

Compilerbau





Martin Plümicke
Andreas Stadelmeier

SS 2024





Beschreibung

In der Vorlesung werden anwendungsnahe Konzepte und Techniken zu Programmiersprachen und Compilerbau vermittelt. Konkret werden zunächst die Phasen des Compilerbaus an Hand eines Java-Compilers vorgestellt. Als Implementierungstechnik wird die funktionale Programmiersprache Haskell verwendet. Dazu werden die notwendigen Grundlagen der funktionalen Programmierung aufbauend auf den Kenntnissen der Grundvorlesung vermittelt. Im 2. Teil der Lehrveranstaltungen werden die Studierenden in Gruppenarbeit einen Mini-Java-Compiler mit den gelernten Techniken implementieren.

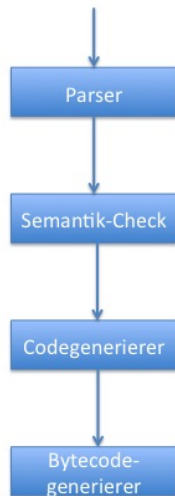
Literatur

-  Bauer and Höllerer.
Übersetzung objektorientierter Programmiersprachen.
Springer-Verlag, 1998, (in german).
-  Alfred V. Aho, Ravi Lam, Monica S.and Sethi, and Jeffrey D. Ullman.
Compiler: Prinzipien, Techniken und Werkzeuge.
Pearson Studium Informatik. Pearson Education Deutschland, 2.
edition, 2008.
(in german).
-  Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman.
Compilers Principles, Techniques and Tools.
Addison Wesley, 1986.
-  Reinhard Wilhelm and Dieter Maurer.
Übersetzerbau.
Springer-Verlag, 2. edition, 1992.
(in german).

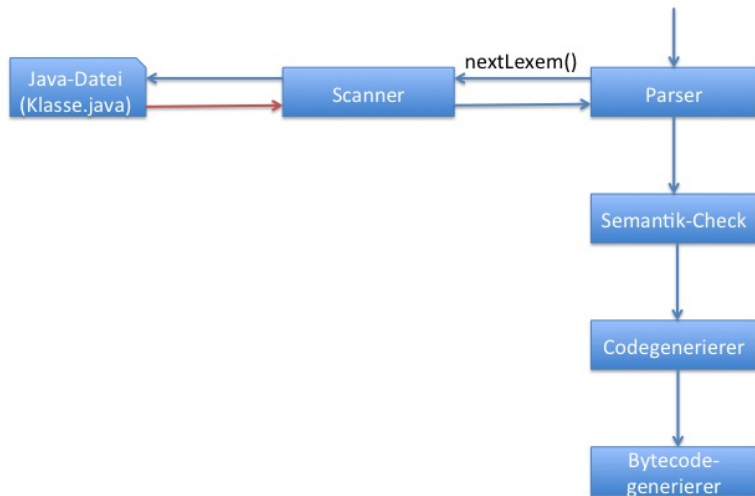
Literatur II

-  James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley.
The Java[®] Language Specification.
The Java series. Addison-Wesley, Java SE 8 edition, 2014.
-  Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley.
The Java[®] Virtual Machine Specification.
The Java series. Addison-Wesley, Java SE 8 edition, 2014.
-  Bryan O'Sullivan, Donald Bruce Stewart, and John Goerzen.
Real World Haskell.
O'Reilly, 2009.
-  Peter Thiemann.
Grundlagen der funktionalen Programmierung.
Teubner, 1994.

