

Compilerbau

Martin Plümicke

SS 2019

Agenda

I. Überblick Vorlesung

Literatur

II. Compiler Überblick

III. Überblick Funktionale Programmierung

Einleitung

Haskell-Grundlagen

IV. Compiler

Scanner


Parser


Abstrakte Syntax

Semantische Analyse/Typecheck


Codegenerierung

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.

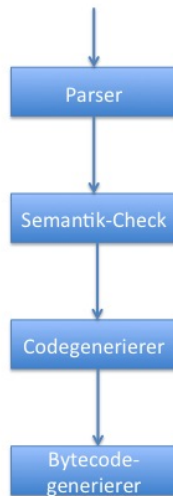
O'Reilly, 2009.



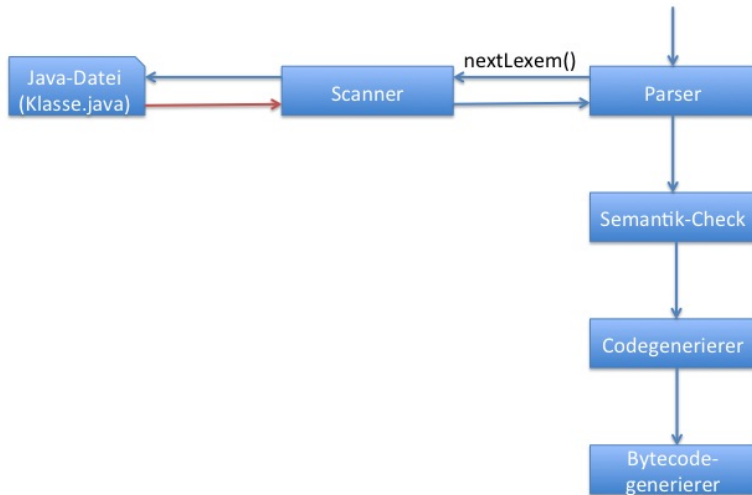
Peter Thiemann.
Grundlagen der funktionalen Programmierung.

Teubner, 1994.

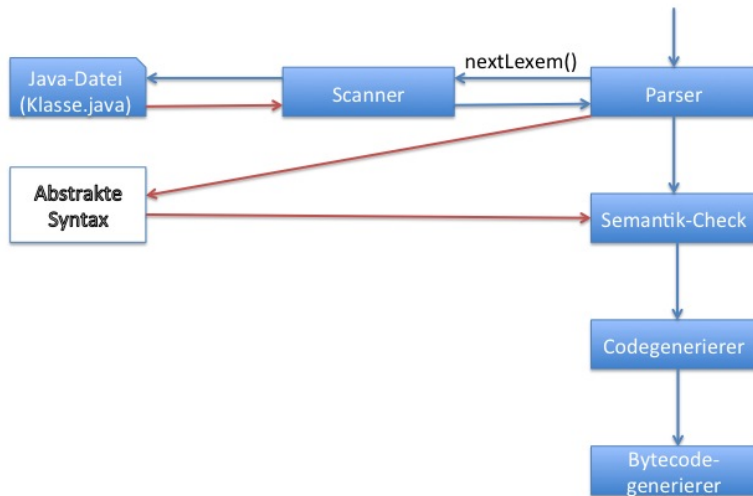
Compiler Überblick



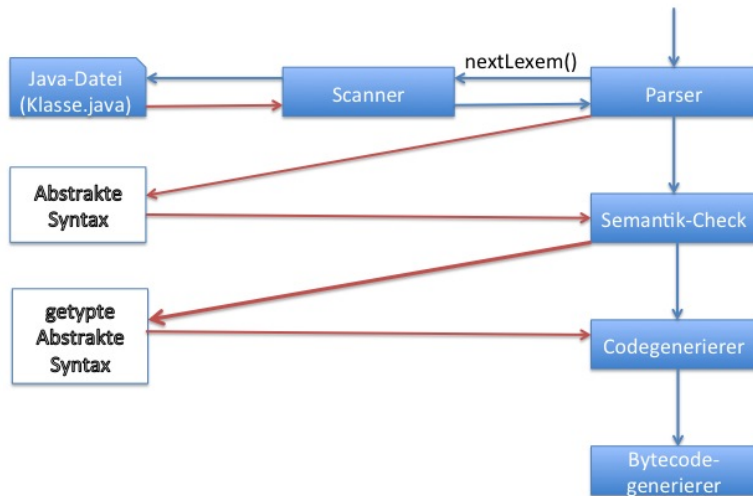
Compiler Überblick



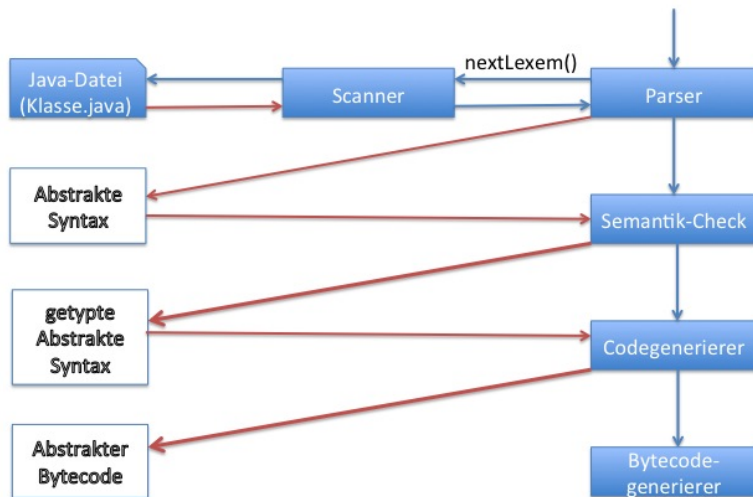
Compiler Überblick



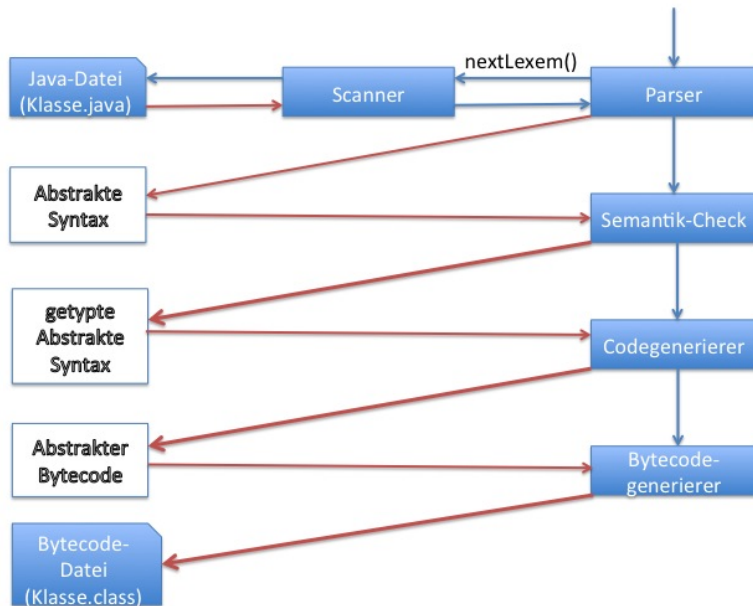
Compiler Überblick



Compiler Überblick



Compiler Überblick



III. 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) = $x \cdot x$

1. Quadratfunktion

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

square(x) = $x \cdot x$

Java:

```
int square(int x) {  
    return x*x;  
}
```

1. Quadratfunktion

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

square(x) = $x \cdot x$

Java:

```
int square(int x) {  
    return x*x;  
}
```

Haskell:

```
square :: Int -> Int  
square(x) = x*x
```

2. Maximumsfunktion

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

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

2. Maximumsfunktion

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

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

Java:

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

2. Maximumsfunktion

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

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

Java:

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

Haskell:

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

3. Kreisfunktion

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

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

3. Kreisfunktion

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

Java:

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

3. Kreisfunktion

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

Java:

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

Haskell:

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

4. Vektorarithmetik

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

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.add(1.0);v1.add(2.0);  
        Vektorarithmetik v2 = new Vektorarithmetik();  
        v2.add(3.0);v2.add(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.add(1.0);v1.add(2.0);  
        Vektorarithmetik v2 = new Vektorarithmetik();  
        v2.add(3.0);v2.add(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

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

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 x + n;  
    }  
}
```

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

```
class Main {  
  
    Fun<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: \rightarrow Folge;

cons : $A \times \text{Folge} \rightarrow \text{Folge}$;

operations

head : Folge \rightarrow A;

tail : Folge \rightarrow Folge;

is_empty: Folge \rightarrow Boolean;

cat: Folge \times Folge \rightarrow Folge;

len: Folge $\rightarrow \mathbb{N}$;

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` \rightarrow `var1` , ... , `varn`

`fundecl` \rightarrow `vars` :: `type`

| | | |
|--------------------|--|--------------------------------|
| <code>type</code> | \rightarrow <code>btype</code> \rightarrow <code>type</code>] | (function type) |
| <code>btype</code> | \rightarrow [<code>btype</code>] <code>atype</code> | (type application) |
| <code>atype</code> | \rightarrow <code>tyvar</code> | |
| | (<code>type_1</code> , ... , <code>type_k</code>) | (tuple type, $k \geq 2$) |
| | [<code>type</code>] | (list type) |
| | (<code>type</code>) | (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 }`

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

`rhs -> = exp`
`| gdrhs [where decls]`

`gdrhs -> gd = exp [gdrhs]`

`gd -> " | " exp`

Deklarationen von Funktionen in Haskell

Expressions

```
exp  -> \ apat1 ... apatn -> exp (lambda abstraction,
                                   n>=1)
      | let decls in exp          (let expression)
      | if exp then exp else exp (conditional)
      | case exp of { alts }      (case expression)
      | do { stmts }              (do expression)
      | fexp
fexp -> [fexp] aexp                (function application)

alts -> alt1 ; ... ; altn         (n>=1)
alt  -> pat -> exp
```

```

aexp -> qvar                (variable)
      | gcon                (general constructor)
      | literal
      | ( exp )              (parenthesized expression)
      | ( exp1 , ... , expk ) (tuple, k>=2)
      | [ exp1 , ... , expk ] (list, k>=1)
literal -> integer | float | char | string

```

Beispiele:

square $x = x * x$

Beispiele:

`square x = x * x`

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
    len0r(x, 0)
```

let/where-Konstrukte

```
len: [a] -> int
len(x) = let
    len0([], n) = n
    len0 (x:xs, n) = len0 xs (n+1)
in
    len0r(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)
```

$$T_{\text{FolgeInt}} =$$

| | | |
|----------------------------|--------------------------------|---------|
| { 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:

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

Funktion als Ergebnis:

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

Currying

Satz: Sei $f : (\tau_1, \dots, \tau_n) \rightarrow \tau$ eine Funktion mit $f(\mathbf{a}_1, \dots, \mathbf{a}_n) = \mathbf{a}$. 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

$$(\dots (((f' \mathbf{a}_1) \mathbf{a}_2) \mathbf{a}_3) \dots \mathbf{a}_n) = \mathbf{a}.$$

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
```

```
sqliist :: [int] -> [int]
```

```
sqliist 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
```

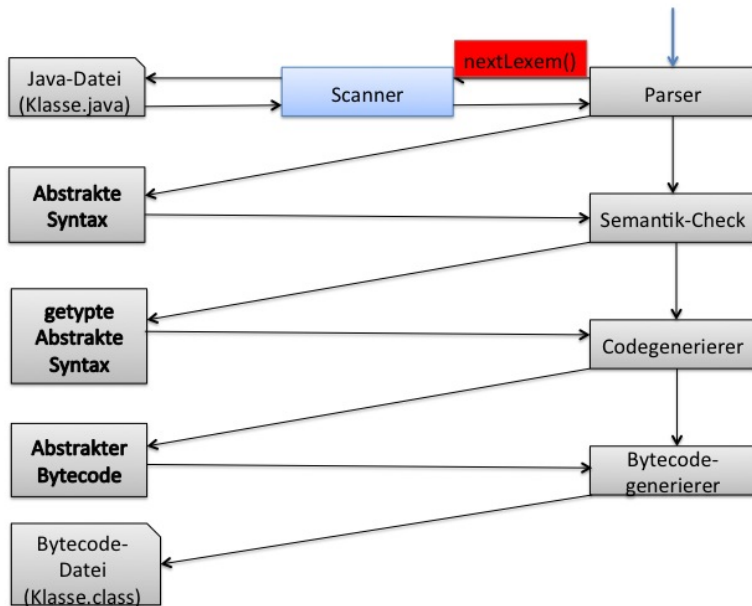
Stdin/Stdout

```
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
```


Scanner



Programmiersprachen

Programmiersprachen werden als formale Sprachen über einem Alphabet von Tokens definiert.

Lexeme, Tokens

Für jede Programmiersprache wird eine Menge von Strings festgelegt, über die die erlaubte Struktur dann definiert wird. Man nennt diese Strings **Lexeme**.

