Compilerbau

Martin Plümicke

WS 2018/19

Agenda

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

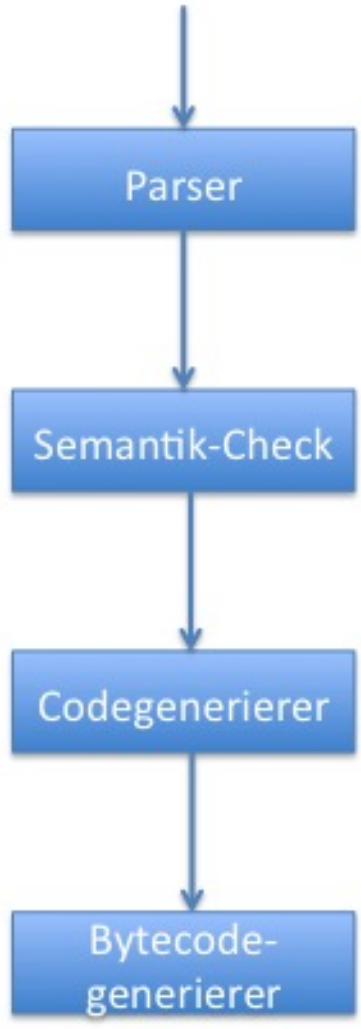

James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley.
The Java® Language Specification.
The Java series. Addison-Wesley, Java SE 8 edition, 2014.


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

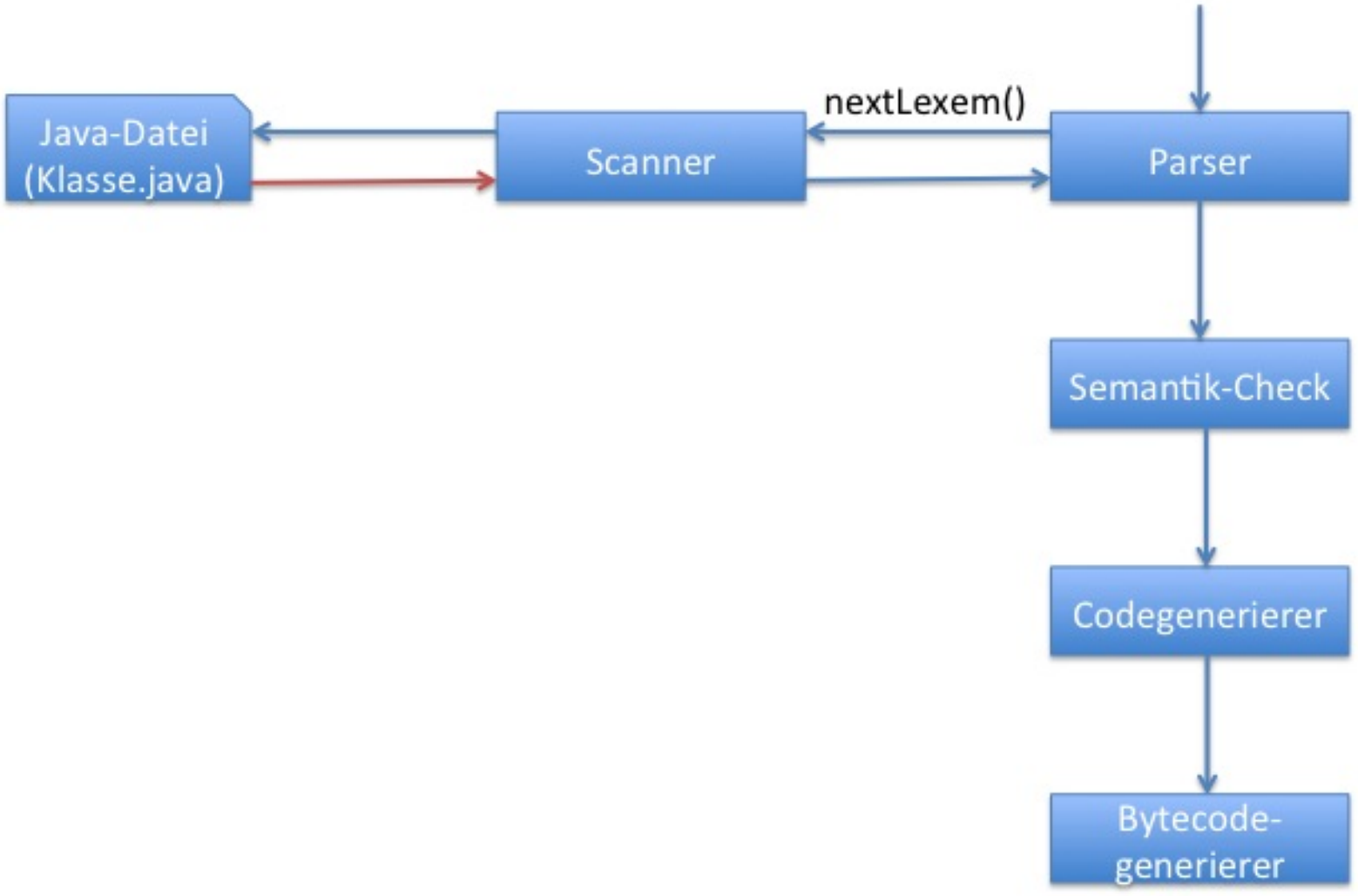

Bryan O'Sullivan, Donald Bruce Stewart, and John Goerzen.
Real World Haskell.
O'Reilly, 2009.