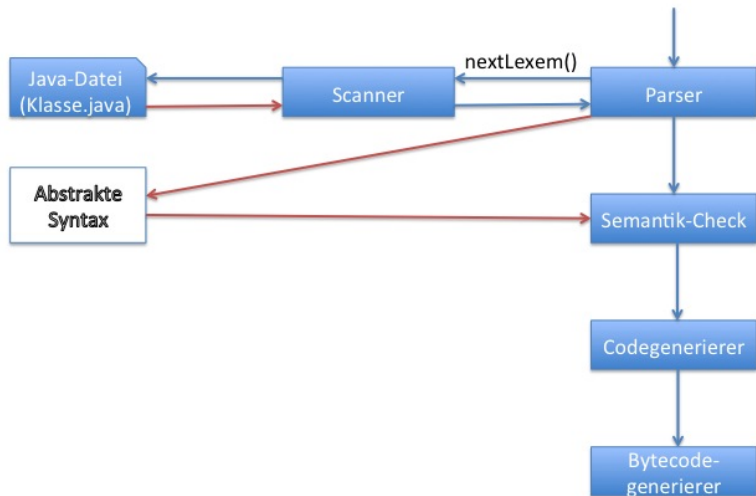
Compiler Überblick



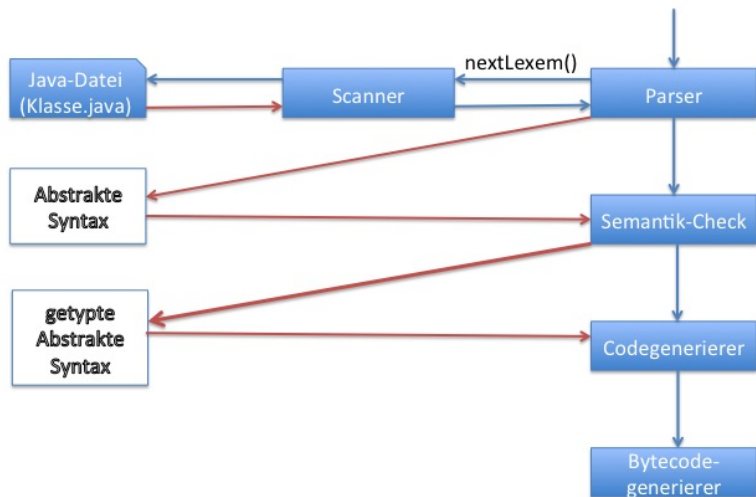
Compiler Überblick



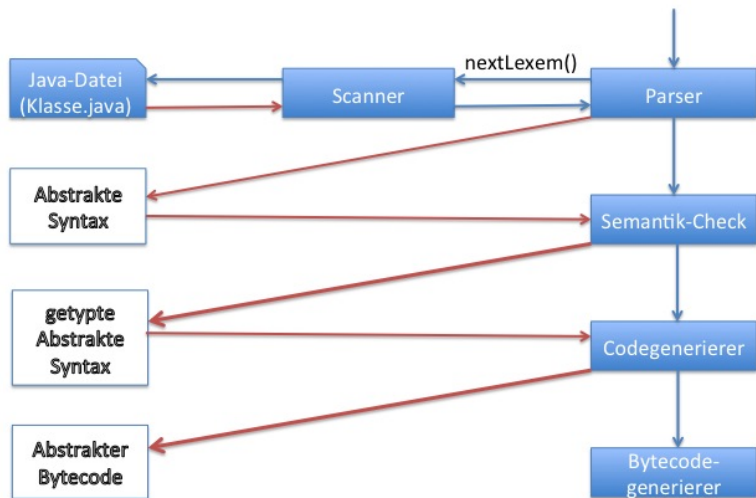
Compiler Überblick



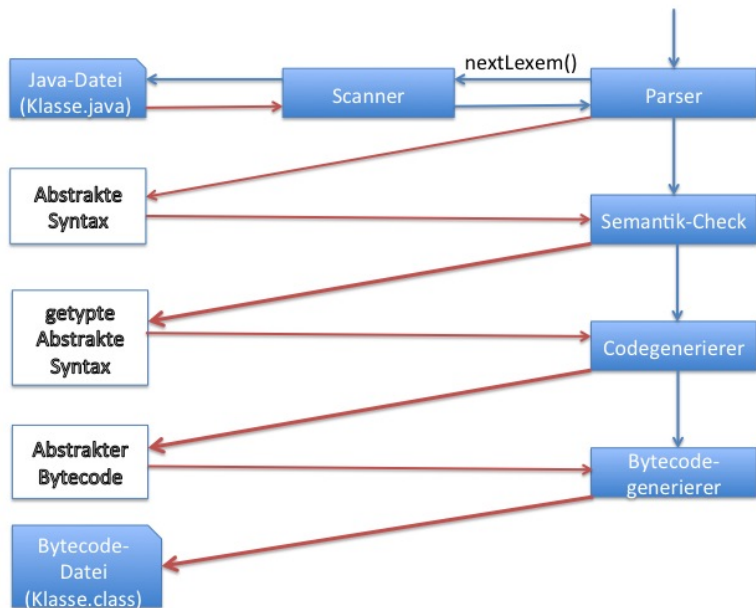
Compiler Überblick



Compiler Überblick



Compiler Überblick



Überblick Funktionale Programmierung

Einleitung

Funktionen

$$f : D \rightarrow W$$

- ▶ Definitionsbereich D
- ▶ Wertebereich W
- ▶ Abbildungsvorschrift: $x \mapsto f(x)$

Spezifikation als Funktion

Eingabe: Spezifikation des Definitionsbereichs

Ausgabe: Spezifikation des Wertebereichs

funktionaler Zusammenhang: Definition der Abbildungsvorschrift

1. Quadratfunktion

square : $\mathbb{Z} \rightarrow \mathbb{Z}$

square(x) = x^2

1. Quadratfunktion

square : $\mathbb{Z} \rightarrow \mathbb{Z}$
square(x) = x^2

Java:

```
int square(int x) {  
    return x^2;  
}
```

1. Quadratfunktion

square : $\mathbb{Z} \rightarrow \mathbb{Z}$
square(x) = x^2

Java:

```
int square(int x) {  
    return x^2;  
}
```

Haskell:

```
square :: Int -> Int  
square(x) = x^2
```

2. Maximumsfunktion

$$\max : \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z}$$

$$\max(x, y) = \begin{cases} x & x \geq y \\ y & \text{sonst} \end{cases}$$

2. Maximumsfunktion

$\max : \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z}$

$$\max(x, y) = \begin{cases} x & x \geq y \\ y & \text{sonst} \end{cases}$$

Java:

```
int max(int x, int y) {  
    if (x >= y) return x  
    else return y;  
}
```

2. Maximumsfunktion

