



Compiler

Lexikalische Analyse

# Kapitel 2

## Lexikalische Analyse

词法分析

# Vorspann

this is some text without spaces and punctuation marks which is therefore quite difficult to read by humans lexical analysis will break this text up into words while the parsing phase will extract the grammatical structure of the text

这段没有空格和标点符号的文字，因此很难被人类阅读，逻辑分析将把这段文字分成几个字，同时解析法将提取文字的语法结构  
this is some text without spaces and punctuation marks which is therefore quite difficult to read by humans lexical analysis will break this text up into words while the parsing phase will extract the grammatical structure of the text

This **is** some **text** without **spaces** and **punctuation-marks** which **is** therefore quite difficult to **read** by **humans**. **Lexical-analysis** **will** **break** this text up into **words** while the **parsing-phase** **will** extract the **grammatical structure** of the text.

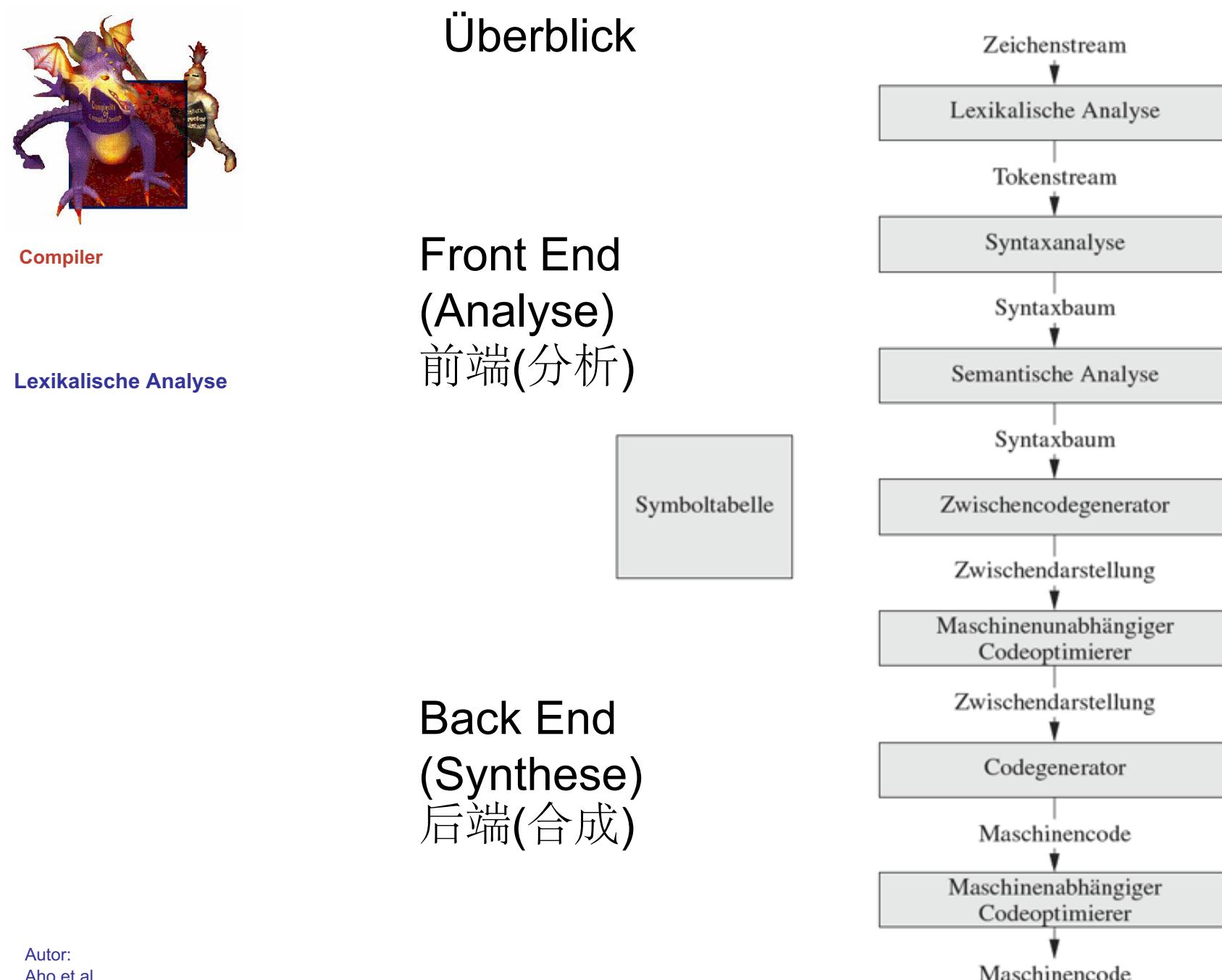
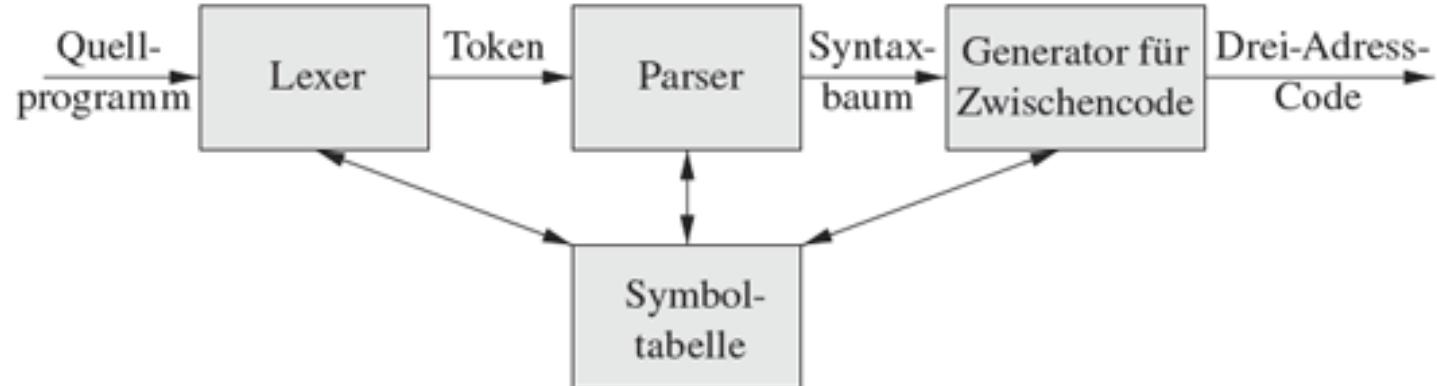


Abbildung 1.6: Phasen eines Compilers



Compiler



## Lexikalische Analyse

Abbildung 2.3: Modell eines Compiler-Front-Ends

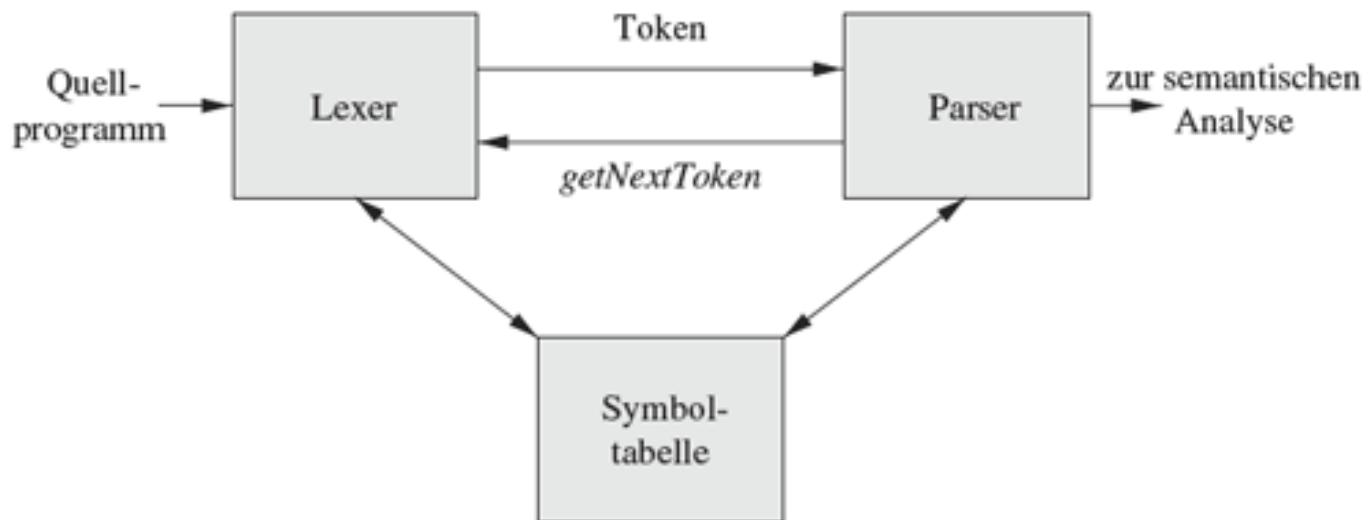


Abbildung 3.1: Interaktionen zwischen dem Lexer und dem Parser



Compiler

Autor:

Folie: 5

# Token, Lexeme 代号、词素

- Ein **Token** ist ein Paar aus einem Namen und einem optionalen Attributwert. Der Name steht für eine Kategorie von lexikalischen Einheiten, beschrieben durch ein **Muster**.
  - Schreibweise: <**Name**,Wert> oder <**Name**> falls kein Wert
- Ein **Lexem** in eine Folge von Zeichen im Quellprogramm, die dem Muster für ein Token entspricht.
- In “const pi = 3.1416;” ist “pi” ein Lexem für den Tokennamen “Bezeichner”



Compiler

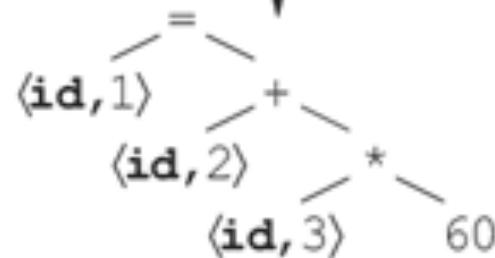
Lexikalische Analyse

position = initial + rate \* 60

Lexikalischer Analysator

$\langle \text{id}, 1 \rangle \langle= \rangle \langle \text{id}, 2 \rangle \langle + \rangle \langle \text{id}, 3 \rangle \langle * \rangle \langle 60 \rangle$

Syntaktischer Analysator



1	position	...
2	initial	...
3	rate	...