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

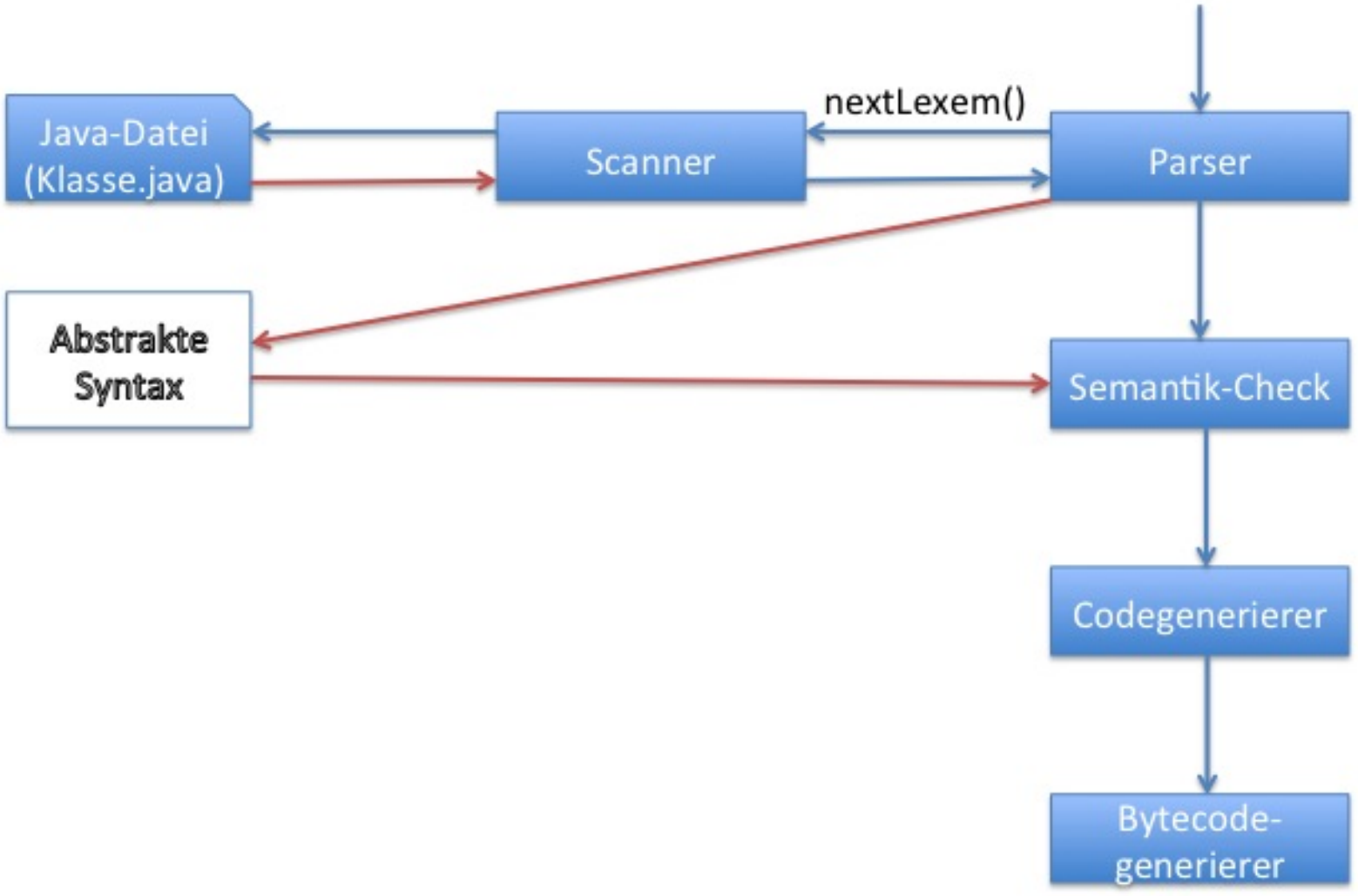
Compiler Überblick



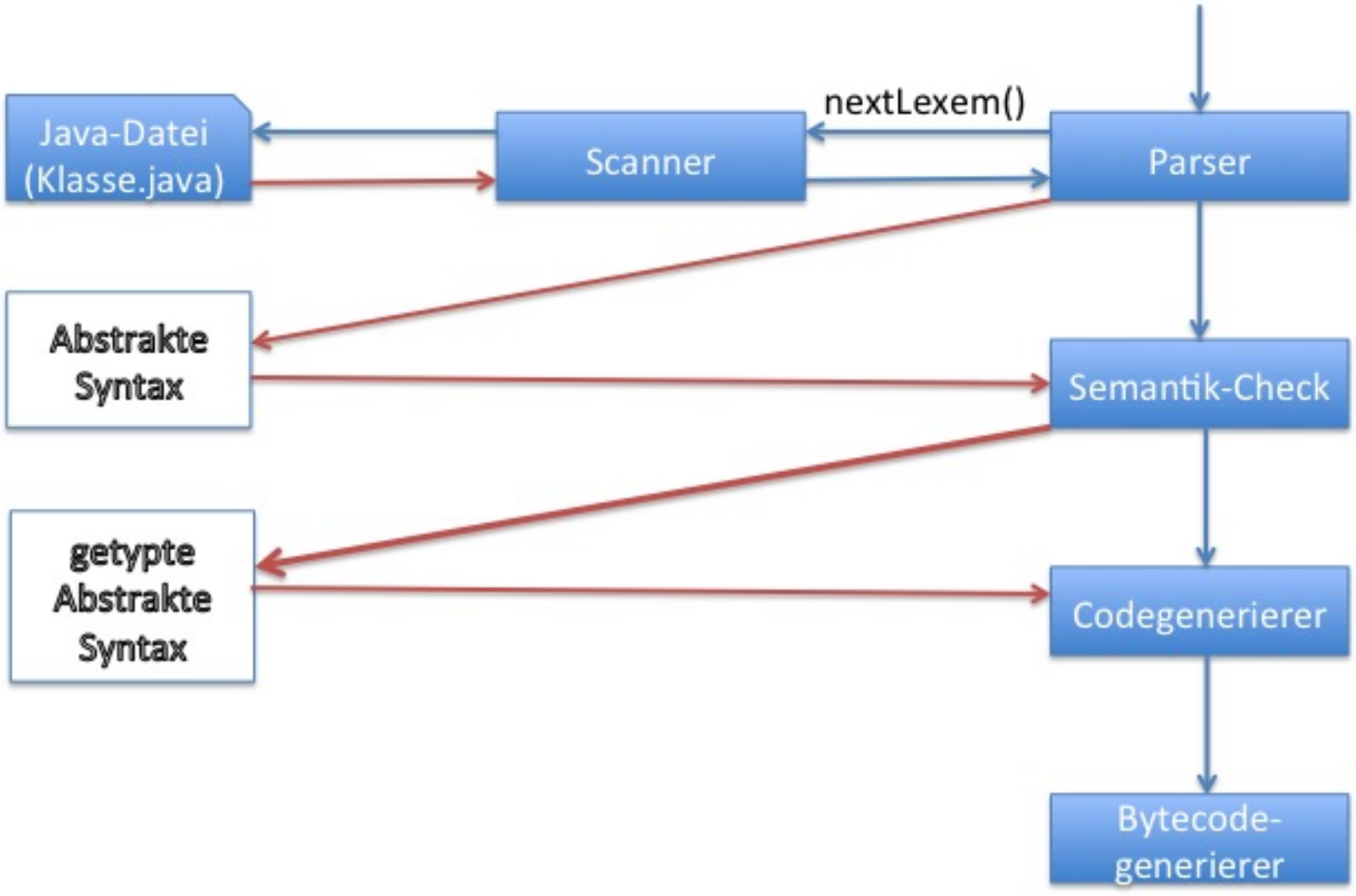
Compiler Überblick



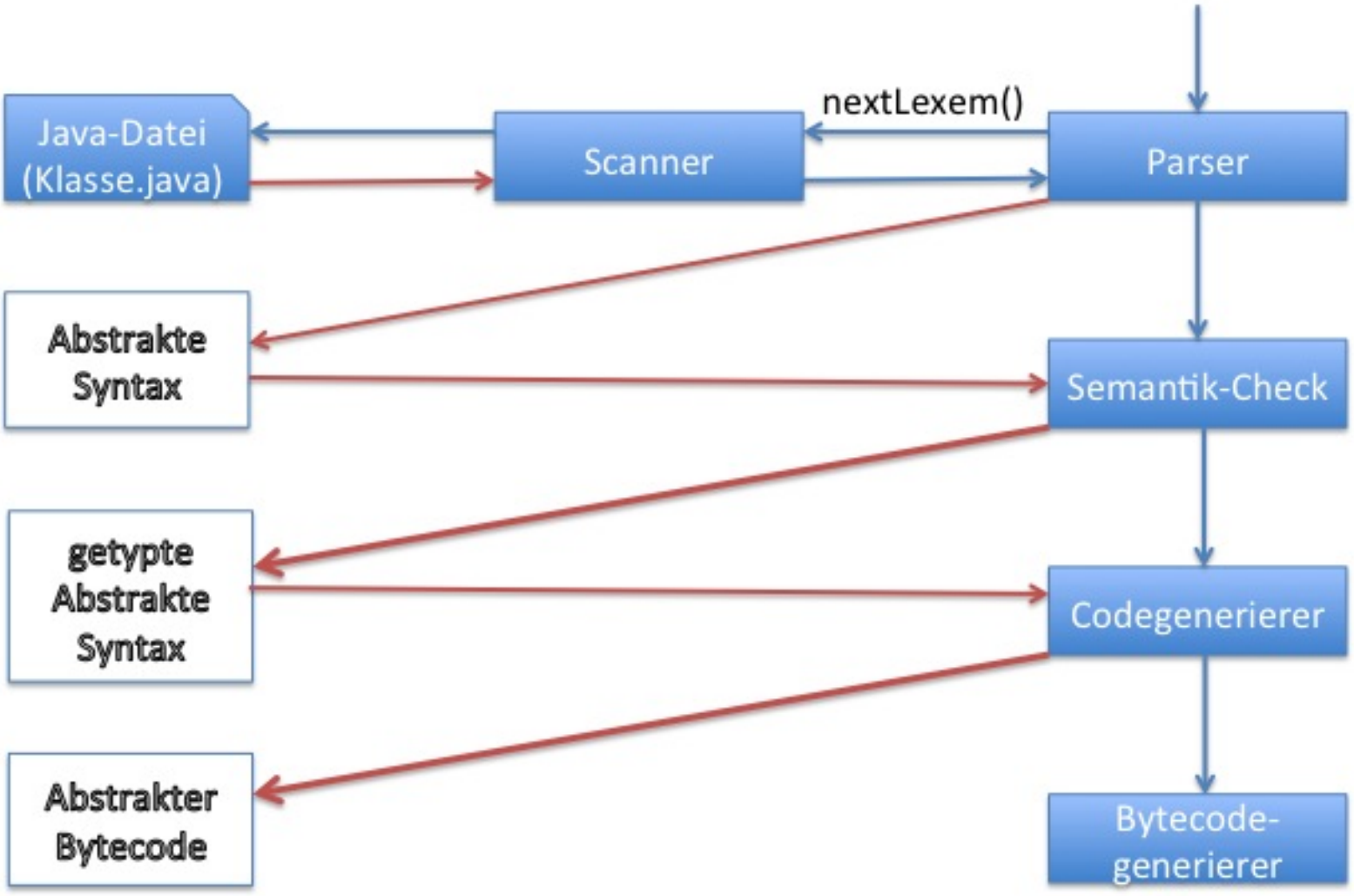
Compiler Überblick