$\max : \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z}$

$$\max(x, y) = \begin{cases} x & x \geq y \\ y & \text{sonst} \end{cases}$$

Java:

```
int max(int x, int y) {  
    if (x >= y) return x  
    else return y;  
}
```

Haskell:

```
maxi :: (Int, Int) -> Int  
maxi(x,y) | x >= y    = x  
          | otherwise = y
```

```
maxi(x, y) = if x >= y then x else y
```

3. Kreisfunktion

kreis : $[0, 2\pi] \rightarrow [-1, 1] \times [-1, 1]$

$x \mapsto (\cos(x), \sin(x))$

3. Kreisfunktion

kreis : $[0, 2\pi] \rightarrow [-1, 1] \times [-1, 1]$

$x \mapsto (\cos(x), \sin(x))$

Java:

```
class Kreis {  
    float a;  
    float b;  
  
    Kreis kreisfunktion(float x) {  
        Kreis k = new Kreis();  
        k.a = Math.cos(x);  
        k.b = Math.sin(x);  
        return k;}  
}
```

3. Kreisfunktion

kreis : $[0, 2\pi] \rightarrow [-1, 1] \times [-1, 1]$
 $x \mapsto (\cos(x), \sin(x))$

Java:

```
class Kreis {  
    float a;  
    float b;  
  
    Kreis kreisfunktion(float x) {  
        Kreis k = new Kreis();  
        k.a = Math.cos(x);  
        k.b = Math.sin(x);  
        return k;}  
}
```

Haskell:

```
kreis :: Float -> (Float,Float)  
kreis(x) = (cos(x), sin(x))
```

4. Vektorarithmetik

$$\begin{aligned} f : \mathbf{VR}(\mathbb{R}) \times \mathbf{VR}(\mathbb{R}) \times (\mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}) &\rightarrow \mathbf{VR}(\mathbb{R}) \\ ((v_1, \dots, v_n), (v'_1, \dots, v'_n), \oplus) &\mapsto ((v_1 \oplus v'_1), \dots, (v_n \oplus v'_n)) \end{aligned}$$

