Compilerbau

Martin Plümicke

WS 2021/22

Agenda

Überblick Vorlesung

- Literatur

Compiler Überblick

Compiler (I)

- Scanner
- Parser

Überblick Funktionale Programmierung

- Einleitung
- Haskell-Grundlagen
- Scanner in Funktionalen Sprachen
- Parser in Funktionalen Sprachen

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

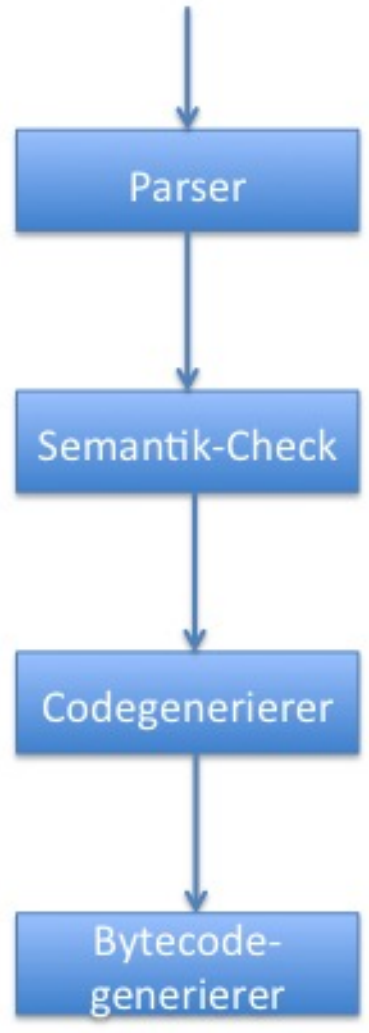

James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley.
The Java® Language Specification.
The Java series. Addison-Wesley, Java SE 8 edition, 2014.


Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley.
The Java® Virtual Machine Specification.
The Java series. Addison-Wesley, Java SE 8 edition, 2014.

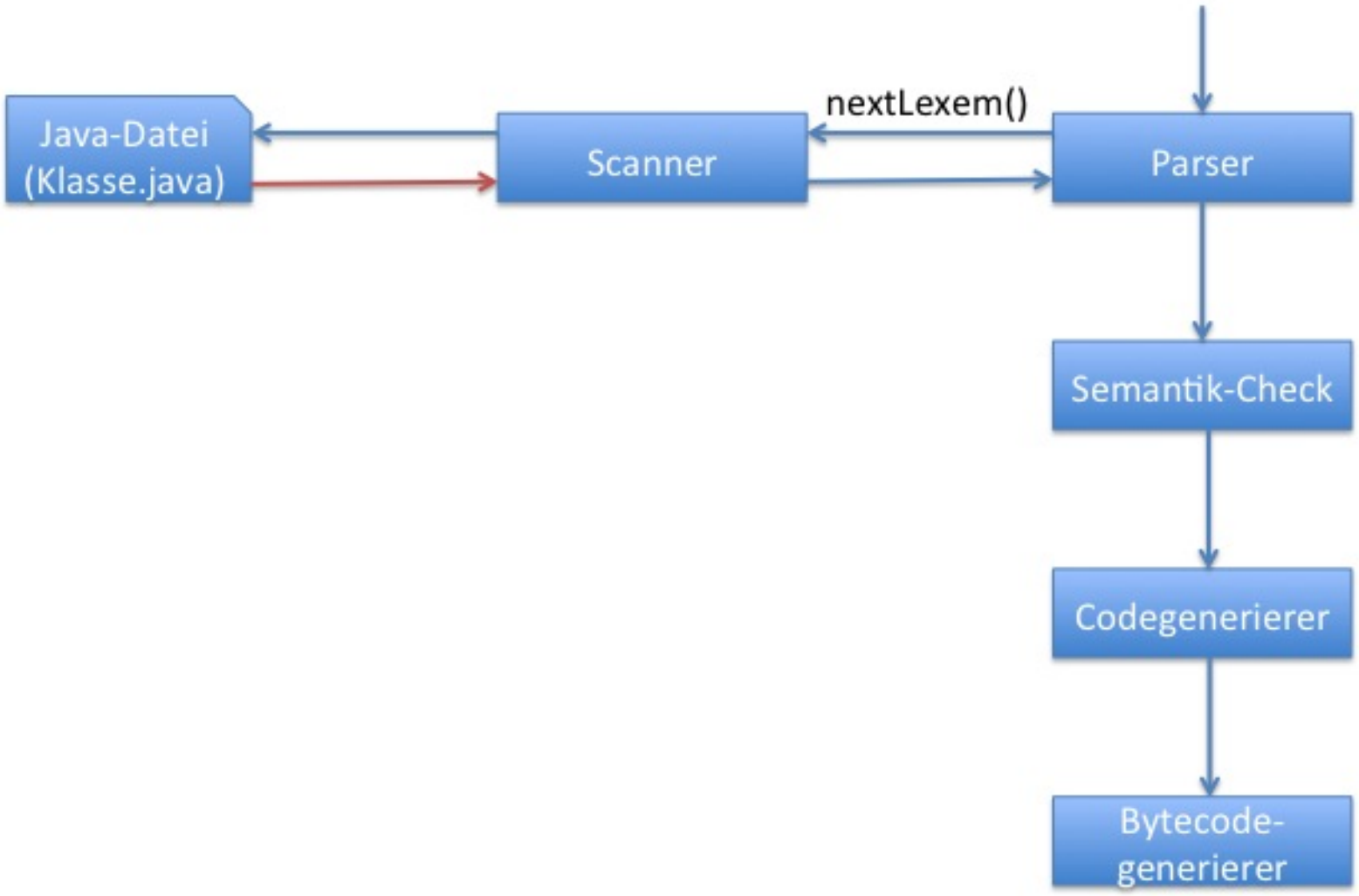

Bryan O'Sullivan, Donald Bruce Stewart, and John Goerzen.
Real World Haskell.
O'Reilly, 2009.


Peter Thiemann.
Grundlagen der funktionalen Programmierung.
Teubner, 1994.