SYMBOLTABELLE

Ziel: Folgen von Buchstaben erkennen die als Einheit behandelt werden können 目的：认识可以作为一个单位的字母序列。



# Beispiele

Compiler

Lexikalische Analyse

## Muster

Token	Informelle Beschreibung	Beispiellexeme
if	Zeichen i, f	if
else	Zeichen e, l, s, e	else
comparison	< oder > oder <= oder >= oder == oder !=	<=, !=
id	Buchstabe, auf den Buchstaben oder Ziffern folgen	pi, score, D2
number	Alle numerischen Konstanten	3.14159, 0, 6.02e23
literal	Alles außer Anführungszeichen, was in Anführungszeichen steht	"core dumped"

Abbildung 3.2: Beispiele für Token



Compiler

Autor:

Folie: 8

- 1. Ein Token für jedes Schlüsselwort.
- 2. Token für Operatoren, entweder einzeln oder in Klassen
- 3. Ein Token für alle Bezeichner
- 4. Ein oder mehrere Token für Konstanten wie Zahlen und Literalstrings
- 5. Token für jedes Satzzeichensymbol wie linke und rechte Klammern, Komma und Semikolon



Compiler

Autor:

Folie: 9

# Gründe für die lexikalische Analyse 词汇分析的原因

- Vereinfachung des Parsers: 简化解  
析器
  - Leerraum (Whitespace) entfernt, 去除  
空白 größere Grundeinheiten erkannt
- Effizienz 效率
  - Puffertechniken 缓冲技术
- Portierbarkeit 便携性
  - LF/CR auf Windows und Unix, Unicode



Compiler

Autor:

# Intermezzo: Lexing

- nowhereisromancementavailable
- nowhere is romancement available
- now here is roman cement available
- no where is roman cement avail able

blumentopferde 献花



Compiler

Autor:

# Fortran

- **Consistently separating words by spaces became a general custom about the tenth century A.D., and lasted until about 1957, when FORTRAN abandoned the practice.**-- Sun FORTRAN Reference Manual

用空格分隔单词成为一种普遍的习惯，一直持续到1957年左右，FORTRAN放弃了这种做法。

If a variable is not declared, it is implicitly given a type based on its first letter (I to N being integers, the rest floats)- 如果一个变量没有被声明，它将根据它的第一个字母被隐含地赋予一个类型（I到N是整数，其余是浮点数）。

```
DO 50 I = 1 . 100
```

```
IF ...
```

```
50 CONTINUE
```



Compiler

Autor:

Folie: 12

# Collection of Software Bugs

## 软件漏洞收集

- **6. NASA Mariner 1 , Venus probe**
- **(period instead of comma in FORTRAN DO-Loop, 1962)**
- **Horror Nr. 25 with reference to**
- **G.J.Myers: Software Reliability: Principles & Practice, p. 25**
- **The FORTRAN-Code**
- **NASA**
- **Additional Information on code bugs**
- **Mariner software bug is now considered to be an urban legend, see zope risks , risk digest 8.75 subj 1 , risk digest 9.54 subj 1 , wikipedia**

An error in a single FORTRAN statement resulted in the loss of the first American probe to Venus. From G. J. Myers, *Software Reliability: Principles & Practice*, p. 25.一条FORTRAN语句的错误导致了美国第一个前往金星的探测器的损失。摘自G. J. Myers, *Software Reliability: 原则与实践*，第25页。



Compiler

Autor:

Folie: 13

## 22. Juli 1962, Cape Canaveral/Florida Start der ersten amerikanischen Venussonde Mariner 1 Trägerrakete Atlas-Agena B (NASA, 15. AAB-Start)

```
• ...
• IF (TVAL .LT. 0.2E-2) GOTO 40
• DO 40 M = 1, 3
• W0 = (M-1)*0.5
• X = H*1.74533E-2*W0
• DO 20 NO = 1, 8
• EPS = 5.0*10.0** (NO-7)
• CALL BESJ(X, 0, B0, EPS, IER)
• IF (IER .EQ. 0) GOTO 10
• 20 CONTINUE
• DO 5 K = 1. 3
• T(K) = W0
• Z = 1.0/(X**2)*B1**2+3.0977E-4*B0**2
• D(K) = 3.076E-2*2.0*(1.0/X*B0*B1+3.0977E-4*
• *(B0**2-X*B0*B1))/Z
• E(K) = H**2*93.2943*W0/SIN(W0)*Z
• H = D(K)-E(K)
• 5 CONTINUE
• 10 CONTINUE
• Y = H/W0-1
• 40 CONTINUE
• ...
• ...
```

**Ausschnitt aus dem FORTRAN-  
Programm zur Steuerung der  
Flugbahn der Trägerrakete:**



## **Entscheidender Fehler:**

### **Punkt statt Komma!**

#### **Wirkung:**

D05K = 1.3

Wertzuweisung an eine nicht deklarierte Variable

Kein Durchlauf der (nicht vorhandenen) Schleife

#### **Folge:**

Abweichung der Trägerrakete von der vorgesehenen Flugbahn

Zerstörung der Rakete nach 290 Sekunden

Kosten dieser Wertzuweisung: \$ 18.5 Millionen

#### **Ursache:**

Schlechte Programmiersprache wegen Blanks (Zwischenräume) in Namen und Zahlen erlaubt

Variablen-Deklaration nicht notwendig

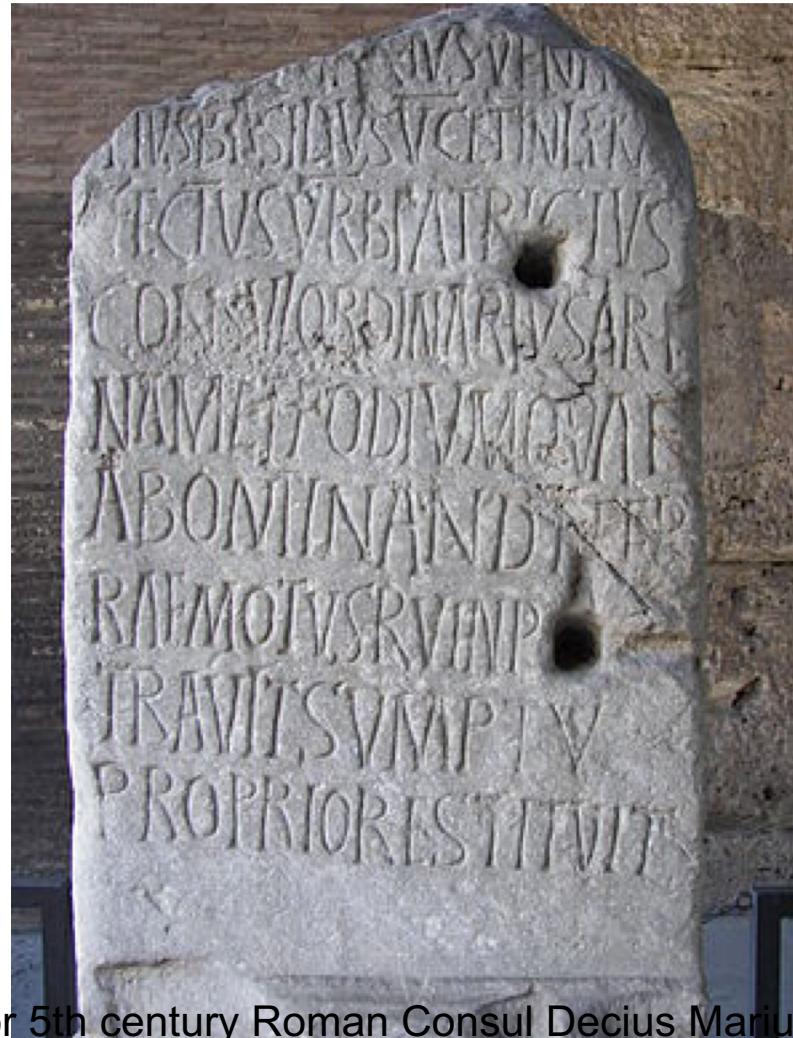
Strukturierte Schleife (END DO) nicht möglich

Würde auch heute noch so fehl-funktionieren!

