if x<z then x else z)  
              else (if y<z then y else z)
```


Deklaration von Funktionen

```
min3 :: (Int,Int,Int) -> Int
```

```
min3 (x,y,z) = if x<y then (if x<z then x else z)  
              else (if y<z then y else z)
```

```
> min3 (5,4,6)
```

```
4
```

Deklaration von Funktionen

```
min3 :: (Int,Int,Int) -> Int
min3 (x,y,z) = if x<y then (if x<z then x else z)
               else (if y<z then y else z)
```

```
> min3 (5,4,6)
```

```
4
```

```
min3' :: Int -> Int -> Int -> Int
min3' x y z = min (min x y) z
```

Deklaration von Funktionen

```
min3 :: (Int,Int,Int) -> Int
min3 (x,y,z) = if x<y then (if x<z then x else z)
               else (if y<z then y else z)
```

```
> min3 (5,4,6)
4
```

```
min3' :: Int -> Int -> Int -> Int
min3' x y z = min (min x y) z
```

```
> min3' 5 4 6
4
```

Deklaration von Funktionen

```
min3 :: (Int,Int,Int) -> Int
min3 (x,y,z) = if x<y then (if x<z then x else z)
                else (if y<z then y else z)
```

```
> min3 (5,4,6)
4
```

```
min3' :: Int -> Int -> Int -> Int
min3' x y z = min (min x y) z
```

```
> min3' 5 4 6
4
```

```
isEven :: Int -> Bool
isEven n = (n `mod` 2) == 0
```

Deklaration von Funktionen

```
min3 :: (Int,Int,Int) -> Int
min3 (x,y,z) = if x<y then (if x<z then x else z)
               else (if y<z then y else z)
```

```
> min3 (5,4,6)
4
```

```
min3' :: Int -> Int -> Int -> Int
min3' x y z = min (min x y) z
```

```
> min3' 5 4 6
4
```

```
isEven :: Int -> Bool
isEven n = (n `mod` 2) == 0
```

```
> isEven 12
True
```

Rekursive Funktionen

```
sumsquare :: Int -> Int
```

```
sumsquare i = if i==0 then 0 else i*i + sumsquare (i-1)
```

Rekursive Funktionen

```
sumsquare :: Int -> Int  
sumsquare i = if i==0 then 0 else i*i + sumsquare (i-1)
```

```
> sumsquare 4  
30
```

Rekursive Funktionen

```
sumsquare :: Int -> Int  
sumsquare i = if i==0 then 0 else i*i + sumsquare (i-1)
```

```
> sumsquare 4  
30
```

```
fac :: Int -> Int  
fac n = if n==0 then 1 else n * fac (n-1)
```


Rekursive Funktionen

```
sumsquare :: Int -> Int  
sumsquare i = if i==0 then 0 else i*i + sumsquare (i-1)
```

```
> sumsquare 4  
30
```

```
fac :: Int -> Int  
fac n = if n==0 then 1 else n * fac (n-1)
```

```
> fac 5  
120
```

Berechnung durch schrittweise Auswertung

```
sumsquare :: Int -> Int
```

```
sumsquare i = if i==0 then 0 else i*i + sumsquare (i-1)
```

```
> sumsquare 3
```

Berechnung durch schrittweise Auswertung

```
sumsquare :: Int -> Int
```

```
sumsquare i = if i==0 then 0 else i*i + sumsquare (i-1)
```

```
> sumsquare 3
```

```
= if 3==0 then 0 else 3*3 + sumsquare (3-1)
```

Berechnung durch schrittweise Auswertung

```
sumsquare :: Int -> Int
sumsquare i = if i==0 then 0 else i*i + sumsquare (i-1)

> sumsquare 3
= if 3==0 then 0 else 3*3 + sumsquare (3-1)
= 3*3 + sumsquare (3-1)
```

Berechnung durch schrittweise Auswertung

```
sumsquare :: Int -> Int  
sumsquare i = if i==0 then 0 else i*i + sumsquare (i-1)
```

```
> sumsquare 3  
= if 3==0 then 0 else 3*3 + sumsquare (3-1)  
= 3*3 + sumsquare (3-1)  
= 9 + sumsquare 2
```

Berechnung durch schrittweise Auswertung

```
sumsquare :: Int -> Int  
sumsquare i = if i==0 then 0 else i*i + sumsquare (i-1)
```

```
> sumsquare 3  
= if 3==0 then 0 else 3*3 + sumsquare (3-1)  
= 3*3 + sumsquare (3-1)  
= 9 + sumsquare 2  
= 9 + if 2==0 then 0 else 2*2 + sumsquare (2-1)
```

Berechnung durch schrittweise Auswertung

```
sumsquare :: Int -> Int  
sumsquare i = if i==0 then 0 else i*i + sumsquare (i-1)
```

```
> sumsquare 3  
= if 3==0 then 0 else 3*3 + sumsquare (3-1)  
= 3*3 + sumsquare (3-1)  
= 9 + sumsquare 2  
= 9 + if 2==0 then 0 else 2*2 + sumsquare (2-1)  
= 9 + 2*2 + sumsquare (2-1)
```

Berechnung durch schrittweise Auswertung

```
sumsquare :: Int -> Int  
sumsquare i = if i==0 then 0 else i*i + sumsquare (i-1)
```

```
> sumsquare 3  
= if 3==0 then 0 else 3*3 + sumsquare (3-1)  
= 3*3 + sumsquare (3-1)  
= 9 + sumsquare 2  
= 9 + if 2==0 then 0 else 2*2 + sumsquare (2-1)  
= 9 + 2*2 + sumsquare (2-1)  
= 9 + 4 + sumsquare 1
```


Berechnung durch schrittweise Auswertung

```
sumsquare :: Int -> Int  
sumsquare i = if i==0 then 0 else i*i + sumsquare (i-1)
```

