

# Compilerbau

Martin Plümicke  
Andreas Stadelmeier

SS 2025

# Beschreibung

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

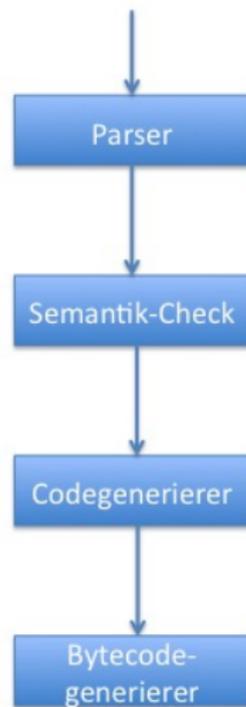
# Literatur

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

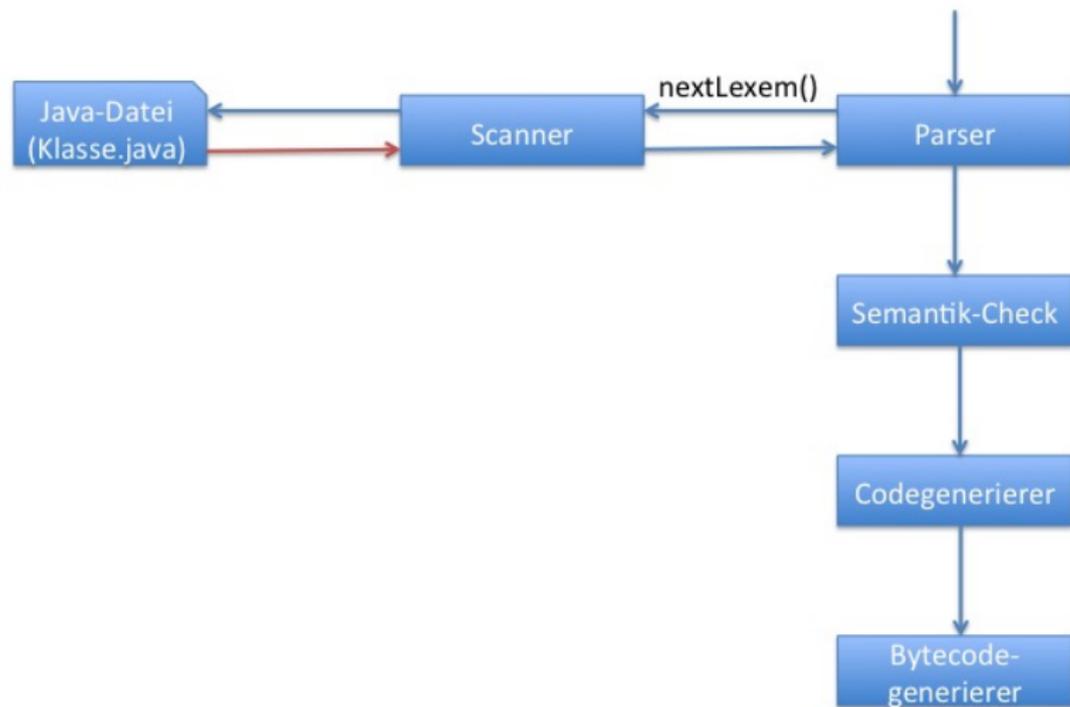
## Literatur II

-  James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley.  
*The Java<sup>®</sup> Language Specification.*  
The Java series. Addison-Wesley, Java SE 8 edition, 2014.
-  Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley.  
*The Java<sup>®</sup> 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.