Verschiedene Lexeme, die eine ähnliche Bedeutung haben, fasst man zu Klassen von Lexemen zusammen. Die Klassen heißen **Tokens**.

Lexeme, Tokens

Für jede Programmiersprache wird eine Menge von Strings festgelegt, über die die erlaubte Struktur dann definiert wird. Man nennt diese Strings **Lexeme**.

Verschiedene Lexeme, die eine ähnliche Bedeutung haben, fasst man zu Klassen von Lexemen zusammen. Die Klassen heißen **Tokens**.

Um Tokens bilden zu können, muss man jedes Lexem (String) durch eine **reguläre Sprache** über den Symbolen eines Zeichensatzes (z.B. ASCII, latin-1, UTF-8, ...) beschreiben.

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 -> Let }
```

```
  in                   { \s -> In }
```

```
  $digit+              { \s -> Int (read s) }
```

```
  [= \+ \- \* \\/ \(\ \)] { \s -> Sym (head s) }
```

```
  $alpha [$alpha $digit \_ \']* { \s -> Var s }
```

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

Alex-Spezifikation Beispiel II

```
{  
  -- The token type:  
  data Token =  
    Let      |  
    In       |  
    Sym Char |  
    Var String |  
    Int Int  
    deriving (Eq,Show)  
  
  main = do  
    s <- getContents  
    print (alexScanTokens s)  
}
```

Wrapper

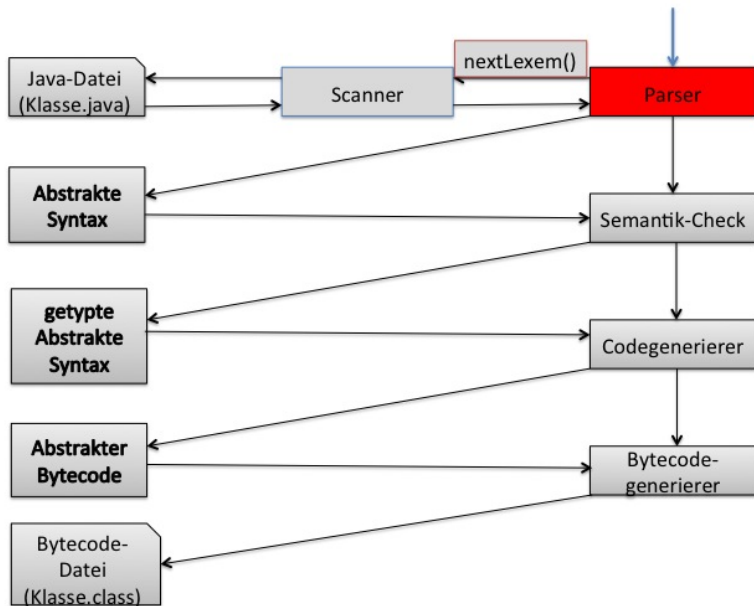
Es gibt in Alex einige vordefinierte Wrapper:

- ▶ 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



Programmiersprachen

Programmiersprachen werden als formale Sprachen über einem Alphabet von Tokens definiert.

Spezifikation eines Parser

Eingabe: Grammatik $G = (N, \Sigma, \Pi, S)$, $w \in \Sigma^*$

Ausgabe: $erg \in \{True, False\}$

Nachbedingung: $erg = (w \in \mathcal{L}(G))$

Mit anderen Worten: Es muss eine Ableitung $S \xrightarrow{*} w$ gefunden werden.

Ableitungsbaum

Man kann die Ableitung eines Wortes als Baum betrachten.

Ableitungsbaum

Man kann die Ableitung eines Wortes als Baum betrachten.

Aufbau:

Top-down: **Linksableitungen** (man erhält eine Ableitung, bei der immer das am weitesten links stehende Nichtterminal abgeleitet wird)

Ableitungsbaum

Man kann die Ableitung eines Wortes als Baum betrachten.

Aufbau:

Top-down: **Linksableitungen** (man erhält eine Ableitung, bei der immer das am weitesten links stehende Nichtterminal abgeleitet wird)

Bottom-Up: **Rechtsableitungen** (man erhält (rückwärts) eine Ableitung bei der immer das am weitesten rechts stehende Nichtterminal abgeleitet wird)

Recursive Decent–Syntaxanalyse

- ▶ Eingabe wird durch eine Menge rekursiver Funktionen abgearbeitet.
- ▶ Jedem Nichtterminal der Grammatik entspricht eine Funktion.
- ▶ Die Folge der Funktionsaufrufe bestimmt implizit den Ableitungsbaum.

Linksrekursive Grammatik

$G = (N, T, \Pi, S)$ mit

$$\Pi = \left\{ \begin{array}{l} \text{Exp} \rightarrow \text{let var = Exp in Exp} \\ \text{Exp} \rightarrow \text{Exp} + \text{Exp} \\ \text{Exp} \rightarrow \text{var} \\ \text{Exp} \rightarrow \text{digits} \end{array} \right\}$$

Problem: Linksrekursion

Linksrekursive Grammatik

$G = (N, T, \Pi, S)$ mit

$$\Pi = \left\{ \begin{array}{l} \text{Exp} \rightarrow \text{let var} = \text{Exp in Exp} \\ \quad \quad \quad | \quad \text{Exp} + \text{Exp} \\ \quad \quad \quad | \quad \text{var} \\ \quad \quad \quad | \quad \text{digits} \end{array} \right\}$$

Problem: Linksrekursion

Elimination der Linksrekursion:

$$\Pi = \left\{ \begin{array}{l} \text{Exp} \rightarrow T\text{Exp Exp}' \\ \text{Exp}' \rightarrow + T\text{Exp Exp}' \\ \quad \quad | \quad \epsilon \\ T\text{Exp} \rightarrow \text{let var} = \text{Exp in Exp} \\ \quad \quad | \quad \text{var} \\ \quad \quad | \quad \text{digits} \end{array} \right\}$$

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 ( $\epsilon$ )
```

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 ( $\epsilon$ )
```

```
-- 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 = satisfy ((==) tok)
```

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
```

Transformation 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

data 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 Nothing

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 Nothing

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
```

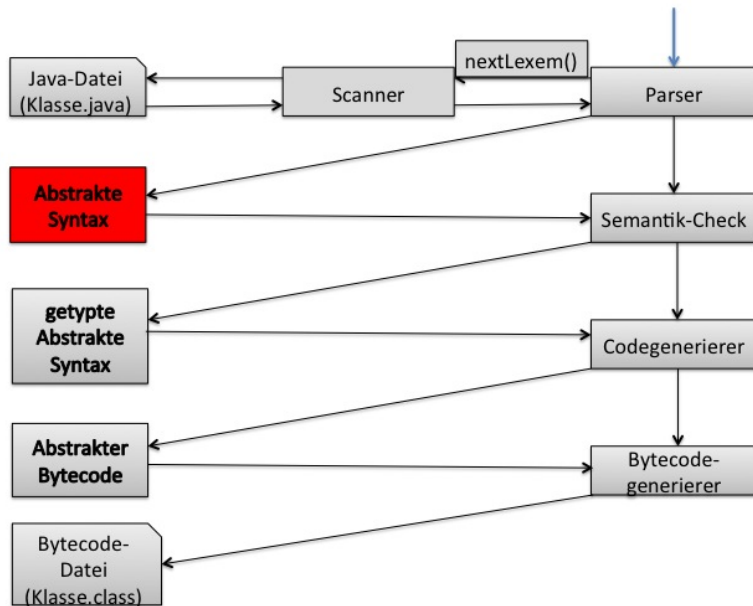
Abschlussbemerkung Kombinator-Parsen

- ▶ Es werden immer alle möglichen Ableitungen gebildet
⇒
 - ▶ keine Vorausschau zur Entscheidung bei Alternativen
 - ▶ Es muss kein Backtracking programmiert werden
 - ▶ `hd` 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
⇒
 - ▶ keine Vorausschau zur Entscheidung bei Alternativen
 - ▶ Es muss kein Backtracking programmiert werden
 - ▶ `hd` 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 Endlosrekursionen führen
⇒ Auflösung von Linksrekursionen

Abstrakte Syntax



Abstrakte Syntax

Unter **abstrakter Syntax** versteht man eine *abstrakte* Repräsentation eines konkreten Programms als **Syntaxbaum**.

Man kann den **Ableitungsbaum** in den **abstrakten Syntaxbaum** abbilden.

Abstrakte Syntax

Klassendeklaration

Datentyp:

```
data Class = Class(Type, [FieldDecl], [MethodDecl])
```


Abstrakte Syntax

Klassendeklaration

Datentyp:

```
data Class = Class(Type, [FieldDecl], [MethodDecl])
```

Java-Programm

```
class Klassenname { }
```

Abstrakte Syntax

Klassendeklaration

Datentyp:

```
data Class = Class(Type, [FieldDecl], [MethodDecl])
```

Java-Programm

```
class Klassenname { }
```

Abstrakte Syntax:

```
Class("Klassenname", [], [])
```

Instanzvariable (fields)

Datentyp:

```
data FieldDecl = FieldDecl(Type, String)
```

Instanzvariable (fields)

Datentyp:

```
data FieldDecl = FieldDecl(Type, String)
```

Java-Programm

```
class Klassenname {  
  
    int v;  
  
}
```

Instanzvariable (fields)

Datentyp:

```
data FieldDecl = FieldDecl(Type, String)
```

Java-Programm

```
class Klassenname {  
  
    int v;  
  
}
```

Abstrakte Syntax:

```
FieldDecl("int", "v")
```

Methoden

Datentyp:

```
data MethodDecl =  
  Method(Type, String, [(Type, String)], Stmt)
```

Methoden

Datentyp:

```
data MethodDecl =  
  Method(Type, String, [(Type, String)], Stmt)
```

Java-Programm

```
class Klassenname {  
  
  void methode (int x, char y) { }  
  
}
```

Methoden

Datentyp:

```
data MethodDecl =  
  Method(Type, String, [(Type, String)], Stmt)
```

Java-Programm

```
class Klassenname {  
  
  void methode (int x, char y) { }  
  
}
```

Abstrakte Syntax:

```
MethodDecl("void", "methode",  
           [("int", "x"), ("char", "y")],  
           Block([]))
```


Expression

```
data Expr = This
  | Super
  | LocalOrFieldVar(String)
  | InstVar(Expr, String)
  | Unary(String, Expr)
  | Binary(String, Expr, Expr)
  | Integer(Integer)
  | Bool(Bool)
  | Char(Char)
  | String(String)
  | Jnull
  | StmtExprExpr(StmtExpr)

data StmtExpr = Assign(Expr, Expr)
  | New(Type, [Expr])
  | MethodCall(Expr, String, [Expr])
```

LocalOrFieldVar

Datentyp:

```
LocalOrFieldVar(String)
```

LocalOrFieldVar

Datentyp:

LocalOrFieldVar(String)

Java-Programm

```
class Klassenname {  
  
    int methode (int x, char y) {  
  
        return x;  
  
    }  
  
}
```

LocalOrFieldVar

Datentyp:

`LocalOrFieldVar(String)`

Java-Programm

```
class Klassenname {  
  
    int methode (int x, char y) {  
  
        return x;  
  
    }  
  
}
```

Abstrakte Syntax:

`LocalOrFieldVar("x")`

InstVar

Datentyp:

`InstVar(Expr,String)`

InstVar

Datentyp:

InstVar(Expr,String)

Java-Programm

```
class Klassenname {  
  
    int methode (Typ x, char y) {  
  
        return x.v;  
  
    }  
  
}
```

InstVar

Datentyp:

`InstVar(Expr, String)`

Java-Programm

```
class Klassenname {  
  
    int methode (Typ x, char y) {  
  
        return x.v;  
  
    }  
  
}
```

Abstrakte Syntax:

`InstVar(LocalOrFieldVar("x"), "v")`

Integer

Datentyp:

`Integer(Integer)`

Integer

Datentyp:

`Integer(Integer)`

Java-Programm

```
class Klassenname {  
  
    int methode (int x, char y) {  
  
        return 1;  
  
    }  
  
}
```

Integer

Datentyp:

Integer(Integer)

Java-Programm

```
class Klassenname {  
  
    int methode (int x, char y) {  
  
        return 1;  
  
    }  
  
}
```

Abstrakte Syntax:

Integer(1)

Binary I

Datentyp:

Binary(String, Expr, Expr)

Binary I

Datentyp:

Binary(String, Expr, Expr)

Java-Programm

```
class Klassenname {  
  
    int methode (int x, char y) {  
  
        return 1 + x;  
  
    }  
  
}
```

Binary I

Datentyp:

Binary(String, Expr, Expr)

Java-Programm

```
class Klassenname {  
  
    int methode (int x, char y) {  
  
        return 1 + x;  
  
    }  
  
}
```

Abstrakte Syntax:

Binary("+", Integer(1), LocalOrFieldVar("x"))

Binary II

Datentyp:

Binary(String, Expr, Expr)

Binary II

Datentyp:

Binary(String, Expr, Expr)

Java-Programm

