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

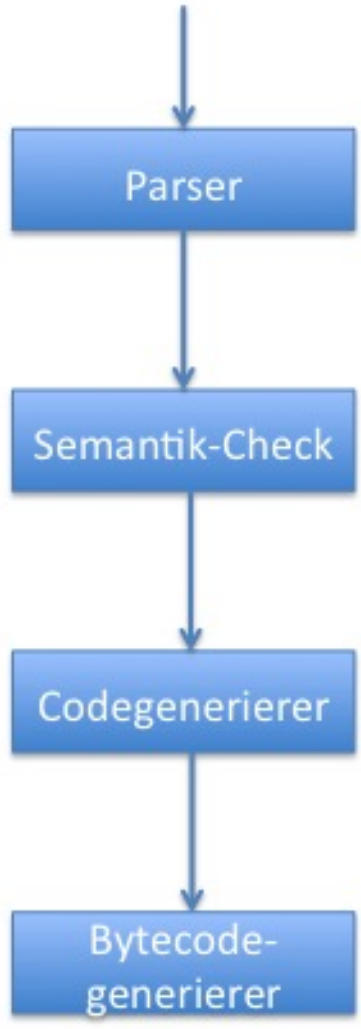

James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley.
The Java® Language Specification.
The Java series. Addison-Wesley, Java SE 8 edition, 2014.


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

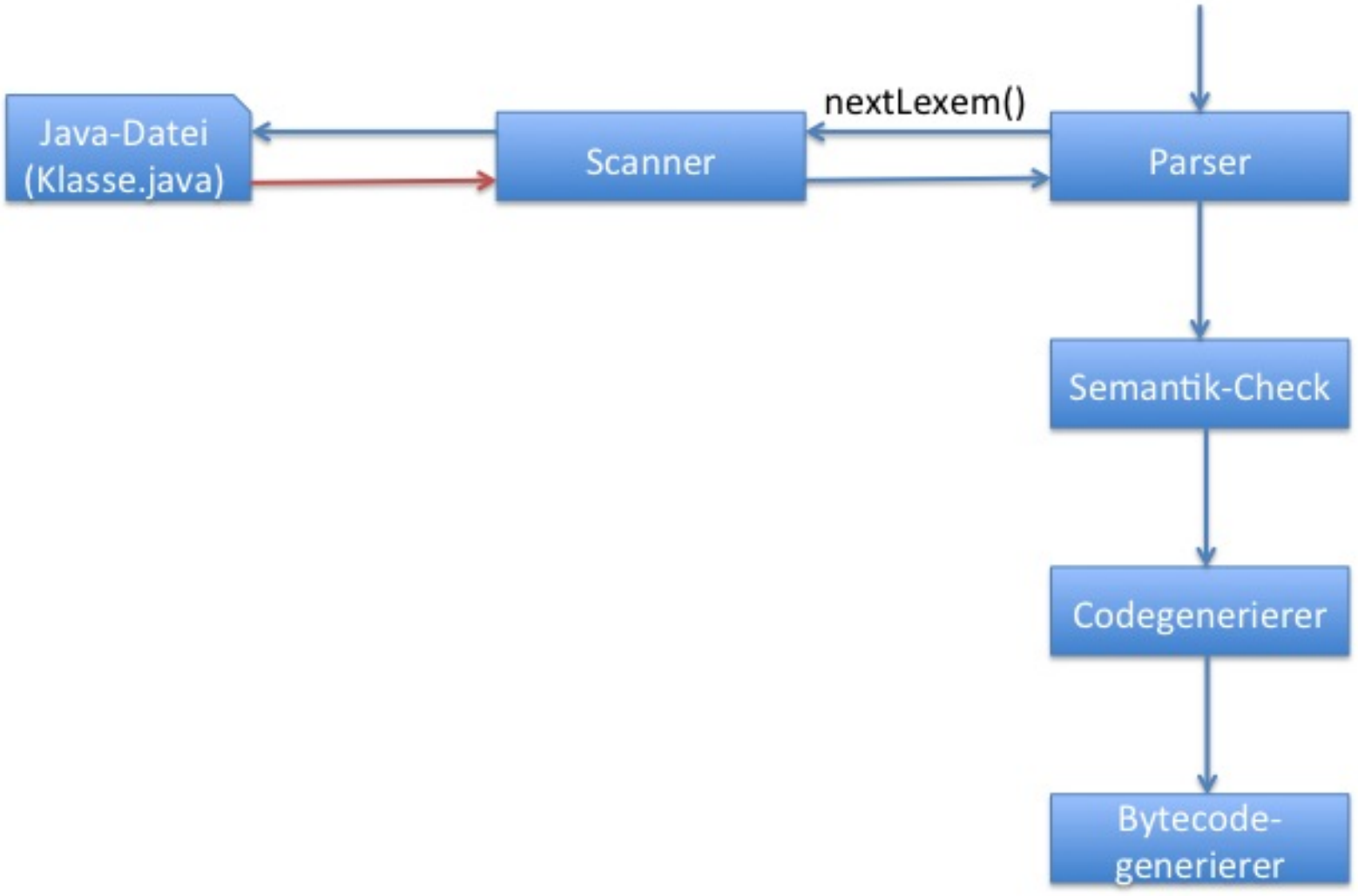

Bryan O'Sullivan, Donald Bruce Stewart, and John Goerzen.
Real World Haskell.
O'Reilly, 2009.