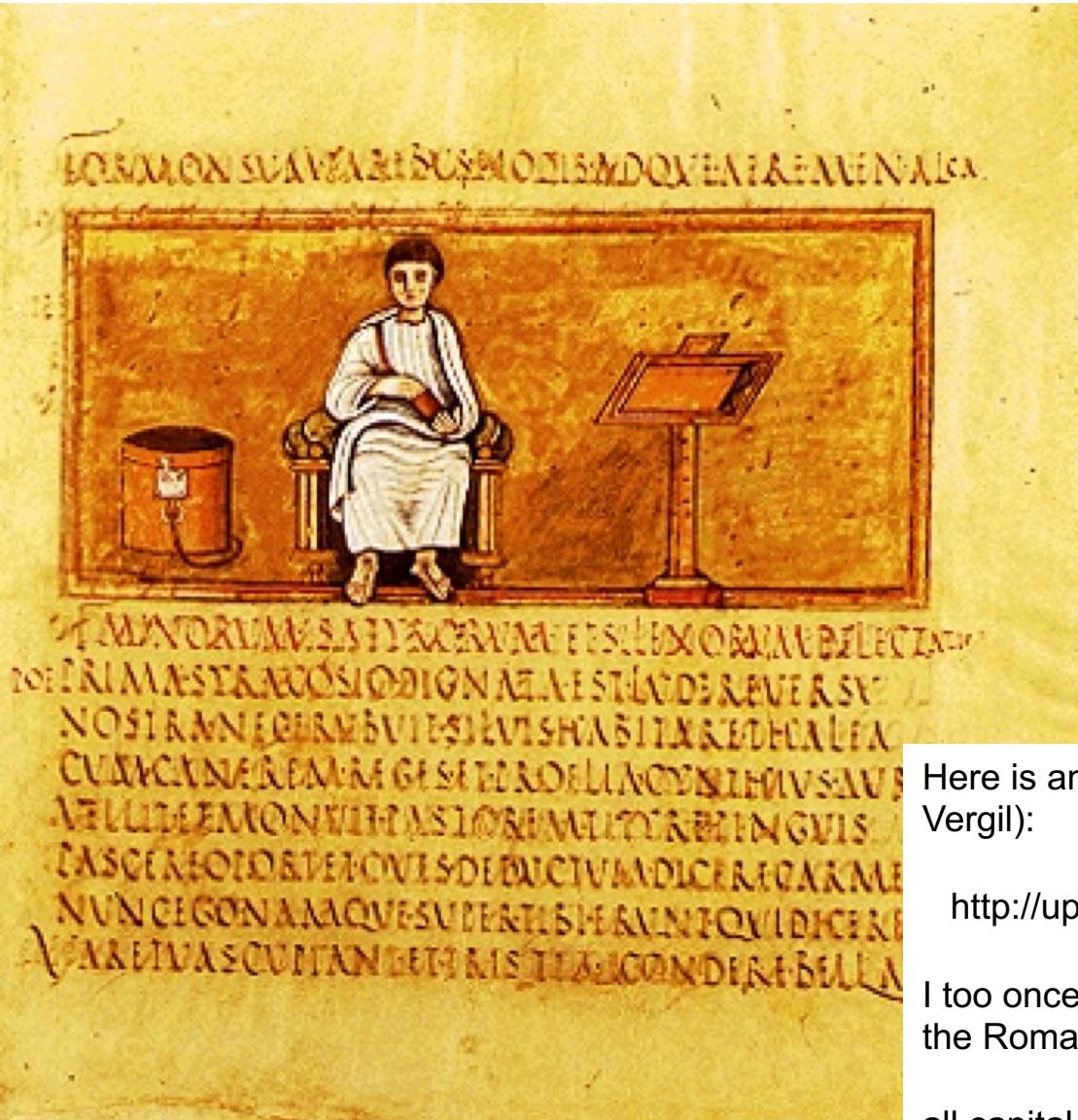


Compiler

## Lexikalische Analyse



Inscription for 5th century Roman Consul Decius Marius Venantius Bassilius in the Colosseum in Rome. CIL VI 1716 c, VI 32094 c. 罗马斗兽场内为5世纪罗马执政官德西乌斯Q马利乌斯Q贝西利乌斯的题词。CIL VI 1716 c, VI 32094 c.



Here is an example of Imperial writing (it's a folio of Vergil):

<http://upload.wikimedia.org/wikipedia/commons/0/0rtrait.jpg>

I too once craved to write in the orthographic style of the Romans, and you can manage it too pretty easily:

- all capital letters
- no spaces
- no punctuation
- no paragraphing



Compiler

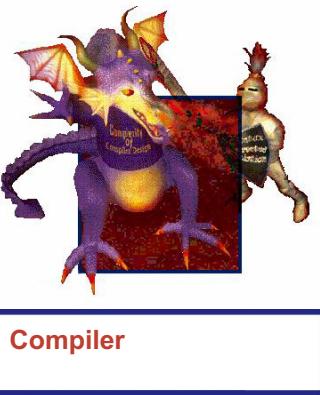
Autor:

Folie: 17

# Gründe des Studiums

## 研究的原因

- Techniken nützlich für andere Anwendungen:
  - Unix, Scripting,... (grep, ...)
  - Pattern matching, Bioinformatik,...
  - Verifikation
- Zusammenhang zur theoretischen Informatik对其他应用有用的技术。
  - - Unix, 脚本, ... (grep, ...)
  - - 模式匹配, 生物信息学, ...
  - - 核查
  - - 与理论计算机科学的联系



# Knifflige Probleme

## 棘手的问题

- if 1 == if1 then then1 = the

- Fortran:

- DO 5 I = 1.25
  - DO 5 I = 1,25

Anmerkung: lexikalische Fehler  
eher selten: m 注意：词汇错误相当罕见。

fi 1 == if1 then then1 = else



Compiler

Autor:

Folie: 19

# Tokenmuster

a aa aaa aaaa

i if ifa

1 11

12

...

Wie können wir Tokenmuster für eine  
Programmiersprache beschreiben?  
我们如何描述一种编程语言的标记模  
式?



Compiler

Autor:

Folie: 20

# Tokenmuster令牌模式

Auf Englisch für Java/C:

一个标识符是一串字母

– An **identifier** is a sequence of letters and digits; the first character must be a letter. The underscore \_ counts as a letter. Upper- and lowercase letters are different. If the input stream has been parsed into tokens up to a given character, the next token is taken to include the **longest** string of characters that could possibly constitute a token.  
Blanks, ...

Andere Vorschläge ?



Compiler

Autor:

Folie: 21

- # Formale Sprachen
- **Alphabet**  $\Sigma$ : endliche Menge von Symbolen
    - $\{0,1\}$ ,  $\{a,b,c,\dots,z\}$ , Ascii, ...
  - **String** (Wort): **endliche Folge** von Symbolen  $\in \Sigma$ 
    - 101, helloworld, if  $a=0$  then  $a := b$
  - **Sprache** = **abzählbare Menge** von Strings
    - Primzahlen im Binärformat, English, Java Programme
    - Lexeme für ein Token !!  
**Identifier** :  $\{a,b,\dots,foo,\dots,x\_3,\dots\}$



Compiler

Lexikalische Analyse

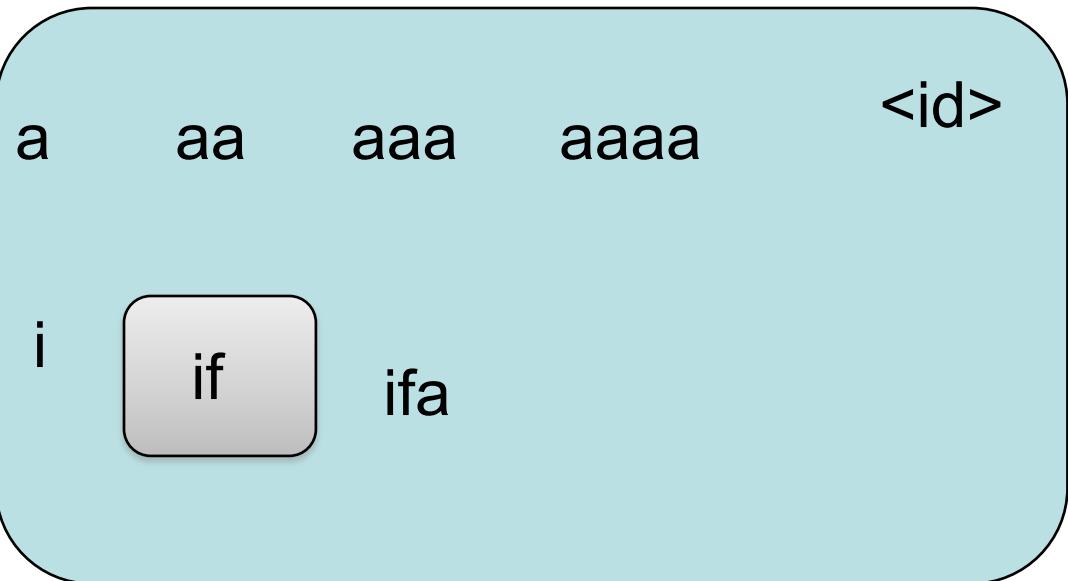
**Alphabet:** {a,b,c,...,z,0,1,...9,...}

## **String**

(Wort): endliche

Folge  
字符串

(有限序列)



## **Sprache:**

Menge von

Strings





# Notationen

Compiler

Autor:

Folie: 23

- Leerer String:  $\epsilon$
- Präfix, Suffix, Teilstring
- Konkatenation von x und y: xy
  - x = pro, y = gramm, xy = programm
  - $x\epsilon = \epsilon x = x$
- Potenzierung
  - $s^0 = \epsilon$
  - $s^i = s^{i-1}s$
  - $s = ab, s^2 = abab$



Compiler

Autor:

Folie: 24

Operation	Definition und Schreibweise
Vereinigung von $L$ und $M$	$L \cup M = \{s   s \text{ ist in } L \text{ oder } s \text{ ist in } M\}$
Konkatenation (Verkettung) von $L$ und $M$	$LM = \{st   s \text{ ist in } L \text{ und } t \text{ ist in } M\}$
Kleene-Hülle von $L$	$L^* = \bigcup_{i=0}^{\infty} L^i$
Positive Hülle von $L$	$L^+ = \bigcup_{i=1}^{\infty} L^i$