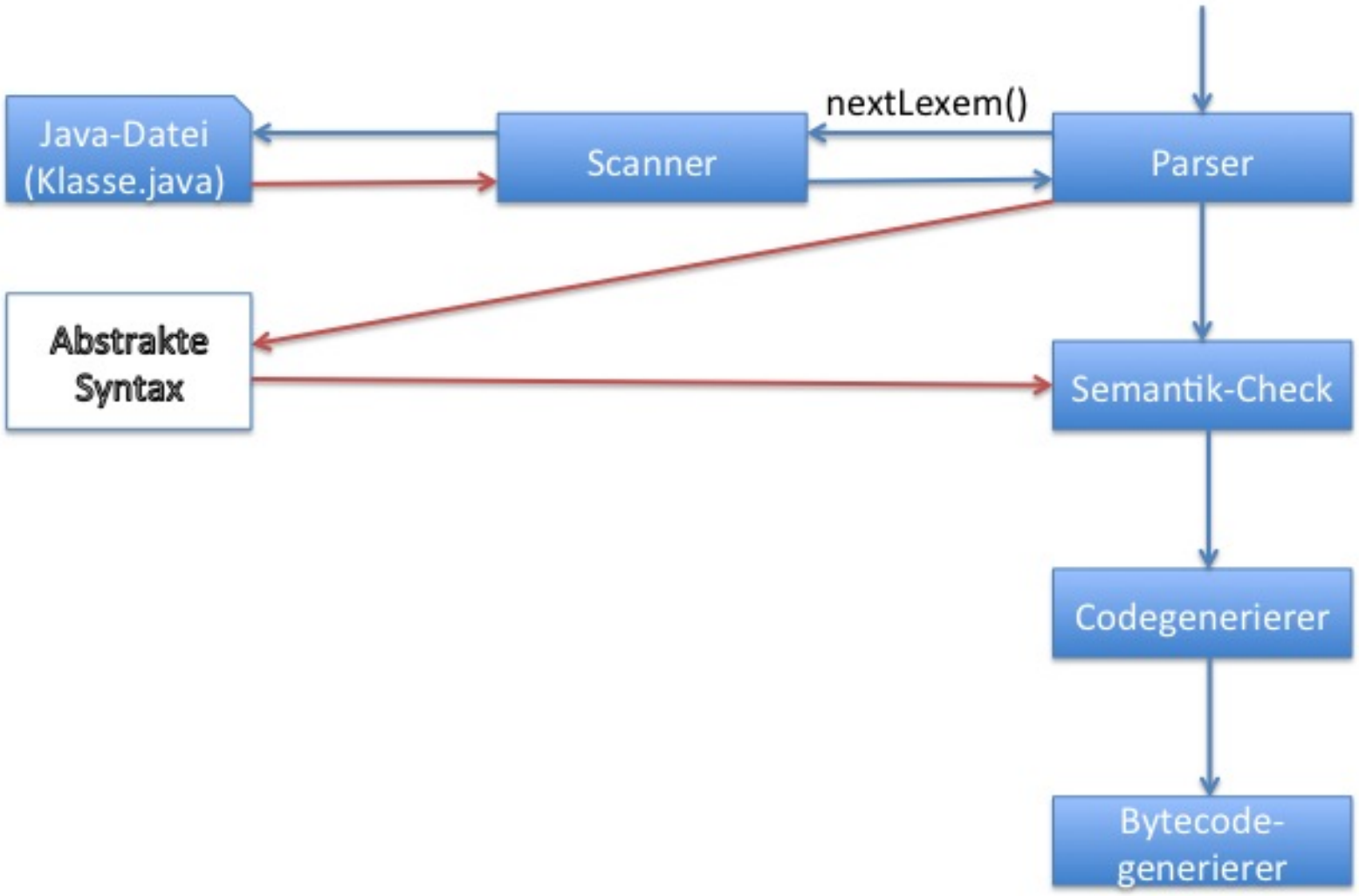
Compiler Überblick



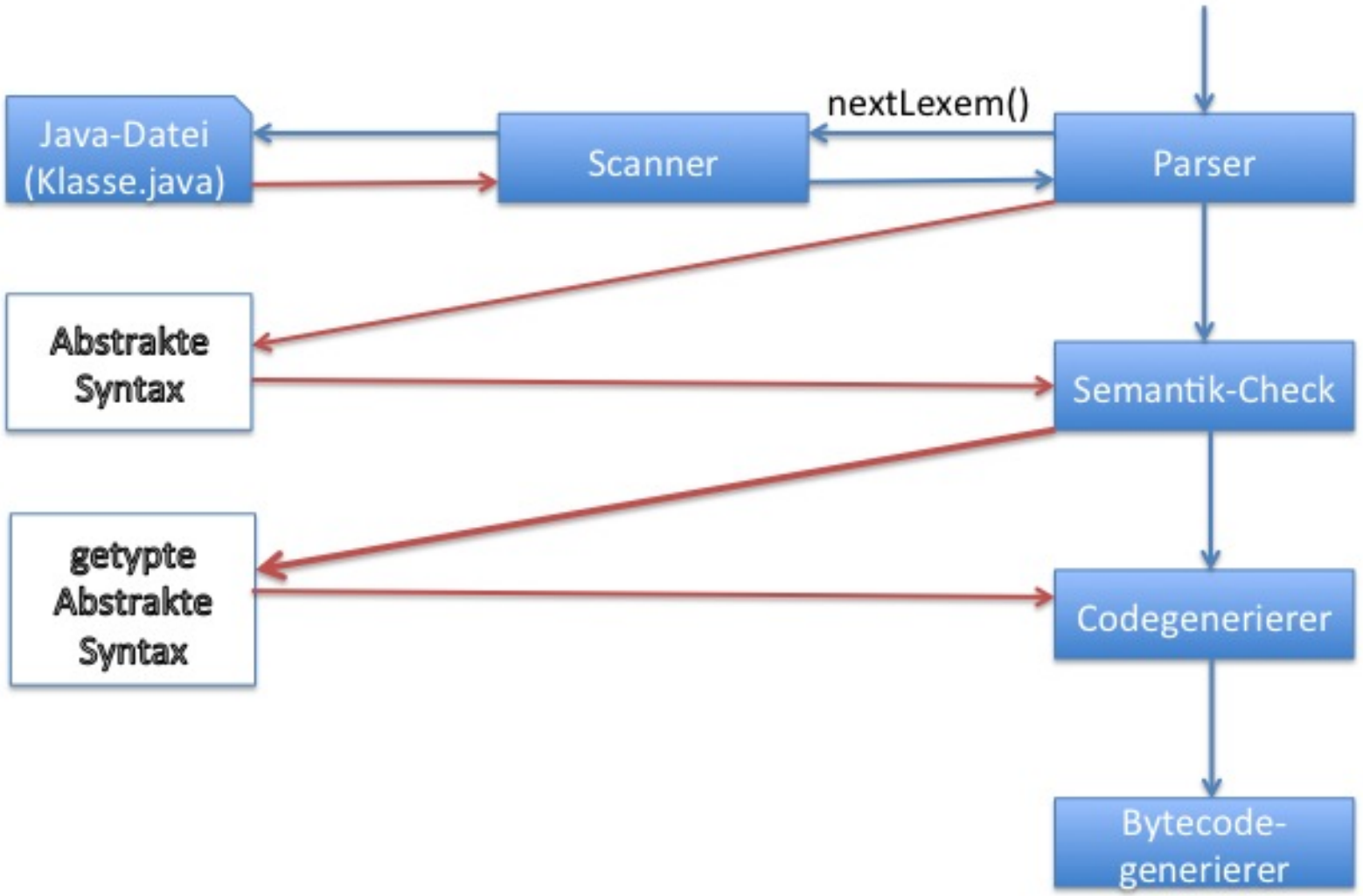
Compiler Überblick



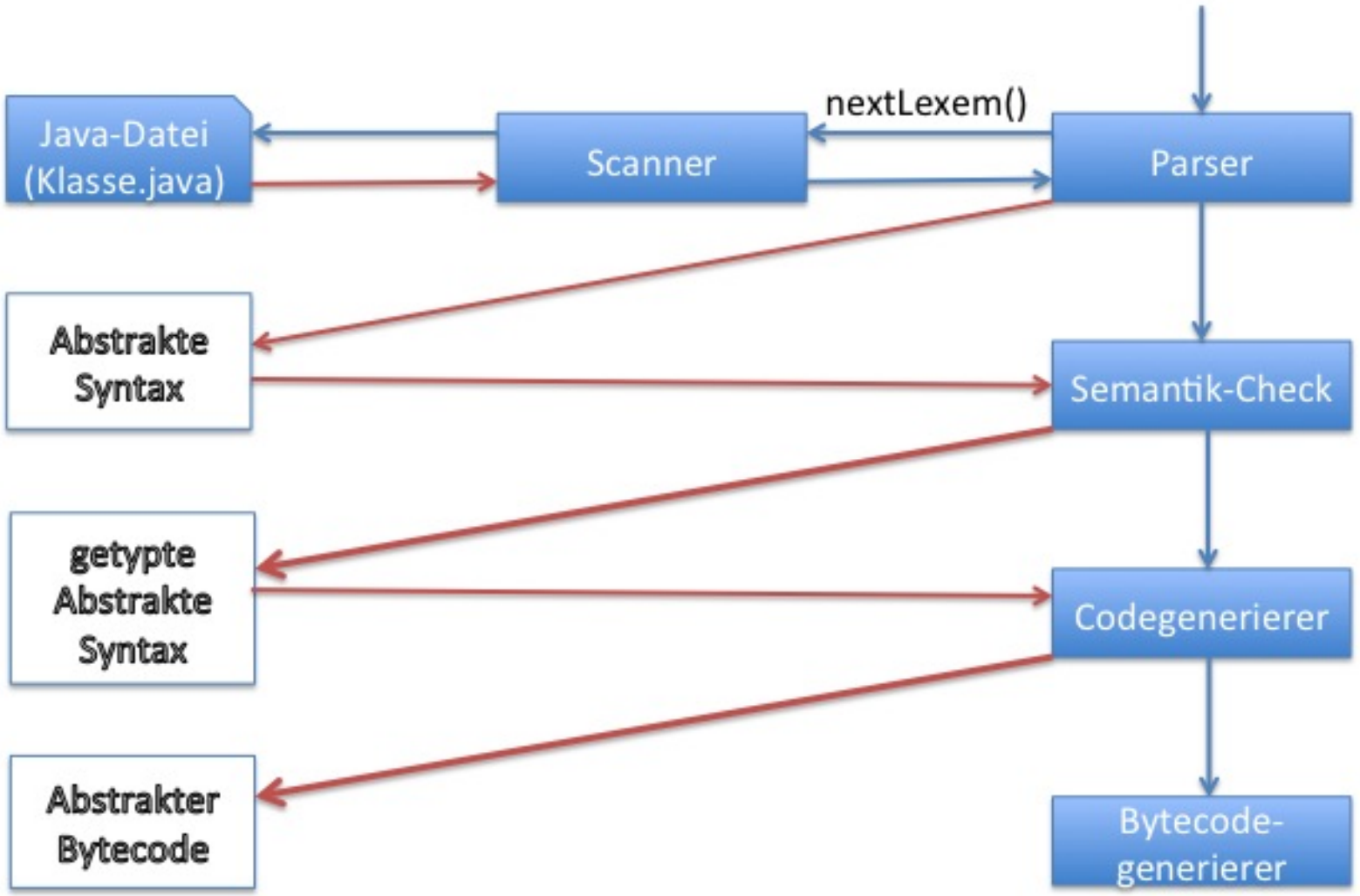
Compiler Überblick



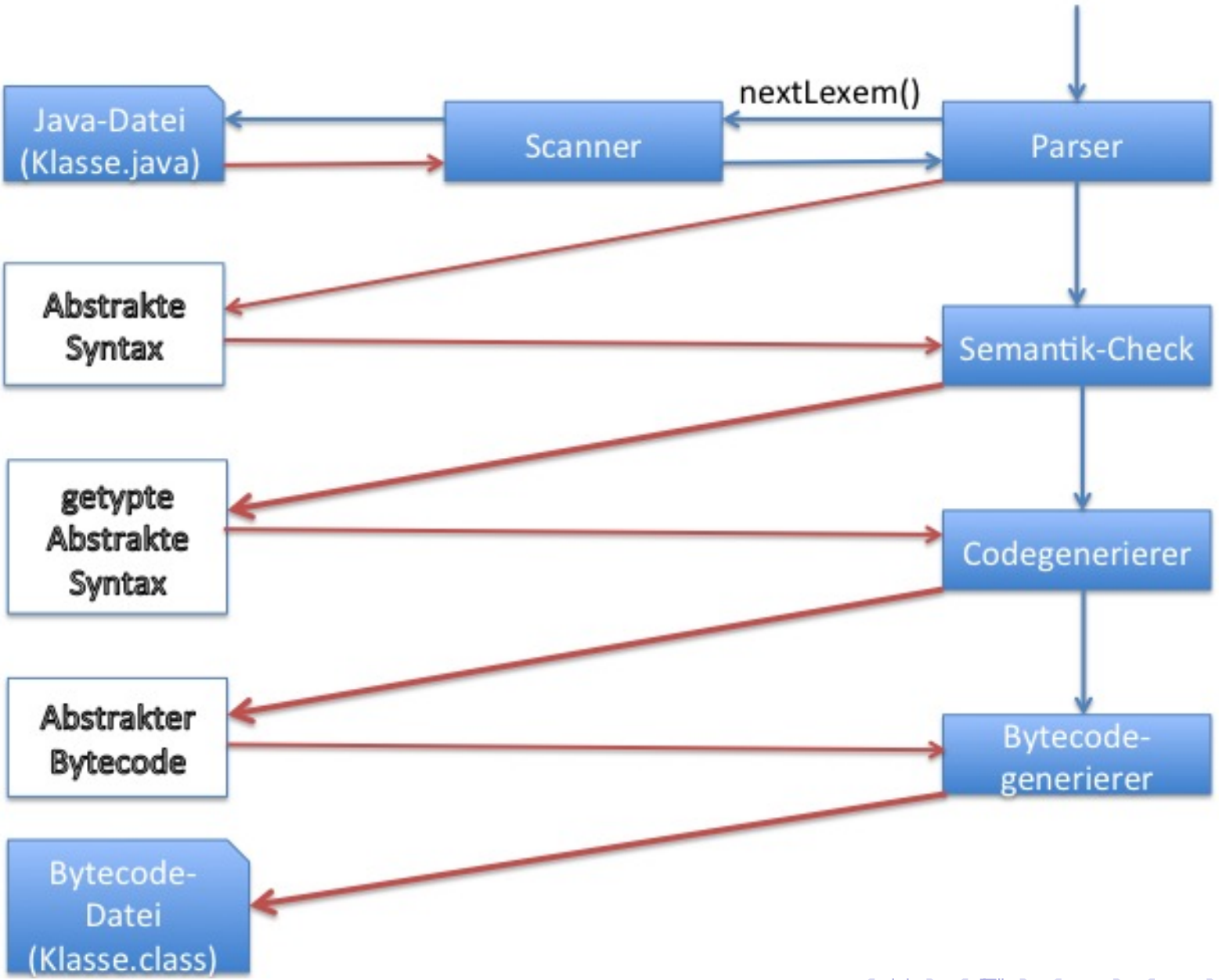
Compiler Überblick



Compiler Überblick



Compiler Überblick



Überblick Funktionale Programmierung
Einleitung

Funktionen

$$f : D \to W$$

- 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 : $\mathbb{Z} \to \mathbb{Z}$
square$(\, x \,) = x^2$

1. Quadratfunktion

$\mathsf{square} : \mathbb{Z} \to \mathbb{Z}$
$\mathbf{square}(\, x \,) = x^2$

Java:

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

1. Quadratfunktion

$\mathsf{square} : \mathbb{Z} \to \mathbb{Z}$
$\mathbf{square}(\, x \,) = x^2$

Java:

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

Haskell:

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

2. Maximumsfunktion

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

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

2. Maximumsfunktion

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

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

Java:

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

2. Maximumsfunktion

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

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

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

$\text{kreis} : [0, 2\pi] \to [-1, 1] \times [-1, 1]$

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

3. Kreisfunktion

$\text{kreis} : [0, 2\pi] \to [-1, 1] \times [-1, 1]$
$x \mapsto (cos(x), sin(x))$

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

$\text{kreis} : [0, 2\pi] \to [-1, 1] \times [-1, 1]$
$x \mapsto (cos(x), sin(x))$

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

$$\mathsf{f} : \mathbf{VR}(\,\mathbb{R}\,) \times \mathbf{VR}(\,\mathbb{R}\,) \times (\mathbb{R} \times \mathbb{R} \to \mathbb{R}) \to \mathbf{VR}(\,\mathbb{R}\,)$$
$$((v_1, \ldots, v_n), (v'_1, \ldots, v'_n), \oplus) \mapsto ((v_1 \oplus v'_1), \ldots, (v_n \oplus v'_n))$$

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

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