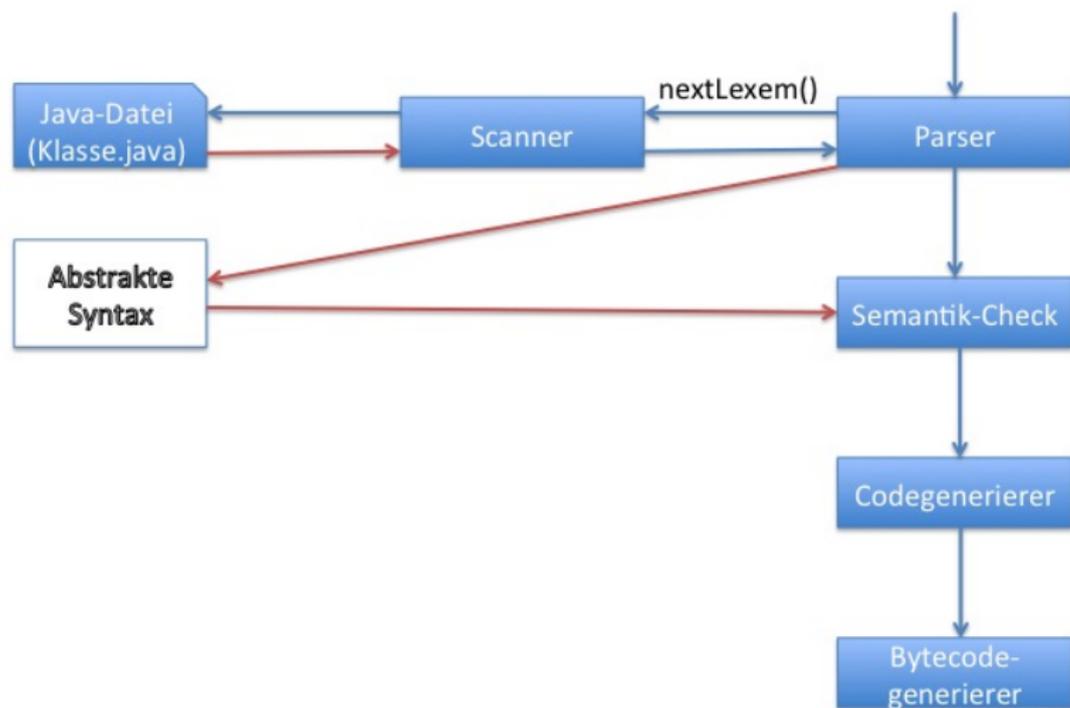
# Compiler Überblick



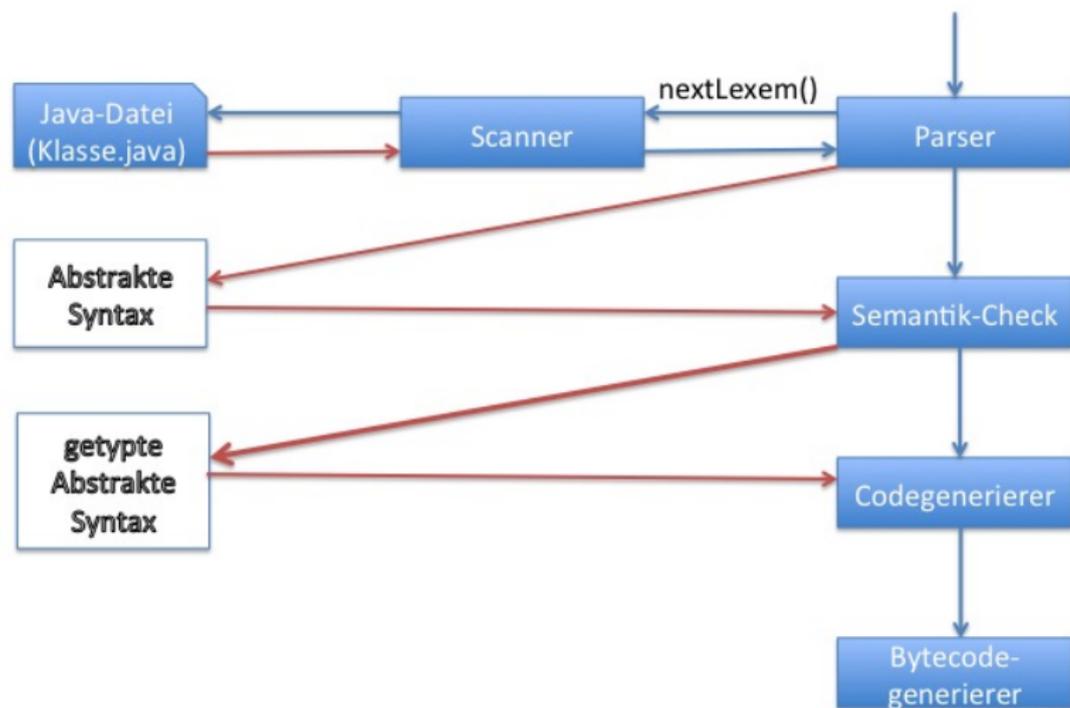
# Compiler Überblick



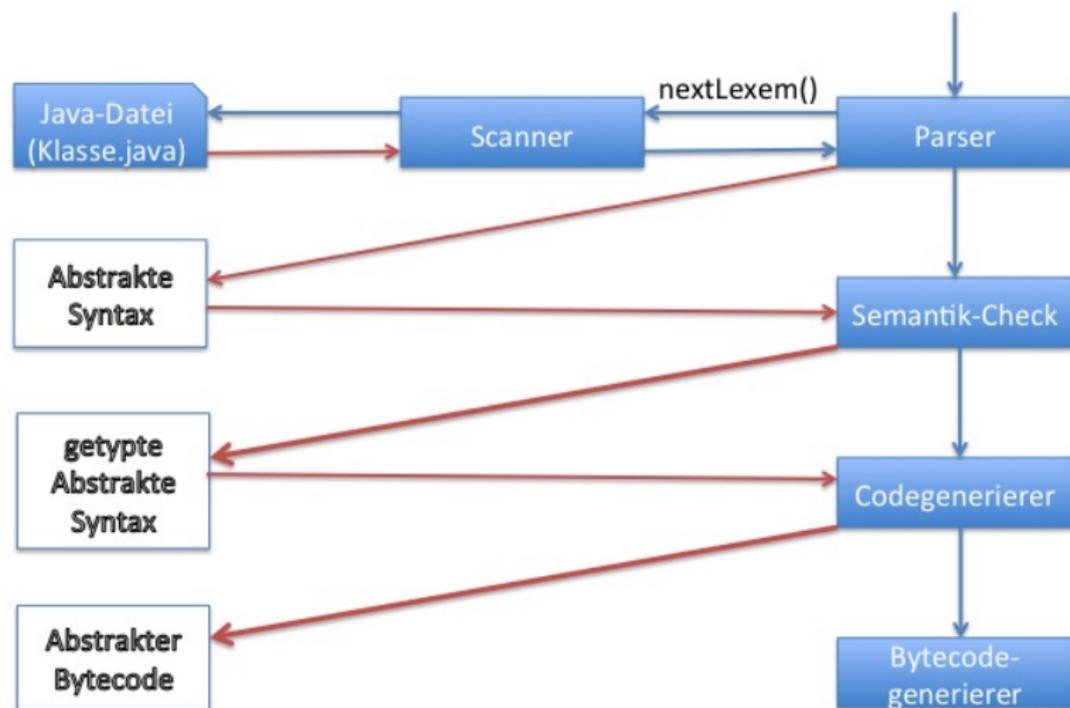
# Compiler Überblick



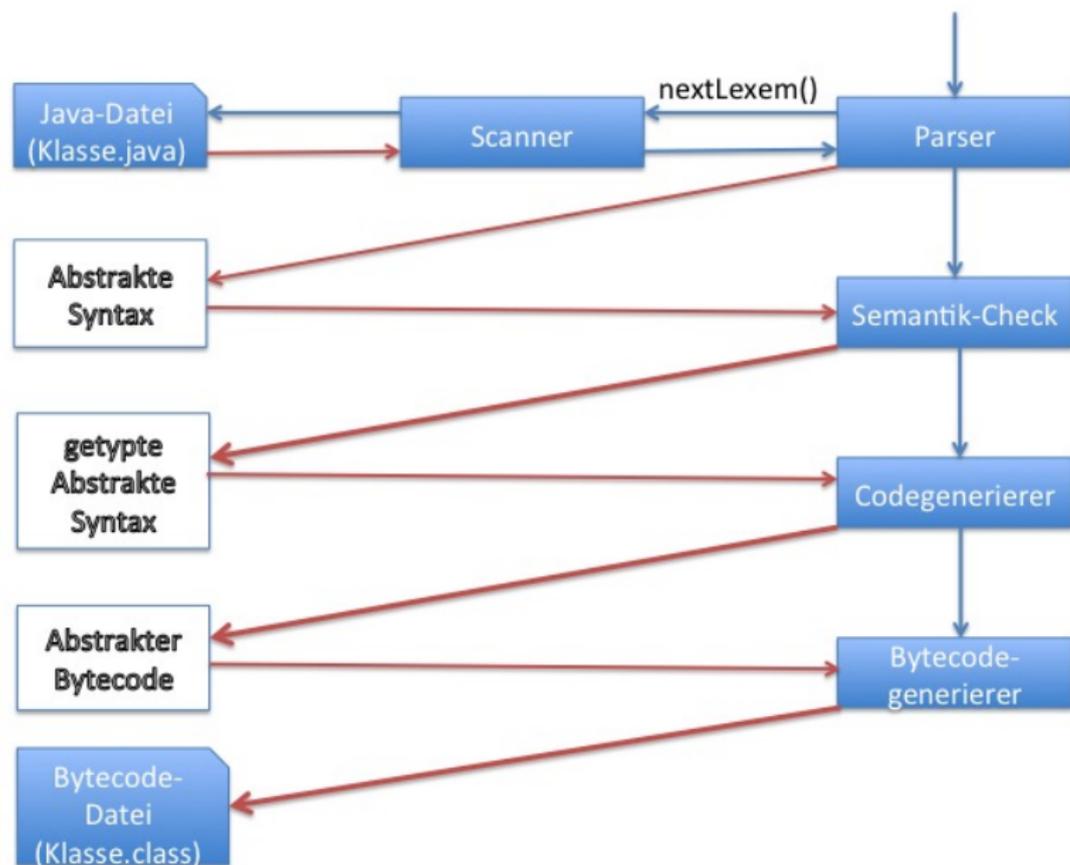
# Compiler Überblick



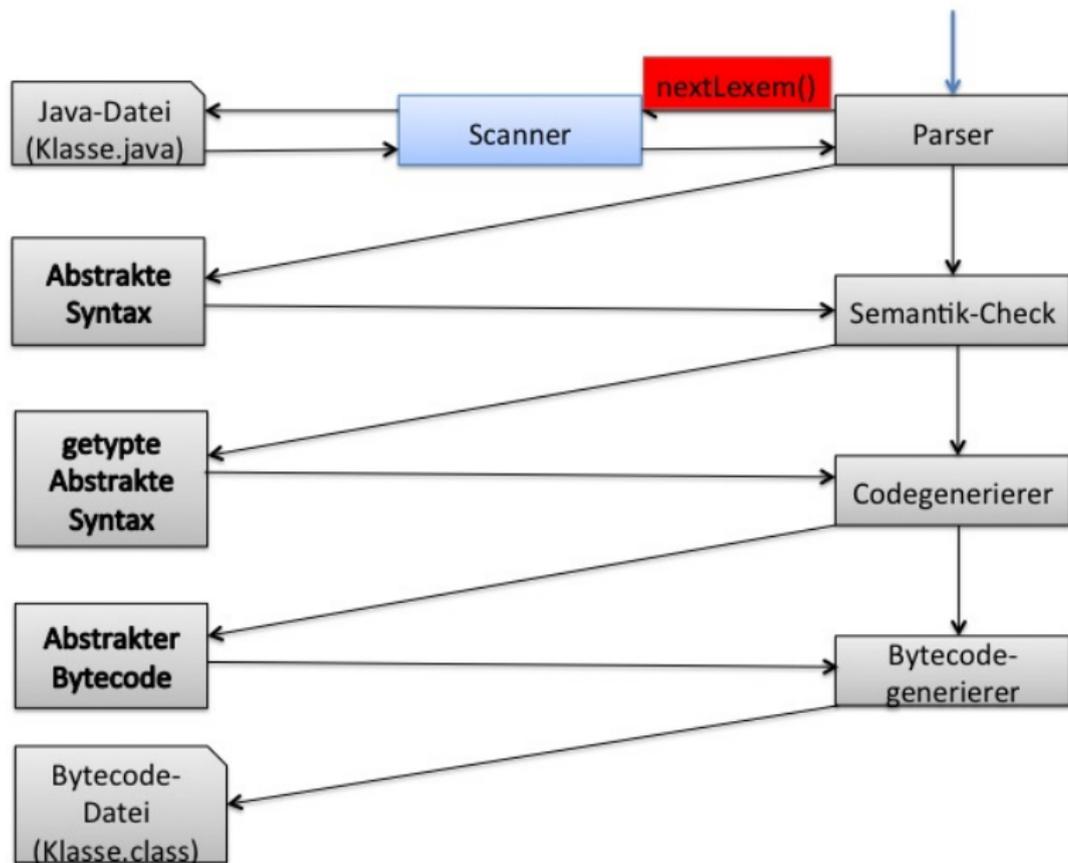
# Compiler Überblick



# Compiler Überblick



# Scanner



# 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  $\Sigma$  ein Alphabet, dann ist die Menge der regulären Ausdrücke über  $\Sigma : R(\Sigma)$  