```
class Klassenname {  
    void methode (int x, int y) {  
  
        if (x == y) { }  
  
    }  
}
```

Binary II

Datentyp:

Binary(String, Expr, Expr)

Java-Programm

```
class Klassenname {  
    void methode (int x, int y) {  
  
        if (x == y) { }  
  
    }  
}
```

Abstrakte Syntax:

```
Binary("==",  
        LocalOrFieldVar("x"),  
        LocalOrFieldVar("y"))
```


Statement Expression: MethodCall

Datentyp:

`StmtExprExpr (StmtExpr)`

`MethodCall (Expr, String, [Expr])`

Statement Expression: MethodCall

Datentyp:

StmtExprExpr(StmtExpr)

MethodCall(Expr,String,[Expr])

Java-Programm

```
class Klassenname {  
  
    int methode (Typ x, int y, int z) {  
        return x.f(y, z);  
    }  
}
```

Statement Expression: MethodCall

Datentyp:

```
StmtExprExpr(StmtExpr)  
MethodCall(Expr,String,[Expr])
```

Java-Programm

```
class Klassenname {  
  
    int methode (Typ x, int y, int z) {  
        return x.f(y, z);  
    }  
}
```

Abstrakte Syntax:

```
StmtExprExpr(MethodCall(LocalOrFieldVar("x")),  
              "f",  
              [LocalOrFieldVar("y"), LocalOrFieldVar("z")])
```

Statements

```
data Stmt = Block([Stmt])
          | Return( Expr )
          | While( Expr , Stmt )
          | LocalVarDecl( Type, String )
          | If( Expr, Stmt , Maybe Stmt )
          | StmtExprStmt(StmtExpr)

data StmtExpr = Assign(Expr, Expr)
              | New(Type, [Expr])
              | MethodCall(Expr, String, [Expr])
```

Return-Statement

Datentyp:

```
Return( Expr )
```

Return-Statement

Datentyp:

`Return(Expr)`

Java-Programm

```
class Klassenname {  
  
    int methode (int x, char y) {  
        return 1 + x;  
    }  
  
}
```

Return-Statement

Datentyp:

```
Return( Expr )
```

Java-Programm

```
class Klassenname {  
  
    int methode (int x, char y) {  
        return 1 + x;  
    }  
  
}
```

Abstrakte Syntax:

```
Return(Binary("+",  
            Integer(1),  
            LocalOrFieldVar("x")))
```

While-Statement

Datentyp:

```
While( Expr, Stmt )
```


While-Statement

Datentyp:

```
While( Expr, Stmt )
```

Java-Programm

```
class Klassenname {  
  
    void methode (int x, char y) {  
        while (x < 1) { }  
    }  
  
}
```

While-Statement

Datentyp:

```
While( Expr, Stmt )
```

Java-Programm

```
class Klassenname {  
  
    void methode (int x, char y) {  
        while (x < 1) { }  
    }  
  
}
```

Abstrakte Syntax:

```
While(Binary("<",  
        LocalOrFieldVar("x"),  
        Integer(1)),  
        Block([]))
```

LocalVarDecl-Statement

Datentyp:

```
LocalVarDecl( Type, String )
```

LocalVarDecl–Statement

Datentyp:

LocalVarDecl(Type, String)

Java–Programm

```
class Klassenname {  
  
    void methode (int x, char y) {  
        int i;  
    }  
  
}
```

LocalVarDecl–Statement

Datentyp:

`LocalVarDecl(Type, String)`

Java–Programm

```
class Klassenname {  
  
    void methode (int x, char y) {  
        int i;  
    }  
  
}
```

Abstrakte Syntax:

`LocalVarDecl("int", "i")`

If (ohne else)

Datentyp:

If(Expr, Stmt, Maybe Stmt)

If (ohne else)

Datentyp:

If(Expr, Stmt, Maybe Stmt)

Java-Programm

```
class Klassenname {  
    int methode (int x, int y) {  
        if (x == y) return 1;  
    }  
}
```

If (ohne else)

Datentyp:

If(Expr, Stmt, Maybe Stmt)

Java-Programm

```
class Klassenname {  
    int methode (int x, int y) {  
        if (x == y) return 1;  
    }  
}
```

Abstrakte Syntax:

```
If(Binary("==",  
        LocalOrFieldVar("x"),  
        LocalOrFieldVar("y")),  
    Return(Integer(1))),  
    Nothing)
```


If (mit else)

Datentyp:

If(Expr, Stmt, Maybe Stmt)

Java-Programm

```
class Klassenname {  
    int methode (int x, int y) {  
        if (x == y) return 1;  
        else return 2;  
    }  
}
```

Abstrakte Syntax:

```
If(Binary("==",  
        LocalOrFieldVar("x"),  
        LocalOrFieldVar("y")),  
    Return(Integer(1)),  
    Just (Return(Integer(2))))
```

Statement Expression: Assign

Datentyp:

StmtExprStmt (StmtExpr)

Assign(Expr, Expr)

Statement Expression: Assign

Datentyp:

StmtExprStmt (StmtExpr)

Assign(Expr, Expr)

Java-Programm

```
class Klassenname {  
  
    void methode (Typ x, int y, int z) {  
        int i;  
        i = x;  
    }  
}
```

Statement Expression: Assign

Datentyp:

StmtExprStmt (StmtExpr)

Assign(Expr, Expr)

Java-Programm

```
class Klassenname {  
  
    void methode (Typ x, int y, int z) {  
        int i;  
        i = x;  
    }  
}
```

Abstrakte Syntax:

```
StmtExprStmt (Assign(LocalOrFieldVar("i"),  
                      LocalOrFieldVar("x")))
```

Komplettes Beispiel

```
class Klassenname {  
    int v;  
    int methode (Typ x, int y, int z) {  
        int i;  
        i = v;  
        return i;  
    }  
}
```

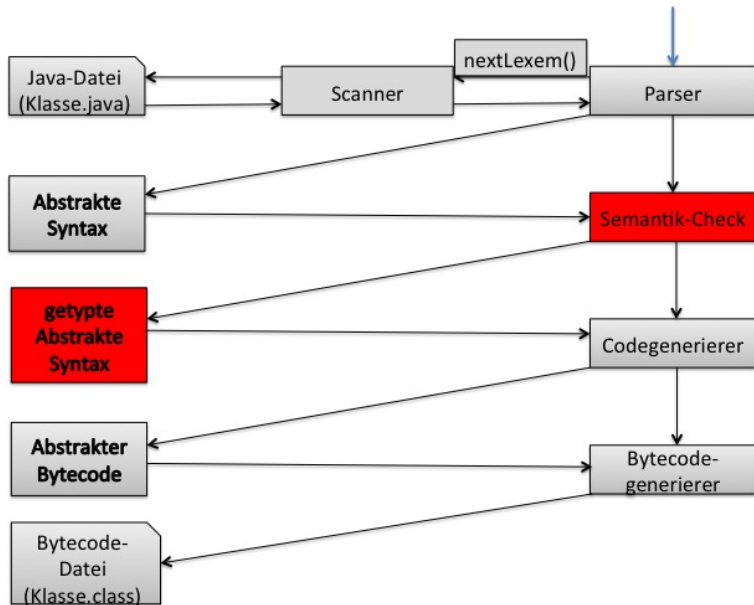
Komplettes Beispiel

```
class Klassenname {  
    int v;  
    int methode (Typ x, int y, int z) {  
        int i;  
        i = v;  
        return i;  
    }  
}
```

Abstrakte Syntax:

```
Class("Klassenname",  
    [FieldDecl("int", "v")],  
    [MethodDecl ("int", "methode",  
        [("Typ", "x"), ("int", "y"), ("int", "z")],  
        Block([LocalVarDecl("int", "i"),  
            StmtExprStmt(  
                Assign(LocalOrFieldVar("i"),  
                    LocalOrFieldVar("v")),  
                Return (LocalOrFieldVar(i)))]))])])])
```

Semantische Analyse/Typecheck



Semantische Analyse/Typecheck

- ▶ Überprüfen der Kontextsensitiven Nebenbedingungen:
 - ▶ alle Variablen/Methoden deklariert/sichtbar?
 - ▶ Typen korrekt?
- ▶ Typisierung aller Sub-Terme

Ungetypte abstrakte Syntax für *Mini-Java* I

```
data Class = Class(Type, [FieldDecl], [MethodDecl])

data FieldDecl = FieldDecl(Type, String)

data MethodDecl = Method(Type, String,[(Type,String)], Stmt)

data Stmt = Block([Stmt])
           | Return( Expr )
           | While( Expr , Stmt )
           | LocalVarDecl(Type, String)
           | If(Expr, Stmt , Maybe Stmt)
           | StmtExprStmt(StmtExpr)

data StmtExpr = Assign(String, Expr)
               | New(Type, [Expr])
               | MethodCall(Expr, String, [Expr])
```

Ungetypte abstrakte Syntax für *Mini-Java* II

```
data Expr = This
  | Super
  | LocalOrFieldVar(String)
  | InstVar(Expr, String)
  | Unary(String, Expr)
  | Binary(String, Expr, Expr)
  | Integer(Integer)
  | Bool(Bool)
  | Char(Char)
  | String(String)
  | Jnull
  | StmtExprExpr(StmtExpr)

type Prg = [Class]
```

Formale Definitionen I

Typableitungen

Menge von Typannahmen: $O = \{ id_1 : ty_1, \dots, id_n : ty_n \}$

Formale Definitionen I

Typableitungen

Menge von Typannahmen: $O = \{ id_1 : ty_1, \dots, id_n : ty_n \}$

$$O \triangleright_{Id} id : ty$$

Aus der Menge der Typannahmen O ist für den Bezeichner id der Typ ty ableitbar.

Formale Definitionen I

Typableitungen

Menge von Typannahmen: $O = \{ id_1 : ty_1, \dots, id_n : ty_n \}$

$$O \triangleright_{Id} id : ty$$

Aus der Menge der Typannahmen O ist für den Bezeichner id der Typ ty ableitbar.

$$O \triangleright_{Expr} e : ty$$

Aus der Menge der Typannahmen O ist für den Ausdruck e der Typ ty ableitbar.

Formale Definitionen I

Typableitungen

Menge von Typannahmen: $O = \{ id_1 : ty_1, \dots, id_n : ty_n \}$

$$O \triangleright_{Id} id : ty$$

Aus der Menge der Typannahmen O ist für den Bezeichner id der Typ ty ableitbar.

$$O \triangleright_{Expr} e : ty$$

Aus der Menge der Typannahmen O ist für den Ausdruck e der Typ ty ableitbar.

$$O \triangleright_{Stmt} s : ty$$

Aus der Menge der Typannahmen O ist für das Statement s der Typ ty ableitbar.

Formale Definitionen II

Typurteile

$$[\text{Regelname}] \frac{O1 \triangleright x : ty1}{O2 \triangleright y : ty2}$$

Aus der Regel *Regelname* folgt, wenn man aus $O1$ ableiten kann, dass x den Typ $ty1$ hat, dann kann man aus $O2$ ableiten dass y den Typ $ty2$ hat.

Ident-Rule

$$[\mathbf{Ident}] \frac{(f : ty) \in O_\tau}{O_\tau \triangleright_{Id} f : ty}$$

O_τ : Menge aller in der Klasse τ sichtbaren Methoden und Attribute

Beispiel Ident–Rule

```
class A {  
    Integer attr;  
    A meth(Boolean x) { ... }  
}
```

Beispiel Ident–Rule

```
class A {  
    Integer attr;  
    A meth(Boolean x) { ... }  
}
```

- ▶ $O_A = \{ \text{attr} : \text{Integer}, \text{meth} : \text{Boolean} \rightarrow A \}$

$$[\text{Ident}] \frac{O_A}{O_A \triangleright_{Id} \text{attr} : \text{Integer}}$$

Beispiel Ident–Rule