Compiler

Autor:

Folie: 25

# Wie beschreiben wir die Sprache für ein Token? 我们如何描述令牌的语言?

- **Identifier** : {a,b,...,foo,...,x\_3,...}



Compiler

Autor:

# Reguläre Ausdrücke I

- Beschreibung formaler Sprachen
- Bausteine:  $a$   $\varepsilon$  | . \* ( )
- $a$ , wobei  $a \in \Sigma$ 
  - bezeichnet:  $L(a) = \{“a”\}$
- $\varepsilon$ , mit  $\varepsilon \notin \Sigma$ 
  - bezeichnet:  $L(\varepsilon) = \{\varepsilon\} = \{“”\}$   
Menge mit einem String ( $\varepsilon$ )
  - verschieden von  $\emptyset = \{\} !!!$



Compiler

Autor:

# Reguläre Ausdrücke II

- $M|N$ , mit M,N als reguläre Ausdrücke
  - $L(M|N) = L(M) \cup L(N)$
- $MN$ , mit M,N als reguläre Ausdrücke
  - $L(MN) = \{\alpha\beta \mid \alpha \in L(M) \wedge \beta \in L(N)\}$
  - $= L(M)L(N)$
- $L((a|b)c) = ?$



Compiler

Autor:

# Kleine Übung

- $L((a|b)c) = ?$
- $L(a) = \{"a"\}$
- $L(b) = \{"b"\}$
- $L((a|b)) = \{"a\} \cup \{"b\} = \{"a", "b"\}$
- $L(c) = \{"c"\}$
- $L((a|b)c) = \{"ac", "bc"\}$



Compiler

Autor:

# Reguläre Ausdrücke III

- $M^*$ , mit M als regulärem Ausdruck
    - Wiederholung (0 oder mehr)
    - “Kleene closure”
    - $L(M^*) = \{\alpha_1 \alpha_2 \dots \alpha_k \mid k \geq 0 \wedge \alpha_i \in L(M)\} = L(M)^*$
    - $L((a|b)^*) = \{ "", "a", "b", "aa", "ab", "ba", "bb", "aaa", \dots \}$
    - $L((ab)^*) = \{ "", "ab", "abab", "ababab", \dots \}$
- Konkatenierung von Folgen



Compiler

Autor:

Folie: 30

# Algebraische Gesetze

Gesetz	Beschreibung
$r s = s r$	ist kommutativ.
$r (s t) = (r s) t$	ist assoziativ.
$r(st) = (rs)t$	Konkatenation ist assoziativ.
$r(s t) = rs rt; \quad (s t)r = sr tr$	Konkatenation ist distributiv über  .
$\epsilon r = r\epsilon = r$	$\epsilon$ ist die Identität für Konkatenation.
$r^* = (r \epsilon)^*$	$\epsilon$ ist in einer Hülle garantiert.
$r^{**} = r^*$	* ist idempotent.

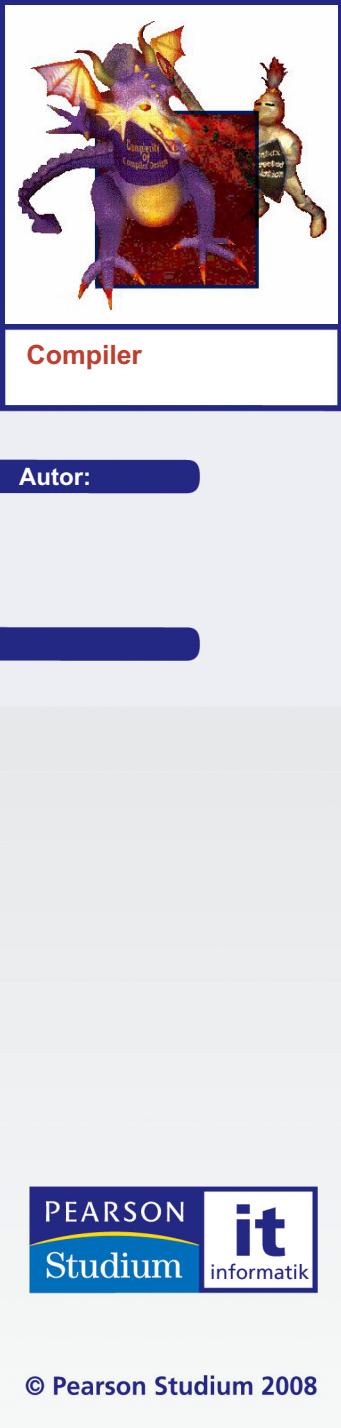


Compiler

Autor:

# Quiz 测验

- Schreiben Sie reguläre Ausdrücke für
  - Binärzahlen als Strings über  $\{0,1\}$ :  
 $\{0,1,10,11,100,101,\dots\}$ 
    - (vermeiden Sie 01, 00, 001, 000,...)
  - Binärzahlen  $> 101$
  - Dezimalzahlen  
 $\{0,1,2,3,4,5,6,7,8,9,10,\dots\}$ 
    - (vermeiden Sie: 01, 02,..., 00, 001, 002,...)

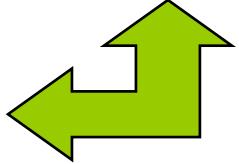


# Quiz: Antworten

höchste Priorität: \*

dann: .

dann: |

- Binärzahlen  $\{0, 1, 10, 11, 100, 101, \dots\}$
- $0 \mid (1((0|1)^*)) = 0 \mid 1(0|1)^*$  
- Binärzahlen  $> 101$
- $1( \ 1(0|1)(0|1)^* \mid 0(0|1)(0|1)(0|1)^* )$
- Dezimalzahlen  
 $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, \dots\}$
- $0 \mid$   
 $(1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)^*$ 

unbequem!



Compiler

Autor:

# Andere Notationen

Syntactic Sugar:  
Does not extend power

- **M<sup>+</sup>** : Wiederholung (mindestens 1 mal)  
–  $L(M^+) = L(M(M^*))$
- **M?** : optionaler Teil 可选部分  
–  $L(M?) = L(\varepsilon|M)$
- **[a-zA-D2-9]** : “character set alternation”  
– z.B.:  $L([b-y]) = L(b|c|d|\dots|x|y)$
- **“a.+”** : Anführung, bezeichnet sich selber
- **.** : jedes Zeichen ausser “newline”  
(Aufpassen, dies steht nicht für Konkatenation!)



Compiler

Autor:

# Beispiele

- $0 \mid (1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)^*$   
wird zu:
  - $0 \mid [1-9][0-9]^*$
  - $1( \ 1(0|1)(0|1)^* \mid 0(0|1)(0|1)(0|1)^* )$   
wird zu:
    - $1( \ 1(0|1)^+ \mid 0(0|1)(0|1)^+ )$
- Nun versuchen Sie die Klasse der Bezeichner zu beschreiben: 现在试着描述一下标识符的类别。



Autor:

# Bezeichner als reguläre Ausdrücke 作为正则表达式的 标识符

- Auf Englisch für Java/C:
  - An **identifier** is a sequence of letters and digits; the first character must be a letter. The underscore \_ counts as a letter. Upper- and lowercase letters are different.

[a-zA-Z\_][a-zA-Z\_0-9]\*



## Compiler

## Lexikalische Analyse

Ausdruck	Entsprechungen	Beispiel
$c$	Das eine Nicht-Operatorzeichen $c$	$a$
$\backslash c$	Das Zeichen $c$ als Literal	$\backslash *$
$"s"$	Das Zeichen $s$ als Literal	$***$
.	Jedes Zeichen außer der Zeilenschaltung	$a.*b$
$^$	Zeilenanfang	$^abc$
$\$$	Zeilenende	$abc\$$
$[s]$	Ein beliebiges Zeichen im String $s$	$[abc]$
$[^s]$	Ein beliebiges Zeichen, das nicht im String $s$ vorkommt	$[^abc]$
$r^*$	0 oder mehr Strings, die $r$ entsprechen	$a^*$
$r^+$	Ein oder mehr Strings, die $r$ entsprechen	$a^+$
$r?$	0 oder 1 $r$	$a?$
$r\{m,n\}$	$m$ bis $n$ Vorkommen von $r$	$a\{1,5\}$
$r_1 r_2$	Ein $r_1$ gefolgt von einem $r_2$	$ab$
$r_1   r_2$	Ein $r_1$ oder ein $r_2$	$a b$
$(r)$	Dasselbe wie $r$	$(a b)$
$r_1 / r_2$	$r_1$ , wenn $r_2$ darauf folgt	$abc/123$



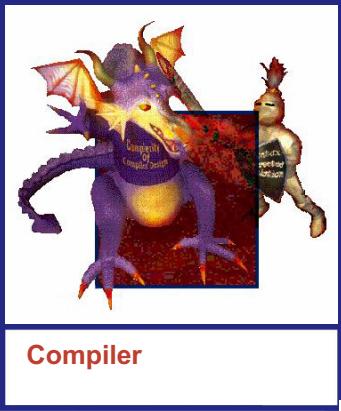
Compiler

Autor:

Folie: 37

# Reguläre Definitionen

- $d_1 \rightarrow r_1$
  - $d_2 \rightarrow r_2$
  - ...
  - $d_n \rightarrow r_n$
- 
- $d_i$  neues Symbol nicht in  $\Sigma$
  - $r_i$  regulärer Ausdruck über  $\Sigma \cup \{d_1, \dots, d_{i-1}\}$



# Reguläre Definitionen

Compiler

Autor:

Folie: 38

```
digit    → [0-9]
digits   → digit+
number   → digits ( . digits)? ( E [+-]? digits )?
letter   → [A-Za-z]
id       → letter ( letter | digit )*
if       → if
then     → then
else     → else
rellop   → < | > | <- | >- | = | <>
```

Abbildung 3.11: Muster für die Token in Beispiel 3.8



Compiler

# Ziel des Lexers:

digit → [0-9]  
digits → digit+  
number → digits (. digits)? ( E [+-]? digits )?  
letter → [A-Za-z]  
id → letter ( letter | digit )\*  
if → if  
then → then  
else → else  
relOp → < | > | <= | >= | - | ◊

Lexikalische Analyse

Ziel (an Parser weitergeben):

Lexeme	Tokenname	Attributwert
Jedes ws	-	-
if	if	-
then	then	-
else	else	-
Jedes id	id	Zeiger auf Tabelleneintrag
Jedes number	number	Zeiger auf Tabelleneintrag
<	relOp	LT
<=	relOp	LE
-	relOp	EQ
◊	relOp	NE
>	relOp	GT
>=	relOp	GE



Compiler

Lexikalische Analyse

# Überblick: Lexing

- ✓ **Reguläre Ausdrücke** (für die verschiedenen Tokenklassen)



- **Nicht-deterministische** endliche Automaten (NFA)
  - **Deterministische** endliche Automaten (DFA)
    - Linearer Quellcode in **C, Java, ...** (für das Lexing)

**Lex,  
javacc  
sablecc**



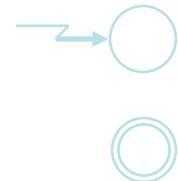


Compiler

Lexikalische Analyse

# Nicht-deterministische endliche Automaten: Bausteine

- Eine endliche Menge von Eingabesymbolen  $\Sigma$
- Eine endliche Menge von **Zuständen**  $S$
- Eine Menge von **Kanten**, gekennzeichnet mit  $\Sigma \cup \{\epsilon\}$
- Einen **Startzustand**  $s_0 \in S$
- Eine Menge von **Endzuständen**  $F \subseteq S$





Compiler

Autor:

Folie: 42

# Darstellungen

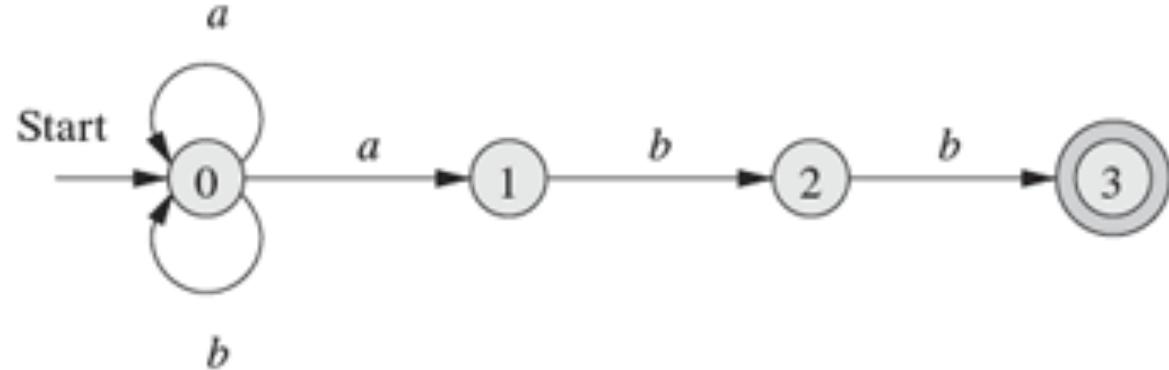


Abbildung 3.24: Ein nichtdeterministischer endlicher Automat

Zustand	a	b	$\epsilon$
0	{0, 1}	{0}	$\emptyset$
1	$\emptyset$	{2}	$\emptyset$
2	$\emptyset$	{3}	$\emptyset$
3	$\emptyset$	$\emptyset$	$\emptyset$

Abbildung 3.25: Übergangstabelle für den NFA in Abbildung 3.24

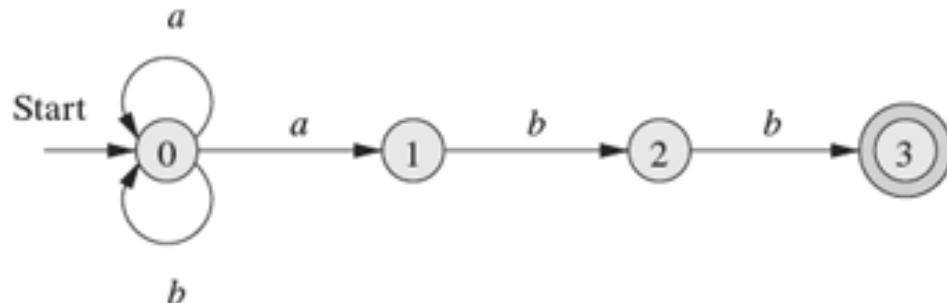


Compiler

Lexikalische Analyse

# Nicht-deterministische endliche Automaten: Sprache

- Ein NFA akzeptiert einen Eingabestring  $x$  dann und nur dann, wenn
  - es (mindestens) einen Pfad von dem Startzustand zu einem Endzustand **gibt** bei dem die Symbole entlang des Pfades  $x$  ergeben
  - Anmerkung:  $\epsilon$  Symbole tragen nicht zu dem akzeptierten String bei.





Compiler

Autor:

Folie: 44

# Weiteres Beispiel

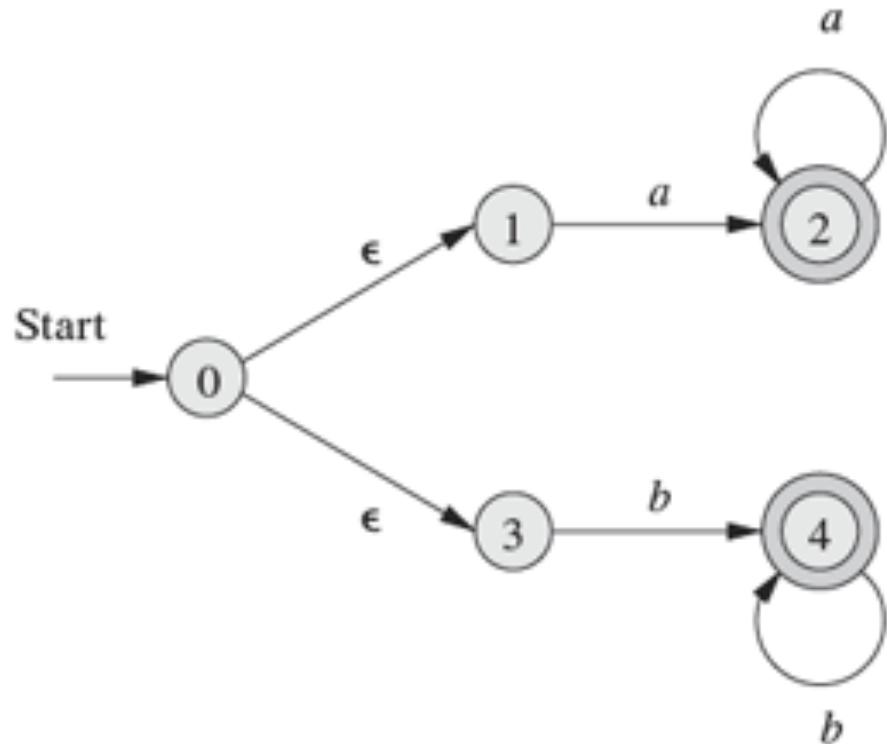


Abbildung 3.26: Ein NFA, der  $aa^*|bb^*$  akzeptiert



Compiler

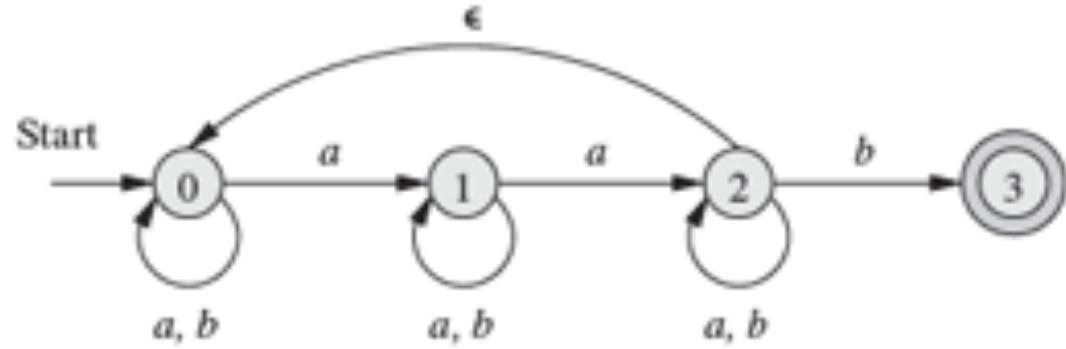


Abbildung 3.29: NFA zu Übung 3.6.3

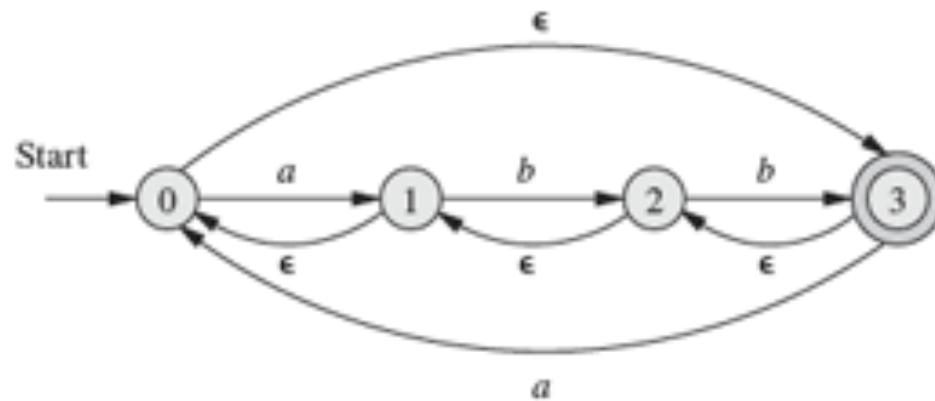


Abbildung 3.30: NFA zu Übung 3.6.4

Wieviele Pfade mit aabb gibt es?  
Akzeptiert der Automat aabb?