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  $\Sigma$  ein Alphabet, dann ist die Menge der regulären Ausdrücke über  $\Sigma : R(\Sigma)$  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:**  $\Sigma = \{a, b, c\}$

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

# 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.
  - ▶  $\alpha = \beta\gamma \Rightarrow \mathcal{L}(\alpha) = \{ww' \mid w \in \mathcal{L}(\beta), w' \in \mathcal{L}(\gamma)\}$
  - ▶  $\alpha = \beta|\gamma \Rightarrow \mathcal{L}(\alpha) = \mathcal{L}(\beta) \cup \mathcal{L}(\gamma)$
  - ▶  $\alpha = \beta^* \Rightarrow \mathcal{L}(\alpha) = \{\varepsilon\} \cup \{ww' \mid w \in \mathcal{L}(\beta), w' \in \mathcal{L}(\beta^*)\}$
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

- ▶  $Q \hat{=}$  Zustandsmenge (endlich)
- ▶  $\Sigma \hat{=}$  Alphabet
- ▶  $\delta \hat{=}$  Übergangsfunktion:  $Q \times \Sigma \rightarrow Q$
- ▶  $s \in Q \hat{=}$  Anfangszustand
- ▶  $F \subseteq Q \hat{=}$  Menge der Finalzustände

# Deterministische endliche Automat (DEA)

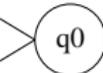
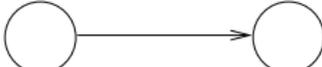
Unter einem DEA versteht man

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

mit

- ▶  $Q \hat{=}$  Zustandsmenge (endlich)
- ▶  $\Sigma \hat{=}$  Alphabet
- ▶  $\delta \hat{=}$  Übergangsfunktion:  $Q \times \Sigma \rightarrow Q$
- ▶  $s \in Q \hat{=}$  Anfangszustand
- ▶  $F \subseteq Q \hat{=}$  Menge der Finalzustände

DEAs kann man grafisch darstellen:

- ▶ **Zustände:** 
- ▶ **Startzustand:** 
- ▶ **Finalzustand:** 
- ▶ **Übergangsfunktion:** 