```
> sumsquare 3  
= if 3==0 then 0 else 3*3 + sumsquare (3-1)  
= 3*3 + sumsquare (3-1)  
= 9 + sumsquare 2  
= 9 + if 2==0 then 0 else 2*2 + sumsquare (2-1)  
= 9 + 2*2 + sumsquare (2-1)  
= 9 + 4 + sumsquare 1  
= 9 + 4 + if 1==0 then 0 else 1*1 + sumsquare (1-1)
```

Berechnung durch schrittweise Auswertung

```
sumsquare :: Int -> Int  
sumsquare i = if i==0 then 0 else i*i + sumsquare (i-1)
```

```
> sumsquare 3  
= if 3==0 then 0 else 3*3 + sumsquare (3-1)  
= 3*3 + sumsquare (3-1)  
= 9 + sumsquare 2  
= 9 + if 2==0 then 0 else 2*2 + sumsquare (2-1)  
= 9 + 2*2 + sumsquare (2-1)  
= 9 + 4 + sumsquare 1  
= 9 + 4 + if 1==0 then 0 else 1*1 + sumsquare (1-1)  
= 9 + 4 + 1*1 + sumsquare (1-1)
```

Berechnung durch schrittweise Auswertung

```
sumsquare :: Int -> Int  
sumsquare i = if i==0 then 0 else i*i + sumsquare (i-1)
```

```
> sumsquare 3  
= if 3==0 then 0 else 3*3 + sumsquare (3-1)  
= 3*3 + sumsquare (3-1)  
= 9 + sumsquare 2  
= 9 + if 2==0 then 0 else 2*2 + sumsquare (2-1)  
= 9 + 2*2 + sumsquare (2-1)  
= 9 + 4 + sumsquare 1  
= 9 + 4 + if 1==0 then 0 else 1*1 + sumsquare (1-1)  
= 9 + 4 + 1*1 + sumsquare (1-1)  
= 9 + 4 + 1 + sumsquare 0
```

Berechnung durch schrittweise Auswertung

```
sumsquare :: Int -> Int
sumsquare i = if i==0 then 0 else i*i + sumsquare (i-1)

> sumsquare 3
= if 3==0 then 0 else 3*3 + sumsquare (3-1)
= 3*3 + sumsquare (3-1)
= 9 + sumsquare 2
= 9 + if 2==0 then 0 else 2*2 + sumsquare (2-1)
= 9 + 2*2 + sumsquare (2-1)
= 9 + 4 + sumsquare 1
= 9 + 4 + if 1==0 then 0 else 1*1 + sumsquare (1-1)
= 9 + 4 + 1*1 + sumsquare (1-1)
= 9 + 4 + 1 + sumsquare 0
= 9 + 4 + 1 + if 0==0 then 0 else 0*0 + sumsquare (0-1)
```

Berechnung durch schrittweise Auswertung

```
sumsquare :: Int -> Int
sumsquare i = if i==0 then 0 else i*i + sumsquare (i-1)

> sumsquare 3
= if 3==0 then 0 else 3*3 + sumsquare (3-1)
= 3*3 + sumsquare (3-1)
= 9 + sumsquare 2
= 9 + if 2==0 then 0 else 2*2 + sumsquare (2-1)
= 9 + 2*2 + sumsquare (2-1)
= 9 + 4 + sumsquare 1
= 9 + 4 + if 1==0 then 0 else 1*1 + sumsquare (1-1)
= 9 + 4 + 1*1 + sumsquare (1-1)
= 9 + 4 + 1 + sumsquare 0
= 9 + 4 + 1 + if 0==0 then 0 else 0*0 + sumsquare (0-1)
= 9 + 4 + 1 + 0
```

Berechnung durch schrittweise Auswertung

```
sumsquare :: Int -> Int
sumsquare i = if i==0 then 0 else i*i + sumsquare (i-1)

> sumsquare 3
= if 3==0 then 0 else 3*3 + sumsquare (3-1)
= 3*3 + sumsquare (3-1)
= 9 + sumsquare 2
= 9 + if 2==0 then 0 else 2*2 + sumsquare (2-1)
= 9 + 2*2 + sumsquare (2-1)
= 9 + 4 + sumsquare 1
= 9 + 4 + if 1==0 then 0 else 1*1 + sumsquare (1-1)
= 9 + 4 + 1*1 + sumsquare (1-1)
= 9 + 4 + 1 + sumsquare 0
= 9 + 4 + 1 + if 0==0 then 0 else 0*0 + sumsquare (0-1)
= 9 + 4 + 1 + 0
= 14
```

Pattern-Matching

Statt: `power :: Int -> Int`

`power n = if n==0 then 1 else 2 * power (n-1)`

Pattern-Matching

Statt: `power :: Int -> Int`
`power n = if n==0 then 1 else 2 * power (n-1)`

Auch: `power :: Int -> Int`
`power 0 = 1`
`power (m+1) = 2 * power m`

Pattern-Matching

```
Statt: power :: Int -> Int
       power n = if n==0 then 1 else 2 * power (n-1)
```

```
Auch: power :: Int -> Int
       power 0      = 1
       power (m+1) = 2 * power m
```

```
Statt: product :: [Int] -> Int
       product l = if null l
                   then 1
                   else head l * product (tail l)
```

Pattern-Matching

```
Statt: power :: Int -> Int
       power n = if n==0 then 1 else 2 * power (n-1)
```

```
Auch: power :: Int -> Int
      power 0      = 1
      power (m+1) = 2 * power m
```

```
Statt: product :: [Int] -> Int
      product l = if null l
                  then 1
                  else head l * product (tail l)
```