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 Ausruf *)

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
```

b

```
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–Kostrukte

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

let/where–Kostrukte

```
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

```
datadecl    -> 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)
```

$T_{\texttt{FolgeInt}} =$

$\{$`Empty`, `Cons(1, Empty)`, , `Cons(1, Cons(1, Empty))`, , ...$\}$

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

`Cons(3, Empty)`, `Cons(1, Cons(3, Empty))`,

... ...

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:

$$(\tau \to \tau') \to \tau''$$

Funktion als Ergebnis:

$$\tau' \to (\tau' \to \tau'')$$

Currying

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

$$f' : \tau_1 \to (\tau_2 \to (\ldots (\tau_n \to \tau) \ldots)$$

mit für alle a_i, a

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

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

$$\tau_1 \to (\tau_2 \to (\tau_3 \to \ldots \tau_n)\ldots)$$

schreibt man

$$\tau_1 \to \tau_2 \to \tau_3 \to \ldots \tau_n.$$

Konventionen

Für

$$\tau_1 \to (\tau_2 \to (\tau_3 \to \ldots \tau_n)\ldots)$$

schreibt man

$$\tau_1 \to \tau_2 \to \tau_3 \to \ldots \tau_n.$$

Für

$$(\ldots(((f(\,a_1\,))(\,a_2\,)\,)(\,a_3\,)\,)\ldots)(\,a_n\,)$$

schreibt man

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

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: $[a_1, \ldots, a_n]$

Verknüpfung:

rechtsassoziativ:

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

fold

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

Verknüpfung:

rechtsassoziativ:

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

linksassoziativ:

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

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

```
openeFile :: 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
```

Klassendeklarationen I

```
classdecl   -> class [scontext =>] tycls tyvar
               [where cdecls]

scontext    -> simpleclass
            |  (simpleclass_1,..., simpleclass_n)
                                              (n>=0)

simpleclass -> qtycls tyvar