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

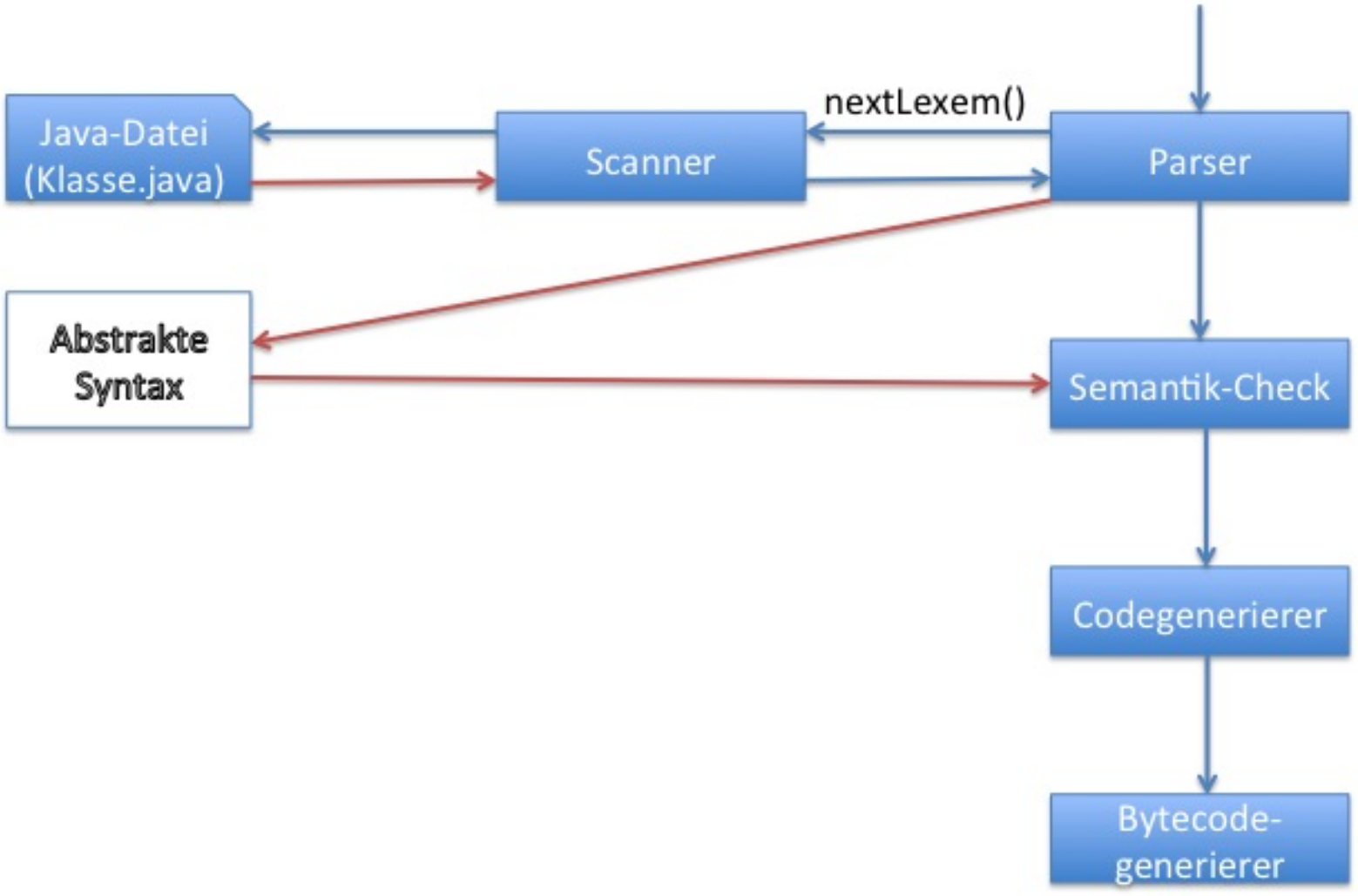
Compiler Überblick



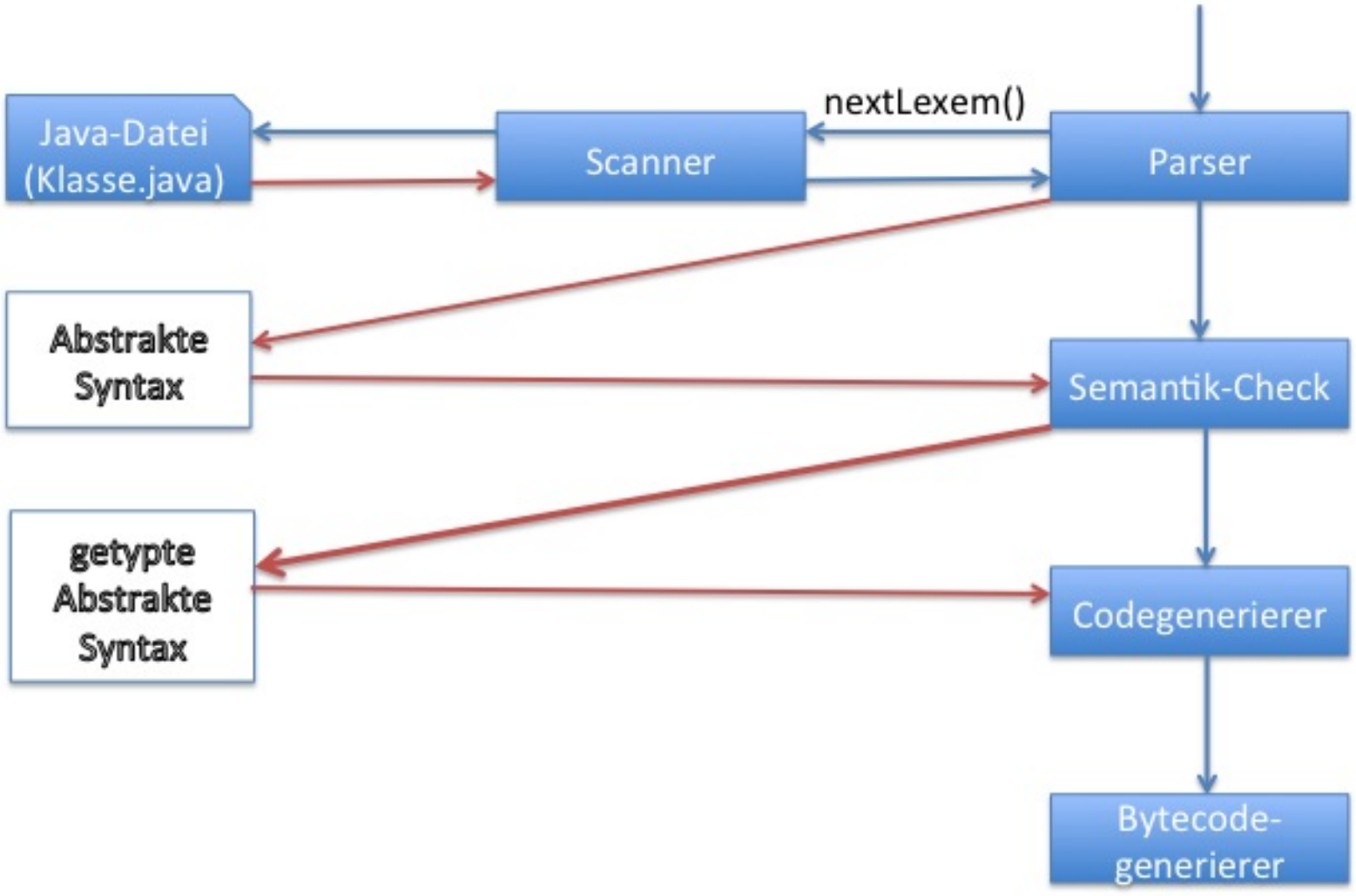