```
Auch: product :: [Int] -> Int
      product []      = 1
      product (x:xs) = x * product xs
```

- **Listen** sind die wichtigsten Datenstrukturen in der funktionalen Programmierung.
- Haskell-Liste: Folge von Elementen **gleichen Typs** (homogene Struktur)
- In Haskell werden Listenelemente in **eckige Klammern** eingeschlossen.

[1, 2, 3]

Liste von ganzen Zahlen (Typ: Integer)

['a', 'b', 'c']

Liste von Buchstaben (Typ: Char)

[]

leere Liste (beliebigen Typs)

[[1,2], [], [2]]

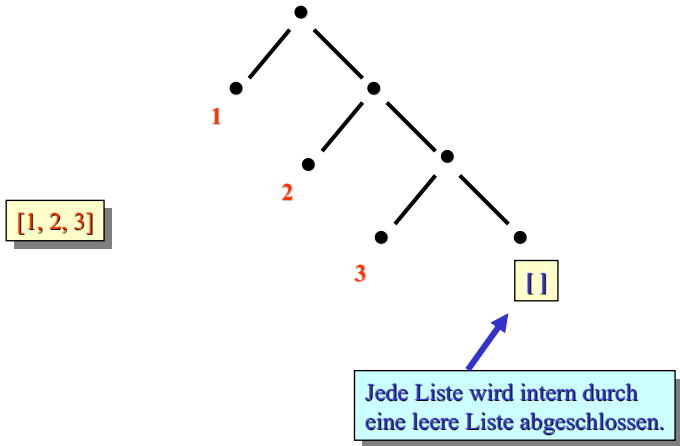
Liste von Integer-Listen

[[1,2], 'a', 3]

keine Liste (verschiedene Elementtypen)

Baumdarstellung von Listen

Listen werden intern als **Binärbäume** dargestellt, deren Blättern mit den einzelnen Listenelemente markiert sind:

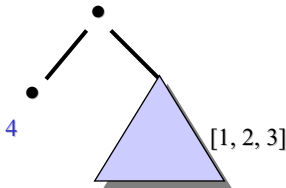


- elementarer **Konstruktor** ('Operator' zum Konstruieren) für Listen in Haskell:



- Der Konstruktor `:` dient zum **Verlängern** einer gegebenen Liste um ein Element, das am Listenkopf eingefügt wird:

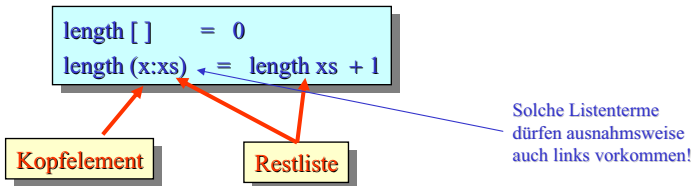
```
> 4 : [1, 2, 3]
[4, 1, 2, 3]
```



- **alternative Notation** für Listen (analog zur Baumdarstellung):

```
4 : 1 : 2 : 3 : []
```

- in Haskell vordefiniert: Funktion zur Bestimmung der **Länge einer Liste**



- Beispiel für die Anwendung von 'length':

```
> length [1,2]
⇒ length [2] + 1
⇒ (length [ ] + 1) + 1
⇒ ( 0 + 1) + 1
⇒ 1 + 1
⇒ 2
```

Komplexes Pattern-Matching

```
risers :: [Int] -> [[Int]]
risers []          = []
risers [x]         = [[x]]
risers (x:y:zs) = if x<=y then (x:s):ts else [x]:(s:ts)
                  where (s:ts) = risers (y:zs)
```

Komplexes Pattern-Matching

```
risers :: [Int] -> [[Int]]
risers []          = []
risers [x]         = [[x]]
risers (x:y:zs) = if x<=y then (x:s):ts else [x]:(s:ts)
                  where (s:ts) = risers (y:zs)

> risers [1,2,0]
```


Komplexes Pattern-Matching

```
risers :: [Int] -> [[Int]]
risers []      = []
risers [x]     = [[x]]
risers (x:y:zs) = if x<=y then (x:s):ts else [x):(s:ts)
                  where (s:ts) = risers (y:zs)

> risers [1,2,0]
= if 1<=2 then (1:s):ts else [1):(s:ts)
  where (s:ts) = risers (2:[0])
```

Komplexes Pattern-Matching

```
risers :: [Int] -> [[Int]]
risers []          = []
risers [x]         = [[x]]
risers (x:y:zs) = if x<=y then (x:s):ts else [x]:(s:ts)
                  where (s:ts) = risers (y:zs)

> risers [1,2,0]
= if 1<=2 then (1:s):ts else [1]:(s:ts)
  where (s:ts) = risers (2:[0])
= (1:s):ts
  where (s:ts) = risers (2:[0])
```

Komplexes Pattern-Matching

```
risers :: [Int] -> [[Int]]
risers []      = []
risers [x]    = [[x]]
risers (x:y:zs) = if x<=y then (x:s):ts else [x]:(s:ts)
                 where (s:ts) = risers (y:zs)
```

```
> risers [1,2,0]
= if 1<=2 then (1:s):ts else [1]:(s:ts)
  where (s:ts) = risers (2:[0])
= (1:s):ts
  where (s:ts) = risers (2:[0])
= (1:s):ts
  where (s:ts) = [2]:(s':ts')
                                     where (s':ts') = risers (0:[])
```

Komplexes Pattern-Matching

```
risers :: [Int] -> [[Int]]
risers []          = []
risers [x]         = [[x]]
risers (x:y:zs) = if x<=y then (x:s):ts else [x]:(s:ts)
                  where (s:ts) = risers (y:zs)

> risers [1,2,0]
= if 1<=2 then (1:s):ts else [1]:(s:ts)
  where (s:ts) = risers (2:[0])
= (1:s):ts
  where (s:ts) = risers (2:[0])
= (1:s):ts
  where (s:ts) = [2]:(s':ts')
                        where (s':ts') = risers (0:[])
= (1:s):ts
  where (s:ts) = [2]:[[0]]
```

Komplexes Pattern-Matching

```
risers :: [Int] -> [[Int]]
risers []          = []
risers [x]         = [[x]]
risers (x:y:zs) = if x<=y then (x:s):ts else [x]:(s:ts)
                  where (s:ts) = risers (y:zs)
```