```
class A {  
    Integer attr;  
    A meth(Boolean x) { ... }  
}
```

► $O_A = \{ \text{attr} : \text{Integer}, \text{meth} : \text{Boolean} \rightarrow A \}$

$$\text{[Ident]} \frac{O_A}{O_A \triangleright_{Id} \text{attr} : \text{Integer}}$$

$$\text{[Ident]} \frac{O_A}{O_A \triangleright_{Id} \text{meth} : \text{Boolean} \rightarrow A}$$

Literal-Regeln

[IntLiteral] $O \triangleright_{Expr} \text{Int}(n) : \text{int}$

[BoolLiteral] $O \triangleright_{Expr} \text{Bool}(b) : \text{boolean}$

[CharLiteral] $O \triangleright_{Expr} \text{Char}(c) : \text{char}$

[NullLiteral] $O \triangleright_{Expr} \text{Null} : \theta'$

Expression-Regel: Simple-Expressions

$$[\text{Unary1}] \frac{O \triangleright_{Expr} e : \text{int}}{O \triangleright_{Expr} \text{Unary}("+"/"-", e) : \text{int}}$$

Expression–Regel: Simple–Expressions

$$\text{[Unary1]} \quad \frac{O \triangleright_{Expr} e : \text{int}}{O \triangleright_{Expr} \text{Unary}("+"/"-", e) : \text{int}}$$

$$\text{[Unary2]} \quad \frac{O \triangleright_{Expr} e : \text{boolean}}{O \triangleright_{Expr} \text{Unary}("!", e) : \text{boolean}}$$

Expression–Regel: Simple–Expressions

$$\text{[Unary1]} \quad \frac{O \triangleright_{Expr} e : \text{int}}{O \triangleright_{Expr} \text{Unary}("+"/"-", e) : \text{int}}$$

$$\text{[Unary2]} \quad \frac{O \triangleright_{Expr} e : \text{boolean}}{O \triangleright_{Expr} \text{Unary}("!", e) : \text{boolean}}$$

$$\text{[Binary1]} \quad \frac{O \triangleright_{Expr} e1 : \text{int}, O \triangleright_{Expr} e2 : \text{int}}{O \triangleright_{Expr} \text{Binary}("+"/"-"/"*"/"\%", e1, e2) : \text{int}}$$

Expression–Regel: Simple–Expressions

$$\begin{array}{c} [Unary1] \quad \frac{O \triangleright_{Expr} e : \text{int}}{O \triangleright_{Expr} \text{Unary}("+"/"-", e) : \text{int}} \end{array}$$

$$\begin{array}{c} [Unary2] \quad \frac{O \triangleright_{Expr} e : \text{boolean}}{O \triangleright_{Expr} \text{Unary}("!", e) : \text{boolean}} \end{array}$$

$$\begin{array}{c} [Binary1] \quad \frac{O \triangleright_{Expr} e1 : \text{int}, O \triangleright_{Expr} e2 : \text{int}}{O \triangleright_{Expr} \text{Binary}("+"/"-"/"*"/"%", e1, e2) : \text{int}} \end{array}$$

$$\begin{array}{c} [Binary2] \quad \frac{O \triangleright_{Expr} e1 : \text{boolean}, O \triangleright_{Expr} e2 : \text{boolean}}{O \triangleright_{Expr} \text{Binary}("&&"/"||", e1, e2) : \text{boolean}} \end{array}$$

Expression-Regel: Variablen

$$\text{[LocalOrFieldVar]} \frac{O \triangleright_{Id} v : \theta}{O \triangleright_{Expr} \text{LocalOrFieldVar}(v) : \theta}$$

Expression–Regel: Variablen

$$\text{[LocalOrFieldVar]} \quad \frac{O \triangleright_{Id} v : \theta}{O \triangleright_{Expr} \text{LocalOrFieldVar}(v) : \theta}$$

$$\text{[InstVar]} \quad \frac{O \triangleright_{Expr} re : \bar{\tau}, \quad O_{\bar{\tau}} \triangleright_{Id} v : \theta}{O \triangleright_{Expr} \text{InstVar}(re, v) : \theta}$$

Beispiel InstVar

```
class A {  
    Integer attr;  
    A meth(Boolean x) { ... }  
}
```

$$O_A = \{ \text{attr} : \text{Integer}, \text{meth} : \text{Boolean} \rightarrow A \}$$

```
...  
A a = new A()  
a.attr = 5;
```

übersetzt in abstrakte syntax: **InstVar(LocalOrFieldVar(*a*), *attr*).**

Beispiel InstVar

```
class A {  
    Integer attr;  
    A meth(Boolean x) { ... }  
}
```

$$O_A = \{ \text{attr} : \text{Integer}, \text{meth} : \text{Boolean} \rightarrow A \}$$

```
...  
A a = new A()  
a.attr = 5;
```

übersetzt in abstrakte syntax: **InstVar**(**LocalOrFieldVar**(*a*), *attr*).

$$\text{[Ident]} \quad \frac{\{ a : A \}}{\{ a : A \} \triangleright_{Id} a : A}$$

Beispiel InstVar

```
class A {  
    Integer attr;  
    A meth(Boolean x) { ... }  
}
```

$$O_A = \{ \text{attr} : \text{Integer}, \text{meth} : \text{Boolean} \rightarrow A \}$$

```
...  
A a = new A()  
a.attr = 5;
```

übersetzt in abstrakte syntax: **InstVar**(**LocalOrFieldVar**(*a*), *attr*).

| | |
|--|--|
| | $\frac{\{a : A\}}{\text{[Ident]}} \quad \frac{\{a : A\} \triangleright_{Id} a : A}{\text{[LocalOrFieldVar]}} \quad \frac{\{a : A\} \triangleright_{Expr} \text{LocalOrFieldVar}(a) : A}{\text{[LocalOrFieldVar]}}$ |
|--|--|

$$\frac{O_A}{\text{[LocalOrFieldVar]}}$$

Beispiel InstVar

```
class A {  
    Integer attr;  
    A meth(Boolean x) { ... }  
}
```

$$O_A = \{ \text{attr} : \text{Integer}, \text{meth} : \text{Boolean} \rightarrow A \}$$

```
...  
A a = new A()  
a.attr = 5;
```

übersetzt in abstrakte syntax: **InstVar**(**LocalOrFieldVar**(*a*), *attr*).

$$\begin{array}{c} \text{[Ident]} \quad \frac{\{ a : A \}}{\text{[LocalOrFieldVar]} \quad \frac{\{ a : A \} \triangleright_{Id} a : A}{\{ a : A \} \triangleright_{Expr} \text{LocalOrFieldVar}(a) : A,}} \end{array} \quad \begin{array}{c} \text{[Ident]} \quad \frac{O_A}{O_A \triangleright_{Id} \text{attr} : \text{Integer}} \end{array}$$

Beispiel InstVar

```
class A {  
    Integer attr;  
    A meth(Boolean x) { ... }  
}
```

$O_A = \{ \text{attr} : \text{Integer}, \text{meth} : \text{Boolean} \rightarrow A \}$

...

```
A a = new A()  
a.attr = 5;
```

übersetzt in abstrakte syntax: **InstVar**(**LocalOrFieldVar**(*a*), *attr*).

$$\begin{array}{c} \text{[Ident]} \quad \frac{\{a : A\}}{\{a : A\} \triangleright_{Id} a : A} \quad \text{[Ident]} \quad \frac{O_A}{O_A \triangleright_{Id} \text{attr} : \text{Integer}} \\ \text{[LocalOrFieldVar]} \quad \frac{\{a : A\} \triangleright_{Id} a : A, \quad O_A \triangleright_{Id} \text{attr} : \text{Integer}}{\{a : A\} \triangleright_{Expr} \text{LocalOrFieldVar}(a) : A, \quad O_A \triangleright_{Id} \text{attr} : \text{Integer}} \\ \text{[InstVar]} \quad \frac{\{a : A\} \triangleright_{Expr} \text{LocalOrFieldVar}(a) : A, \quad O_A \triangleright_{Id} \text{attr} : \text{Integer}}{\{a : A\} \triangleright_{Expr} \text{InstVar}(\text{LocalOrFieldVar}(a), \text{attr}) : \text{Integer}} \end{array}$$

Expression–Regel: Statement–Expressions

[New] $O \triangleright_{Expr} \text{New}(\theta) : \theta$

Expression–Regel: Statement–Expressions

[New] $O \triangleright_{Expr} \text{New}(\theta) : \theta$

[Assign]
$$\frac{O \triangleright_{Expr} ve : \theta', O \triangleright_{Expr} e : \theta}{O \triangleright_{Expr} \text{Assign}(ve, e) : \theta'} \quad \theta \leq^* \theta'^1$$

¹ \leq^* ist die Subtypen–Relation

Expression–Regel: Statement–Expressions

[New] $O \triangleright_{Expr} \text{New}(\theta) : \theta$

[Assign]
$$\frac{O \triangleright_{Expr} ve : \theta', O \triangleright_{Expr} e : \theta}{O \triangleright_{Expr} \text{Assign}(ve, e) : \theta'} \quad \theta \leq^* \theta'^1$$

[Method-Call]
$$\frac{\begin{array}{l} O \triangleright_{Expr} re : \bar{\tau} \\ O_{\bar{\tau}} \triangleright_{Id} m : \theta'_1 \times \dots \times \theta'_n \rightarrow \theta \\ \forall 1 \leq i \leq n : O \triangleright_{Expr} e_i : \theta_i \end{array}}{O \triangleright_{Expr} \text{MethodCall}(re, m, (e_1, \dots, e_n)) : \theta} \quad \theta_i \leq^* \theta'_i$$

¹ \leq^* ist die Subtypen–Relation

Beispiel MethodCall

```
class A {  
    Integer attr;  
    A meth(Boolean x) { ... }  
}
```

$$O_A = \{ \text{attr} : \text{Integer}, \text{meth} : \text{Boolean} \rightarrow A \}$$

...

```
A a = new A()
```

```
A aa = a.meth(true);
```

übers. in abstrakte Syntax:

MethodCall(**LocalOrFieldVar**(*a*), *meth*, **Bool**(*true*)).

Beispiel MethodCall

```
class A {  
    Integer attr;  
    A meth(Boolean x) { ... }  
}
```

$$O_A = \{ \text{attr} : \text{Integer}, \text{meth} : \text{Boolean} \rightarrow A \}$$

...

```
A a = new A()
```

```
A aa = a.meth(true);
```

übers. in abstrakte Syntax:

MethodCall(**LocalOrFieldVar**(*a*), *meth*, **Bool**(true)).

$$\text{[Ident]} \quad \frac{\{a : A\}}{\{a : A\} \triangleright_{Id} a : A}$$

Beispiel MethodCall

```
class A {  
    Integer attr;  
    A meth(Boolean x) { ... }  
}
```

$O_A = \{ \text{attr} : \text{Integer}, \text{meth} : \text{Boolean} \rightarrow A \}$

...

A a = new A()

A aa = a.meth(true);

übers. in abstrakte Syntax:

MethodCall(**LocalOrFieldVar**(a), meth, Bool(true)).

| | | |
|-------------------|---|--|
| | $\{a : A\}$ | |
| [Ident] | _____ | |
| | $\{a : A\} \triangleright_{Id} a : A$ | |
| [LocalOrFieldVar] | _____ | |
| | $\{a : A\} \triangleright_{Expr} \text{LocalOrFieldVar}(a) : A$ | |

_____ O_A

Beispiel MethodCall

```
class A {  
    Integer attr;  
    A meth(Boolean x) { ... }  
}
```

$O_A = \{ \text{attr} : \text{Integer}, \text{meth} : \text{Boolean} \rightarrow A \}$

...

A a = new A()

A aa = a.meth(true);

übers. in abstrakte Syntax:

MethodCall(**LocalOrFieldVar**(a), meth, Bool(true)).

$$\begin{array}{c} \text{[Ident]} \quad \frac{\{a : A\}}{\{a : A\} \triangleright_{Id} a : A} \\ \text{[LocalOrFieldVar]} \quad \frac{\{a : A\} \triangleright_{Id} a : A}{\{a : A\} \triangleright_{Expr} \text{LocalOrFieldVar}(a) : A,} \end{array} \quad \begin{array}{c} \text{[Ident]} \quad \frac{O_A}{O_A \triangleright_{Id} \text{meth} : \text{Boolean} \rightarrow A} \end{array}$$

Beispiel MethodCall

```
class A {  
    Integer attr;  
    A meth(Boolean x) { ... }  
}
```

$$O_A = \{ \text{attr} : \text{Integer}, \text{meth} : \text{Boolean} \rightarrow A \}$$

...