Compiler Überblick



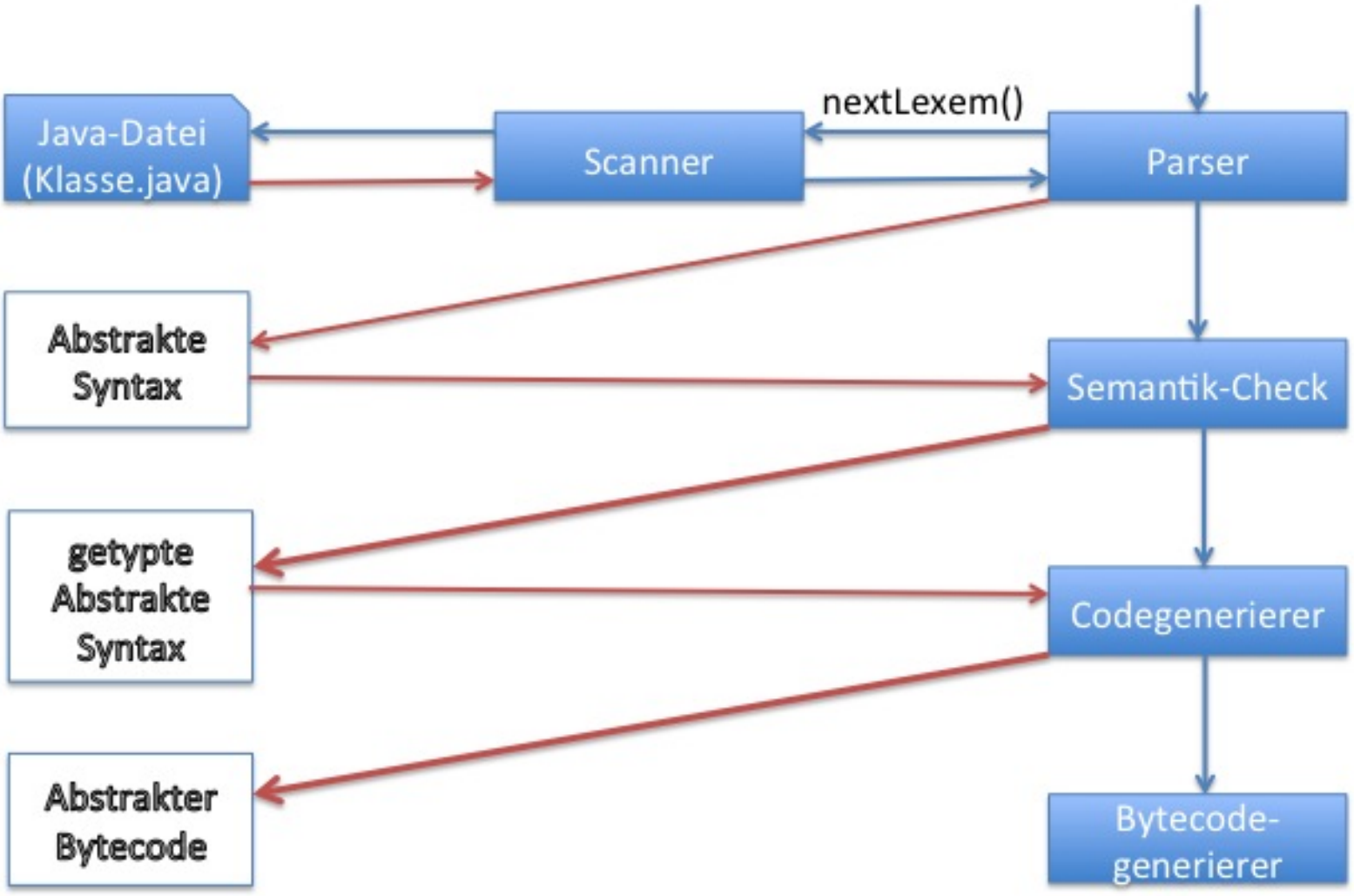
Compiler Überblick



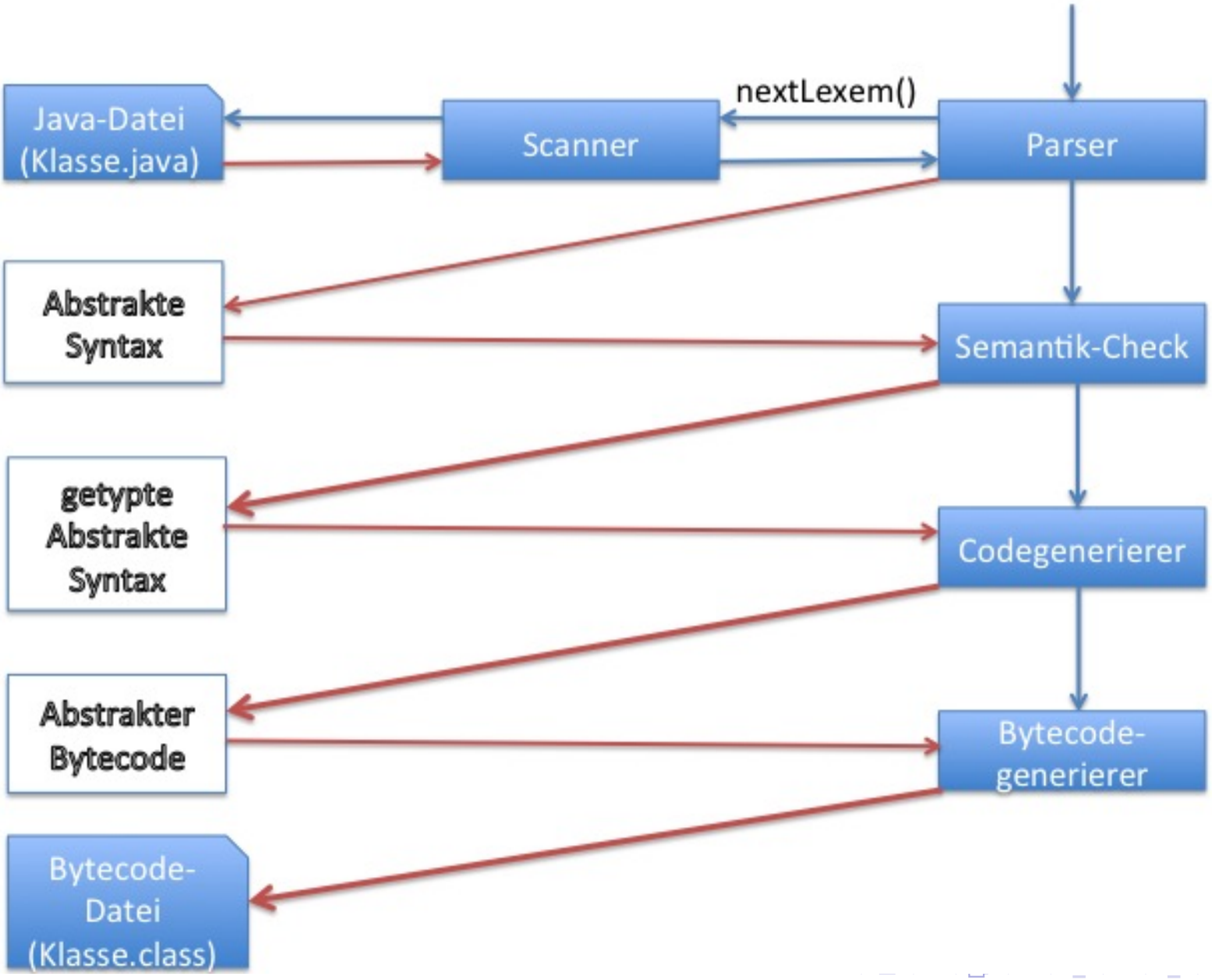
Compiler Überblick