4. Vektorarithmetik (Java)

```
interface Arth {
    Double verkn (Double x, Double y);
}

class Vektorarithmetik extends Vector<Double> {

    Vektorarithmetik f (Vektorarithmetik v, Arth a) {
        Vektorarithmetik ret = new Vektorarithmetik();
        for (int i=0;i<v.size();i++) {
            ret.setElementAt(a.verkn(this.elementAt(i),
                                     v.elementAt(i)), i);
        }
        return ret;
    }
}
```

```
class Add implements Arth {  
    public Double verkn (Double x, Double y) {  
        return x + y;  
    }  
}
```

```
class Sub implements Arth {  
    public Double verkn (Double x, Double y) {  
        return x - y;  
    }  
}
```



```
class Main {
    public static void main(String[] args) {
        Vektorarithmetik v1 = new Vektorarithmetik();
        v1.addElement(1.0);v1.addElement(2.0);
        Vektorarithmetik v2 = new Vektorarithmetik();
        v2.addElement(3.0);v2.addElement(4.0);
        Add add = new Add();
        Sub sub = new Sub();
        System.out.println(v1.f(v2, add));
        System.out.println(v1.f(v2, sub));
    }
}
```

Java 8

```
class Main {
    public static void main(String[] args) {
        Vektorarithmetik v1 = new Vektorarithmetik();
        v1.addElement(1.0);v1.addElement(2.0);
        Vektorarithmetik v2 = new Vektorarithmetik();
        v2.addElement(3.0);v2.addElement(4.0);

        //nicht mehr notwendig
        //Add add = new Add();
        //Sub sub = new Sub();
        //System.out.println(v1.f(v2, add));
        //System.out.println(v1.f(v2, sub));

        //Lambda-Expressions
        System.out.println(v1.f(v2, (x,y) -> x+y));
        System.out.println(v1.f(v2, (x,y) -> x-y));
    }
}
```

4. Vektorarithmetik (Haskell)

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

4. Vektorarithmetik (Haskell)

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

```
f([], y, g) = []
```

```
f((v : vs), (w : ws), g) = (g(v,w)) : (f (vs, ws, g))
```

5. Addition einer Konstanten

$$\begin{aligned} \text{addn} &: \mathbb{N} \rightarrow (\mathbb{N} \rightarrow \mathbb{N}) \\ n &\mapsto (x \mapsto x + n) \end{aligned}$$

5. Addition einer Konstanten (bis Java-7)

```
class addn {  
    int n;  
  
    addn(int n) {  
        this.n = n;  
    }  
  
    static addn add1(int n) {  
        return new addn(n);  
    }  
  
    int add2(int x) {  
        return n + x;  
    }  
}
```

```
public static void main(String[] args) {  
    System.out.println(add1(5).n);  
    System.out.println(add1(5).add2(4));  
}  
}
```

5. Addition einer Konstanten (Java-8)

```
interface Fun1<A,R> {  
    R apply(A arg);  
}
```

```
class Main {  
  
    Fun1<Integer, Integer> addn(int n) {  
        return x -> x + n;  
    }  
}
```


5. Addition einer Konstanten (Haskell)

```
addn :: Int -> (Int -> Int)
addn(n) = \x -> x + n
```

Grundlegende Eigenschaften Funktionaler Sprachen

1. Keine Seiteneffekte

Wird eine Funktion mehrfach auf das **gleiche Argument** angewandt, so erhält man **IMMER** das **gleiche Ergebnis**.