## Reg. Ausdruck nach DEA

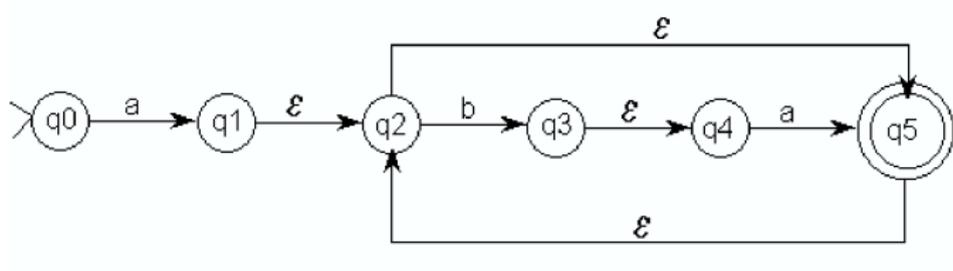
Man kann reguläre Ausdruck in DEA's übersetzen, die die Sprachen der regulären Ausdruck erkennen:

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

Beispiel:  $a(ba)^*$

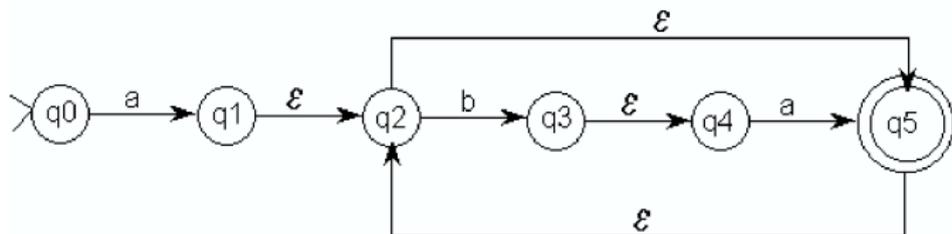
Beispiel:  $a(ba)^*$

NEA:

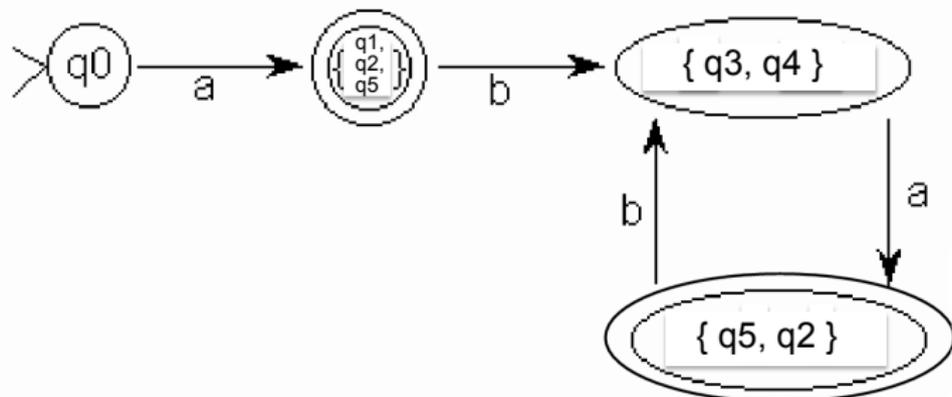


Beispiel:  $a(ba)^*$

NEA:



DEA:



# Lex-Spezifikation

$r_1$     { action<sub>1</sub> }

$r_2$     { action<sub>2</sub> }

...

$r_n$     { action<sub>n</sub> }

$r_i \hat{=}$  regulärer Ausdruck

action<sub>i</sub>  $\hat{=}$  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.