cdecls      ->  { cdecl_1 ; ... ; cdecl_n } (n>=0)

cdecl       -> vars :: [context =>] type
            |  (funlhs | var) rhs
```

Klassendeklarationen II

```
context --> cls
        | ( class_1 , ... , class_n ) (n>=0)

cls --> qtycls tyvar
    |  qtycls ( tyvar atype_1 ... atype_n ) (n>=1)

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   --> = expr
```

Instance-Deklarationen

```
instancedecl → instance [scontext =>] qtycls inst
               [where idecls]

inst → gtycon
     | (gtycon tyvar1 ... tyvark) (k>=0, tyvars
                                   distinct)
     | (tyvar1,..., tyvark)       (k>=2, tyvars
                                   distinct)
     | [ tyvar ]
     | ( tyvar1 → tyvar2 )        (tyvars
                                   distinct)

idecls → { idecl1 ; ... ; idecln } (n>=0)
idecl  → (funlhs | var) rhs
       |                           (empty)
```

Ord a

```
class (Eq a) => Ord a where
    compare                 :: a -> a -> Ordering
    (<), (<=), (>=), (>)    :: a -> a -> Bool
    max, min                :: a -> a -> a

    compare x y | x==y       = EQ
                | x<=y       = LT
                | otherwise = GT

-- Minimal complete definition: (<=) or compare
-- using compare can be more efficient for complex
-- types
    x <= y = compare x y /= GT
    x <  y = compare x y == LT
    x >= y = compare x y /= LT
    x >  y = compare x y == GT
```

```
max x y   | x <= y      = y
          | otherwise   = x
min x y   | x <= y      = x
          | otherwise   = y
```

Typklasse Show

```
type  ShowS   = String -> String

class  Show a  where
    showsPrec :: Int -> a -> ShowS
    show      :: a -> String
    showList  :: [a] -> ShowS

    showsPrec _ x s   = show x ++ s
    show x            = showsPrec 0 x ""
-- ... default decl for showList given in Prelude

shows   :: (Show a) => a -> ShowS
shows   =  showsPrec 0
```

Instanz Show für Folge a

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

instance Show a => Show (Folge a) where
    show f = "[" ++ (show1 f) where
        show1 Empty = "]"
        show1 (Cons(e, Empty)) = (show e) ++ "]"
        show1 (Cons(e, es)) = (show e) ++ ","
            ++ (show1 es)
```