Grundlegende Eigenschaften Funktionaler Sprachen

2. Verzögerte Auswertung

$f(x) = f(x)$ (* rekursiver Ausruf *)

$g(x, y) = y+1$

Grundlegende Eigenschaften Funktionaler Sprachen

2. Verzögerte Auswertung

$f(x) = f(x)$ (* rekursiver Aufruf *)

$g(x, y) = y+1$

Was passiert beim Aufruf

$g(f(2), 2)$

Grundlegende Eigenschaften Funktionaler Sprachen

3. Polymorphes Typsystem

```
datatype Folge(A);
```

```
sorts A, Folge;
```

```
constructors
```

```
empty: → Folge;
```

```
cons : A × Folge → Folge;
```

```
operations
```

```
head : Folge → A;
```

```
tail : Folge → Folge;
```

```
is_empty: Folge → Boolean;
```

```
cat: Folge × Folge → Folge;
```

```
len: Folge → N;
```

Grundlegende Eigenschaften Funktionaler Sprachen

4. Automatische Speicherverwaltung

Die Programmierung von Speicherverwaltung entfällt. Die Speicher-Allocation und Konstruktion von Datenobjekten und die Freigabe von Speicherplatz (garbage-collection) geschieht ohne Einwirkung des Programmierers.

Grundlegende Eigenschaften Funktionaler Sprachen

5. Funktionen als Bürger 1. Klasse

Funktionen können sowohl als **Argumente** als auch als **Rückgabewerte** von **Funktionen** verwendet werden. (Vgl. Beispiele *Vektorarithmetik* und *Addition einer Konstanten*)

Deklarationen von Funktionen in Haskell

Def. und Wertebereich

```
vars    → var_1 , ..., var_n
```

```
fundecl → vars :: type
```

```
type → btype [→ type]      (function type)
```

```
btype → [btype] atype      (type application)
```

```
atype → qtycon
```

```
  | tyvar
```

```
  | "(" type_1, ..., type_k ")" (tuple type, k>=2)
```

```
  | "[" type "]"                (list type)
```

```
  | "(" type ")"                (parenthesized
```

```
    constructor)
```

```
(Haskell-Grammatik: https://www.haskell.org/onlinereport/syntax-iso.html)
```


Beispiele:

```
square:: int → int
```

```
maxi:: (int, int) → int
```

Deklarationen von Funktionen in Haskell

Abbildungsvorschrift

```
fundecl  -> funlhs rhs

funlhs   -> var apat { apat }

apat     -> var [@ apat]           (as pattern)
         | literal
         | _                       (wildcard)
         | "(" pat ")"            (parenthesized pattern)
         | "(" pat1, ... , patk ")" (tuple pattern, k>=2)
         | "[" pat1, ... , patk "]" (list pattern, k>=1)

rhs      -> = exp
         | guardrhs

guardrhs -> gd = exp [gdrhs]
gd       -> "|" exp
```

Deklarationen von Funktionen in Haskell

Expressions

```
expr → \apat_1...apat_n → expr (lambda abstraction,
                                n>=1)
  | let decls in expr          (let expression)
  | if expr then expr else expr (conditional)
  | case expr of { alts }      (case expression)
  | do { stmts }                (do expression)
  | fexp
fexp → [fexp] aexp              (function application)

alts → alt_1 ; ... ; alt_n      (n>=1)
alt  → pat → expr
```

```
aexp → qvar          (variable)
      | gcon          (general constructor)
      | literal
      | "(" expr ")"  (parenthesized expression)
      | "(" expr_1, ... , expr_k ")" (tuple, k>=2)
      | "[" expr_1, ... , expr_k "]" (list, k>=1)
```

```
literal → integer | float | char | string
```

Beispiele:

square $x = x^2$

Beispiele:

```
square x = x^2
```

1. Variante:

```
maxi(x,y) = if x > y then x else y
```

2. Variante (guarded equations):

```
maxi(x,y) | x > y      = x  
          | otherwise = y
```

Pattern-Matching

Vordefinierter Typ [a]

[] steht für leere Liste

: steht für den Listenkonstruktor

```
head :: [a] -> a
```

```
tail :: [a] -> [a]
```

```
head(x : xs) = x
```

```
tail(x : xs) = xs
```

Pattern-Matching

Vordefinierter Typ [a]

[] steht für leere Liste

: steht für den Listenkonstruktor

```
head :: [a] -> a  
tail :: [a] -> [a]
```

```
head(x : xs) = x  
tail(x : xs) = xs
```