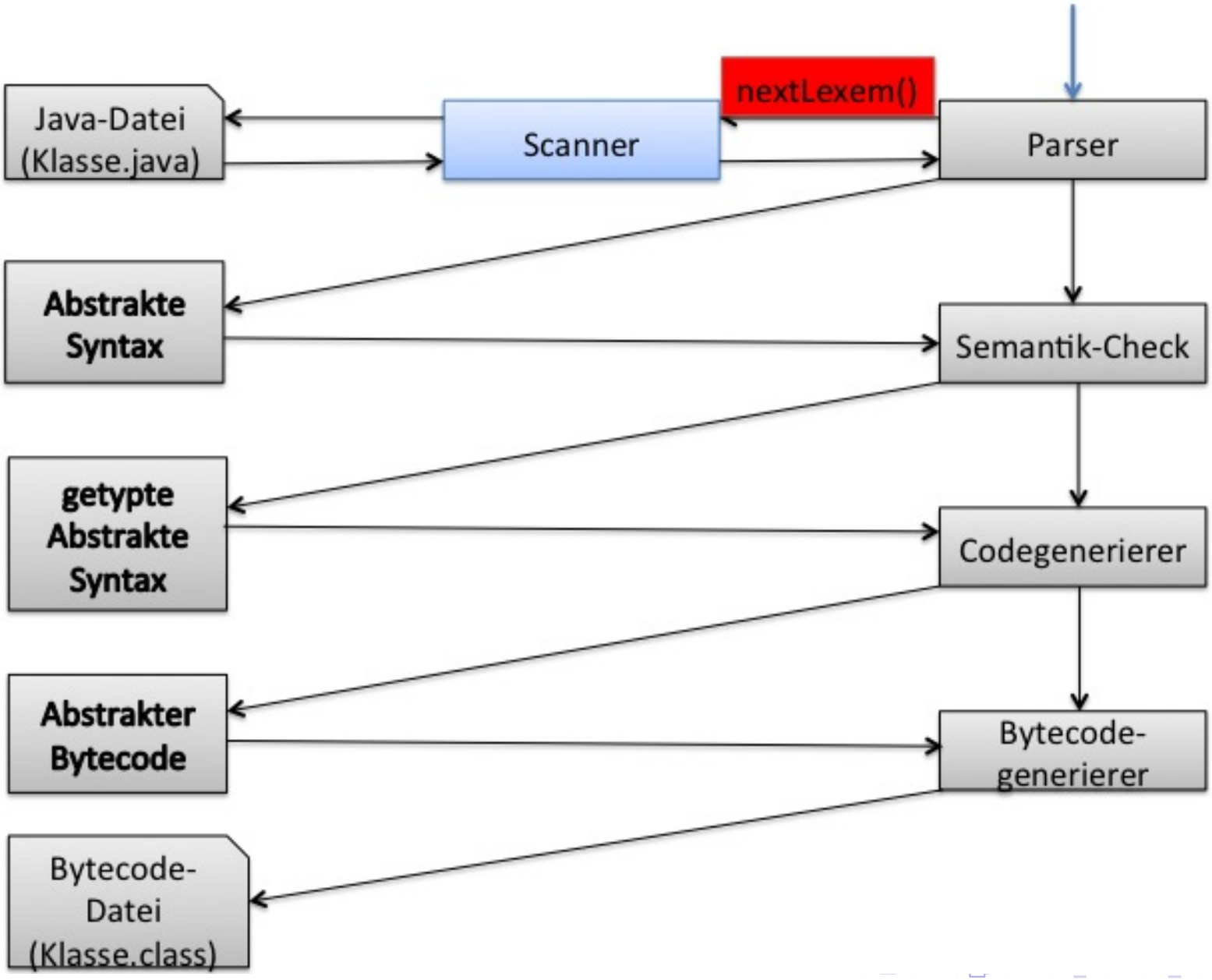
Compiler Überblick



Compiler Überblick



Scanner


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.