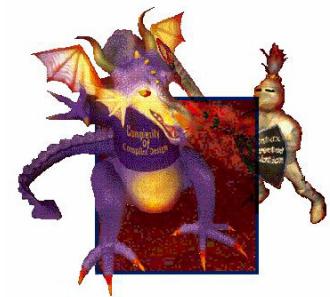
Compiler

Autor:

Folie: 46

# DFA

- Ein deterministischer endlicher Automat ist ein Sonderfall eines NFA:
  - Es gibt keine Kanten die mit  $\epsilon$  gekennzeichnet sind
  - Für jeden Zustand und jedes Symbol a:
    - genau\*\* eine Kante aus s die mit a gekennzeichnet ist
- Alternativ-Definition: maximal\*\*



# DFA Beispiel

Compiler

Lexikalische Analyse

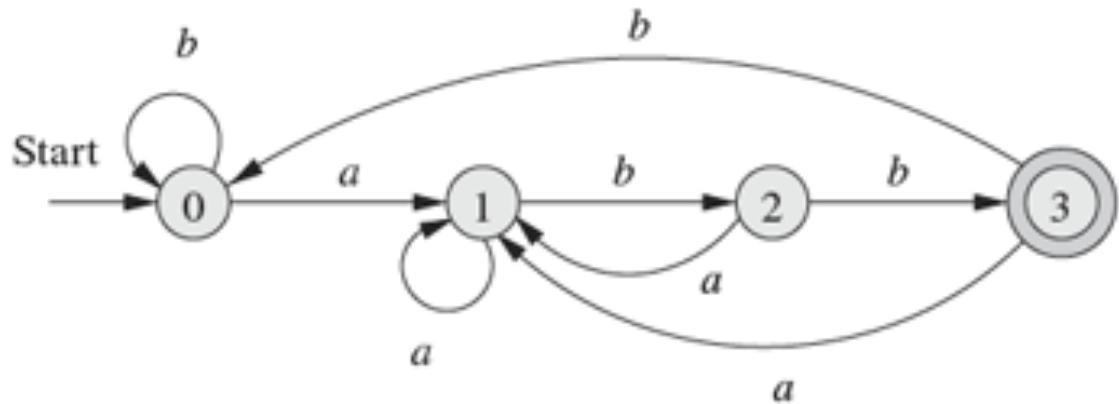


Abbildung 3.28: DFA, der die Sprache  $(a|b)^*abb$  akzeptiert



# DFA Implementation

Compiler

Lexikalische Analyse

```
s =  $s_0$ ;  
c = nextChar();  
while ( c != eof ) {  
    s = move(s,c);  
    c = nextChar();  
}  
if ( s is in F ) return "yes";  
else return "no";
```

Abbildung 3.27: Simulation eines DFA



# Überblick: Lexing

Compiler

Lexikalische Analyse

- **Reguläre Ausdrücke** (für die verschiedenen Tokenklassen)
  - ✓
  - ↓↓
- **Nicht-deterministische** endliche Automaten (NFA)
  - ↓
- **Deterministische** endliche Automaten (DFA)
  - ↓
- Linearer Quellcode in **C, Java, ...** (für das Lexing)

**Lex,  
javacc  
sablecc**



Compiler

Autor:

# R.E. → NFA (1)

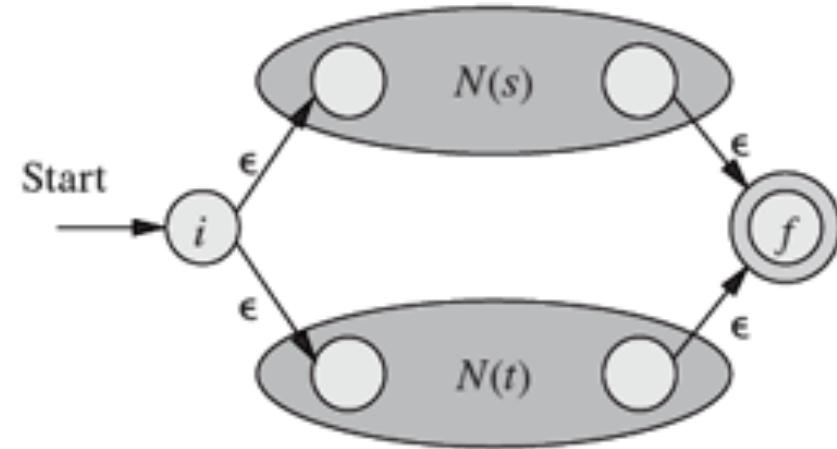
- Konstruktion aus dem Drachenbuch
  - Verschieden von [Appel]
  - $N(r) = \text{NFA der } L(r) \text{ akzeptiert}$
- **a**, wo  $a \in \Sigma$ 
  - $N(a) =$
- **$\epsilon$** 
  - $N(\epsilon) =$



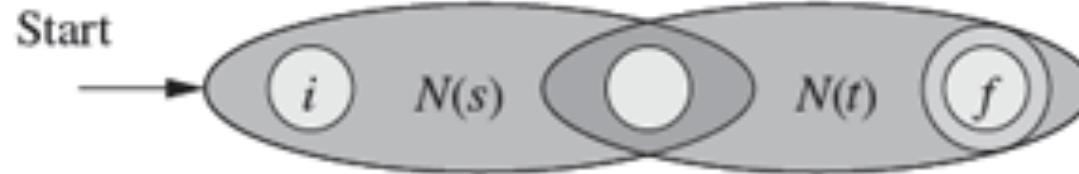
Compiler

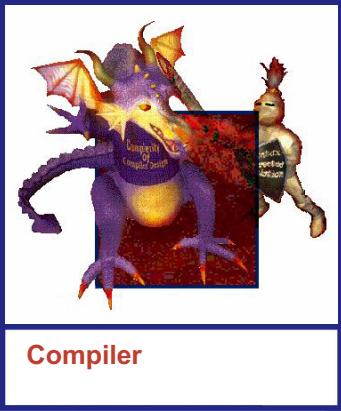
Autor:

- $s|t$ 
  - $N(s|t) =$



- $st$ 
  - $N(st) =$

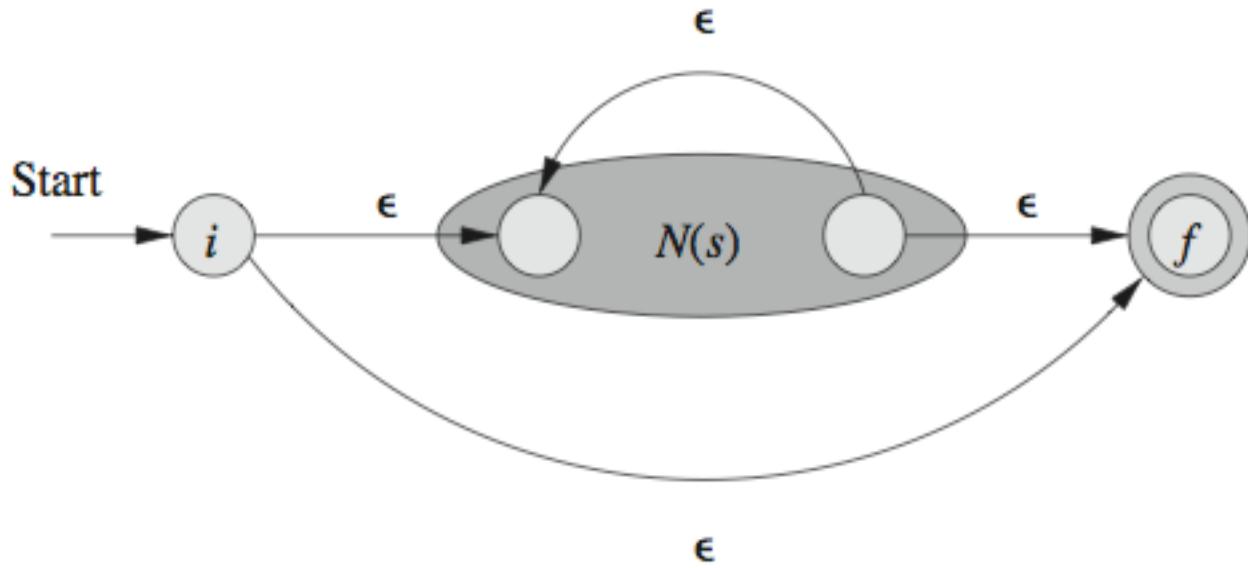




Autor:

# R.E. → NFA (3)

- $s^*$ 
  - $N(s^*) =$

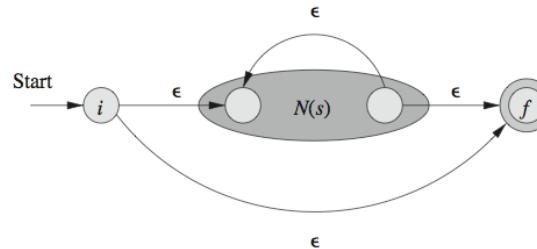




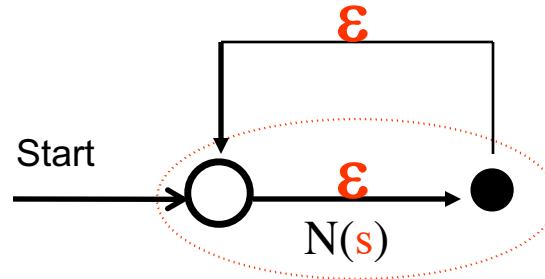
Compiler

Autor:

# Frage: Warum nicht:



- $s^*$ 
  - $N(s^*) =$



Viel einfacher (2  $\epsilon$  statt 4, weniger Zustände)

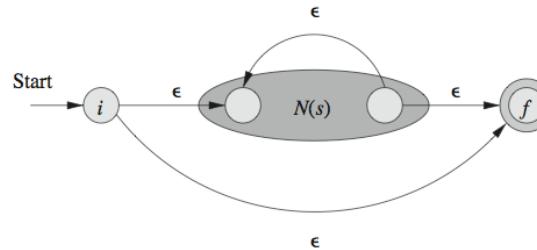
Was ist das Problem?