```
> risers [1,2,0]
= if 1<=2 then (1:s):ts else [1]:(s:ts)
  where (s:ts) = risers (2:[0])
= (1:s):ts
  where (s:ts) = risers (2:[0])
= (1:s):ts
  where (s:ts) = [2]:(s':ts')
                                where (s':ts') = risers (0:[])
= (1:s):ts
  where (s:ts) = [2]:[[0]]
= [[1,2],[0]]
```

Komplexes Pattern-Matching

```
unzip :: [(Int,Int)] -> ([Int],[Int])
unzip []           = ([],[ ])
unzip ((x,y):zs) = (x:xs,y:ys)
    where (xs,ys) = unzip zs
```

Komplexes Pattern-Matching

```
unzip :: [(Int,Int)] -> ([Int],[Int])
```

```
unzip [] = ([],[Int])
```

```
unzip ((x,y):zs) = (x:xs,y:ys)
```

```
    where (xs,ys) = unzip zs
```

```
> unzip [(1,2),(3,4)]
```

Komplexes Pattern-Matching

```
unzip :: [(Int,Int)] -> ([Int],[Int])
```

```
unzip [] = ([],[Int])
```

```
unzip ((x,y):zs) = (x:xs,y:ys)
```

```
    where (xs,ys) = unzip zs
```

```
> unzip [(1,2),(3,4)]
```

```
= (1:xs,2:ys)
```

```
    where (xs,ys) = unzip [(3,4)]
```


Komplexes Pattern-Matching

```
unzip :: [(Int,Int)] -> ([Int],[Int])
```

```
unzip [] = ([],[Int])
```

```
unzip ((x,y):zs) = (x:xs,y:ys)
```

```
  where (xs,ys) = unzip zs
```

```
> unzip [(1,2),(3,4)]
```

```
= (1:xs,2:ys)
```

```
  where (xs,ys) = unzip [(3,4)]
```

```
= (1:xs,2:ys)
```

```
  where (xs,ys) = (3:xs',4:ys')
```

```
    where (xs',ys') = unzip []
```

Komplexes Pattern-Matching

```
unzip :: [(Int,Int)] -> ([Int],[Int])
```

```
unzip [] = ([],[])
```

```
unzip ((x,y):zs) = (x:xs,y:ys)
```

```
  where (xs,ys) = unzip zs
```

```
> unzip [(1,2),(3,4)]
```

```
= (1:xs,2:ys)
```

```
  where (xs,ys) = unzip [(3,4)]
```

```
= (1:xs,2:ys)
```

```
  where (xs,ys) = (3:xs',4:ys')
```

```
                    where (xs',ys') = unzip []
```

```
= (1:xs,2:ys)
```

```
  where (xs,ys) = (3:xs',4:ys')
```

```
                    where (xs',ys') = ([],[])
```

Komplexes Pattern-Matching

```
unzip :: [(Int,Int)] -> ([Int],[Int])
```

```
unzip [] = ([],[Int])
```

```
unzip ((x,y):zs) = (x:xs,y:ys)
```

```
    where (xs,ys) = unzip zs
```

```
> unzip [(1,2),(3,4)]
```

```
= (1:xs,2:ys)
```

```
    where (xs,ys) = unzip [(3,4)]
```

```
= (1:xs,2:ys)
```

```
    where (xs,ys) = (3:xs',4:ys')
```

```
                where (xs',ys') = unzip []
```

```
= (1:xs,2:ys)
```

```
    where (xs,ys) = (3:xs',4:ys')
```

```
                where (xs',ys') = ([],[Int])
```

```
= ([1,3],[2,4])
```

Pattern-Matching über mehreren Argumenten

```
drop :: Int -> [Int] -> [Int]
drop 0      xs      = xs
drop n     []      = []
drop (n+1) (x:xs) = drop n xs
```

Pattern-Matching über mehreren Argumenten

```
drop :: Int -> [Int] -> [Int]
drop 0      xs      = xs
drop n      []      = []
drop (n+1) (x:xs) = drop n xs
```

```
> drop 0 [1,2,3]
[1,2,3]
```

```
> drop 5 [1,2,3]
[]
```

```
> drop 3 [1,2,3,4,5]
[4,5]
```

Reihenfolge beim Pattern-Matching

```
zip :: [Int] -> [Int] -> [(Int,Int)]
zip (x:xs) (y:ys) = (x,y):(zip xs ys)
zip xs      ys      = []
```

Reihenfolge beim Pattern-Matching

```
zip :: [Int] -> [Int] -> [(Int,Int)]  
zip (x:xs) (y:ys) = (x,y):(zip xs ys)  
zip xs      ys      = []
```

```
> zip [1..3] [10..15]  
[(1,10),(2,11),(3,12)]
```

Reihenfolge beim Pattern-Matching

```
zip :: [Int] -> [Int] -> [(Int,Int)]  
zip (x:xs) (y:ys) = (x,y):(zip xs ys)  
zip xs      ys      = []
```

```
> zip [1..3] [10..15]  
[(1,10),(2,11),(3,12)]
```

```
zip :: [Int] -> [Int] -> [(Int,Int)]  
zip xs      ys      = []  
zip (x:xs) (y:ys) = (x,y):(zip xs ys)
```


Reihenfolge beim Pattern-Matching

```
zip :: [Int] -> [Int] -> [(Int,Int)]
zip (x:xs) (y:ys) = (x,y):(zip xs ys)
zip xs      ys      = []
```

```
> zip [1..3] [10..15]
[(1,10),(2,11),(3,12)]
```

```
zip :: [Int] -> [Int] -> [(Int,Int)]
zip xs      ys      = []
zip (x:xs) (y:ys) = (x,y):(zip xs ys)
```

```
> zip [1..3] [10..15]
[]
```

Einfache Ein- und Ausgabe

```
module Main where

product :: [Int] -> Int
product []      = 1
product (x:xs) = x * product xs

main = do n <- readLn
          m <- readLn
          print (product [n..m])
```

Einfache Ein- und Ausgabe