Pattern-Matching

```
head(1 : 2 : 3 : 4 : []) = 1  
tail(1 : 2 : 3 : 4 : []) = 2 : 3 : 4 : []
```


let/where-Konstrukte

```
len: [a] -> int
len(x) = let
    len0([], n) = n
    len0(x:xs, n) = len0(xs, n+1)
in
    len0(x, 0)
```

let/where-Konstrukte

```
len: [a] -> int
len(x) = let
    len0([], n) = n
    len0(x:xs, n) = len0(xs, n+1)
in
    len0(x, 0)
```

```
len(x) = len0(x, 0)
where len0([], n) = n
      len0(x:xs, n)
          = len0(xs, n+1)
```

Namenlose Funktionen

```
addn :: Int -> (Int -> Int)
addn n = \x -> x+n
```

Datentypen

Abkürzungen mit `type`

```
type String = [Char]
type Floatpair = (float, float)
```

Datentypen

Algebraische Datentypen

```
datadec1  -> data [context =>] simpletype  
          = constrs [deriving]
```

```
simpletype -> tycon tyvar_1 ... tyvar_k (k>=0)
```

```
constrs   -> constr_1 | ... | constr_n (n>=1)
```

```
constr    -> con [!] atype_1 ... [!] atype_k (arity con = k,  
                                             k>=0)  
          | con { fielddecl_1 , ... , fielddecl_n } (n>=0)
```

```
fielddecl -> vars :: (type | ! atype)
```

```
deriving  -> deriving (dclass |  
                    (dclass_1, ... , dclass_n)) (n>=0)
```

Beispiel:

```
data Folge a = Empty
             | Cons (a , Folge a)
```

$$T_{\text{FolgeInt}} =$$
$$\{ \text{Empty}, \text{Cons}(1, \text{Empty}), \text{Cons}(1, \text{Cons}(1, \text{Empty})), \dots \}$$
$$\text{Cons}(2, \text{Empty}), \text{Cons}(1, \text{Cons}(2, \text{Empty})),$$
$$\text{Cons}(3, \text{Empty}), \text{Cons}(1, \text{Cons}(3, \text{Empty})),$$
$$\dots \quad \dots$$

head und tail über dem Datentyp Folge

```
head :: Folge(a) -> a
tail :: Folge(a) -> Folge(a)
```

```
head(Cons(x, xs)) = x
```

```
tail(Cons(x, xs)) = xs
```

Pattern-Matching:

```
head(Cons(1, Cons(2, Empty))) = 1
```

```
tail(Cons(1, Cons(2, Empty))) = Cons(2, Empty)
```

Funktionen höherer Ordnung

Funktion als Argument:

$$(\tau \rightarrow \tau') \rightarrow \tau''$$

Funktion als Ergebnis:

$$\tau' \rightarrow (\tau' \rightarrow \tau'')$$

Currying

Satz: Sei $f : (\tau_1, \dots, \tau_n) \rightarrow \tau$ eine Funktion mit $f(a_1, \dots, a_n) = a$. Dann gibt es genau eine Funktion

$$f' : \tau_1 \rightarrow (\tau_2 \rightarrow (\dots (\tau_n \rightarrow \tau) \dots))$$

mit für alle a_i, a

$$(\dots (((f' a_1) a_2) a_3) \dots a_n) = a.$$

Currying Beispiel

```
curry :: ((a,b) -> c) -> (a -> (b -> c))  
curry f = \x -> (\y -> f(x,y))
```

Currying Beispiel

```
curry :: ((a,b) -> c) -> (a -> (b -> c))  
curry f = \x -> (\y -> f(x,y))
```

```
uncurry :: (a -> (b -> c)) -> ((a,b) -> c)  
uncurry f = \ (x, y) -> ((f x) y)
```