Typklassen Read

```
type  ReadS a = String -> [(a,String)]

class  Read a  where
    readsPrec :: Int -> ReadS a
    readList  :: ReadS [a]
-- ... default decl for readList given in Prelude
```

Instanz Read für Folge a

```
instance Read a => Read (Folge a) where
    readsPrec _ inp = let [(x,y)] = lex inp in
      case x of
      "[" -> readsPrec 0 y
      "]" -> readsPrec 0 y
      "," -> readsPrec 0 y
      " " -> readsPrec 0 y
      "" -> [(Empty, "")]
      x ->  let
                [(res, "")] = readsPrec 0 y i
            in
                [((Cons(read x, res)), "")]
```

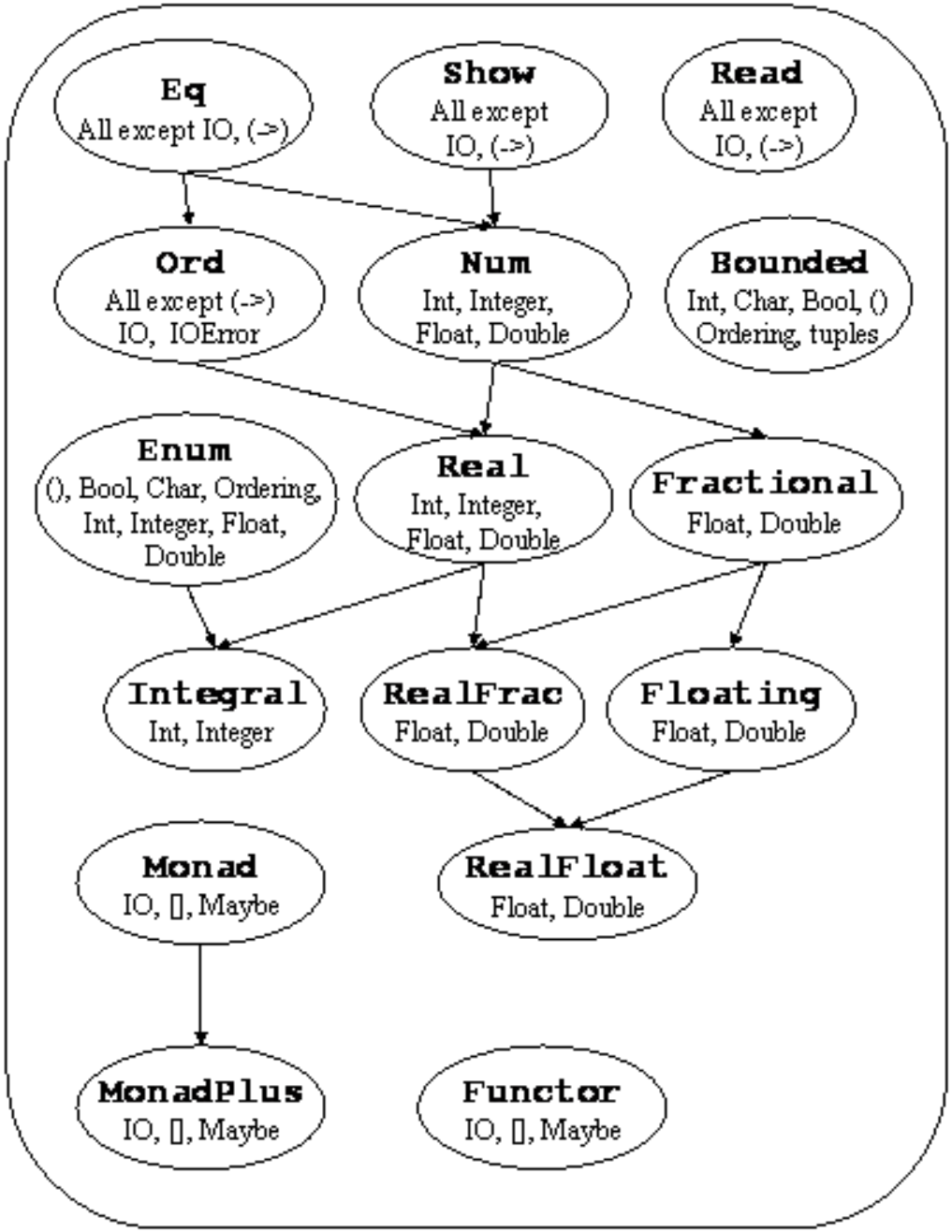
read :: (Read a) => String -> a

```
type  ReadS a = String -> [(a,String)]


reads   :: (Read a) => ReadS a
reads   =  readsPrec 0

read    :: (Read a) => String -> a
read s  =  case [x | (x,t) <- reads s,
                     ("","") <- lex t] of
   [x] -> x
   []  -> error "PreludeText.read: no parse"
   _   -> error "PreludeText.read: ambiguous parse"
```

Standard Haskell class hierarchry



Standard Haskell class hierarchry II

```
class  (Eq a, Show a) => Num a  where
    (+), (−), (*)  :: a −> a −> a
    negate         :: a −> a
    abs, signum    :: a −> a
    fromInteger    :: Integer −> a

class  (Num a, Ord a) => Real a  where
    toRational ::  a −> Rational

class  (Real a, Enum a) => Integral a  where
    quot, rem, div, mod :: a −> a −> a
    quotRem, divMod     :: a −> a −> (a,a)
    toInteger           :: a −> Integer
```

```
class  (Num a) => Fractional a  where
    (/)          :: a -> a -> a
    recip        :: a -> a
    fromRational :: Rational -> a

class  (Fractional a) => Floating a  where
    pi                  :: a
    exp, log, sqrt      :: a -> a
    (**), logBase       :: a -> a -> a
    sin, cos, tan       :: a -> a
    asin, acos, atan    :: a -> a
    sinh, cosh, tanh    :: a -> a
    asinh, acosh, atanh :: a -> a
```

Typklasse Monade

```
class Applicative m => Monad m  where
    (>>=)   :: m a -> (a -> m b) -> m b
    (>>)    :: m a -> m b -> m b
    return  :: a -> m a
    fail    :: String -> m a

    m >> k  =  m >>= \_ -> k
    fail s  = error s
```

1. Ansatz: Scanner direkt implementieren I

```
module Lexer (Token(..),lexer) where

import Data.Char

data Token = LetToken
           | InToken
           | SymToken Char
           | VarToken String
           | IntToken Int
  deriving (Eq,Show)
```