```
module Main where

product :: [Int] -> Int
product []      = 1
product (x:xs) = x * product xs

main = do n <- readLn
          m <- readLn
          print (product [n..m])
```

Programmablauf:

5

Einfache Ein- und Ausgabe

```
module Main where

product :: [Int] -> Int
product []      = 1
product (x:xs) = x * product xs

main = do n <- readLn
          m <- readLn
          print (product [n..m])
```

Programmablauf:

5

8

Einfache Ein- und Ausgabe

```
module Main where

product :: [Int] -> Int
product []      = 1
product (x:xs) = x * product xs

main = do n <- readLn
          m <- readLn
          print (product [n..m])
```

Programmablauf:

5

8

1680

Algebraische Datentypen (I)

```
data Days = Monday | Tuesday | Wednesday | Thursday |  
          Friday | Saturday | Sunday
```

- Typ Days hat mögliche Werte Monday, Tuesday, ...

Algebraische Datentypen (I)

```
data Days = Monday | Tuesday | Wednesday | Thursday |  
          Friday | Saturday | Sunday
```

- Typ Days hat mögliche Werte Monday, Tuesday, ...
- kann beliebig als Komponente in anderen Typen auftreten, etwa [(Days,Int)] mit z.B. [], [(Sunday,-5)] und [(Monday,1),(Wednesday,3),(Monday,0),(Friday,5)] als möglichen Werten

Algebraische Datentypen (I)

```
data Days = Monday | Tuesday | Wednesday | Thursday |  
          Friday | Saturday | Sunday
```

- Typ Days hat mögliche Werte Monday, Tuesday, ...
- kann beliebig als Komponente in anderen Typen auftreten, etwa [(Days,Int)] mit z.B. [], [(Sunday,-5)] und [(Monday,1),(Wednesday,3),(Monday,0),(Friday,5)] als möglichen Werten
- Berechnung mittels Pattern-Matching möglich:

```
workingday :: Days -> Bool  
workingday Saturday = False  
workingday Sunday   = False  
workingday day      = True
```


Algebraische Datentypen (II)

```
data Date = Date Int Int Int
```

```
data Time = Hour Int
```

```
data Connection = Flight String Date Time Time |  
                 Train Date Time Time
```

Algebraische Datentypen (II)

```
data Date = Date Int Int Int
data Time = Hour Int
data Connection = Flight String Date Time Time |
                 Train Date Time Time
```

- mögliche Werte für Connection:
Flight "DBA" (Date 20 06 2007) (Hour 9) (Hour 11),
Train (Date 21 06 2007) (Hour 9) (Hour 13), ...

Algebraische Datentypen (II)

```
data Date = Date Int Int Int
data Time = Hour Int
data Connection = Flight String Date Time Time |
                 Train Date Time Time
```

- mögliche Werte für Connection:
Flight "DBA" (Date 20 06 2007) (Hour 9) (Hour 11),
Train (Date 21 06 2007) (Hour 9) (Hour 13), ...
- Berechnung mittels Pattern-Matching:
travelTime :: Connection -> Int
travelTime (Flight _ _ (Hour d) (Hour a)) = a-d+2
travelTime (Train _ (Hour d) (Hour a)) = a-d+1

Algebraische Datentypen (III)

```
data Expr = Lit Int | Add Expr Expr | Sub Expr Expr
```

Algebraische Datentypen (III)

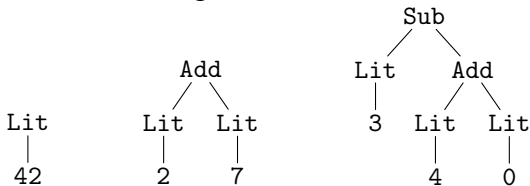
```
data Expr = Lit Int | Add Expr Expr | Sub Expr Expr
```

- mögliche Werte: Lit 42, Add (Lit 2) (Lit 7),
Sub (Lit 3) (Add (Lit 4) (Lit 0)), ...

Algebraische Datentypen (III)

```
data Expr = Lit Int | Add Expr Expr | Sub Expr Expr
```

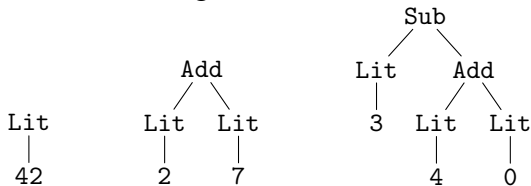
- mögliche Werte: Lit 42, Add (Lit 2) (Lit 7), Sub (Lit 3) (Add (Lit 4) (Lit 0)), ...
- Baumdarstellung:



Algebraische Datentypen (III)

```
data Expr = Lit Int | Add Expr Expr | Sub Expr Expr
```

- mögliche Werte: Lit 42, Add (Lit 2) (Lit 7), Sub (Lit 3) (Add (Lit 4) (Lit 0)), ...
- Baumdarstellung:



- Berechnung:

```
eval :: Expr -> Int
```

```
eval (Lit n)      = n
```

```
eval (Add e1 e2) = (eval e1) + (eval e2)
```

```
eval (Sub e1 e2) = (eval e1) - (eval e2)
```

Konkatenation von Listen

- wichtige Grundoperation für alle Listen: **Konkatenieren** zweier Listen
(= Aneinanderhängen)

```
concatenation [ ] ys      = ys
concatenation (x:xs) ys  = x : (concatenation xs ys)
```

- Beispielanwendung:

```
> concatenation [1, 2] [3, 4]
[1, 2, 3, 4]
```

- Auch diese Funktion ist in Haskell **als Infixoperator** vordefiniert :

```
> [ 1, 2] ++ [3, 4]
[1, 2, 3, 4]
```


Polymorphe Typen

```
concatenation :: [Int] -> [Int] -> [Int]
concatenation []      ys = ys
concatenation (x:xs) ys = x:(concatenation xs ys)
```

Polymorphe Typen