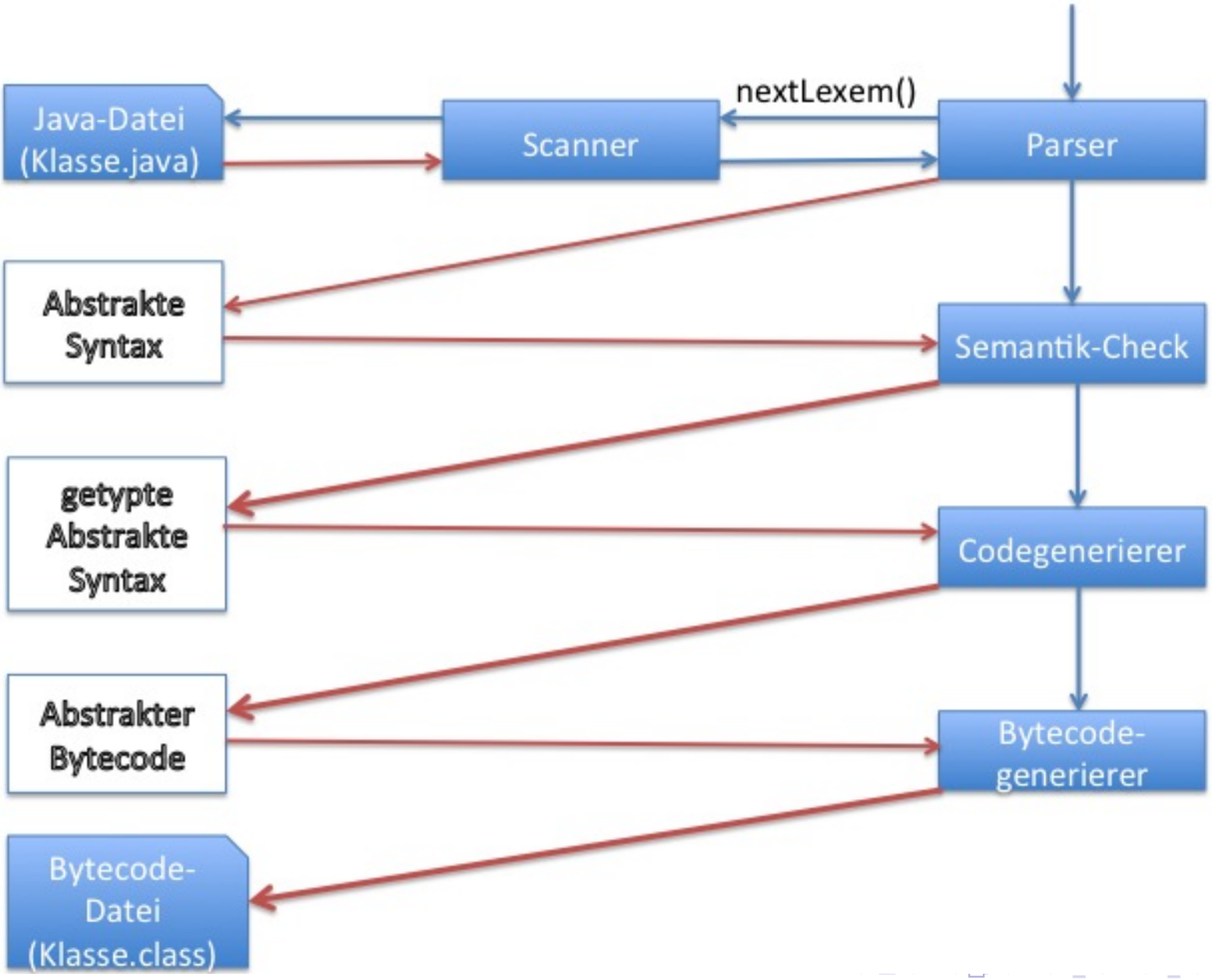
Compiler Überblick



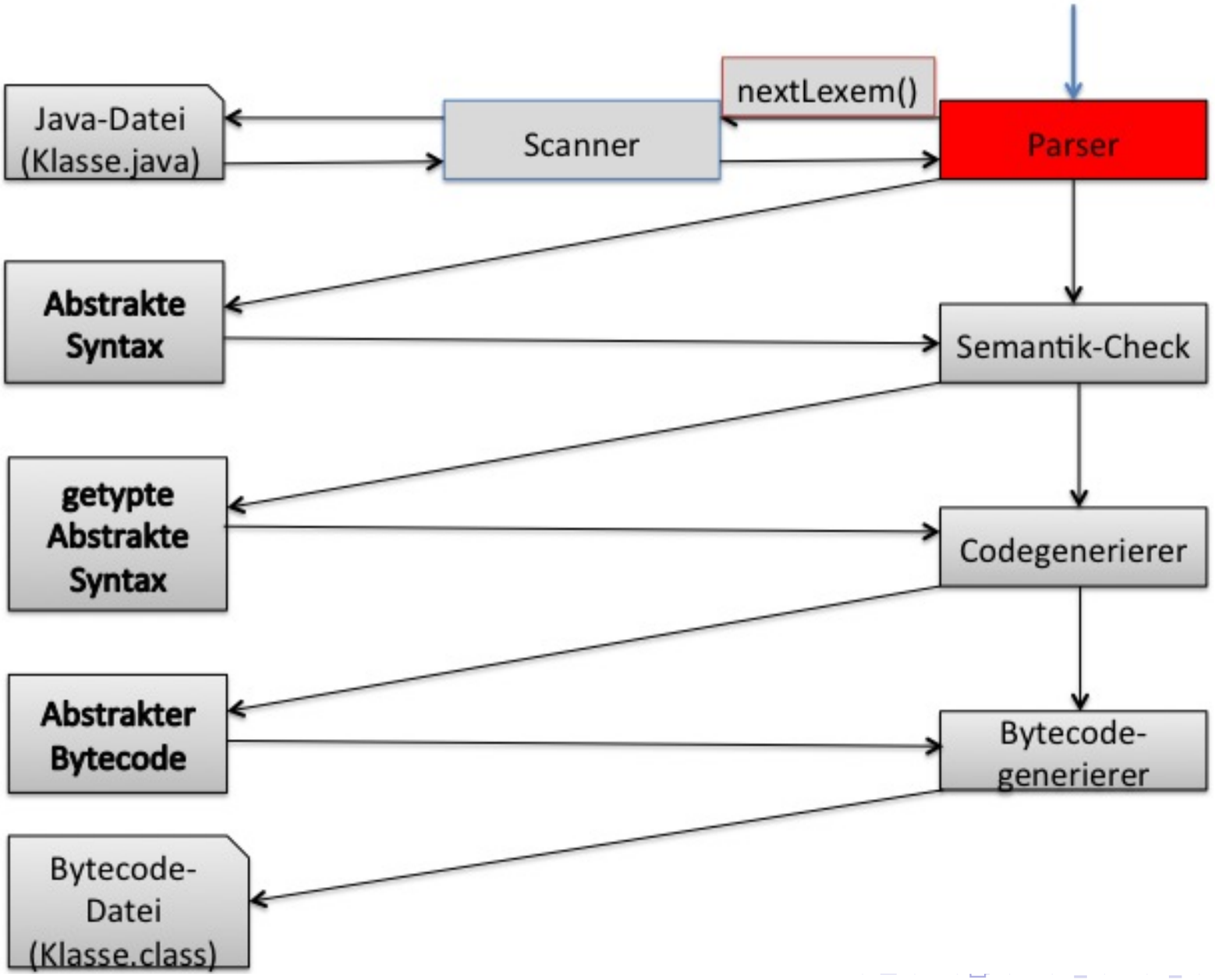
Compiler Überblick



Compiler Überblick



Parser


Java-Datei
(Klasse.java)
Scanner
nextLexem()
Parser
Abstrakte
Syntax
Semantik-Check
getypte
Abstrakte
Syntax
Codegenerierer
Abstrakter
Bytecode
Bytecode-
generierer
Bytecode-
Datei
(Klasse.class)

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 $S \stackrel{*}{\rightarrow} w$ gefunden werden.