1. Ansatz: Scanner direkt implementieren II

```
lexer :: String -> [Token]
lexer [] = []
lexer (c:cs)
      | isSpace c = lexer cs
      | isAlpha c = lexVar (c:cs)
      | isDigit c = lexInt (c:cs)
lexer ('+':cs) = SymToken '+' : lexer cs
lexer ('-':cs) = SymToken '-' : lexer cs
lexer ('*':cs) = SymToken '*' : lexer cs
lexer ('/':cs) = SymToken '/' : lexer cs
lexer ('(':cs) = SymToken '(' : lexer cs
lexer (')':cs) = SymToken ')' : lexer cs
```

1. Ansatz: Scanner direkt implementieren III

```
lexInt cs = IntToken (read num) : lexer rest
      where (num,rest) = span isDigit cs

lexVar cs =
   case span isAlpha cs of
      ("let",rest) -> LetToken : lexer rest
      ("in",rest)  -> InToken : lexer rest
      (var,rest)   -> VarToken var : lexer rest
```

2. Ansatz: Scanner-Tools

- lex (Programmiersprache C, Standard-Tool Unix)
- JLex (Programmiersprache Java, `https://www.cs.princeton.edu/~appel/modern/java/JLex/`)
- Alex (Programmiersprache Haskell, `http://www.haskell.org/alex`)

Alex-Spezifikation

```
{
 Haskell-code
}
```

Alex-Spezifikation

```
{
 Haskell-code
}

$ abk1  =  regExp1
$ abk2  =  regExp2
...
$ abkn  =  regExpn
```

Alex-Spezifikation

```
{
 Haskell-code
}

$ abk1 =  regExp1
$ abk2 =  regExp2
...
$ abkn =  regExpn

%wrapper " wrapper"
```

Alex-Spezifikation

```
{
 Haskell-code
}

$ abk1 =  regExp1
$ abk2 =  regExp2
...
$ abkn =  regExpn

%wrapper " wrapper"

tokens :=

 lex--Spezifikation
```

Alex-Spezifikation

```
{
 Haskell-code
}

$ abk1 =  regExp1
$ abk2 =  regExp2
...
$ abkn =  regExpn

%wrapper " wrapper"

tokens :=

 lex--Spezifikation

{
 Haskell-code
}
```

Alex-Spezifikation Beispiel

```
{
}

%wrapper "basic"

$digit = 0-9                    -- digits
$alpha = [a-zA-Z]               -- alphabetic characters

tokens :-
  $white+                ;
  "--".*                 ;
  let                 {  \s -> LetToken }
  in                  {  \s -> InToken }
  $digit+               {  \s -> IntToken (read s) }
  [\=\+\-\*\/\(\)]            {  \s -> SymToken (head s) }
  $alpha [$alpha $digit \_ \']*    {  \s -> VarToken s }

  -- Each action has type :: String -> Token
```

Alex-Spezifikation Beispiel II

```
{
-- The token type:

data Token = LetToken
           | InToken
           | SymToken Char
           | VarToken String
           | IntToken Int
  deriving (Eq,Show)


main = do
  s <- getContents
  print (alexScanTokens s)
}
```

Wrapper

Es gibt in Alex einige vordefinierte Wrapper:

- The *basic* wrapper
- The *posn* wrapper
- The *monad* wrapper
- The *monadUserState* wrapper
- The *gscan* wrapper
- The *bytestring* wrappers

Types der token actions (basic Wrapper)

```
String -> Token
```

Parser–Kombinatoren I

Funktionale Implementierung eines recursive decent Parsers:

```
type Parser tok a = [tok] -> [(a, [tok])]
```

Parser–Kombinatoren I

Funktionale Implementierung eines recursive decent Parsers:

```
type Parser tok a = [tok] -> [(a, [tok])]

failure :: Parser a b                    -- Parser der leeren Sprache
failure = _ -> []                        -- liefert immer fail
```

Parser–Kombinatoren I

Funktionale Implementierung eines recursive decent Parsers:

```
type Parser tok a = [tok] -> [(a, [tok])]

failure :: Parser a b                    -- Parser der leeren Sprache
failure = _ -> []                        -- liefert immer fail

succeed :: a -> Parser tok a             -- Parser der Sprache des
succeed value toks = [(value, toks)] -- leeren Worts (ϵ)
```

Parser–Kombinatoren I

Funktionale Implementierung eines recursive decent Parsers:

```
type Parser tok a = [tok] -> [(a, [tok])]

failure :: Parser a b                    -- Parser der leeren Sprache
failure = _ -> []                        -- liefert immer fail

succeed :: a -> Parser tok a             -- Parser der Sprache des
succeed value toks = [(value, toks)] -- leeren Worts (ϵ)

-- bedingte Erkennung
satisfy :: (tok -> Bool) -> Parser tok tok
satisfy cond [] = []
satisfy cond (tok : toks) | cond tok  = succeed tok toks
                          | otherwise = failure toks
```

Parser–Kombinatoren I

Funktionale Implementierung eines recursive decent Parsers:

```
type Parser tok a = [tok] -> [(a, [tok])]

failure :: Parser a b                    -- Parser der leeren Sprache
failure = _ -> []                        -- liefert immer fail

succeed :: a -> Parser tok a             -- Parser der Sprache des
succeed value toks = [(value, toks)] -- leeren Worts (ϵ)

-- bedingte Erkennung
satisfy :: (tok -> Bool) -> Parser tok tok
satisfy cond [] = []
satisfy cond (tok : toks) | cond tok  = succeed tok toks
                          | otherwise = failure toks

-- erkennen eines bestimmten Lexems (Terminals)
lexem :: Eq tok => tok -> Parser tok tok
lexem tok toks = satisfy ((==) tok) toks
```

Parser–Kombinatoren II

Umsetzen der Produktionen

```
-- nacheinander Erkennen
(+.+) :: Parser tok a -> Parser tok b -> Parser tok (a,b)
(p1 +.+ p2) toks = [((v1, v2), rest2) | (v1, rest1) <- p1 toks,
                                        (v2, rest2) <- p2 rest1]
```

Parser–Kombinatoren II

Umsetzen der Produktionen

```
-- nacheinander Erkennen
(+.+) :: Parser tok a -> Parser tok b -> Parser tok (a,b)
(p1 +.+ p2) toks = [((v1, v2), rest2) | (v1, rest1) <- p1 toks,
                                        (v2, rest2) <- p2 rest1]

-- Alternative
(|||) :: Parser tok a -> Parser tok a -> Parser tok a
(p1 ||| p2) toks = p1 toks ++ p2 toks
```

Parser–Kombinatoren II

Umsetzen der Produktionen

```
-- nacheinander Erkennen
(+.+) :: Parser tok a -> Parser tok b -> Parser tok (a,b)
(p1 +.+ p2) toks = [((v1, v2), rest2) | (v1, rest1) <- p1 toks,
                                        (v2, rest2) <- p2 rest1]
```

```
-- Alternative
(|||) :: Parser tok a -> Parser tok a -> Parser tok a
(p1 ||| p2) toks = p1 toks ++ p2 toks
```