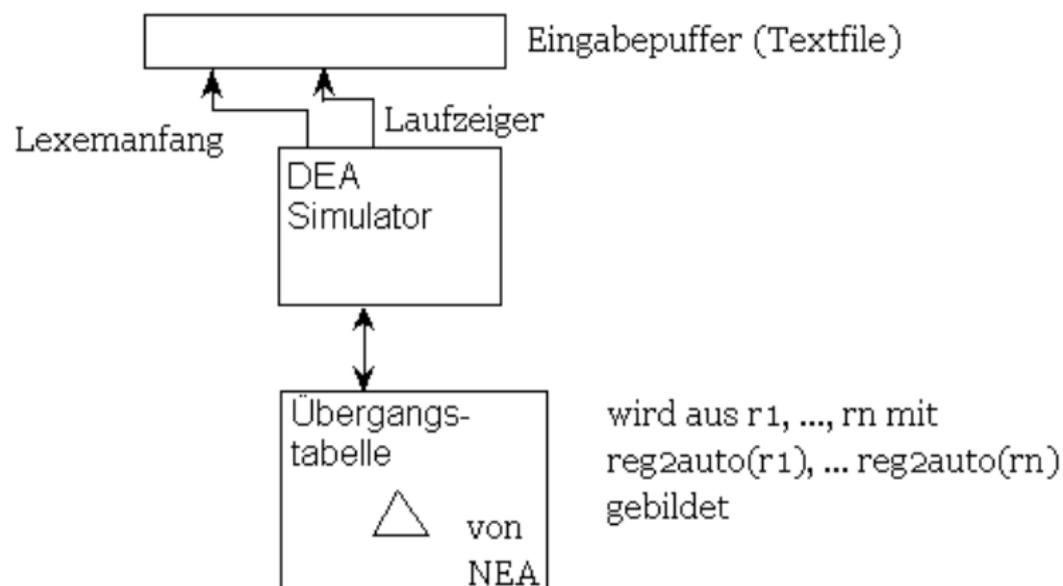
Genügt das längste passende Lexem immernoch mehreren regulären Ausdrücken, so wird der reguläre Ausdruck genommen, der den kleinsten Index hat.