```
concatenation :: [Int] -> [Int] -> [Int]
concatenation []      ys = ys
concatenation (x:xs) ys = x:(concatenation xs ys)
```

```
concatenation' :: [Bool] -> [Bool] -> [Bool]
concatenation' []      ys = ys
concatenation' (x:xs) ys = x:(concatenation' xs ys)
```

Polymorphe Typen

```
concatenation :: [Int] -> [Int] -> [Int]
concatenation []      ys = ys
concatenation (x:xs) ys = x:(concatenation xs ys)
```

```
concatenation' :: [Bool] -> [Bool] -> [Bool]
concatenation' []      ys = ys
concatenation' (x:xs) ys = x:(concatenation' xs ys)
```

```
concatenation'' :: String -> String -> String
concatenation'' []      ys = ys
concatenation'' (x:xs) ys = x:(concatenation'' xs ys)
```

Polymorphe Typen

```
concatenation :: [a] -> [a] -> [a]
```

```
concatenation [] ys = ys
```

```
concatenation (x:xs) ys = x:(concatenation xs ys)
```

Polymorphe Typen

```
concatenation :: [a] -> [a] -> [a]
```

```
concatenation [] ys = ys
```

```
concatenation (x:xs) ys = x:(concatenation xs ys)
```

```
> concatenation [1,2,3] [4,5,6]
```

Polymorphe Typen

```
concatenation :: [a] -> [a] -> [a]
```

```
concatenation [] ys = ys
```

```
concatenation (x:xs) ys = x:(concatenation xs ys)
```

```
> concatenation [1,2,3] [4,5,6]
```

```
[1,2,3,4,5,6]
```

Polymorphe Typen

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

```
> concatenation [1,2,3] [4,5,6]
[1,2,3,4,5,6]
```

```
> concatenation [True] [False,True,False]
```

Polymorphe Typen

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

```
> concatenation [1,2,3] [4,5,6]
[1,2,3,4,5,6]
```

```
> concatenation [True] [False,True,False]
[True,False,True,False]
```


Polymorphe Typen

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

```
> concatenation [1,2,3] [4,5,6]
[1,2,3,4,5,6]
```

```
> concatenation [True] [False,True,False]
[True,False,True,False]
```

```
> concatenation "abc" "def"
```

Polymorphe Typen

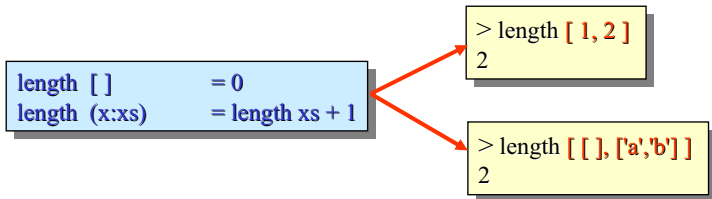
```
concatenation :: [a] -> [a] -> [a]
concatenation []      ys = ys
concatenation (x:xs) ys = x:(concatenation xs ys)
```

```
> concatenation [1,2,3] [4,5,6]
[1,2,3,4,5,6]
```

```
> concatenation [True] [False,True,False]
[True,False,True,False]
```

```
> concatenation "abc" "def"
"abcdef"
```

- Das Beispiel 'zwischen' hat gezeigt, dass es manchmal sinnvoll ist, Typen nicht exakt festzulegen, um **flexibleren** Gebrauch einer Funktion zu ermöglichen.
- Für die meisten Listenoperatoren ist diese "Flexibilität" sogar unerlässlich, weil sie für Listen aus beliebigen Elementtypen gedacht sind:



- Solche Funktionen werden **polymorph** genannt.

("vielgestaltig", von griech. "poly": "viel"; "morph..": "Gestalt")

Typvariablen und parametrisierte Typen

- Um Funktionen wie 'length' einen Typ zuzuordnen zu können, werden Variablen verwendet, die als Platzhalter für beliebige Typen stehen:

Typvariablen

- Mit Typvariablen können für polymorphe Funktionen **parametrisierte Typen** gebildet werden:

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

- Ist auch der **Resultattyp** nur durch eine Typvariable beschrieben, dann bestimmt wieder der Typ der aktuellen Parameter den Typ des Resultats:

```
> last
last :: [a] -> a
```

```
> last [1,2,3]
3 :: Integer
```

Polymorphe Typen

```
drop :: Int -> [Int] -> [Int]
drop 0      xs      = xs
drop n     []      = []
drop (n+1) (x:xs) = drop n xs
```

Polymorphe Typen

```
drop :: Int -> [Int] -> [Int]
drop 0      xs      = xs
drop n      []      = []
drop (n+1) (x:xs) = drop n xs
```

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

Polymorphe Typen

```
concatenation :: [a] -> [a] -> [a]
concatenation []      l = l
concatenation (x:xs) l = x:(concatenation xs l)

> concatenation "abc" [True]
```

Polymorphe Typen

```
concatenation :: [a] -> [a] -> [a]
concatenation []      l = l
concatenation (x:xs) l = x:(concatenation xs l)
```

```
> concatenation "abc" [True]
```

```
    Couldn't match 'Char' against 'Bool'
```

```
      Expected type: Char
```

```
      Inferred type: Bool
```

```
    In the list element: True
```

```
    In the second argument of 'concatenation',
    namely '[True]'
```


Polymorphe Typen

```
zip :: [a] -> [b] -> [(a,b)]  
zip (x:xs) (y:ys) = (x,y):(zip xs ys)  
zip xs      ys      = []
```

Polymorphe Typen

```
zip :: [a] -> [b] -> [(a,b)]
zip (x:xs) (y:ys) = (x,y):(zip xs ys)
zip xs      ys      = []

> zip "abc" [True,False,True]
```

Polymorphe Typen

```
zip :: [a] -> [b] -> [(a,b)]  
zip (x:xs) (y:ys) = (x,y):(zip xs ys)  
zip xs      ys      = []
```