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.

Reguläre Ausdrücke

Sei Σ ein Alphabet, dann ist die Menge der regulären Ausdrücke über Σ : $R(\Sigma)$ definiert als kleinste Menge mit folgenden Eigenschaften.

a) $\varepsilon \in R(\Sigma)$

b) $\Sigma \subseteq R(\Sigma)$

c) $a \in R(\Sigma) \wedge b \in R(\Sigma) \Rightarrow ab \in R(\Sigma)$
$a|b \in R(\Sigma)$
$a^* \in R(\Sigma)$
$(a) \in R(\Sigma)$

Reguläre Ausdrücke

Sei Σ ein Alphabet, dann ist die Menge der regulären Ausdrücke über Σ : $R(\Sigma)$ definiert als kleinste Menge mit folgenden Eigenschaften.

a) $\varepsilon \in R(\Sigma)$

b) $\Sigma \subseteq R(\Sigma)$

c) $a \in R(\Sigma) \wedge b \in R(\Sigma) \Rightarrow ab \in R(\Sigma)$
$a|b \in R(\Sigma)$
$a^* \in R(\Sigma)$
$(a) \in R(\Sigma)$

Beispiel: $\Sigma = \{a, b, c\}$

$$R(\Sigma) = \{\varepsilon, a, b, c, ab, ac, aa, \ldots, abac, aaa, \ldots, a|b, a|c, b|c, aa|bc, \ldots, aa|bc|aa, a^*, b^*, aaaa^*, (a), (b), (a|c), (a|c)^*, (a|c)^*a, \ldots\}$$

Reguläre Sprache

Sei $\alpha \in R(\Sigma)$ ein regulärer Ausdruck, so ist die reguläre Sprache $\mathcal{L}(\alpha)$ definiert durch die kleinste Menge mit folgenden Eigenschaften:

1. $\mathcal{L}(\varepsilon) = \varepsilon(=\text{""})$
2. $\alpha \in \Sigma \Rightarrow \mathcal{L}(\alpha) = \{\alpha\}$
3. - $\alpha = \beta\gamma \Rightarrow \mathcal{L}(\alpha) = \{ww' | w \in \mathcal{L}(\beta), w' \in \mathcal{L}(\gamma)\}$
   - $\alpha = \beta|\gamma \Rightarrow \mathcal{L}(\alpha) = \mathcal{L}(\beta) \cup \mathcal{L}(\gamma)$
   - $\alpha = \beta^* \Rightarrow \mathcal{L}(\alpha) = \{\varepsilon\} \cup \{ww' | w \in \mathcal{L}(\beta), w' \in \mathcal{L}(\beta^*)\}$
4. $\alpha = (\beta) \Rightarrow \mathcal{L}(\alpha) = \mathcal{L}(\beta)$

Deterministische endliche Automat (DEA)

Unter einem DEA versteht man

$$A = (Q, \Sigma, \delta, s, F)$$

mit

- $Q \triangleq$ Zustandsmenge (endlich)
- $\Sigma \triangleq$ Alphabet
- $\delta \triangleq$ Übergangsfunktion: $Q \times \Sigma \to Q$
- $s \in Q \triangleq$ Anfangszustand
- $F \subseteq Q \triangleq$ Menge der Finalzustände

Deterministische endliche Automat (DEA)

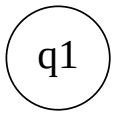
Unter einem DEA versteht man

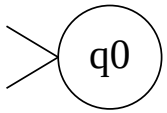
$$A = (Q, \Sigma, \delta, s, F)$$

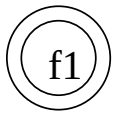
mit

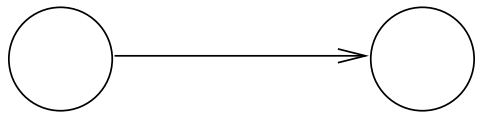
- $Q \triangleq$ Zustandsmenge (endlich)
- $\Sigma \triangleq$ Alphabet
- $\delta \triangleq$ Übergangsfunktion: $Q \times \Sigma \to Q$
- $s \in Q \triangleq$ Anfangszustand
- $F \subseteq Q \triangleq$ Menge der Finalzustände

DEAs kann man grafisch darstellen:

- **Zustände:** 

- **Startzustand:** 

- **Finalzustand:** 

- **Übergangsfunktion:** 

Reg. Ausdruck nach DEA

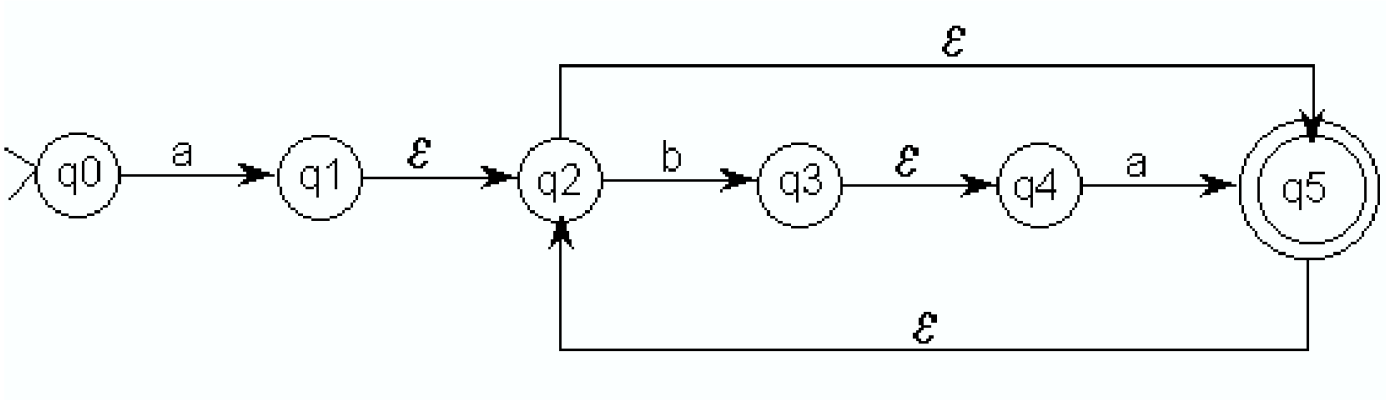
Man kann reguläre Ausdruck in DEA's übersetzen, die die Sprachen der regulären Ausdruck erkennen:

$$\text{reg.Ausdruck} \overset{reg2auto}{\Rightarrow} \text{NEA} \overset{NEA2DEA}{\Rightarrow} \text{DEA}$$

Beispiel: $a(ba)*$

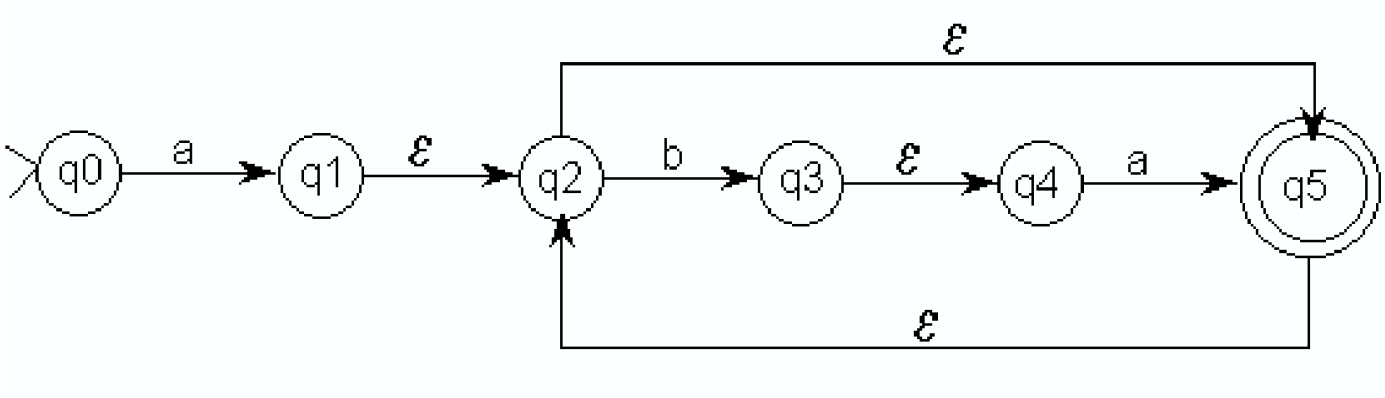
Beispiel: $a(ba)*$

NEA:

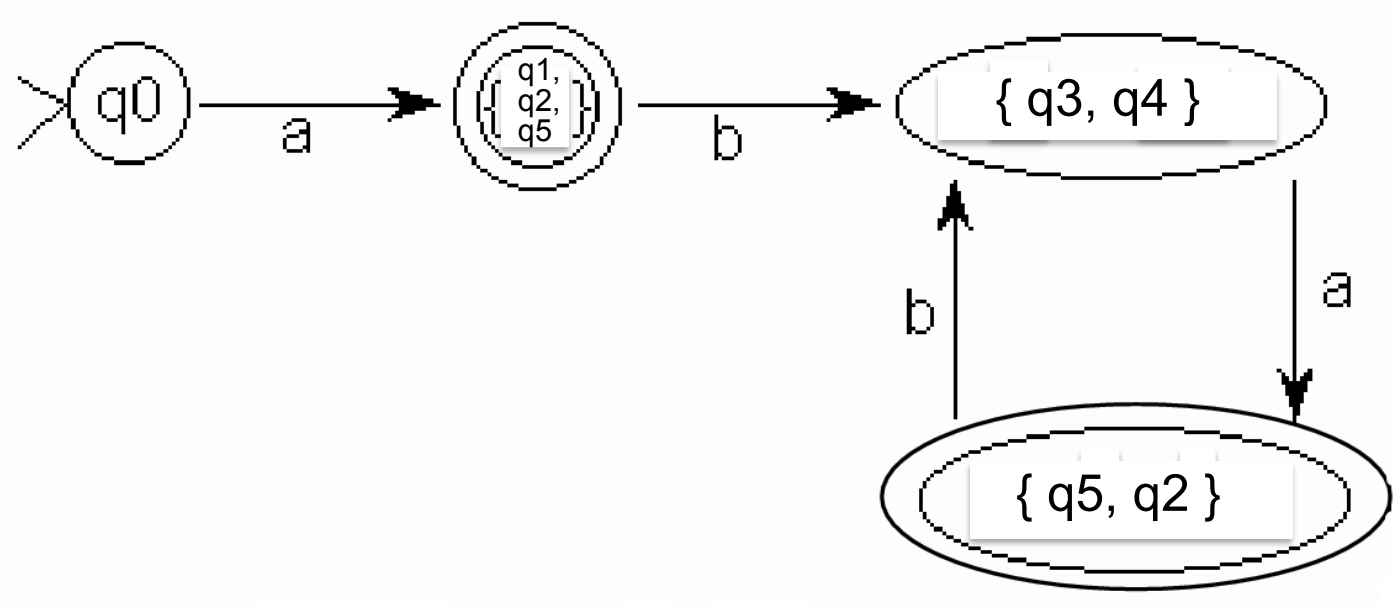


Beispiel: $a(ba)*$

NEA:



DEA:



Lex–Spezifikation

r_1 { action_1 }
r_2 { action_2 }
...
r_n { action_n }

$r_i \triangleq$ regulärer Ausdruck
$\text{action}_i \triangleq$ Aktion in bestimmter Programmiersprache

Beispiel

`public` | `protected` | `private` { } // (Token *Zugriffsrechte*)
`static` { } // (Token *STATIC*)
`abstract` { } // (Token *ABSTRACT*)
`class` { } // (Token *CLASS*)
`while` { } // (Token *WHILE*)
`do` { } // (Token *DO*)
`if` { } // (Token *IF*)
(`a`|...|`z`|`A`|...|`Z`)(`a`|...|`z`|`A`|...|`Z`|`0`|...|`9`)* { } // (Token *IDENTIFIER*)
`;` { } // (Token *SEMIKOLON*)
“Σ*” { } // (Token *STRING*)

Principle of longest match

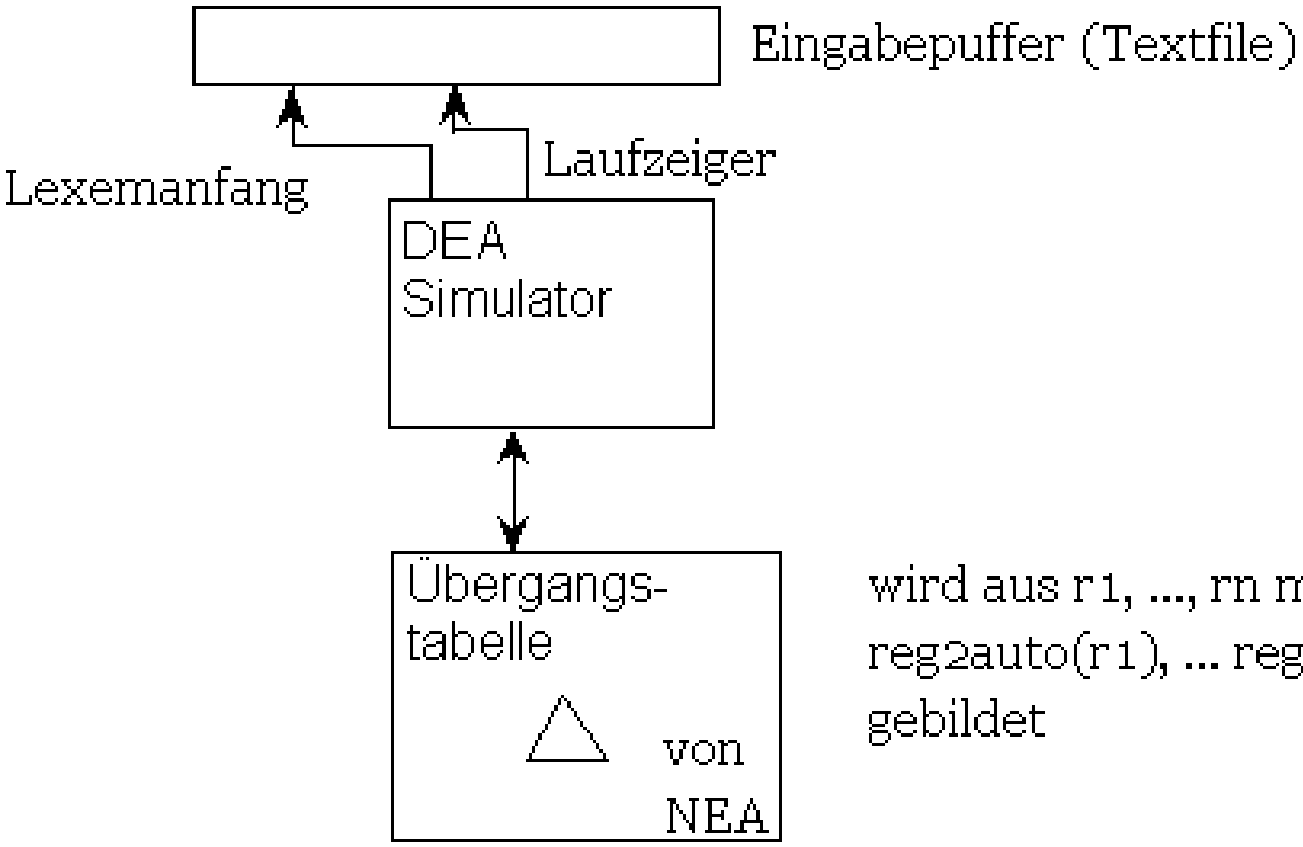
Es wird immer so weit gelesen, dass nach dem nächsten Zeichen kein regulärer Ausdruck der Lex–Spezifikation mehr passen würde.
Genügt das längste passende Lexem immernoch mehreren regulären Ausdrücken, so wird der reguläre Ausdruck genommen, der den kleinsten Index hat.