### Beispiel:

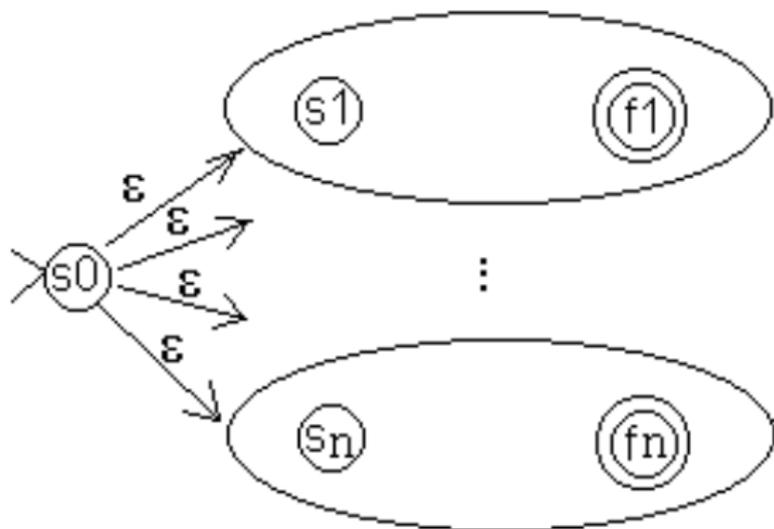
whilei → Token *Identifier*

while → Token *WHILE*

# Arbeitsweise eines Scanners



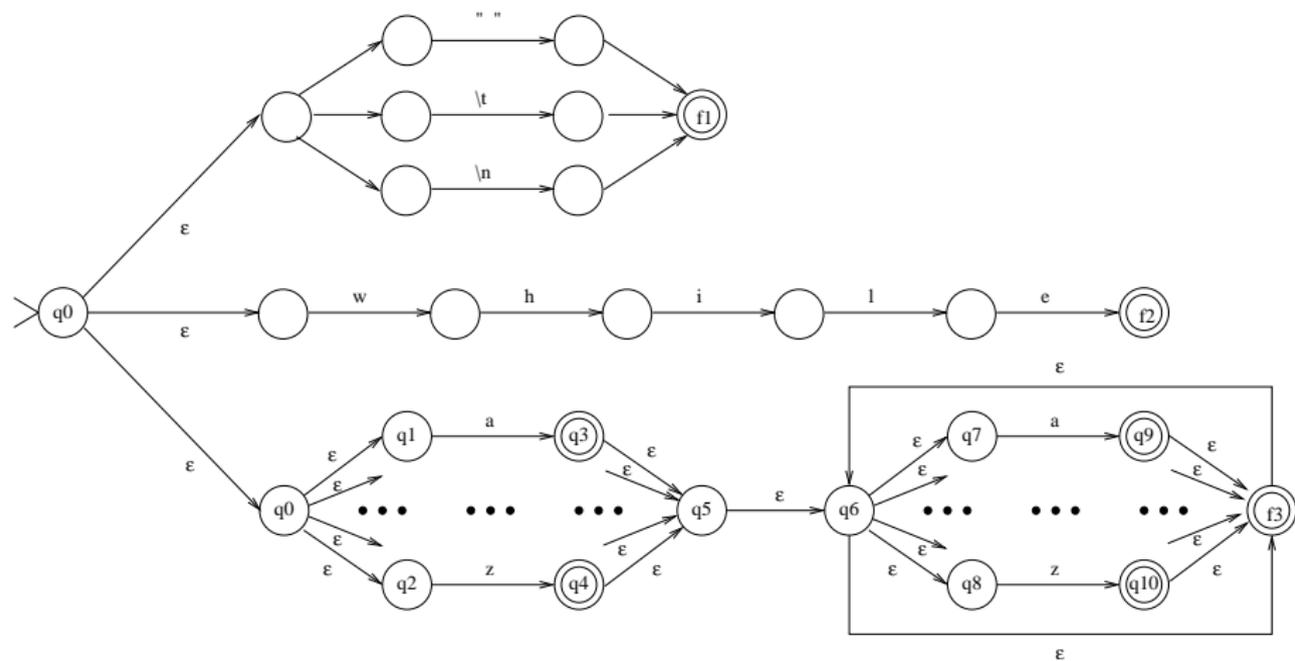
## NEA der Lex-Spezifikation (Übergangstabelle)



# Beispiel Mini-Java

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

# Beispiel Mini-Java



# 1. Ansatz: Scanner direkt implementieren I

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

import Data.Char

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

# 1. Ansatz: Scanner direkt implementieren II

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

# 1. Ansatz: Scanner direkt implementieren III

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

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

## 2. Ansatz: Scanner-Tools

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

# Alex-Spezifikation

```
{  
  Haskell-code  
}
```

# Alex-Spezifikation

```
{  
  Haskell-code  
}
```

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

# Alex-Spezifikation

```
{  
  Haskell-code  
}  
  
$ abk1 = regExp1  
$ abk2 = regExp2  
...  
$ abkn = regExpn  
  
%wrapper " wrapper"
```

# Alex-Spezifikation

```
{  
  Haskell-code  
}
```

```
$ abk1 = regExp1
```

```
$ abk2 = regExp2
```

```
...
```

```
$ abkn = regExpn
```

```
%wrapper " wrapper"
```

```
tokens :=
```

*lex--Spezifikation*

# Alex-Spezifikation

```
{  
  Haskell-code  
}  
  
$ abk1 = regExp1  
$ abk2 = regExp2  
...  
$ abkn = regExpn  
  
%wrapper " wrapper"  
  
tokens :=
```

*lex--Spezifikation*

```
{  
  Haskell-code  
}
```

# Alex-Spezifikation Beispiel

```
{  
}
```

```
%wrapper "basic"
```

```
$digit = 0-9          -- digits
```

```
$alpha = [a-zA-Z]    -- alphabetic characters
```

```
tokens :-
```

```
  $white+           ;
```

```
  "--".*           ;
```

```
  let              { \s -> LetToken }
```

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

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

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

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

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

## Alex-Spezifikation Beispiel II

```
{  
  -- The token type:  
  
  data Token = LetToken  
             | InToken  
             | SymToken Char  
             | VarToken String  
             | IntToken Int  
             deriving (Eq, Show)  
  
  main = do  
    s <- getContents  
    print (alexScanTokens s)  
}
```

# Wrapper

Es gibt in Alex einige vordefinierte Wrapper:

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

# Types der token actions (basic Wrapper)

String -> Token