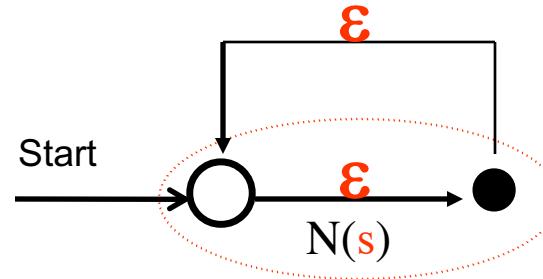
Compiler

Autor:

# Frage: Warum nicht:

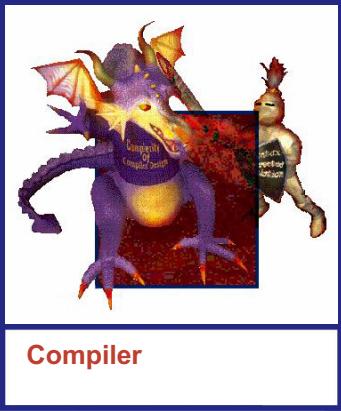


- $s^*$ 
  - $N(s^*) =$



Viel einfacher (2  $\epsilon$  statt 4, weniger Zustände)

Was ist das Problem? Falsch bei Konkatenation:  
Z.B. Automat für  $a^*b^*$  würde auch **aba** akzeptieren!



Compiler

Autor:

Folie: 55

# Beispiel: $(a|b)^*abb$

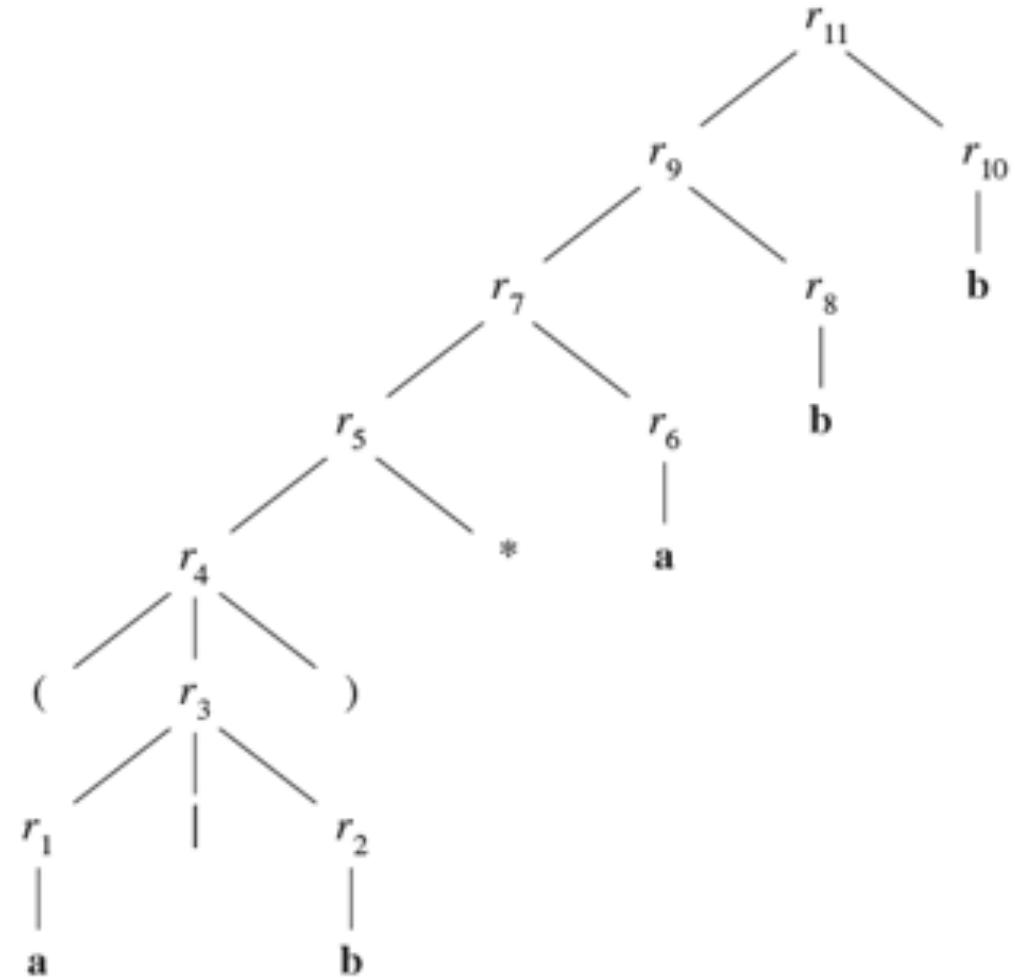
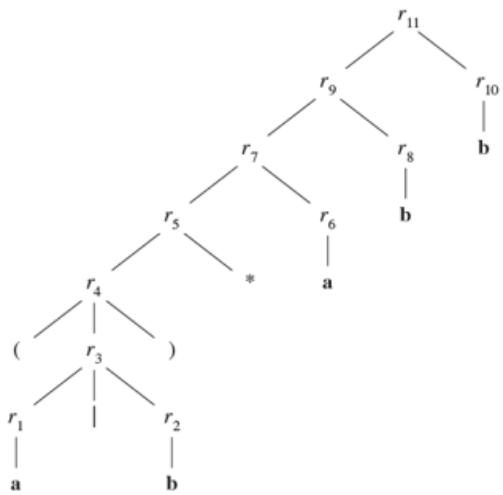


Abbildung 3.43: Parse-Baum für  $(a|b)^*abb$

Die Erstellung von Parsebäumen wird später noch besprochen.



Compiler



Lexikalische Analyse

## Tiefensuche auf Parsebaum:

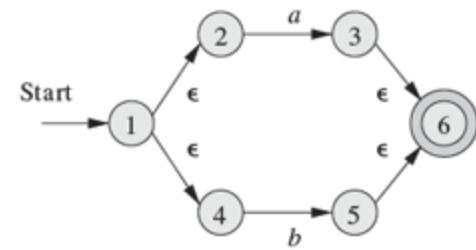


Abbildung 3.44: NFA für  $r_3$

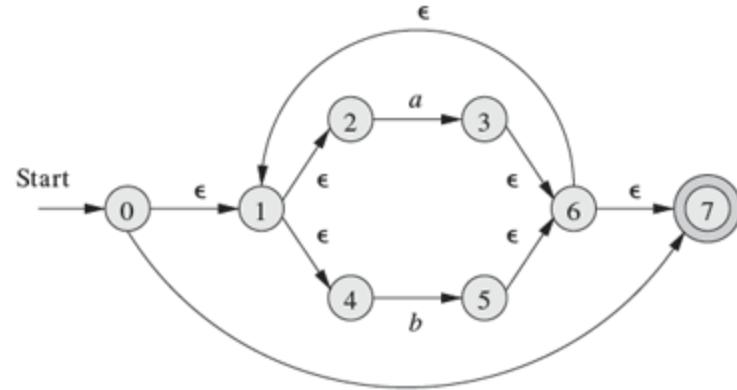


Abbildung 3.45: NFA für  $r_6$

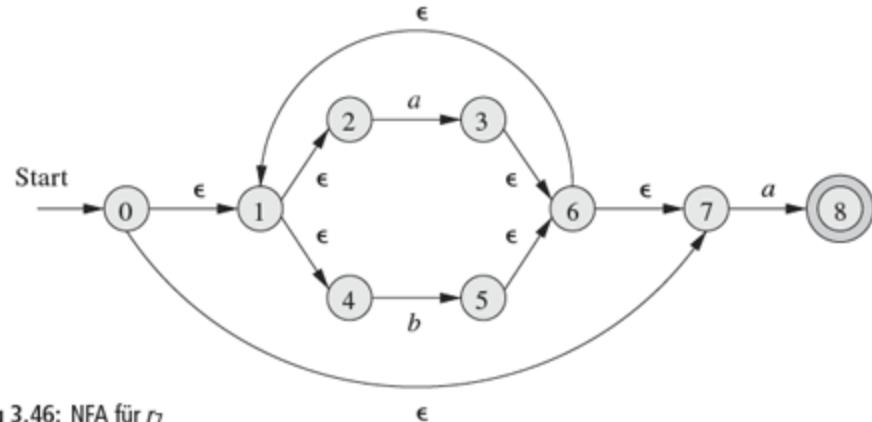
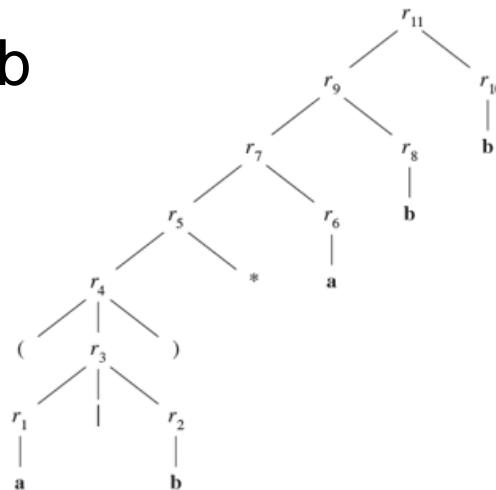