Principle of longest match

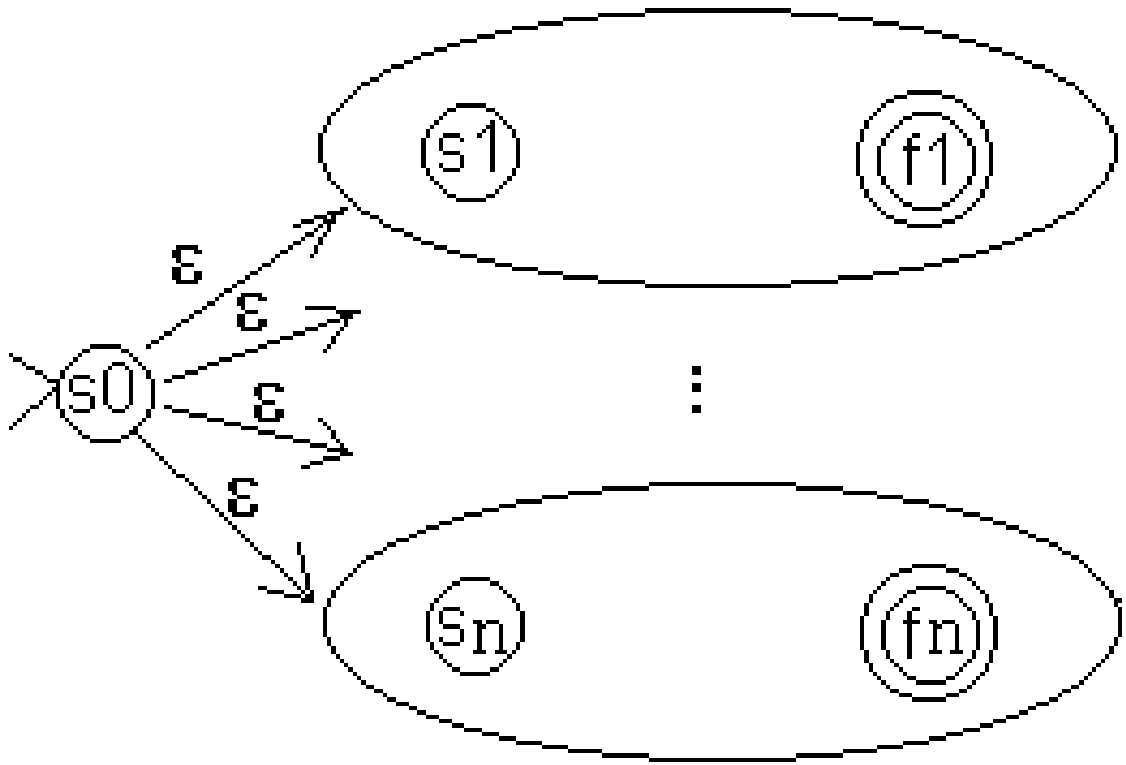
Es wird immer so weit gelesen, dass nach dem nächsten Zeichen kein regulärer Ausdruck der Lex–Spezifikation mehr passen würde.
Genügt das längste passende Lexem immernoch mehreren regulären Ausdrücken, so wird der reguläre Ausdruck genommen, der den kleinsten Index hat.

Beispiel:
whilei $\rightarrow$ Token *Identifier*
while $\rightarrow$ Token *WHILE*

Arbeitsweise eines Scanners



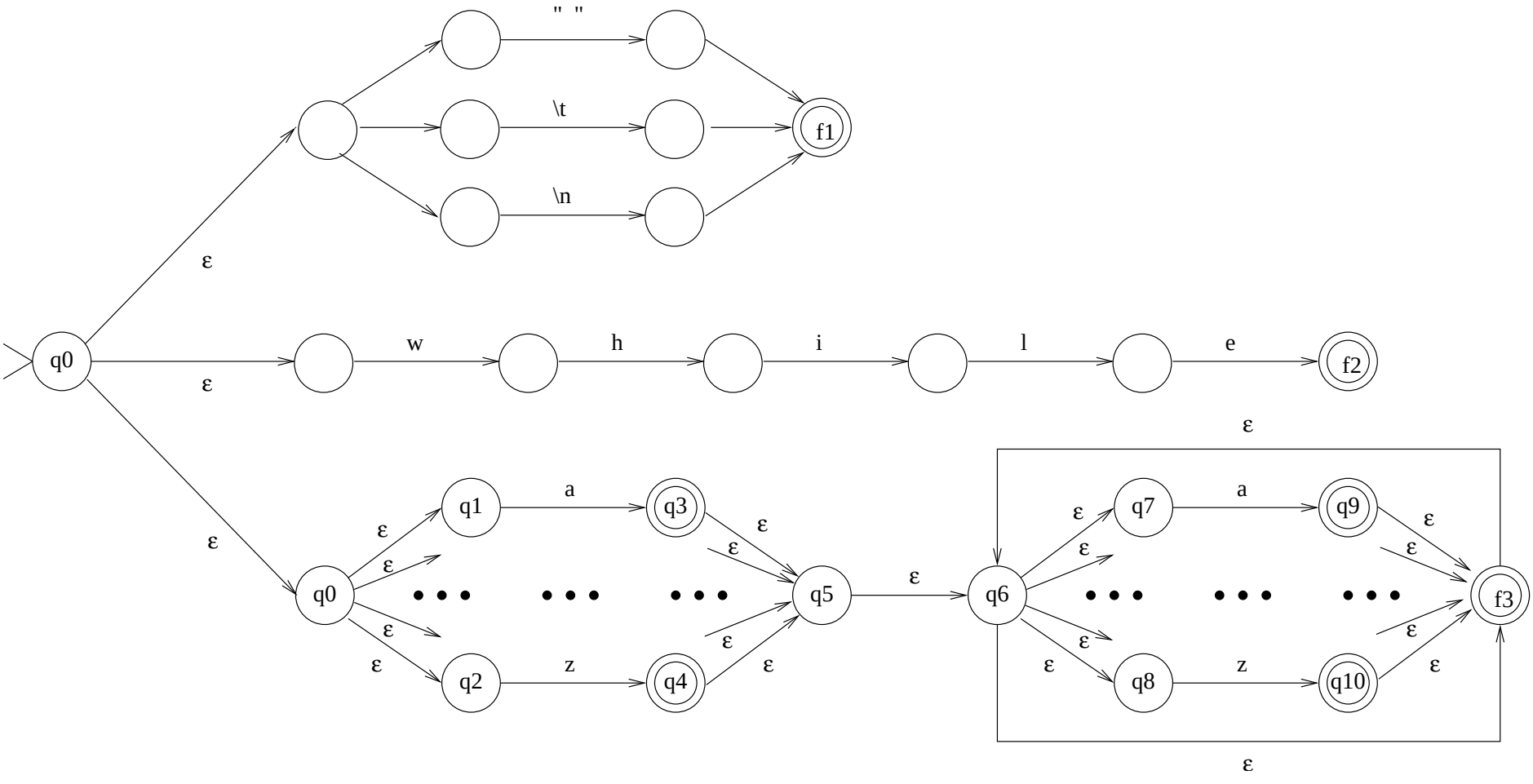
NEA der Lex–Spezifikation (Übergangstabelle)



Beispiel Mini-Java

```
[ \t\n]
while
[a-z][a-z]*
```

Beispiel Mini-Java



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
```