Transfomation der Ergebnisse

```
(<<<) :: Parser tok a -> (a -> b) -> Parser tok b
(p <<< f) toks = [ (f v, rest) | (v, rest) <- p toks]
```

Beispiel Parser–Kombinatoren

Lexeme

```
data Token = LetToken
           | InToken
           | SymToken Char
           | VarToken String
           | IntToken Int

isVar (VarToken x) = True
isVar _ = False

isSym x (SymToken y) = x == y
isSym _ _ = False

isInt (IntToken n) = True
isInt _ = False

date Maybe a = Just a
             | Nothing
```

Beispiel Parser–Kombinatoren II

$G = (N, T, \Pi, S)$ mit $N = \{Exp, Exp', TExp\}$
$T = \{let, in, digits, var, =, +\}$ und
$\Pi = \{Exp \rightarrow TExp\ Exp'$
$Exp' \rightarrow + TExp\ Exp' \mid \epsilon$
$TExp \rightarrow let\ var\ =\ Exp\ in\ Exp \mid var \mid digits\}$

Beispiel Parser–Kombinatoren II

$G = (N, T, \Pi, S)$ mit $N = \{Exp, Exp', TExp\}$
$T = \{let, in, digits, var, =, +\}$ und
$\Pi = \{Exp \rightarrow TExp\ Exp'$
$Exp' \rightarrow + TExp\ Exp' \mid \epsilon$
$TExp \rightarrow let\ var\ =\ Exp\ in\ Exp \mid var \mid digits\}$

```
expr :: Parser Token ???
expr = (texp +.+ expr')

expr' :: Parser Token ???
expr' = ((satisfy (isSym '+')) +.+ texp +.+ expr')
 ||| succeed ???

texp :: Parser Token ???
texp = ((lexem LetToken) +.+ (satisfy isVar)
  +.+ (satisfy (isSym '=')) +.+ expr +.+ (lexem InToken)
  +.+ expr)
 ||| (satisfy isVar)
 ||| (satisfy isInt)
```

Beispiel Parser–Kombinatoren II

$G = (N, T, \Pi, S)$ mit $N = \{Exp, Exp', TExp\}$
$T = \{let, in, digits, var, =, +\}$ und
$\Pi = \{Exp \rightarrow TExp\ Exp'$
$Exp' \rightarrow + TExp\ Exp' \mid \epsilon$
$TExp \rightarrow let\ var\ =\ Exp\ in\ Exp \mid var \mid digits\}$

```
expr :: Parser Token ???
expr = (texp +.+ expr')

expr' :: Parser Token ???
expr' = ((satisfy (isSym '+')) +.+ texp +.+ expr')
 ||| succeed ???

texp :: Parser Token ???
texp = ((lexem LetToken) +.+ (satisfy isVar)
  +.+ (satisfy (isSym '=')) +.+ expr +.+ (lexem InToken)
  +.+ expr)
 ||| (satisfy isVar)
 ||| (satisfy isInt)
```

Typfehler!!!

Abstrakte Syntax *Mini funktionale Expressions*

```
data MiniFunkExpr =
       Let String MiniFunkExpr MiniFunkExpr
     | Plus MiniFunkExpr MiniFunkExpr
     | Const Int
     | Var String
     deriving (Eq, Show)
```

Transformation in abstrakte Syntax

```
expr :: Parser Token MiniFunkExpr
expr = (texp +.+ expr')
```

```
expr' :: Parser Token (Maybe MiniFunkExpr)
expr' =
 (((satisfy (isSym '+')) +.+ texp +.+ expr')

 ||| succeed Nothing

texp :: Parser Token MiniFunkExpr
texp = (((lexem LetToken) +.+ (satisfy isVar) +.+
   (satisfy (isSym '=')) +.+ expr +.+ (lexem InToken) +.+ expr)

 ||| ((satisfy isVar)
 ||| ((satisfy isInt)
```

Transformation in abstrakte Syntax

```
expr :: Parser Token MiniFunkExpr
expr = (texp +.+ expr')
       <<< \(e1, e2) ->
                 if (e2 == Nothing) then e1
                                    else Plus e1 (fromJust e2)

expr' :: Parser Token (Maybe MiniFunkExpr)
expr' =
 (((satisfy (isSym '+')) +.+ texp +.+ expr')
     <<< (\(_,(e1, e2)) ->
           if (e2 == Nothing) then Just e1
                              else Just (Plus e1 (fromJust e2))))
 ||| succeed Nothing

texp :: Parser Token MiniFunkExpr
texp = (((lexem LetToken) +.+ (satisfy isVar) +.+
   (satisfy (isSym '=')) +.+ expr +.+ (lexem InToken) +.+ expr)
 <<< (\(_,(VarToken id, (_, (e, (_, e2))))) -> (Let id e e2)))
 ||| ((satisfy isVar) <<< (\(VarToken id) -> Var id))
 ||| ((satisfy isInt) <<< (\(IntToken n) -> Const n))
```

Anpassung Alex-Spezifikation

```
{
module Scanner (alexScanTokens, Token(..)) where
}

%wrapper "basic"

$digit = 0-9                    -- digits
$alpha = [a-zA-Z]               -- alphabetic characters

tokens :-
  $white+                ;
  "--".*                 ;
  let                { \s -> LetToken }
  in                 { \s -> InToken }
  $digit+              { \s -> IntToken (read s) }
  [\=\+\-\*\/\(\)]           { \s -> SymToken (head s) }
  $alpha [$alpha $digit \_ ']*   { \s -> VarToken s }
```

```
{
data Token =
   LetToken
|  InToken
|  SymToken Char
|  VarToken String
|  IntToken Int
  deriving (Eq,Show)
}
```

main Funktion

```
-- nur wenn keine Tokens übrig sind ist die Loesung korrekt
correctsols :: [(t, [a])] -> [(t, [a])]
correctsols sols =
  (filter (\(_, resttokens) -> null resttokens)) sols

parser :: String -> MiniFunkExpr
parser = fst . head . correctsols. expr . alexScanTokens
```

main Funktion

```
-- nur wenn keine Tokens übrig sind ist die Loesung korrekt
correctsols :: [(t, [a])] -> [(t, [a])]
correctsols sols =
  (filter (\(_, resttokens) -> null resttokens)) sols

parser :: String -> MiniFunkExpr
parser = fst . head . correctsols. expr . alexScanTokens


main = do
  s <- readFile "Pfad/fst.mfe"
  print (parser s)
```

main Funktion

```
-- nur wenn keine Tokens übrig sind ist die Loesung korrekt
correctsols :: [(t, [a])] -> [(t, [a])]
correctsols sols =
  (filter (\(_, resttokens) -> null resttokens)) sols

parser :: String -> MiniFunkExpr
parser = fst . head . correctsols. expr . alexScanTokens


main = do
  s <- readFile "Pfad/fst.mfe"
  print (parser s)
```

Mögliche Eingabe: `fst.mfe`

```
let x = 10
in let y = 20
   in x + y
```

Abschlussbemerkung Kombinator-Parsen

- Es werden immer alle möglichen Ableitungen gebildet
  $\Rightarrow$
  - keine Vorausschau zur Endscheidung bei Alternativen
  - Es muss kein Backtracking programmiert werden
  - `head` in der Funktion `parser` führt dazu, dass nur die erste Lösung bestimmt wird.
  - Nicht-strikte Auswertung führt zu trotzdem effizienten Ergebnissen.