Beispiel

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

$N = \{S, A\}$
$T = \{a, b, c\}$
$\Pi = \{S \to cAb$
$\quad A \to ab \mid a\}$

Beispiel

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

$N = \{S, A\}$
$T = \{a, b, c\}$
$\Pi = \{S \to cAb$
$\quad A \to ab \mid a\}$

$w = cab$

Beispiel

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

$N = \{S, A\}$
$T = \{a, b, c\}$
$\Pi = \{S \to cAb$
$\quad A \to ab \mid a\}$

$w = cab$

$\Rightarrow Ableitung : S \to cAb \to cab$

Beispiel

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

$N = \{S, A\}$
$T = \{a, b, c\}$
$\Pi = \{S \to cAb$
$\quad A \to ab \mid a\}$

$w = cab$

$\Rightarrow Ableitung : S \to cAb \to cab$

Ergebnis: $erg = True$

Ansatz für einen Parser

Programmiersprachen werden als kontextfreie Grammatiken mit kontextsensitiven Nebenbedingungen beschrieben

Ansatz für einen Parser

Programmiersprachen werden als kontextfreie Grammatiken mit kontextsensitiven Nebenbedingungen beschrieben

- Cocke-Younger-Kasami–Algorithmus
  Nachteil:
  - Voraussetzung: Grammatik Chomsky-Normalform
  - Aufwand $O(n^3)$

Ansatz für einen Parser

Programmiersprachen werden als kontextfreie Grammatiken mit kontextsensitiven Nebenbedingungen beschrieben

- Cocke-Younger-Kasami–Algorithmus
  Nachteil:
  - Voraussetzung: Grammatik Chomsky-Normalform
  - Aufwand $O(n^3)$
- Push–Down–Automaten
  Nachteil:
  - Polynomialer Aufwand

$\Rightarrow$ Betrachtung einer Teilmenge der Chomsky-2–Sprachen (LR–Sprachen)

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)

Linksableitungen (top–down)

$G = (N, T, \Pi, S)$ mit $N = \{S, A, B, C\}$ $T = \{a, b, c\}$ und
$\Pi = \{S \to AB, A \to a, B \to bC, C \to c\}$

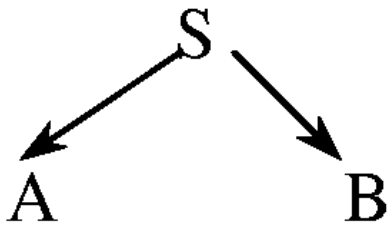
Eingabewort: $w = abc$

Linksableitungen (top–down)

$G = (N, T, \Pi, S)$ mit $N = \{S, A, B, C\}$ $T = \{a, b, c\}$ und
$\Pi = \{S \to AB, A \to a, B \to bC, C \to c\}$

Eingabewort: $w = abc$

$nexttoken() = a$

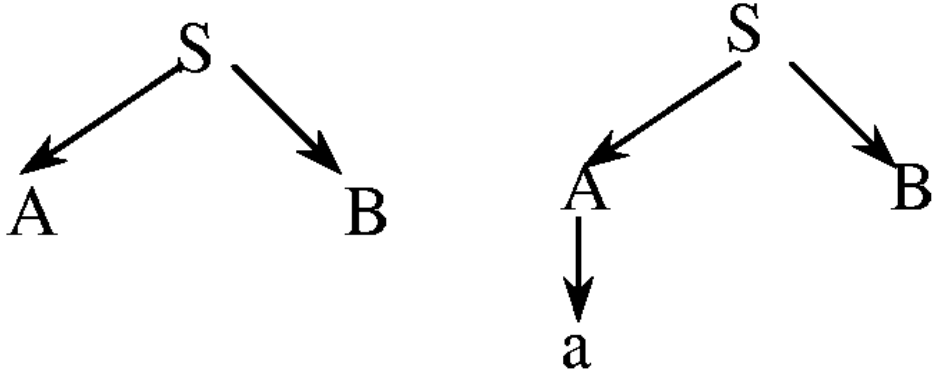


Linksableitungen (top–down)

$G = (N, T, \Pi, S)$ mit $N = \{S, A, B, C\}$ $T = \{a, b, c\}$ und
$\Pi = \{S \to AB, A \to a, B \to bC, C \to c\}$

Eingabewort: $w = abc$

$nexttoken() = a$

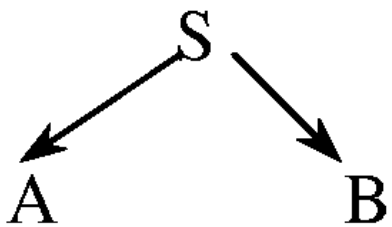


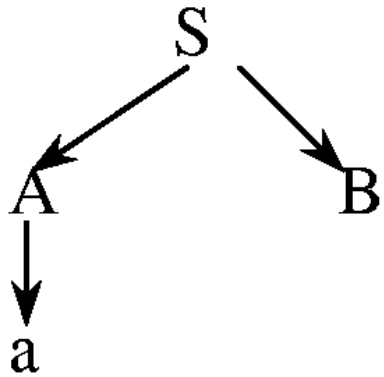
Linksableitungen (top–down)

$G = (N, T, \Pi, S)$ mit $N = \{S, A, B, C\}$ $T = \{a, b, c\}$ und
$\Pi = \{S \to AB, A \to a, B \to bC, C \to c\}$

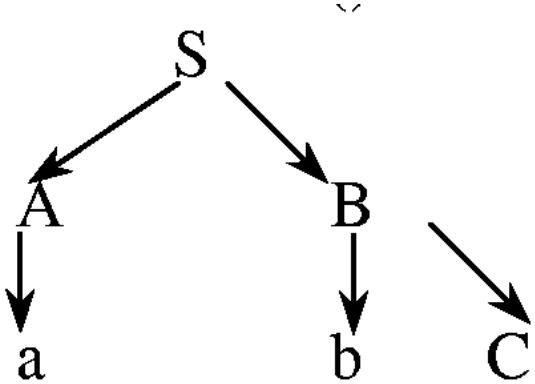
Eingabewort: $w = abc$

$nexttoken() = a$





$nexttoken() = b$

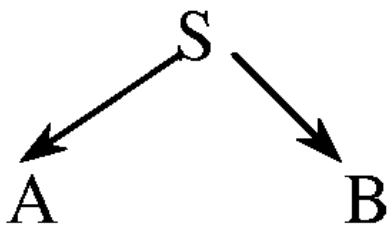


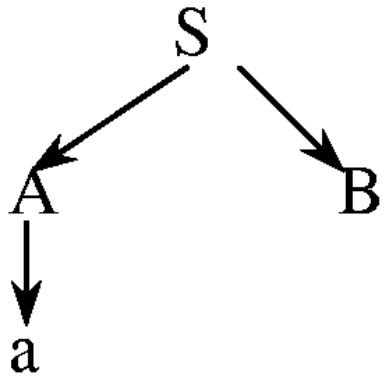
Linksableitungen (top–down)