```
> zip "abc" [True,False,True]  
[('a',True),('b',False),('c',True)]
```

Polymorphe Typen

```
zip :: [a] -> [b] -> [(a,b)]  
zip (x:xs) (y:ys) = (x,y):(zip xs ys)  
zip xs      ys      = []
```

```
> zip "abc" [True,False,True]  
[('a',True),('b',False),('c',True)]
```

```
> :t "abc"  
"abc" :: [Char]
```

```
> :t [True,False,True]  
[True,False,True] :: [Bool]
```

```
> :t [('a',True),('b',False),('c',True)]  
 [('a',True),('b',False),('c',True)] :: [(Char,Bool)]
```

Polymorphe Typen

```
data Tree a = Nil | Node a (Tree a) (Tree a)
```

Polymorphe Typen

```
data Tree a = Nil | Node a (Tree a) (Tree a)
```

- mögliche Werte: Nil

Polymorphe Typen

```
data Tree a = Nil | Node a (Tree a) (Tree a)
```

- mögliche Werte: Nil,
Node 4 Nil Nil :: Tree Int

Polymorphe Typen

```
data Tree a = Nil | Node a (Tree a) (Tree a)
```

- mögliche Werte: Nil,
Node 4 Nil Nil :: Tree Int,
Node 'a' Nil (Node 'b' Nil Nil) :: Tree Char,
...

Polymorphe Typen

```
data Tree a = Nil | Node a (Tree a) (Tree a)
```

- mögliche Werte: Nil,
Node 4 Nil Nil :: Tree Int,
Node 'a' Nil (Node 'b' Nil Nil) :: Tree Char,
...
- aber nicht: Node 4 (Node 'a' Nil Nil) Nil

Polymorphe Typen

```
data Tree a = Nil | Node a (Tree a) (Tree a)
```

- mögliche Werte: Nil,
Node 4 Nil Nil :: Tree Int,
Node 'a' Nil (Node 'b' Nil Nil) :: Tree Char,
...

- aber nicht: Node 4 (Node 'a' Nil Nil) Nil

- Berechnung:

```
height :: Tree a -> Int
```

```
height Nil = 0
```

```
height (Node n t1 t2) = 1 + (max (height t1)  
                                (height t2))
```

Funktionen als Parameter

- Ein sehr nützliches Beispiel einer Funktion, die eine andere Funktion als **Parameter** akzeptiert und sie dann auf alle Elemente einer Liste anwendet, ist die map-Funktion:

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

Funktion als Parameter

- Zwei unterschiedliche Applikationen dieser Funktion:

```
> map square [1,2,3]  
[1,4,9] :: [Integer]
```

```
> map sqrt [2,3,4]  
[1.41421,1.73205,2.0] :: [Double]
```

- Die Funktion map ist polymorph:

```
> map  
map :: (a -> b) -> ([a] -> [b])
```

square :: (Integer -> Integer)

sqrt :: (Integer -> Double)

Funktionen höherer Ordnung

Das Beispiel:

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

Funktionen höherer Ordnung

Das Beispiel:

$$\begin{aligned}\text{map } f [] &= [] \\ \text{map } f (x : xs) &= (f x) : (\text{map } f xs)\end{aligned}$$

Einige Aufrufe:

$$\text{map succ [1, 2, 3]} = [2, 3, 4]$$

Funktionen höherer Ordnung

Das Beispiel:

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

Einige Aufrufe:

```
map succ [1, 2, 3] = [2, 3, 4]
```

```
map not [True, False] = [False, True]
```

Funktionen höherer Ordnung

Das Beispiel:

$$\begin{aligned}\text{map } f [] &= [] \\ \text{map } f (x : xs) &= (f x) : (\text{map } f xs)\end{aligned}$$

Einige Aufrufe:

$$\text{map succ } [1, 2, 3] = [2, 3, 4]$$
$$\text{map not } [\text{True}, \text{False}] = [\text{False}, \text{True}]$$
$$\text{map even } [1, 2, 3] = [\text{False}, \text{True}, \text{False}]$$

Funktionen höherer Ordnung

Das Beispiel:

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

Einige Aufrufe:

```
map succ [1, 2, 3]    = [2, 3, 4]  
map not  [True, False] = [False, True]  
map even [1, 2, 3]    = [False, True, False]  
map not  [1, 2, 3]
```


Funktionen höherer Ordnung

Das Beispiel:

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

Einige Aufrufe:

```
map succ [1, 2, 3]    = [2, 3, 4]
map not  [True, False] = [False, True]
map even [1, 2, 3]    = [False, True, False]
map not  [1, 2, 3]
```

Funktionen höherer Ordnung

Das Beispiel:

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

Einige Aufrufe:

```
map succ [1, 2, 3]    = [2, 3, 4]           — a, b ↦ Int, Int
map not  [True, False] = [False, True]      — a, b ↦ Bool, Bool
map even [1, 2, 3]    = [False, True, False] — a, b ↦ Int, Bool
map not  [1, 2, 3]
```

Funktionen höherer Ordnung

Das Beispiel:

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

Einige Aufrufe:

```
map succ [1, 2, 3]    = [2, 3, 4]           — a, b ↦ Int, Int
map not  [True, False] = [False, True]       — a, b ↦ Bool, Bool
map even [1, 2, 3]    = [False, True, False] — a, b ↦ Int, Bool
map not  [1, 2, 3]    ⚡ zur Compile-Zeit zurückgewiesen
```

Funktionen höherer Ordnung

Das Beispiel:

$$\text{map} :: (a \rightarrow b) \rightarrow [a] \rightarrow [b]$$

Einige Aufrufe:

<code>map succ</code>	<code>[1, 2, 3]</code>	<code>= [2, 3, 4]</code>	<code>— a, b ↦ Int, Int</code>
<code>map not</code>	<code>[True, False]</code>	<code>= [False, True]</code>	<code>— a, b ↦ Bool, Bool</code>
<code>map even</code>	<code>[1, 2, 3]</code>	<code>= [False, True, False]</code>	<code>— a, b ↦ Int, Bool</code>
<code>map not</code>	<code>[1, 2, 3]</code>	<code>⚡ zur Compile-Zeit zurückgewiesen</code>	

Funktionen höherer Ordnung

Ein weiteres Beispiel:

```
filter :: (a -> Bool) -> [a] -> [a]
filter p []          = []
filter p (x : xs) | p x          = x : (filter p xs)
                  | otherwise = filter p xs
```

Funktionen höherer Ordnung

Ein weiteres Beispiel:

```
filter :: (a -> Bool) -> [a] -> [a]
filter p []      = []
filter p (x : xs) | p x      = x : (filter p xs)
                  | otherwise = filter p xs
```

Problem: Ausdrücke wie `map f (filter p l)` erfordern Konstruktion von Zwischenergebnissen.

Funktionen höherer Ordnung

Ein weiteres Beispiel:

```
filter :: (a → Bool) → [a] → [a]
filter p []          = []
filter p (x : xs) | p x      = x : (filter p xs)
                  | otherwise = filter p xs
```

Problem: Ausdrücke wie `map f (filter p l)`* erfordern Konstruktion von Zwischenergebnissen.

* `sum [f x | x ← [1..n], p x] ∼= sum (map f (filter p (enumFromTo 1 n)))`

Funktionen höherer Ordnung

Ein weiteres Beispiel:

```
filter :: (a → Bool) → [a] → [a]
filter p []      = []
filter p (x : xs) | p x      = x : (filter p xs)
                  | otherwise = filter p xs
```

Problem: Ausdrücke wie `map f (filter p l)`* erfordern Konstruktion von Zwischenergebnissen.

Lösung?: Explizite Regeln

```
map f (filter p l)  ~> ...
filter p (map f l) ~> ...
    map f1 (map f2 l) ~> ...
filter p1 (filter p2 l) ~> ...
```

* `sum [f x | x ← [1..n], p x] ~> sum (map f (filter p (enumFromTo 1 n)))`

- Nach ähnlichem Prinzip programmiert: vordefinierte Funktion zum **Aufsummieren** von Elementen einer Liste aus ganzen Zahlen

```
sum [ ]           = 0
sum (x:xs)        = sum xs + x
```

- und noch eine Variante: **Multiplizieren** aller Listenelemente

```
product [ ]       = 1
product (x:xs)    = product xs * x
```

- Beide Funktionen sind vordefiniert, aber intern iterativ realisiert.

Funktionen höherer Ordnung

```
sum :: [Int] -> Int
sum []      = 0
sum (x:xs) = x + sum xs
```

```
product :: [Int] -> Int
product []      = 1
product (x:xs) = x * product xs
```

Funktionen höherer Ordnung

```
sum :: [Int] -> Int
sum []      = 0
sum (x:xs) = x + sum xs
```

```
product :: [Int] -> Int
product []      = 1
product (x:xs) = x * product xs
```

Funktionen höherer Ordnung

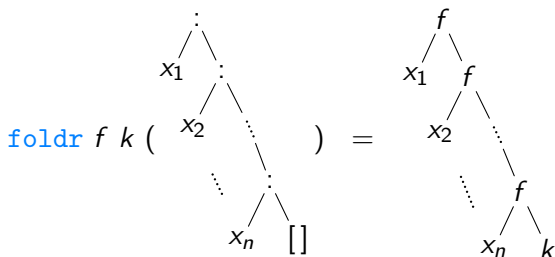
```
sum :: [Int] -> Int
sum []      = 0
sum (x:xs) = x + sum xs
```

```
product :: [Int] -> Int
product []      = 1
product (x:xs) = x * product xs
```

```
foldr :: (Int -> Int -> Int) -> Int -> [Int] -> Int
foldr f k []      = k
foldr f k (x:xs) = f x (foldr f k xs)
```

Funktionen höherer Ordnung

Berechnung mittels `foldr`:



Funktionen höherer Ordnung

```
data Tree = Node Int Tree Tree | Nil
```

```
insert :: Int -> Tree -> Tree
```

```
insert x Nil = Node x Nil Nil
```

```
insert x (Node key left right) = ...
```

Funktionen höherer Ordnung

```
data Tree = Node Int Tree Tree | Nil
```

```
insert :: Int -> Tree -> Tree
```

```
insert x Nil = Node x Nil Nil
```

```
insert x (Node key left right) = ...
```

```
buildTree :: [Int] -> Tree
```

```
buildTree [] = Nil
```

```
buildTree (x:xs) = insert x (buildTree xs)
```

Funktionen höherer Ordnung

```
data Tree = Node Int Tree Tree | Nil
```

```
insert :: Int -> Tree -> Tree
```

```
insert x Nil = Node x Nil Nil
```

```
insert x (Node key left right) = ...
```

```
buildTree :: [Int] -> Tree
```

```
buildTree [] = Nil
```

```
buildTree (x:xs) = insert x (buildTree xs)
```


Funktionen höherer Ordnung

```
data Tree = Node Int Tree Tree | Nil
```

```
insert :: Int -> Tree -> Tree
```

```
insert x Nil = Node x Nil Nil
```

```
insert x (Node key left right) = ...
```

```
buildTree :: [Int] -> Tree
```

```
buildTree [] = Nil
```

```
buildTree (x:xs) = insert x (buildTree xs)
```

```
foldr :: (a -> b -> b) -> b -> [a] -> b
```

```
foldr f k [] = k
```

```
foldr f k (x:xs) = f x (foldr f k xs)
```

Funktionen höherer Ordnung

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

```
buildTree :: [Int] -> Tree
buildTree xs = foldr insert Nil xs
```

```
sum :: [Int] -> Int
sum xs = foldr (+) 0 xs
```

```
product :: [Int] -> Int
product xs = foldr (*) 1 xs
```

...