Abschlussbemerkung Kombinator-Parsen

- Es werden immer alle möglichen Ableitungen gebildet
  $\Rightarrow$
  - keine Vorausschau zur Endscheidung bei Alternativen
  - Es muss kein Backtracking programmiert werden
  - `head` in der Funktion `parser` führt dazu, dass nur die erste Lösung bestimmt wird.
  - Nicht-strikte Auswertung führt zu trotzdem effizienten Ergebnissen.

- Linksrekursive Grammatiken können zu Endlosrekusionen führen
  $\Rightarrow$ Auflösung von Linksrekursionen

Parsergenerator Happy

- Yacc für Haskell
- **Aufruf:** `> happy -info JavaParser.y`
- Option `-info` erzeugt die Info-Datei: `nfo`

Das Happy-File

Haskell–Source–Code:

```
{
module Parser (parse-Funktion) where
}
```

- Das erzeugte Haskell–File definiert das Module *Parser*.
- Die Funktion *parse-Funktion* wird exportiert.

Deklarationen

```
%name { parse-Funktion }
%tokentype { Tokentyp }
%error { parseError-Funktion }
```

- `name`: Name der Parserfunktion.
- `tokentype`: Type der einzelnen Tokens, die der Parser liest.
- `error`: Name der Funktion, die bei einem Fehler aufgerufen wird.

Beispiel

Data–Deklaration des %tokentype's

Leicht modifizierte Datenstruktur in Scanner.x:

```
data Token =
  LetToken|
  InToken |
  PlusToken |
  AssignToken |
  VarToken String  |
  IntToken Int
  deriving (Eq,Show)
```

Tokens

```
%token
    Let { LetToken }
    In { InToken }
    Plus { PlusToken }
    Assign { AssignToken }
    Var  { VarToken $$  }
    Int { IntToken $$ }
```

- Die Tokens werden definiert durch das Paar
  - *Terminal* in der Grammatik (links)
  - *Haskell-Konstruktor des Typs* `%tokentype` (rechts in geschweifter Klammer)
- Wert des Tokens: normalerweise das Token selbst
  `$$` bedeutet, der Wert ist das Argument des Tokens

Modifizierte Grammatik

Die Grammatik aus dem Kombinator-Parsen Beispiel

$$
\begin{aligned}
\Pi = \{\ & Exp \rightarrow TExp\ Exp' \\
& Exp' \rightarrow + TExp\ Exp' \\
& \quad\ \mid \epsilon \\
& TExp \rightarrow let\ var\ =\ Exp\ in\ Exp \\
& \quad\ \mid var \\
& \quad\ \mid digits\ \}
\end{aligned}
$$

wird vereinfacht zu:

$$
\begin{aligned}
\Pi = \{\ & Exp \rightarrow let\ var\ =\ Exp\ in\ Exp \\
& \quad\ \mid Exp\ +\ Exp \\
& \quad\ \mid var \\
& \quad\ \mid digits\}
\end{aligned}
$$

Beim Bottom–Up Parsen dürfen Grammatiken linksrekursiv sein.

Grammatik im Happy–File

```
%% -- aus yacc-Tradition

expr              : Let Var Assign expr In expr { }
                  | expr Plus expr { }
                  | Var { }
                  | Int { }
```

Grammatik und Übersetzung

Erinnerung: Abstrakte Syntax *Mini funktionale Expressions*

```
data MiniFunkExpr =
       Let String MiniFunkExpr MiniFunkExpr
     | Plus MiniFunkExpr MiniFunkExpr
     | Const Int
     | Var String
     deriving (Eq, Show)
```

Grammatik und Übersetzung

Erinnerung: Abstrakte Syntax *Mini funktionale Expressions*

```
data MiniFunkExpr =
       Let String MiniFunkExpr MiniFunkExpr
     | Plus MiniFunkExpr MiniFunkExpr
     | Const Int
     | Var String
     deriving (Eq, Show)
```

Happy–File mit

```
%% -- aus yacc-Tradition

expr              : Let Var Assign expr In expr { Let $2 $4 $6 }
                  | expr Plus expr { Plus $1 $3 }
                  | Var { Var $1 }
                  | Int { Const $1 }
```

Grammatik und Übersetzung

Erinnerung: Abstrakte Syntax *Mini funktionale Expressions*

```
data MiniFunkExpr =
       Let String MiniFunkExpr MiniFunkExpr
     | Plus MiniFunkExpr MiniFunkExpr
     | Const Int
     | Var String
     deriving (Eq, Show)
```

Happy–File mit

```
%% -- aus yacc-Tradition

expr              : Let Var Assign expr In expr { Let $2 $4 $6 }
                  | expr Plus expr { Plus $1 $3 }
                  | Var { Var $1 }
                  | Int { Const $1 }
```

- Hinter jeder Regel wird eine Haskell–Anweisung angegeben, die beim *Reduce*–Schritt des Parsers ausgeführt wird.
- $n gibt das Ergebnis des n. Symbols der rechten Seite an.

Beispiel

```
Let Var Assign expr In expr { Let $2 $4 $6 }
```

bedeutet: Beim *reduce* wird ein `Let`–Element erzeugt, das als Argumente

1. das Argument des Terminals `Var` ($2) und
2. das Ergebnis von `expr` ($4) und
3. das Ergebnis von `expr` ($6) und

hat.

Beispiel

```
Let Var Assign expr In expr { Let $2 $4 $6 }
```

bedeutet: Beim *reduce* wird ein `Let`–Element erzeugt, das als Argumente

1. das Argument des Terminals `Var` ($2) und
2. das Ergebnis von `expr` ($4) und
3. das Ergebnis von `expr` ($6) und

hat.

```
let x = 2 in x
```

gibt das Paar `Let "x" (Const 2) (Var "x")` zurück.

Haskell-Code

Am Ende der Datei gibt es einen Abschnitt, in dem Haskell-Code programmiert werden kann.

```
{

parseError :: [Token] -> a
parseError _ = error "Parse error"

parser :: String -> MiniFunkExpr
parser = expr . alexScanTokens

main = do
  s <- readFile "Pfad/fst.mfe"
  print (parser s)


}
```