```
A a = new A()
```

```
A aa = a.meth(true);
```

übers. in abstrakte Syntax:

MethodCall(**LocalOrFieldVar**(*a*), *meth*, Bool(*true*)).

| | |
|--|--|
| $\frac{\text{[Ident]} \quad \frac{\{a : A\}}{\{a : A\} \triangleright_{Id} a : A}}{\text{[LocalOrFieldVar]} \quad \{a : A\} \triangleright_{Expr} \text{LocalOrFieldVar}(a) : A,}$ | $\frac{\text{[Ident]} \quad \frac{O_A}{O_A \triangleright_{Id} \text{meth} : \text{Boolean} \rightarrow A}}{\text{[BoolLiteral]} \quad \text{Bool}(\text{true}) : \text{boolean}}$ |
|--|--|

Beispiel MethodCall

```
class A {  
    Integer attr;  
    A meth(Boolean x) { ... }  
}
```

$O_A = \{ \text{attr} : \text{Integer}, \text{meth} : \text{Boolean} \rightarrow A \}$

...

A a = new A()

A aa = a.meth(true);

übers. in abstrakte Syntax:

MethodCall(**LocalOrFieldVar**(a), meth, Bool(true)).

| | | |
|-------------------|---|--|
| | $\{ a : A \}$ | |
| [Ident] | <hr/> | |
| | $\{ a : A \} \triangleright_{Id} a : A$ | |
| [LocalOrFieldVar] | <hr/> | |
| | $\{ a : A \} \triangleright_{Expr} \text{LocalOrFieldVar}(a) : A,$ | |
| | | O_A |
| | | <hr/> |
| | | $O_A \triangleright_{Id} \text{meth} : \text{Boolean} \rightarrow A$ |
| | | [Ident] |
| | | $O_A \triangleright_{Id} \text{meth} : \text{Boolean} \rightarrow A$ |
| | | |
| | | [BoolLiteral] Bool(true) : boolean |
| [MethodCall] | <hr/> | |
| | $\{ a : A \} \triangleright_{Expr} \text{MethodCall}(\text{LocalOrFieldVar}(a), \text{meth}, \text{Bool}(\text{true})) : A$ | |

Statement-Regeln

$$\text{[Return]} \frac{O \triangleright_{Expr} e : \theta}{O \triangleright_{Stmt} \text{Return}(e) : \theta}$$

Statement-Regeln

$$\text{[Return]} \quad \frac{O \triangleright_{Expr} e : \theta}{O \triangleright_{Stmt} \text{Return}(e) : \theta}$$

$$\text{[If]} \quad \frac{\begin{array}{c} O \triangleright_{Stmt} s_1 : \theta_1, O \triangleright_{Stmt} s_2 : \theta_2 \\ O \triangleright_{Expr} e : \text{boolean} \end{array}}{O \triangleright_{Stmt} \text{If}(e, s_1, s_2) : \bar{\theta}, \text{ wobei } \bar{\theta} \in \mathbf{UB}^2(\theta_1, \theta_2)}$$

Statement-Regeln

$$\text{[Return]} \quad \frac{O \triangleright_{Expr} e : \theta}{O \triangleright_{Stmt} \text{Return}(e) : \theta}$$

$$\text{[If]} \quad \frac{\begin{array}{c} O \triangleright_{Stmt} s_1 : \theta_1, O \triangleright_{Stmt} s_2 : \theta_2 \\ O \triangleright_{Expr} e : \text{boolean} \end{array}}{O \triangleright_{Stmt} \text{If}(e, s_1, s_2) : \bar{\theta}, \text{ wobei } \bar{\theta} \in \mathbf{UB}^2(\theta_1, \theta_2)}$$

$$\text{[While]} \quad \frac{O \triangleright_{Expr} e : \text{boolean}, O \triangleright_{Stmt} \text{Block}(B) : \theta}{O \triangleright_{Stmt} \text{While}(e, \text{Block}(B)) : \theta}$$

Statement–Regeln: Statement–Expressions

[New] $O \triangleright_{Stmt} \text{New}(\theta) : \text{void}$

Statement–Regeln: Statement–Expressions

[New] $O \triangleright_{Stmt} \text{New}(\theta) : \text{void}$

[Assign]
$$\frac{O \triangleright_{Expr} \text{ve} : \theta', O \triangleright_{Expr} e : \theta}{O \triangleright_{Stmt} \text{Assign}(\text{ve}, e) : \text{void}} \quad \theta \leq^* \theta'$$

Statement–Regeln: Statement–Expressions

[New] $O \triangleright_{Stmt} \text{New}(\theta) : \text{void}$

[Assign]
$$\frac{O \triangleright_{Expr} \text{ve} : \theta', O \triangleright_{Expr} e : \theta}{O \triangleright_{Stmt} \text{Assign}(\text{ve}, e) : \text{void}} \quad \theta \leq^* \theta'$$

[Method-Call]
$$\frac{\begin{array}{l} O \triangleright_{Expr} re : \bar{\tau} \\ O_{\bar{\tau}} \triangleright_{Id} m : \theta'_1 \times \dots \times \theta'_n \rightarrow \theta \\ \forall 1 \leq i \leq n : O \triangleright_{Expr} e_i : \theta_i \end{array}}{O \triangleright_{Stmt} \text{MethodCall}(re, m, (e_1, \dots, e_n)) : \text{void}} \quad \theta_i \leq^* \theta'_i$$

Block-Statement Regeln

$$\frac{O \triangleright_{Stmt} stmt : \theta}{O \triangleright_{Stmt} \text{Block}(stmt) : \theta} [\text{BlockInit}]$$

Block-Statement Regeln

$$\text{[BlockInit]} \quad \frac{O \triangleright_{Stmt} \text{stmt} : \theta}{O \triangleright_{Stmt} \text{Block}(\text{stmt}) : \theta}$$

$$\text{[Block]} \quad \frac{O \triangleright_{Stmt} s_1 : \theta, O \triangleright_{Stmt} \text{Block}(s_2; \dots; s_n) : \theta'}{O \triangleright_{Stmt} \text{Block}(s_1; s_2; \dots; s_n) : \bar{\theta}, \text{ wobei } \bar{\theta} \in \mathbf{UB}(\theta, \theta')}$$

Block-Statement Regeln

$$\text{[BlockInit]} \quad \frac{O \triangleright_{\text{Stmt}} \text{stmt} : \theta}{O \triangleright_{\text{Stmt}} \text{Block}(\text{stmt}) : \theta}$$

$$\text{[Block]} \quad \frac{O \triangleright_{\text{Stmt}} s_1 : \theta, \quad O \triangleright_{\text{Stmt}} \text{Block}(s_2; \dots; s_{n_i}) : \theta'}{O \triangleright_{\text{Stmt}} \text{Block}(s_1; s_2; \dots; s_{n_i}) : \bar{\theta}, \text{ wobei } \bar{\theta} \in \mathbf{UB}(\theta, \theta')}$$

$$\text{[Blockvoid]} \quad \frac{\begin{array}{l} O \triangleright_{\text{Stmt}} s_1 : \text{void}, \\ O \triangleright_{\text{Stmt}} \text{Block}(s_2; \dots; s_{n_i}) : \theta \end{array}}{O \triangleright_{\text{Stmt}} \text{Block}(s_1; s_2; \dots; s_{n_i}) : \theta}$$

Block-Statement Regeln

$$\text{[BlockInit]} \quad \frac{O \triangleright_{\text{Stmt}} \text{stmt} : \theta}{O \triangleright_{\text{Stmt}} \text{Block}(\text{stmt}) : \theta}$$

$$\text{[Block]} \quad \frac{O \triangleright_{\text{Stmt}} s_1 : \theta, \quad O \triangleright_{\text{Stmt}} \text{Block}(s_2; \dots; s_n) : \theta'}{O \triangleright_{\text{Stmt}} \text{Block}(s_1; s_2; \dots; s_n) : \bar{\theta}, \text{ wobei } \bar{\theta} \in \mathbf{UB}(\theta, \theta')}$$

$$\text{[Blockvoid]} \quad \frac{\begin{array}{l} O \triangleright_{\text{Stmt}} s_1 : \text{void}, \\ O \triangleright_{\text{Stmt}} \text{Block}(s_2; \dots; s_n) : \theta \end{array}}{O \triangleright_{\text{Stmt}} \text{Block}(s_1; s_2; \dots; s_n) : \theta}$$

$$\text{[Block-Local-VarDecl]} \quad \frac{O \setminus \{v : \theta'\} \cup \{v : \bar{\theta}\} \triangleright_{\text{Stmt}} \text{Block}(s_2; \dots; s_n) : \theta}{O \triangleright_{\text{Stmt}} \text{Block}(\text{LocalVarDecl}(v, \bar{\theta}); s_2; \dots; s_n) : \theta}$$

Beispiel

```
while (true) {  
    return 1;  
}
```

Beispiel

```
while (true) {  
    return 1;  
}
```

$O = \emptyset$

Beispiel

```
while (true) {  
    return 1;  
}
```

$O = \emptyset$

[IntLiteral]

$O \triangleright_{Expr} \text{Int}(1) : \text{int}$

Beispiel

```
while (true) {  
    return 1;  
}
```

$O = \emptyset$

$$\frac{\begin{array}{l} [\text{IntLiteral}] \\ [\text{Return}] \end{array} \quad O \triangleright_{Expr} \text{Int}(1) : \text{int}}{O \triangleright_{Stmt} \text{Return}(\text{Int}(1)) : \text{int}}$$

Beispiel

```
while (true) {  
    return 1;  
}
```

$O = \emptyset$

$$\begin{array}{c} \text{[IntLiteral]} \quad O \triangleright_{Expr} \text{Int}(1) : \text{int} \\ \text{[Return]} \quad \hline O \triangleright_{Stmt} \text{Return}(\text{Int}(1)) : \text{int} \\ \text{[Block -]} \quad \hline \text{[Init]} \quad O \triangleright_{Stmt} \text{Block}(\text{Return}(\text{Int}(1))) : \text{int} \end{array}$$

Beispiel

```
while (true) {  
    return 1;  
}
```

$O = \emptyset$

[Bool-Literal] $O \triangleright_{Expr} \text{Bool}(True) : \text{boolean}$

[IntLiteral] $O \triangleright_{Expr} \text{Int}(1) : \text{int}$
[Return] $O \triangleright_{Stmt} \text{Return}(\text{Int}(1)) : \text{int}$
[Block-Init] $O \triangleright_{Stmt} \text{Block}(\text{Return}(\text{Int}(1))) : \text{int}$

Beispiel

```
while (true) {  
    return 1;  
}
```

$O = \emptyset$

$$\begin{array}{c} \text{[Bool-Literal]} \quad O \triangleright_{Expr} \text{Bool}(True) : \text{boolean} \\ \text{[while]} \quad \frac{}{O \triangleright_{Stmt} \text{While}(\text{Bool}(True), \text{Block}(\text{Return}(\text{Int}(1)))) : \text{int}} \end{array}$$
$$\begin{array}{c} \text{[IntLiteral]} \quad O \triangleright_{Expr} \text{Int}(1) : \text{int} \\ \text{[Return]} \quad \frac{}{O \triangleright_{Stmt} \text{Return}(\text{Int}(1)) : \text{int}} \\ \text{[Block - Init]} \quad \frac{}{O \triangleright_{Stmt} \text{Block}(\text{Return}(\text{Int}(1))) : \text{int}} \end{array}$$

Datenstruktur typisierter Expressions

```
type Type = String
```

| | |
|---------------------------------------|--------------------|
| -- data Type = TVar(String) | — Typvariable |
| -- TC(String, [Type]) | — Typkonstruktor |
| -- WC | — Wildscard |
| -- WC_Super(Type) | — Super-Wildscard |
| -- WC_Extends(Type) | — Extends-Wildcard |

Datenstruktur typisierter Expressions

```
type Type = String
```

| | |
|---------------------------------------|--------------------|
| -- data Type = TVar(String) | — Typvariable |
| -- TC(String, [Type]) | — Typkonstruktor |
| -- WC | — Wildscard |
| -- WC_Super(Type) | — Super-Wildscard |
| -- WC_Extends(Type) | — Extends-Wildcard |

```
data Expr = This
  | Super
  | LocalOrFieldVar(String)
  | InstVar(Expr, String)
  | Unary(String, Expr)
  | Binary(String, Expr, Expr)
  | Integer(Integer)
  | Bool(Bool)
  | Char(Char)
  | String(String)
  | Jnull
  | StmtExprExpr(StmtExpr)
  | TypedExpr(Expr, Type)
```

Datenstruktur typisierter Statements

```
data Stmt = Block([Stmt])
          | Return( Expr )
          | While( Expr , Stmt )
          | LocalVarDecl(String)
          | If(Expr, Stmt , Maybe Stmt)
          | StmtExprStmt(StmtExpr)
          | TypedStmt(Stmt, Type)

data StmtExpr = Assign(String, Expr)
              | New(Type, [Expr])
              | MethodCall(Expr, String, [Expr])
              | TypedStmtExpr(StmtExpr, Type)
```

Typisierung von Simple-Expressions

```
return 1 + 2
```

Erg. der Parsers:

```
Binary("+", Integer(1), Integer(2))
```

Typisierung von Simple-Expressions

```
return 1 + 2
```

Erg. der Parsers:

```
Binary("+", Integer(1), Integer(2))
```

Typisierung:

```
TypedExpr(Binary("+",  
                  TypedExpr(Integer(1), "int"),  
                  TypedExpr(Integer(2), "int"))  
            "int")
```

LocalOrFieldVar

```
class Klassenname {  
    int v;  
    int methode (int x) {  
        return v + x;  
    }  
}
```

LocalOrFieldVar

```
class Klassenname {  
    int v;  
    int methode (int x) {  
        return v + x;  
    }  
}
```

Erg. der Parsers:

```
Return(Binary("+",  
              LocalOrFieldVar("v")      ← FieldVar  
              LocalOrFieldVar("x")))    ← LocalVar
```


LocalOrFieldVar

```
class Klassenname {  
    int v;  
    int methode (int x) {  
        return v + x;  
    }  
}
```

Erg. der Parsers:

```
Return(Binary("+",  
              LocalOrFieldVar("v")      ← FieldVar  
              LocalOrFieldVar("x")))    ← LocalVar
```

Typisierung:

```
Return(  
    TypedExpr(  
        Binary("+",  
                TypedExpr(LocalOrFieldVar("v"), "int"),  
                TypedExpr(LocalOrFieldVar("x"), "int")),  
        "int"))
```

InstVar, MethodCall

```
class C11 {  
  char m1 () {  
    int b;  
    C12 x = new C12 ()  
    return x.m2(x.v, b);  
  }  
}
```

```
class C12 {  
  C13 v;  
  char m2(C13 v, int w) { ...}  
}  
  
class C13 { ...}
```

InstVar, MethodCall

```
class C11 {  
  char m1 () {  
    int b;  
    C12 x = new C12 ()  
    return x.m2(x.v, b);  
  }  
}
```

```
class C12 {  
  C13 v;  
  char m2(C13 v, int w) { ...}  
}  
  
class C13 { ...}
```

Erg. der Parsers:

```
Return(MethodCall(LocalOrFieldVar("x"), "m2",  
  [InstVar(LocalOrFieldVar("x"), "v"),  
   LocalOrFieldVar("b")]))
```

InstVar, MethodCall

```
class C11 {  
    char m1 () {  
        int b;  
        C12 x = new C12 ()  
        return x.m2(x.v, b);  
    }  
}
```

```
class C12 {  
    C13 v;  
    char m2(C13 v, int w) { ...}  
}  
  
class C13 { ...}
```

Erg. der Parsers:

```
Return(MethodCall(LocalOrFieldVar("x"), "m2",  
    [InstVar(LocalOrFieldVar("x"), "v"),  
     LocalOrFieldVar("b")]))
```

Typisierung:

```
Return(TypedExpr(  
    MethodCall(TypedExpr(LocalOrFieldVar("x"), "C12"), "m2",  
        [TypedExpr(  
            InstVar(TypedExpr(LocalOrFieldVar("x"), "C12"),  
                "v"), "C13"),  
            TypedExpr(LocalOrFieldVar("b"), "int")]), "char"))
```

Semantische Analyse/Algorithmus typecheck

typecheckExpr :: Expr -> [(String, Type)] -> [Class] -> Expr

typecheckStmt :: Stmt -> [(String, Type)] -> [Class] -> Stmt

1. Argument: Ungetypter Ausdruck/Statement
 2. Argument: Tabelle mit lokalen Variablen, die durch Deklarationen LocalVarDecl verändert werden.
 3. Argument: Alle sichtbaren Klassen mit Fields und Methoden
- Ergebnis: getypeter Ausdruck/Statement

Semantische Analyse/Algorithmus typecheck

typecheckExpr :: Expr -> [(String, Type)] -> [Class] -> Expr

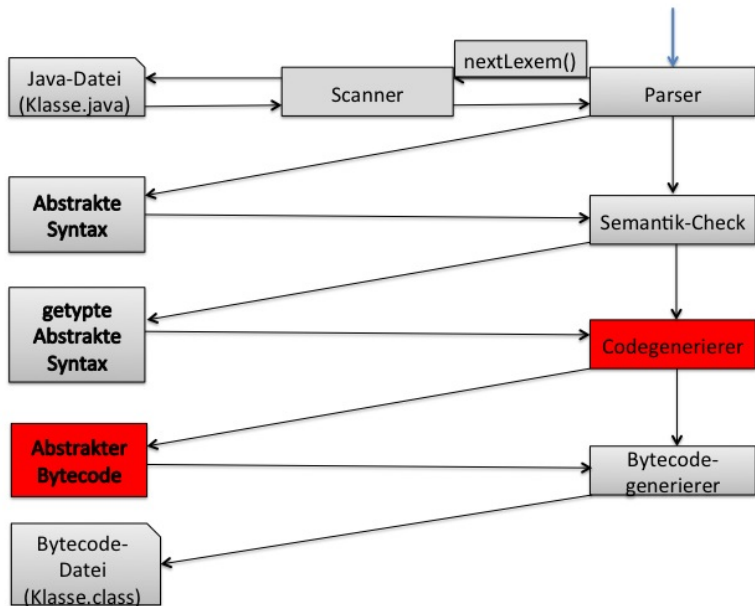
typecheckStmt :: Stmt -> [(String, Type)] -> [Class] -> Stmt

1. Argument: Ungetypter Ausdruck/Statement
 2. Argument: Tabelle mit lokalen Variablen, die durch Deklarationen LocalVarDecl verändert werden.
 3. Argument: Alle sichtbaren Klassen mit Fields und Methoden
- Ergebnis: getypeter Ausdruck/Statement

Ablauf: Lauf über alle Methoden aller Klassen:

- ▶ Check ob Variablen/Methoden deklariert
- ▶ Check, ob die Typen der Metdodenaufrufe/Zuweisung korrekt sind
- ▶ Einfügen der Typisierungen

Codegenerierung



Codegenerierung

Typisierte abstrakte Syntax -> Abstrakten Bytecode

Abstrakter Bytecode (HackageDB)³

```
data ClassFile =
ClassFile { magic           :: Magic           -- CAFEBABE
           , minver         :: MinorVersion    -- Versionen
           , maxver         :: MajorVersion
           , count_cp       :: ConstantPool_Count -- Anz. Eintr. Konst.p
           , array_cp       :: CP_Infos        -- Konstantenpool
           , acfg           :: AccessFlags     -- Berechtigungen
           , this           :: ThisClass       -- This-Klasse
           , super          :: SuperClass      -- Super-Klasse
           , count_interfaces :: Interfaces_Count -- Anz. Interfaces
           , array_interfaces :: Interfaces     -- Interfaces
           , count_fields   :: Fields_Count    -- Anzahl Fields
           , array_fields   :: Field_Infos     -- Fields
           , count_methods  :: Methods_Count   -- Methoden
           , array_methods  :: Method_Infos    -- Methoden
           , count_attributes :: Attributes_Count -- Anz. Attribute
           , array_attributes :: Attribute_Infos -- Attribute
           }
}
```



```
type CP_Infos      = [CP_Info]
type Interfaces    = [Interface]
type Field_Infos   = [Field_Info]
type Method_Infos  = [Method_Info]
type Attribute_Infos = [Attribute_Info]

data Magic = Magic

data MinorVersion = MinorVersion {
    numMinVer :: Int
}

data MajorVersion = MajorVersion {
    numMaxVer :: Int
}
```

Konstantenpool-Einträge I

```
data CP_Info =
    Class_Info
        { tag_cp           :: Tag
        , index_cp         :: Index_Constant_Pool
        , desc             :: String           }
    | FieldRef_Info
        { tag_cp           :: Tag
        , index_name_cp     :: Index_Constant_Pool
        , index_nameandtype_cp :: Index_Constant_Pool
        , desc             :: String           }
    | MethodRef_Info
        { tag_cp           :: Tag
        , index_name_cp     :: Index_Constant_Pool
        , index_nameandtype_cp :: Index_Constant_Pool
        , desc             :: String           }
    | InterfaceMethodRef_Info
        { tag_cp           :: Tag
        , index_name_cp     :: Index_Constant_Pool
        , index_nameandtype_cp :: Index_Constant_Pool
        , desc             :: String           }
```

Konstantenpool-Einträge II

```
| String_Info
    { tag_cp          :: Tag
      , index_cp      :: Index_Constant_Pool
      , desc          :: String
    }

| Integer_Info
    { tag_cp          :: Tag
      , numi_cp       :: Int
      , desc          :: String
    }

| Float_Info
    { tag_cp          :: Tag
      , numf_cp       :: Float
      , desc          :: String
    }

| Long_Info
    { tag_cp          :: Tag
      , numi_l1_cp    :: Int
      , numi_l2_cp    :: Int
      , desc          :: String
    }

| Double_Info
    { tag_cp          :: Tag
      , numi_d1_cp    :: Int
      , numi_d2_cp    :: Int
      , desc          :: String
    }
```

Konstantenpool-Einträge III

```
| NameAndType_Info
    { tag_cp           :: Tag
      , index_name_cp  :: Index_Constant_Pool
      , index_descr_cp :: Index_Constant_Pool
      , desc           :: String           }
| Utf8_Info
    { tag_cp           :: Tag
      , tam_cp         :: Int
      , cad_cp         :: String
      , desc           :: String           }
```

```
data Tag = TagClass
  | TagFieldRef
  | TagMethodRef
  | TagInterfaceMethodRef
  | TagString
  | TagInteger
  | TagFloat
  | TagLong
  | TagDouble
  | TagNameAndType
  | TagUtf8
```

Field_Info und Method_Infos

```
data Field_Info = Field_Info
    { af-fi          :: AccessFlags
    , index_name-fi  :: Index_Constant_Pool    -- name_index
    , index_descr-fi :: Index_Constant_Pool    -- descriptor_index
    , tam-fi         :: Int                    -- count_attributte
    , array_attr-fi  :: Attribute_Infos
    }
```

```
data Method_Info = Method_Info
    { af-mi          :: AccessFlags
    , index_name-mi   :: Index_Constant_Pool    -- name_index
    , index_descr-mi  :: Index_Constant_Pool    -- descriptor_index
    , tam-mi          :: Int                    -- attributes_count
    , array_attr-mi   :: Attribute_Infos
    }
```

Attribute_Info

```
data Attribute_Info =
  ...
| AttributeCode
  { index_name_attr      :: Index_Constant_Pool  -- attribute_name_index
  , tam_len_attr         :: Int                  -- attribute_length
  , len_stack_attr       :: Int                  -- max_stack
  , len_local_attr       :: Int                  -- max_local
  , tam_code_attr        :: Int                  -- code_length
  , array_code_attr      :: ListaInt             -- code como array
                                              -- de bytes
-- , array_code_attr      :: [Code]              -- code array (altern.)
  , tam_ex_attr          :: Int                  -- exceptions_length
  , array_ex_attr        :: Tupla4Int            -- no usamos
  , tam_attr_attr        :: Int                  -- attributes_count
  , array_attr_attr      :: Attribute_Infos
  }
| ...
```

Beispiel: Byte-Code

```
class bct {  
    Integer i;  
}
```

```
> javac -g:none bct.java
```

```
magic = 0x CAFEBAFE
minor_version = 0
major_version = 52
constant_pool_count = 12
constant_pool =
{
1| tag = CONSTANT_Methodref, class_index = 3, name_and_type_index = 9
2| tag = CONSTANT_Class, name_index = 10
3| tag = CONSTANT_Class, name_index = 11
4| tag = CONSTANT_Utf8, length = 1, bytes = i
5| tag = CONSTANT_Utf8, length = 19, bytes =Ljava/lang/Integer;
6| tag = CONSTANT_Utf8, length = 6, bytes = <init>
7| tag = CONSTANT_Utf8, length = 3, bytes = ()V
8| tag = CONSTANT_Utf8, length = 4, bytes = Code
9| tag = CONSTANT_NameAndType, name_index = 6, descriptor_index = 7
10| tag = CONSTANT_Utf8, length = 3, bytes = bct
11| tag = CONSTANT_Utf8, length = 16, bytes = java/lang/Object
}
access_flags = 32 // ACC_SUPER
this_class = #2 // bct
super_class = #3 // java/lang/Object
interfaces_count = 0
interfaces = {}
```



```
fields_count = 1
fields [0] =
{
  access_flags = 0
  name_index = #4 // i
  descriptor_index = #5 // Ljava/lang/Integer;
  attributes_count = 0
  attributes = {}
}
methods_count = 1
methods [0] =
{
  access_flags = 0
  name_index = #6 // <init>
  descriptor_index = #7 // ()V
```

```
attributes_count = 1
attributes [0] =
{
  attribute_name_index = #8  // Code
  attribute_length = 17
  max_stack = 1, max_locals = 1
  code_length = 5
  code =
  {
    0  aload_0
    1  invokespecial #1  // java/lang/Object.<init> ()V
    4  return
  }
  exception_table_length = 0
  exception_table = {}
  attributes_count = 0
  attributes = {}
}
}
attributes_count = 0
attributes = {}
```

JVM – Zur Laufzeit

Frame: Für jeden Methodenaufruf wird ein Frame erzeugt:

- ▶ Array von lokalen Variablen
- ▶ Operanden-Stack
- ▶ Referenz zum zugehörigen Konstantenpool

JVM – Zur Laufzeit

Frame: Für jeden Methodenaufruf wird ein Frame erzeugt:

- ▶ Array von lokalen Variablen
- ▶ Operanden-Stack
- ▶ Referenz zum zugehörigen Konstantenpool

Stack: Auf dem Stack liegen die Frames, der aufgerufenen Methoden

JVM – Zur Laufzeit

Frame: Für jeden Methodenaufruf wird ein Frame erzeugt:

- ▶ Array von lokalen Variablen
- ▶ Operanden-Stack
- ▶ Referenz zum zugehörigen Konstantenpool

Stack: Auf dem Stack liegen die Frames, der aufgerufenen Methoden

Heap: Speicher für alle Objekte

JVM – Zur Laufzeit

Frame: Für jeden Methodenaufruf wird ein Frame erzeugt:

- ▶ Array von lokalen Variablen
- ▶ Operanden-Stack
- ▶ Referenz zum zugehörigen Konstantenpool

Stack: Auf dem Stack liegen die Frames, der aufgerufenen Methoden

Heap: Speicher für alle Objekte

Method Area: Speicher für alle Methoden

Code-Übersetzung

Standard-Konstruktor