Konventionen

Für

$$\tau_1 \rightarrow (\tau_2 \rightarrow (\tau_3 \rightarrow \dots \tau_n) \dots)$$

schreibt man

$$\tau_1 \rightarrow \tau_2 \rightarrow \tau_3 \rightarrow \dots \tau_n.$$

Konventionen

Für

$$\tau_1 \rightarrow (\tau_2 \rightarrow (\tau_3 \rightarrow \dots \tau_n) \dots)$$

schreibt man

$$\tau_1 \rightarrow \tau_2 \rightarrow \tau_3 \rightarrow \dots \tau_n.$$

Für

$$(\dots(((f(a_1))(a_2))(a_3))\dots)(a_n)$$

schreibt man

$$f a_1 a_2 a_3 \dots a_n.$$

map

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

map

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

```
map f [] = []
```

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

Bsp.:

```
square :: int -> int  
square x = x*x
```

```
qu :: int -> int  
qu x = x * x * x
```


Bsp.:

```
square :: int -> int  
square x = x*x
```

```
qu :: int -> int  
qu x = x * x * x
```

```
sqlist :: [int] -> [int]  
sqlist li = map square li
```

```
qulist :: [int] -> [int]  
qulist li = map qu li
```

fold

Gegeben: $[a_1, \dots, a_n]$

Verknüpfung:

rechtsassoziativ:

$$a_1 \oplus (a_2 \oplus (\dots (a_{n-1} \oplus a_n) \dots))$$

fold

Gegeben: $[a_1, \dots, a_n]$

Verknüpfung:

rechtsassoziativ:

$$a_1 \oplus (a_2 \oplus (\dots (a_{n-1} \oplus a_n) \dots))$$

linksassoziativ:

$$(\dots ((a_1 \oplus a_2) \oplus a_3) \dots) \oplus a_n$$

foldr

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

foldr

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

```
foldr f e [] = e
```

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

foldl

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

foldl

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

```
foldl f e [] = e
```

```
foldl f e (x : xs) = foldl f (f e x) xs
```

fold Beispiele

```
sum, prod :: [int] -> int
```

```
sum = foldr (+) 0
```

```
prod = foldl (*) 1
```


fold Beispiele

```
sum, prod :: [int] -> int
```

```
sum = foldr (+) 0
```

```
prod = foldl (*) 1
```

```
foldr (^) 1 [4,3,2] = ?
```

```
foldl (^) 1 [4,3,2] = ?
```

fold Beispiele

```
sum, prod :: [int] -> int
```

```
sum = foldr (+) 0
```

```
prod = foldl (*) 1
```

```
foldr (^) 1 [4,3,2] = 262144
```

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

I/O über die Konsole

```
main = do
  putStrLn "Hallo! Wie heißen Sie? "
  inpStr <- getLine
  putStrLn $ "Willkommen bei Haskell, " ++
    inpStr ++ "!"
```

I/O über die Konsole

```
main = do
  putStrLn "Hallo! Wie heissen Sie? "
  inpStr <- getLine
  putStrLn $ "Willkommen bei Haskell, " ++
    inpStr ++ "!"
```

Ausführen

```
pl@martin-pluemickes-macbook.local% runhaskell IO.hs
Hallo! Wie heissen Sie?
Martin
Willkommen bei Haskell. Martin!
```

Das Modul System I/O

```
openFile :: FilePath -> IO Mode -> IO Handle
hgetChar :: Handle -> IO Char
hgetLine :: Handle -> IO String
hIsEOF   :: Handle -> IO Bool
hPutStr  :: Handle -> String -> IO ()
hPutStrLn :: Handle -> String -> IO ()
hClose   :: Handle -> IO()
```

File-Handling

```
import System.IO
import Data.Char(toUpper)

main = do
  inh <- openFile "input.txt" ReadMode
  outh <- openFile "output.txt" WriteMode
  mainloop inh outh
  hClose inh
  hClose outh
```

File-Handling II

```
mainloop :: Handle -> Handle -> IO ()
mainloop inh outh =
  do ineof <- hIsEOF inh
     if ineof then return ()
        else do inpStr <- hGetLine inh
               hPutStrLn outh (map toUpper inpStr)
               mainloop inh outh
```

Stdin/Stdout

```
import System.IO
import Data.Char(toUpper)

main = mainloop stdin stdout

mainloop :: Handle -> Handle -> IO ()
mainloop inh outh =
    do ineof <- hIsEOF inh
       if ineof then return ()
       else do inpChar <- hGetChar inh
              hPutChar outh inpChar
              mainloop inh outh
```