$G = (N, T, \Pi, S)$ mit $N = \{S, A, B, C\}$ $T = \{a, b, c\}$ und
$\Pi = \{S \to AB, A \to a, B \to bC, C \to c\}$

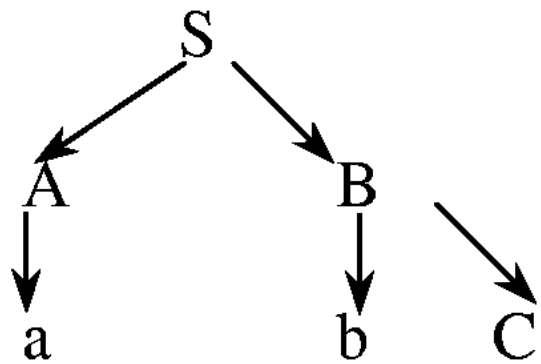
Eingabewort: $w = abc$

nexttoken() = *a*

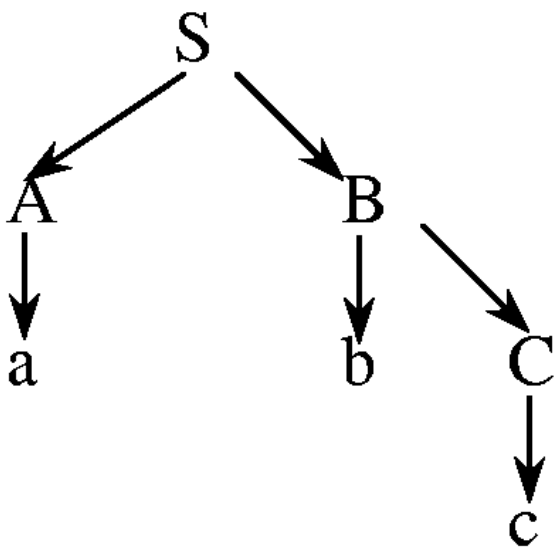




nexttoken() = *b*



nexttoken() = *c*



Recursive Decent–Syntaxanalyse

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

Parser–Kombinatoren I

Funktionale Implementierung eines recursive decent Parsers:

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

Parser–Kombinatoren I

Funktionale Implementierung eines recursive decent Parsers:

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

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

Parser–Kombinatoren I

Funktionale Implementierung eines recursive decent Parsers:

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

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

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

Parser–Kombinatoren I

Funktionale Implementierung eines recursive decent Parsers:

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

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

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

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

Parser–Kombinatoren I

Funktionale Implementierung eines recursive decent Parsers:

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

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

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

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

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

Transfomation der Ergebnisse

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

Beispiel Parser–Kombinatoren

Lexeme

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

isVar (VarToken x) = True
isVar _ = False

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

isInt (IntToken n) = True
isInt _ = False

date Maybe a = Just a
             | Nothing
```

Beispiel Parser–Kombinatoren II

$G = (N, T, \Pi, S)$ mit $N = \{Exp, Exp', TExp\}$

$T = \{let, in, digits, var, =, +\}$ und

$\Pi = \{Exp \rightarrow TExp\ Exp'$

$Exp' \rightarrow\ +\ TExp\ Exp' \mid \epsilon$

$TExp \rightarrow let\ var\ =\ Exp\ in\ Exp \mid var \mid digits\}$

Beispiel Parser–Kombinatoren II

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

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

expr' :: Parser Token ???
expr' = ((satisfy (isSym '+')) +.+ texp +.+ expr')
 ||| succeed 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
  $\Rightarrow$
  - keine Vorausschau zur Endscheidung bei Alternativen
  - Es muss kein Backtracking programmiert wreden
  - `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
  $\Rightarrow$
  - keine Vorausschau zur Endscheidung bei Alternativen
  - Es muss kein Backtracking programmiert wreden
  - `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 Endlosrekusionen führen
  $\Rightarrow$ Auflösung von Linksrekursionen

Rechtsableitungen (bottom–up)

$\Pi = \{S \to AB, A \to a, B \to bC, C \to c\}$

Eingabewort: $w = abc$

Rechtsableitungen (bottom–up)

$\Pi = \{S \to AB, A \to a, B \to bC, C \to c\}$
Eingabewort: $w = abc$

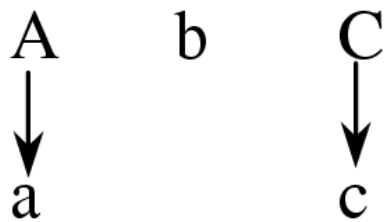
nexttoken$() = a$
wende an: $A \to a$

Rechtsableitungen (bottom–up)

$\Pi = \{S \to AB, A \to a, B \to bC, C \to c\}$
Eingabewort: $w = abc$

nexttoken$() = a$
wende an: $A \to a$

nexttoken$() = b$
nexttoken$() = c$
wende an: $C \to c$

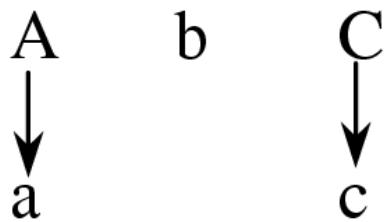


Rechtsableitungen (bottom–up)

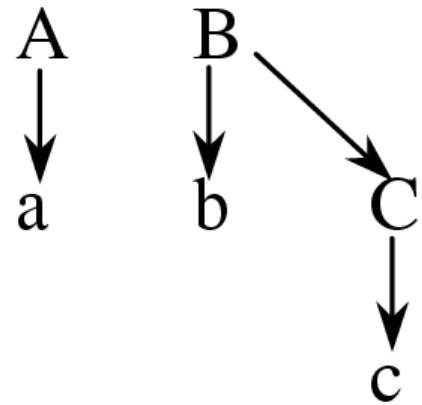
$\Pi = \{S \to AB, A \to a, B \to bC, C \to c\}$
Eingabewort: $w = abc$

nexttoken$() = a$
wende an: $A \to a$

nexttoken$() = b$
nexttoken$() = c$
wende an: $C \to c$



wende an: $B \to bC$

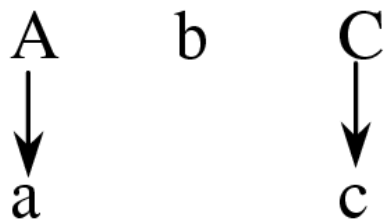


Rechtsableitungen (bottom–up)

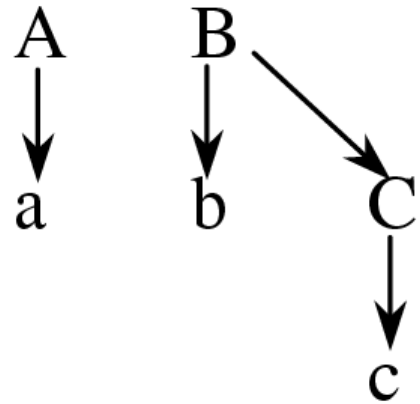
$\Pi = \{S \to AB, A \to a, B \to bC, C \to c\}$
Eingabewort: $w = abc$

nexttoken$() = a$
wende an: $A \to a$

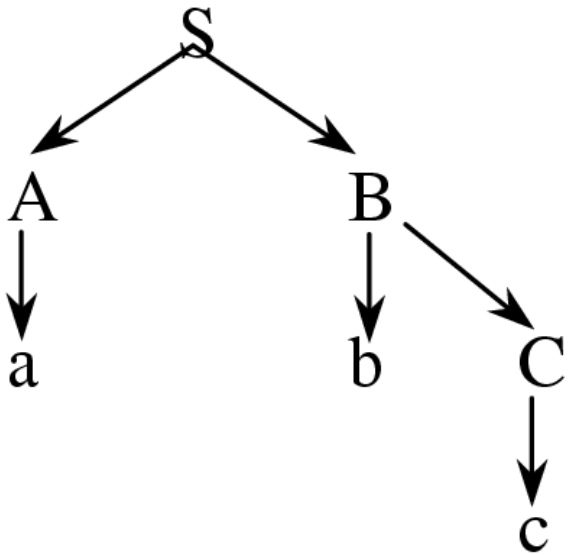
nexttoken$() = b$
nexttoken$() = c$
wende an: $C \to c$



wende an: $B \to bC$



wende an: $S \to AB$

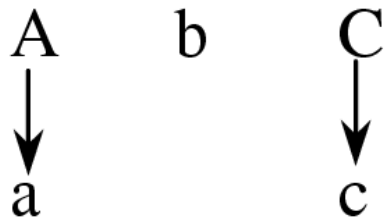


Rechtsableitungen (bottom–up)

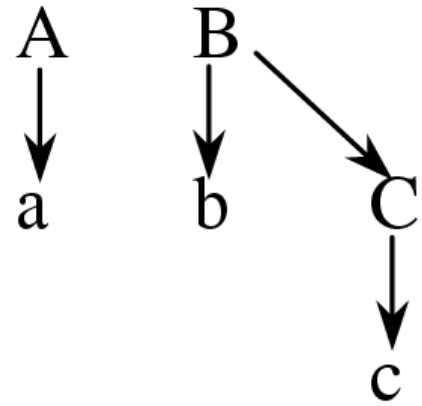
$\Pi = \{S \to AB, A \to a, B \to bC, C \to c\}$
Eingabewort: $w = abc$

nexttoken$() = a$
wende an: $A \to a$

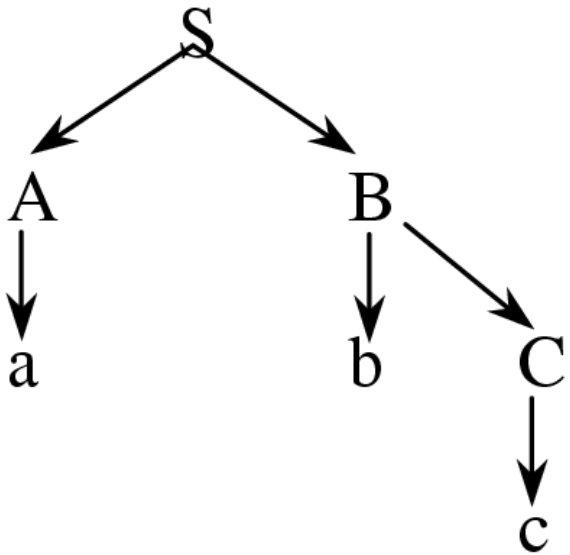
nexttoken$() = b$
nexttoken$() = c$
wende an: $C \to c$



wende an: $B \to bC$



wende an: $S \to AB$



Betrachtet man die Konstruktion rückwärts:

$$\underline{S} \to A\underline{B} \to Ab\underline{C} \to \underline{A}bc \to abc,$$

Weiteres Beispiel:

$G = (N, T, \Pi, S)$ mit
$N = \{S, A, B\}$,
$T = \{a, b, c, d, e\}$ und
$\Pi = \{S \to aABe, A \to Abc \mid b \mid d, B \to d\}$

Eingabestring: $w = abbcde$

Weiteres Beispiel:

$G = (N, T, \Pi, S)$ mit
$N = \{S, A, B\}$,
$T = \{a, b, c, d, e\}$ und
$\Pi = \{S \to aABe, A \to Abc \mid b \mid d, B \to d\}$

Eingabestring: $w = abbcde$

Bottom–Up Syntaxanalyse:

$a.bbcde$

Weiteres Beispiel:

$G = (N, T, \Pi, S)$ mit
$N = \{S, A, B\}$,
$T = \{a, b, c, d, e\}$ und
$\Pi = \{S \to aABe, A \to Abc \mid b \mid d, B \to d\}$

Eingabestring: $w = abbcde$

Bottom–Up Syntaxanalyse:

$a.bbcde \longleftarrow ab.bcde$

Weiteres Beispiel:

$G = (N, T, \Pi, S)$ mit
$N = \{S, A, B\}$,
$T = \{a, b, c, d, e\}$ und
$\Pi = \{S \to aABe, A \to Abc \mid b \mid d, B \to d\}$

Eingabestring: $w = abbcde$

Bottom–Up Syntaxanalyse:

$a.bbcde \longleftarrow ab.bcde \longleftarrow aA.bcde$

Weiteres Beispiel:

$G = (N, T, \Pi, S)$ mit
$N = \{S, A, B\}$,
$T = \{a, b, c, d, e\}$ und
$\Pi = \{S \to aABe, A \to Abc \mid b \mid d, B \to d\}$

Eingabestring: $w = abbcde$

Bottom–Up Syntaxanalyse:

$a.bbcde \longleftarrow ab.bcde \longleftarrow aA.bcde$

Shift–Reduce–Konflikt

$abb.cde$

Weiteres Beispiel:

$G = (N, T, \Pi, S)$ mit
$N = \{S, A, B\}$,
$T = \{a, b, c, d, e\}$ und
$\Pi = \{S \to aABe, A \to Abc \mid b \mid d, B \to d\}$

Eingabestring: $w = abbcde$

Bottom–Up Syntaxanalyse:

$a.bbcde \longleftarrow ab.bcde \longleftarrow aA.bcde \longleftarrow aAb.cde$

Shift–Reduce–Konflikt

$abb.cde$

Weiteres Beispiel:

$G = (N, T, \Pi, S)$ mit
$N = \{S, A, B\}$,
$T = \{a, b, c, d, e\}$ und
$\Pi = \{S \to aABe, A \to Abc \mid b \mid d, B \to d\}$

Eingabestring: $w = abbcde$

Bottom–Up Syntaxanalyse:

$a.bbcde \longleftarrow ab.bcde \longleftarrow aA.bcde \longleftarrow aAb.cde \longleftarrow aAbc.de$

Shift–Reduce–Konflikt

$abb.cde$

Weiteres Beispiel:

$G = (N, T, \Pi, S)$ mit
$N = \{S, A, B\}$,
$T = \{a, b, c, d, e\}$ und
$\Pi = \{S \to aABe, A \to Abc \mid b \mid d, B \to d\}$

Eingabestring: $w = abbcde$

Bottom–Up Syntaxanalyse:

$$a.bbcde \longleftarrow ab.bcde \longleftarrow aA.bcde \longleftarrow aAb.cde \longleftarrow aAbc.de$$

Shift–Reduce–Konflikt: $abb.cde$

Shift–Reduce–Konflikt: $aAA.cde$

Weiteres Beispiel:

$G = (N, T, \Pi, S)$ mit
$N = \{S, A, B\}$,
$T = \{a, b, c, d, e\}$ und
$\Pi = \{S \to aABe, A \to Abc \mid b \mid d, B \to d\}$

Eingabestring: $w = abbcde$

Bottom–Up Syntaxanalyse:

$a.bbcde \longleftarrow ab.bcde \longleftarrow aA.bcde \longleftarrow aAb.cde \longleftarrow aAbc.de \longleftarrow$

Shift–Reduce–Konflikt: $abb.cde$

Shift–Reduce–Konflikt: $aAA.cde$

$aA.de \longleftarrow aAd.e$

Weiteres Beispiel:

$G = (N, T, \Pi, S)$ mit
$N = \{S, A, B\}$,
$T = \{a, b, c, d, e\}$ und
$\Pi = \{S \to aABe, A \to Abc \mid b \mid d, B \to d\}$

Eingabestring: $w = abbcde$

Bottom–Up Syntaxanalyse:

$a.bbcde \longleftarrow ab.bcde \longleftarrow aA.bcde \longleftarrow aAb.cde \longleftarrow aAbc.de \longleftarrow$

Shift–Reduce–Konflikt: $abb.cde$

Shift–Reduce–Konflikt: $aAA.cde$

$aA.de \longleftarrow aAd.e \longleftarrow aAB.e$

Weiteres Beispiel:

$G = (N, T, \Pi, S)$ mit
$N = \{S, A, B\}$,
$T = \{a, b, c, d, e\}$ und
$\Pi = \{S \to aABe, A \to Abc \mid b \mid d, B \to d\}$

Eingabestring: $w = abbcde$

Bottom–Up Syntaxanalyse:

$a.bbcde \longleftarrow ab.bcde \longleftarrow aA.bcde \longleftarrow aAb.cde \longleftarrow aAbc.de \longleftarrow$

Shift–Reduce–Konflikt

$abb.cde$

Shift–Reduce–Konflikt

$aAA.cde$

$aA.de \longleftarrow aAd.e \longleftarrow aAB.e$

Reduce–Reduce–Konflikt

$aAA.e$

Weiteres Beispiel:

$G = (N, T, \Pi, S)$ mit
$N = \{S, A, B\}$,
$T = \{a, b, c, d, e\}$ und
$\Pi = \{S \to aABe, A \to Abc \mid b \mid d, B \to d\}$

Eingabestring: $w = abbcde$

Bottom–Up Syntaxanalyse:

$a.bbcde \longleftarrow ab.bcde \longleftarrow aA.bcde \longleftarrow aAb.cde \longleftarrow aAbc.de \longleftarrow$

Shift–Reduce–Konflikt: $abb.cde$

Shift–Reduce–Konflikt: $aAA.cde$

$aA.de \longleftarrow aAd.e \longleftarrow aAB.e \longleftarrow aABe \longleftarrow S$

Reduce–Reduce–Konflikt: $aAA.e$

Konfliktlösungsansätze

LR(0): Es muss ohne Vorausschau möglich sein zu entscheiden, ob geshiftet oder reduziert wird.

SLR(1): An Hand der Bildung der Menge aller möglichen folgenden Terminalsymbole auf ein Nichtterminal, wird entschieden ob reduziert wird.

LR(1): Für jeden möglichen Ableitungsschritt in einer Produktion wird die Menge der darauffolgenden Terminalsymbolde zur Unterscheidung betrachtet.

LALR(1): Es werden alle Mengen von LR(1)–Elementen zusammengefasst, die den gleichen Ableitungsschritt vollziehen.

$$LR(0) \subset SLR(1) \subset LALR(1) \subset LR(1) (= \mathcal{L}(\mathcal{DPDA}))$$

Parsergenerator Happy

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

Das Happy-File

Haskell–Source–Code:

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

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

Deklarationen

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

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

Beispiel

Data–Deklaration des %tokentype's

Leicht modifizierte Datenstruktur in Scanner.x:

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

Tokens

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

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

Modifizierte Grammatik

Die Grammatik aus dem Kombinator-Parsen Beispiel

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

wird vereinfacht zu:

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

Beim Bottom–Up Parsen dürfen Grammatiken linksrekursiv sein.

Grammatik im Happy–File

```
%% -- aus yacc-Tradition

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

Grammatik und Übersetzung

Erinnerung: Abstrakte Syntax *Mini funktionale Expressions*

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

Grammatik und Übersetzung

Erinnerung: Abstrakte Syntax *Mini funktionale Expressions*

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

Happy–File mit

```
%% -- aus yacc-Tradition

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

Grammatik und Übersetzung

Erinnerung: Abstrakte Syntax *Mini funktionale Expressions*

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

Happy–File mit

```
%% -- aus yacc-Tradition

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

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

Beispiel

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

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

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

hat.

Beispiel

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

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

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

hat.

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

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

Haskell-Code

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

```
{

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

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

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


}
```