```
class bct {  
    bct() {  
        super();  
    }  
}
```

führt zu

```
0  aload_0  
1  invokespecial #1    // java/lang/Object.<init> ()V  
4  return
```

aload_<n>, aload

Code: aload_<n>

Format: aload_<n>

Bechreibung: Lädt die Referenz der n -ten lokalen Variablen auf den Stack

Formen: aload_0 = 42 (0x2a)

aload_1 = 43 (0x2b)

aload_2 = 44 (0x2c)

aload_3 = 45 (0x2d)

aload_<n>, aload

Code: **aload_<n>**

Format: **aload_<n>**

Bechreibung: Lädt die Referenz der n -ten lokalen Variablen auf den Stack

Formen: $\text{aload_0} = 42$ (0x2a)

$\text{aload_1} = 43$ (0x2b)

$\text{aload_2} = 44$ (0x2c)

$\text{aload_3} = 45$ (0x2d)

Code: **aload**

Format: **aload *index***

Bechreibung: Lädt die Variable *index*-te Variable auf den Stack

Formen: $\text{aload} = 25$ (0x19)

invokespecial

Code: `invokespecial`

Format: `invokespecial indexbyte1 indexbyte2`

Bechreibung: Ruft die Instanzmethode auf, die durch die Referenz `indexbyte1 << 8 | indexbyte2` in den Konstantenpool bestimmt wird. (Ohne dynamische Bindung!!!)

- ▶ Wenn die Methode nicht vorhanden ist wird in Superklassen gesucht.
- ▶ Die Argumente der Methoden müssen ebenfalls auf den Stack liegen.
- ▶ Am Ende werden die Argumente und das Objekt vom Stack gelöscht und das Ergebnis drauf gelegt.

Formen: `invokespecial = 183 (0xb7)`

Opcode: return

Code: return

Format: return

Bechreibung: Gibt void von einer Methode zurück.
Gibt die Kontrolle an die aufrufende Methode zurück.

Formen: return = 177 (0xb1)

Variablen beschreiben

```
class bct {  
    int i;  
  
    void m () {  
        i = 1;  
    }  
}
```

führt zu

```
0  aload_0      //this  
1  iconst_1  
2  putfield #2  // bct.i I  
5  return
```

iconst_<n>

Code: iconst_<i>

Format: iconst_<i>

Bechreibung: Push int Konstante auf den Stack

Formen: iconst_m1 = 2 (0x2)

iconst_0 = 3 (0x3)

iconst_1 = 4 (0x4)

iconst_2 = 5 (0x5)

iconst_3 = 6 (0x6)

iconst_4 = 7 (0x7)

iconst_5 = 8 (0x8)

bipush

Code: **bipush**

Format: **bipush byte**

Bechreibung: Push *byte*

Formen: **bipush** = 16 (0x10)

putfield

Code: **putfield**

Format: **putfield indexbyte1 indexbyte2**

Bechreibung: Ordnet das Oberste Element des Stacks dem Attribut **indexbyte1** $\ll 8$ | **indexbyte2** des Konstantenpools im Objekt des zweiobersten Elements des Stacks zu

Formen: **putfield = 181 (0xb5)**

Variablen auslesen und beschreiben

```
class bct {  
    int i;  
  
    void m () {  
        i = i + 1;  
    }  
}
```

führt zu

```
0  aload_0  //this  
1  aload_0  //this  
2  getfield #2  // bct.i I  
5  iconst_1  
6  iadd  
7  putfield #2  // bct.i I  
10 return
```


getfield

Code: `getfield`

Format: `getfield indexbyte1 indexbyte2`

Bechreibung: Liest das Attribut *indexbyte1* << 8 | *indexbyte2* im Konstantenpool des Obersten Element des Stacks aus, löscht das oberste Element und legt den gelesenen Wert auf den Stack.

Formen: `getfield = 180 (0xb4)`

iadd

Code: **iadd**

Format: **iadd**

Bechreibung: Addiert die beiden obersten Elemente des Stacks und nimmt sie vom Stack und legt das Ergebnis drauf.

Formen: $iadd = 96$ (0x60)

iadd

Code: **iadd**

Format: **iadd**

Bechreibung: Addiert die beiden obersten Elemente des Stacks und nimmt sie vom Stack und legt das Ergebnis drauf.

Formen: $iadd = 96$ (0x60)

Anlog: *isub*, *imul*, *idiv*, *iand*, *ior*, *ixor*, *ineg*, ...

Inkrement/Dekrement

```
class bct {  
  
    void m () {  
        int i = 0;  
        i++;  
        i--;  
    }  
}
```

führt zu

```
0  iconst_0  
1  istore_1  
2  iinc 1 1    //Variable 1 (+1)  
5  iinc 1 255  //Variable 1 (-1)  
8  return
```

istore_<n>

Code: istore_<n>

Format: istore_<n>

Bechreibung: Schreibt den Integerwert des obersten Elements des Operanden-Stacks in die n -te lokale Variable und löscht das oberste Element des Operanden-Stacks.

Formen: istore_0 = 59 (0x3b)

istore_1 = 60 (0x3c)

istore_2 = 61 (0x3d)

istore_3 = 62 (0x3e)

istore

Code: **istore**

Format: **istore index**

Bechreibung: Schreibt den Integerwert des obersten Elements des Operanden-Stacks in die *index*-te lokale Variable und löscht das oberste Element des Operanden-Stacks.

Formen: `istore = 54 (0x36)`

iinc

Code: **iinc**

Format: **iinc index const**

Bechreibung: Inkrement der lokalen Variable **index** durch
vozeichenbehaftete 8-Bit **const**.

Formen: iinc = 132 (0x84)

New

```
class A { }  
  
class bct {  
  
    void m () {  
        A aa = new A();  
    }  
}
```

führt zu

```
0  new #2  // A  
3  dup  
4  invokespecial #3  // A.<init> ()V  
7  astore_1  
8  return
```


Opcode: new

Code: new

Format: new indexbyte1 indexbyte2

Bechreibung: Erzeugt ein neues Objekt der Klasse *indexbyte1* << 8 | *indexbyte2* des Konstantenpool auf dem Heap und initialisiert die Attribute. Eine Referenz auf den Speicher in Heap wird auf den Stack gelegt.

Formen: new = 187 (0xbb)

dup

Code: **dup**

Format: **dup**

Bechreibung: Dubliziert das oberste stack Element

Formen: **dup** = 89 (0x59)

astore_<n>, astore

Code: astore_<n>

Format: astore_<n>

Bechreibung: Schreibt die oberste Referenz des Stacks in die *n*-te lokale Variable und löscht die Referenz vom Stack

Formen: astore_0 = 75 (0x4b)

astore_1 = 76 (0x4c)

astore_2 = 77 (0x4d)

astore_3 = 78 (0x4e)

astore_<n>, astore

Code: astore_<n>

Format: astore_<n>

Bechreibung: Schreibt die oberste Referenz des Stacks in die *n*-te lokale Variable und löscht die Referenz vom Stack

Formen: astore_0 = 75 (0x4b)

astore_1 = 76 (0x4c)

astore_2 = 77 (0x4d)

astore_3 = 78 (0x4e)

Code: astore

Format: astore index

Bechreibung: Schreibt die oberste Referenz des Stacks in die *index*-te lokale Variable und löscht die Referenz vom Stack

Formen: astore = 58 (0x3a)

Methodenaufruf

```
class A { int m2(int a)  return a;  }  
class bct {  
    void m () {  
        int j = 0;  
        A aa = new A();  
        int i = aa.m2(j); }  
}
```

führt zu

```
...  
2  new #2  // A  
5  dup  
6  invokespecial #3  // A.<init> ()V  
9  astore_2  
10 aload_2  
11 iload_1  
12 invokevirtual #4  // A.m2 (I)I  
15 istore_3  
16 return
```

iload_<n>, iload

Code: **iload_<n>**

Format: **iload_<n>**

Bechreibung: Lädt den Integerwert aus der n -ten lokalen Variable auf den Stack.

Formen: iload_0 = 26 (0x1a)

iload_1 = 27 (0x1b)

iload_2 = 28 (0x1c)

iload_3 = 29 (0x1d)

iload_<n>, iload

Code: **iload_<n>**

Format: **iload_<n>**

Bechreibung: Lädt den Integerwert aus der *n*-ten lokalen Variable auf den Stack.

Formen: iload_0 = 26 (0x1a)

iload_1 = 27 (0x1b)

iload_2 = 28 (0x1c)

iload_3 = 29 (0x1d)

Code: **iload**

Format: **iload index**

Bechreibung: Lädt den Integerwert aus der *index*-ten lokalen Variable auf den Stack.

Formen: iload = 21 (0x15)

invokevirtual

Code: `invokevirtual`

Format: `invokevirtual indexbyte1 indexbyte2`

Bechreibung: Ruft die Instanzmethode *abhängig von der aktuellen Receiver-Klasse* auf, die durch die Referenz `indexbyte1 << 8 | indexbyte2` in dem Konstantenpool bestimmt wird.

- ▶ Wenn die Methode nicht vorhanden ist wird in Superklassen gesucht.
- ▶ Die Argumente der Methoden müssen ebenfalls auf dem Stack liegen.
- ▶ Am Ende werden die Argumente und das Objekt vom Stack gelöscht und das Ergebnis drauf gelegt.

Formen: `invokevirtual = 182 (0xb6)`

Dynamische Bindung

```
class Superclass {  
  
    private void interestingMethod() {  
        System.out.println("Superclass's interesting method."); }  
  
    void exampleMethod() {  
        interestingMethod(); }  
}  
  
class Subclass extends Superclass {  
  
    void interestingMethod() {  
        System.out.println("Subclass's interesting method."); }  
  
    public static void main(String args[]) {  
        Subclass me = new Subclass();  
        me.exampleMethod(); }  
}
```

Dynamische Bindung

```
class Superclass {  
  
    private void interestingMethod() {  
        System.out.println("Superclass's interesting method."); }  
  
    void exampleMethod() {  
        interestingMethod(); }  
}  
  
class Subclass extends Superclass {  
  
    void interestingMethod() {  
        System.out.println("Subclass's interesting method."); }  
  
    public static void main(String args[]) {  
        Subclass me = new Subclass();  
        me.exampleMethod(); }  
}
```

Erg: Superclass's interesting method.

Dynamische Bindung

```
class Superclass {  
  
    void interestingMethod() {  
        System.out.println("Superclass's interesting method."); }  
  
    void exampleMethod() {  
        interestingMethod(); }  
}  
  
class Subclass extends Superclass {  
  
    void interestingMethod() {  
        System.out.println("Subclass's interesting method."); }  
  
    public static void main(String args[]) {  
        Subclass me = new Subclass();  
        me.exampleMethod(); }  
}
```

Dynamische Bindung

```
class Superclass {  
  
    void interestingMethod() {  
        System.out.println("Superclass's interesting method."); }  
  
    void exampleMethod() {  
        interestingMethod(); }  
}  
  
class Subclass extends Superclass {  
  
    void interestingMethod() {  
        System.out.println("Subclass's interesting method."); }  
  
    public static void main(String args[]) {  
        Subclass me = new Subclass();  
        me.exampleMethod(); }  
}
```

Erg: Subclass's interesting method.

Dynamische Bindung (Bytecode)

```
class Superclass {  
  
    private void interestingMethod() {  
        System.out.println("Superclass's interesting method."); }  
  
    void exampleMethod() {  
        interestingMethod(); }  
}
```

führt zu

```
0  aload_0  
1  invokespecial #5  // Superclass.interestingMethod ()V  
4  return
```

Dynamische Bindung (Bytecode)

```
class Superclass {  
  
    void interestingMethod() {  
        System.out.println("Superclass's interesting method."); }  
  
    void exampleMethod() {  
        interestingMethod(); }  
}
```

führt zu

```
0  aload_0  
1  invokevirtual #5  // interestingMethod ()V (dynamisch)  
4  return
```

Return

```
class bct {  
    bct b;  
  
    int m1() { return 1; }  
  
    bct m2() { return b; }  
}
```

führt zu

```
0  iconst_1          //m1  
1  ireturn  
  
0  aload_0           //m2  
1  getfield #2       // bct.b Lbct;  
4  areturn
```

areturn, ireturn

Code: areturn

Format: areturn

Bechreibung: Rückgabe des obersten Elements des (Operanden-)Stacks als Referenz auf ein Objekt. Legt die Referenz auf den (Operanden-)Stack des Frames der aufrufenden Methode.

Formen: areturn = 176 (0xb0)

areturn, ireturn

Code: areturn

Format: areturn

Bechreibung: Rückgabe des obersten Elements des (Operanden-)Stacks als Referenz auf ein Objekt. Legt die Referenz auf den (Operanden-)Stack des Frames der aufrufenden Methode.

Formen: areturn = 176 (0xb0)

Code: ireturn

Format: ireturn

Bechreibung: Rückgabe des obersten Elements des (Operanden-)Stacks als int, boolean, byte, short, char und legt Element auf den (Operanden-)Stack des Frames der aufrufenden Methode.

Formen: ireturn = 172 (0xb0)

While

```
class bct {  
  
    void m () {  
        int i = 0;  
        while (i == 1) { i++; }  
    }  
}
```

führt zu

```
0  iconst_0  
1  istore_1  
2  iload_1  
3  iconst_1  
4  if_icmpne 0 9  
7  iinc 1 1      //Variable 1 (+1)  
10 goto 255 248  
13 return
```

if_icmp<cond>

Code: if_icmp<cond>

Format: if_icmp<cond> branchbyte1 branchbyte2

Bechreibung: relativer Sprung nach 16 bit vorzeichenbehaftete Zahl
branchbyte1 << 8 | *branchbyte2* wenn jeweiliger Integervergleich des
zweitobersten mit den obersten Stack Element erfolgreich ist

Formen: if_icmpeq = 159 (0x9f)

if_icmpne = 160 (0xa0)

if_icmplt = 161 (0xa1)

if_icmpge = 162 (0xa2)

if_icmpgt = 163 (0xa3)

if_icmple = 164 (0xa4)

goto

Code: `goto`

Format: `goto branchbyte1 branchbyte2`

Bechreibung: relativer Sprung nach 16 bit vorzeichenbehaftete Zahl
 $\text{branchbyte1} \ll 8 \mid \text{branchbyte2}$

Formen: `goto = 167 (0xa7)`

If

```
class bct {  
    void m () {  
        int i = 0;  
        if (i == 1) { i++; }  
    }  
}
```

führt zu

```
0  iconst_0  
1  istore_1  
2  iload_1  
3  iconst_1  
4  if_icmpne 0 6  
7  iinc 1 1    //Variable 1 (+1)  
10 return
```

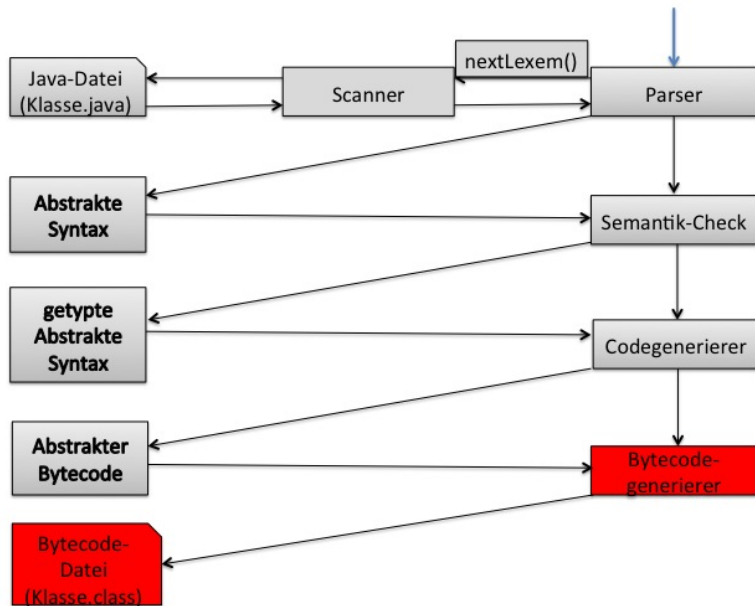
If_else

```
class bct {  
    void m () {  
        int i = 0;  
        if (i == 1) { i++; }  
        else { i--; }  
    }  
}
```

führt zu

```
0  iconst_0  
1  istore_1  
2  iload_1  
3  iconst_1  
4  if_icmpne 0 9  
7  iinc 1 1      //Variable 1 (+1)  
10 goto 0 6  
13 iinc 1 255    //Variable 1 (-1)  
16 return
```

Bytecodegenerierung



Abstrakter Bytecode -> Bytecode

Analyse des Binär-Files

Hexadezimal:

```
od -tx1 bct.class
```

| | | | | | | | | | | | | | | | | |
|----------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 00000000 | ca | fe | ba | be | 00 | 00 | 00 | 34 | 00 | 0c | 0a | 00 | 03 | 00 | 09 | 07 |
| 00000020 | 00 | 0a | 07 | 00 | 0b | 01 | 00 | 01 | 69 | 01 | 00 | 13 | 4c | 6a | 61 | 76 |
| 00000040 | 61 | 2f | 6c | 61 | 6e | 67 | 2f | 49 | 6e | 74 | 65 | 67 | 65 | 72 | 3b | 01 |
| 00000060 | 00 | 06 | 3c | 69 | 6e | 69 | 74 | 3e | 01 | 00 | 03 | 28 | 29 | 56 | 01 | 00 |
| 00000100 | 04 | 43 | 6f | 64 | 65 | 0c | 00 | 06 | 00 | 07 | 01 | 00 | 03 | 62 | 63 | 74 |
| 00000120 | 01 | 00 | 10 | 6a | 61 | 76 | 61 | 2f | 6c | 61 | 6e | 67 | 2f | 4f | 62 | 6a |
| 00000140 | 65 | 63 | 74 | 00 | 20 | 00 | 02 | 00 | 03 | 00 | 00 | 00 | 01 | 00 | 00 | 00 |
| 00000160 | 04 | 00 | 05 | 00 | 00 | 00 | 01 | 00 | 00 | 00 | 06 | 00 | 07 | 00 | 01 | 00 |
| 00000200 | 08 | 00 | 00 | 00 | 11 | 00 | 01 | 00 | 01 | 00 | 00 | 00 | 05 | 2a | b7 | 00 |
| 00000220 | 01 | b1 | 00 | 00 | 00 | 00 | 00 | 00 | | | | | | | | |
| 00000230 | | | | | | | | | | | | | | | | |


```
magic = 0x CAFEBAFE  
minor_version = 0  
major_version = 52
```

```
ca fe ba be 00 00 00 34
```

```

constant_pool_count = 12
constant_pool =
{
1| tag = CONSTANT_Methodref, class_index = 3, name_and_type_index = 9
2| tag = CONSTANT_Class, name_index = 10
3| tag = CONSTANT_Class, name_index = 11
4| tag = CONSTANT_Utf8, length = 1, bytes = i
5| tag = CONSTANT_Utf8, length = 19, bytes = Ljava/lang/Integer;
6| tag = CONSTANT_Utf8, length = 6, bytes = <init>
7| tag = CONSTANT_Utf8, length = 3, bytes = ()V
8| tag = CONSTANT_Utf8, length = 4, bytes = Code
9| tag = CONSTANT_NameAndType, name_index = 6, descriptor_index = 7
10| tag = CONSTANT_Utf8, length = 3, bytes = bct
11| tag = CONSTANT_Utf8, length = 16, bytes = java/lang/Object
}

```

| | | | | | | | | | | | | | | | |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| | | | | | | | | 00 | 0c | 0a | 00 | 03 | 00 | 09 | 07 |
| 00 | 0a | 07 | 00 | 0b | 01 | 00 | 01 | 69 | 01 | 00 | 13 | 4c | 6a | 61 | 76 |
| 61 | 2f | 6c | 61 | 6e | 67 | 2f | 49 | 6e | 74 | 65 | 67 | 65 | 72 | 3b | 01 |
| 00 | 06 | 3c | 69 | 6e | 69 | 74 | 3e | 01 | 00 | 03 | 28 | 29 | 56 | 01 | 00 |
| 04 | 43 | 6f | 64 | 65 | 0c | 00 | 06 | 00 | 07 | 01 | 00 | 03 | 62 | 63 | 74 |
| 01 | 00 | 10 | 6a | 61 | 76 | 61 | 2f | 6c | 61 | 6e | 67 | 2f | 4f | 62 | 6a |
| 65 | 63 | 74 | | | | | | | | | | | | | |

```
access_flags = 32 // ACC_SUPER
this_class = #2 // bct
super_class = #3 // java/lang/Object
interfaces_count = 0
interfaces = {}
```

```
00 20 00 02 00 03 00 00
```

```
fields_count = 1
fields [0] =
{
  access_flags = 0
  name_index = #4 // i
  descriptor_index = #5 // Ljava/lang/Integer;
  attributes_count = 0
  attributes = {}
}
```

00 01 00 00 00 04 00 05 00 00

```
methods_count = 1
methods [0] =
{
access_flags = 0
name_index = #6 // <init>
descriptor_index = #7 // ()V
```

00 01 00 00 00 06 00 07

```

attributes_count = 1
attributes [0] =
{
attribute_name_index = #8 // Code
attribute_length = 17
max_stack = 1, max_locals = 1
code_length = 5
code =
{
    0  aload_0
    1  invokespecial #1 // java/lang/Object.<init> ()V
    4  return
}

```

```

00 01 00 08 00 00 00 11 00 01 00 01 00 00 00 05
2a b7 00 01 b1

```

```
exception_table_length = 0  
exception_table = {}  
attributes_count = 0  
attributes = {}  
}  
}
```

00 00 00 00

```
exception_table_length = 0  
exception_table = {}  
attributes_count = 0  
attributes = {}  
}  
}
```

00 00 00 00

```
attributes_count = 0  
attributes = {}
```

00 00

Dezimal (8 Bit)

```
od -td1 bct.class
```

| | | | | | | | | | | | | | | | | |
|---------|-----|-----|-----|-----|-----|-----|-----|----|-----|-----|-----|-----|-----|-----|-----|-----|
| 0000000 | -54 | -2 | -70 | -66 | 0 | 0 | 0 | 52 | 0 | 12 | 10 | 0 | 3 | 0 | 9 | 7 |
| 0000020 | 0 | 10 | 7 | 0 | 11 | 1 | 0 | 1 | 105 | 1 | 0 | 19 | 76 | 106 | 97 | 118 |
| 0000040 | 97 | 47 | 108 | 97 | 110 | 103 | 47 | 73 | 110 | 116 | 101 | 103 | 101 | 114 | 59 | 1 |
| 0000060 | 0 | 6 | 60 | 105 | 110 | 105 | 116 | 62 | 1 | 0 | 3 | 40 | 41 | 86 | 1 | 0 |
| 0000100 | 4 | 67 | 111 | 100 | 101 | 12 | 0 | 6 | 0 | 7 | 1 | 0 | 3 | 98 | 99 | 116 |
| 0000120 | 1 | 0 | 16 | 106 | 97 | 118 | 97 | 47 | 108 | 97 | 110 | 103 | 47 | 79 | 98 | 106 |
| 0000140 | 101 | 99 | 116 | 0 | 32 | 0 | 2 | 0 | 3 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 0000160 | 4 | 0 | 5 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 6 | 0 | 7 | 0 | 1 | 0 |
| 0000200 | 8 | 0 | 0 | 0 | 17 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 5 | 42 | -73 | 0 |
| 0000220 | 1 | -79 | 0 | 0 | 0 | 0 | 0 | 0 | | | | | | | | |
| 0000230 | | | | | | | | | | | | | | | | |

ASCII

```
od -ta bct.class
```

```
00000000          nul  ff  nl nul etx nul  ht bel
00000020  nul  nl bel nul  vt soh nul soh  i soh nul dc3  L  j  a  v
00000040    a  /  l  a  n  g  /  I  n  t  e  g  e  r  ; soh
00000060  nul ack  <  i  n  i  t  > soh nul etx  (  )  V soh nul
00000100  eot  C  o  d  e  ff nul ack nul bel soh nul etx  b  c  t
00000120  soh nul dle  j  a  v  a  /  l  a  n  g  /  0  b  j
00000140    e  c  t nul  sp nul stx nul etx nul nul nul soh nul nul nul
00000160  eot nul enq nul nul nul soh nul nul nul ack nul bel nul soh nul
00000200  bs nul nul nul dc1 nul soh nul soh nul nul nul enq
00000220  soh      nul nul nul nul nul nul
00000230
```

Haskell: Codegen

```
codegen :: Class -> ClassFile
-- bildet abstrakte Syntax in abstrakten Bytecode ab

compiler :: String -> ClassFile
compiler = codegen . typecheck . parse . alexScanTokens

encodeClassFile :: FilePath -> ClassFile -> IO()
-- erzeugt Bytecode--Datei
```

Haskell: Codegen

```
codegen :: Class -> ClassFile
-- bildet abstrakte Syntax in abstrakten Bytecode ab

compiler :: String -> ClassFile
compiler = codegen . typecheck . parse . alexScanTokens

encodeClassFile :: FilePath -> ClassFile -> IO()
-- erzeugt Bytecode--Datei

main = do
  s <- readFile "name.java"
  encodeClassFile "name.class" (compiler s)
```