Abbildung 3.46: NFA für  $r_7$

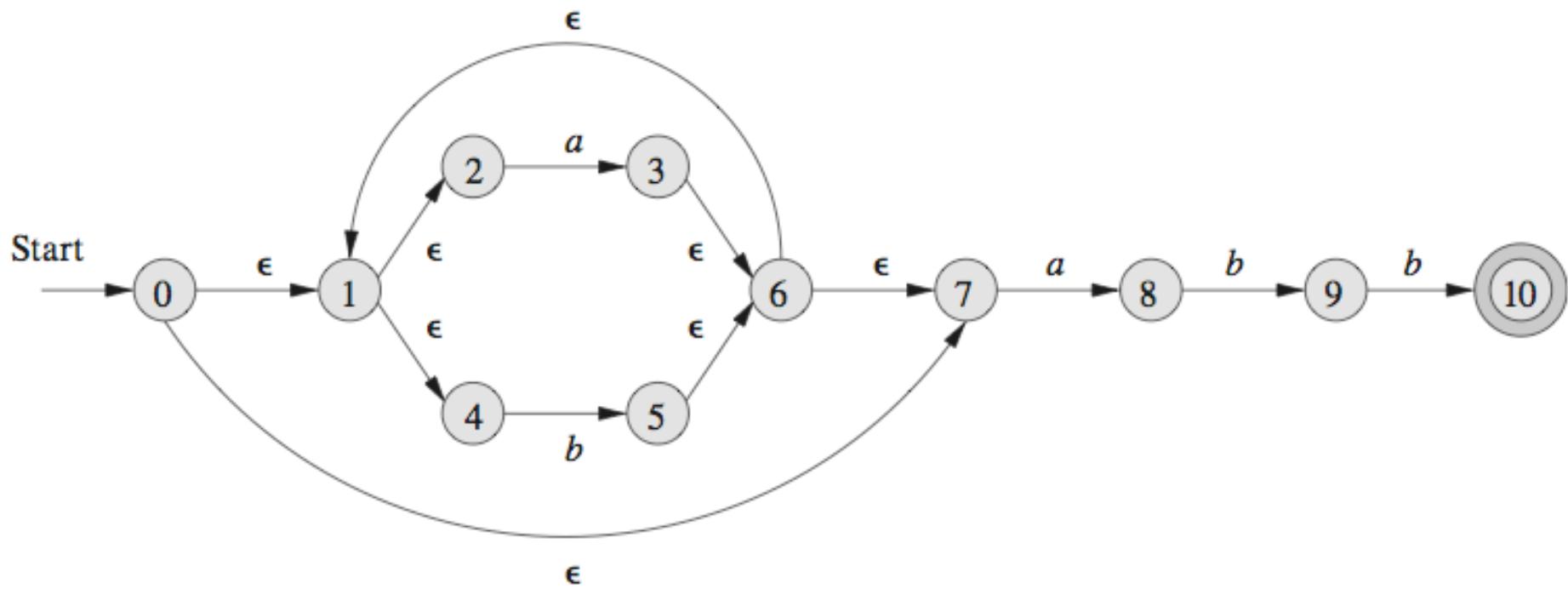


$(a|b)^*abb$



Compiler

Lexikalische Analyse





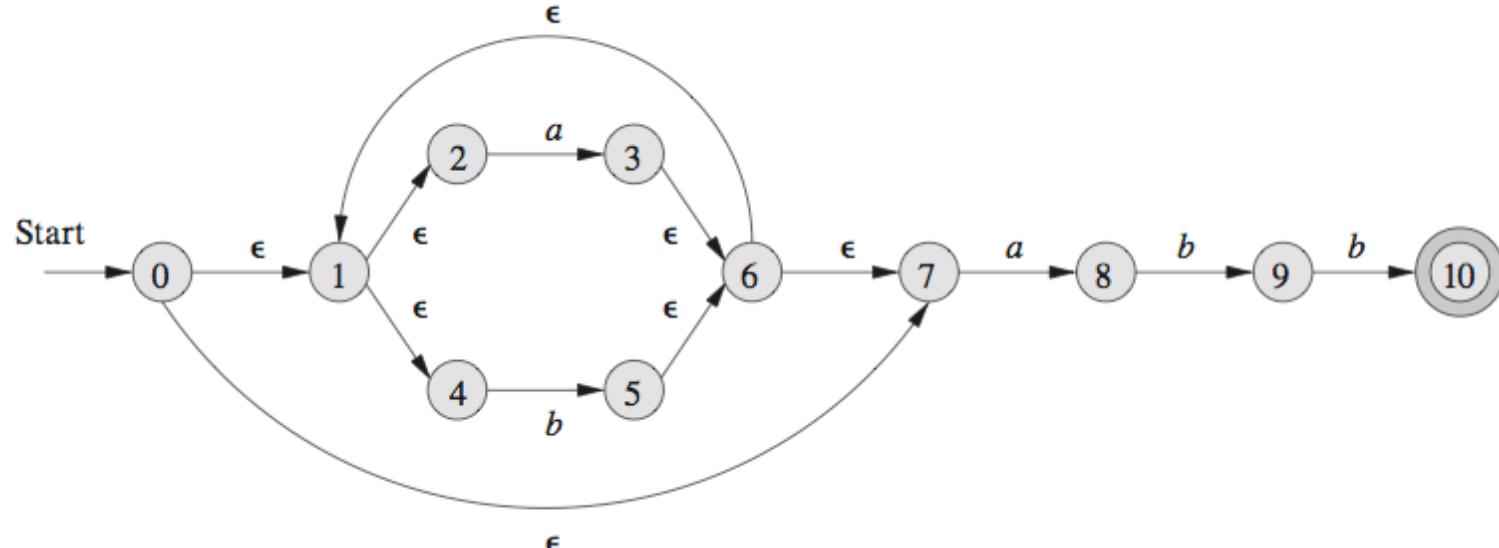
Compiler

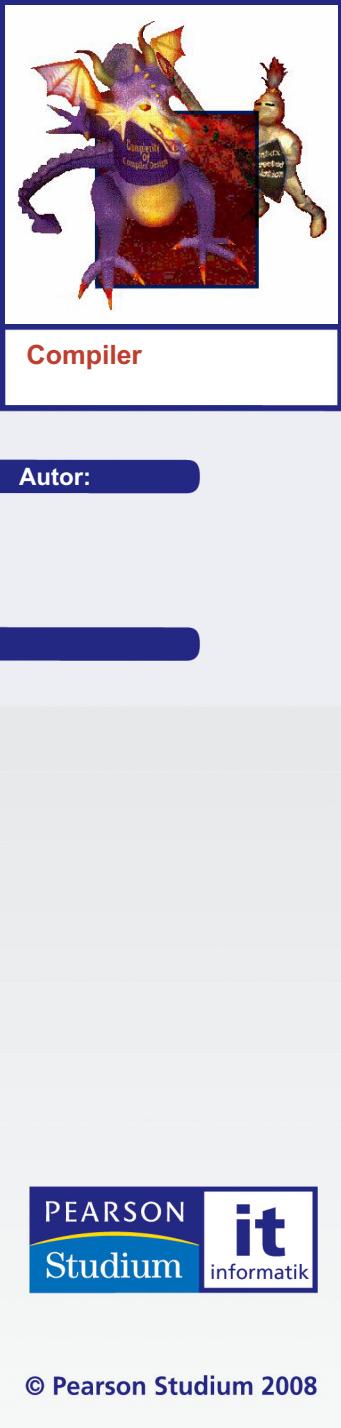
Autor:

Folie: 58

# Effiziente Implementation von NFAs ?NFA的高效实施？

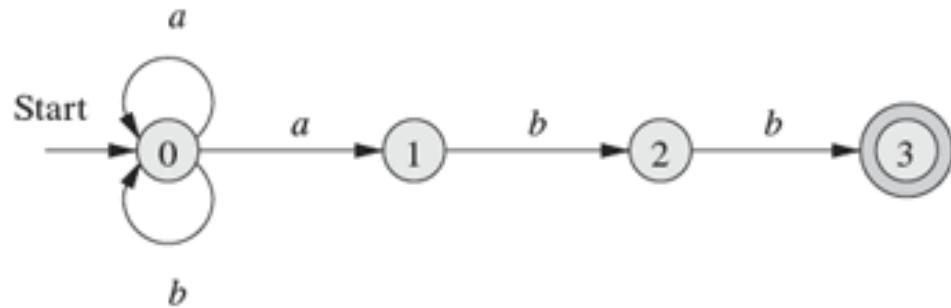
- Immer die richtige Transition wählen ;-) 始终选择正确的过渡
- NFA in DFA konvertieren !





# NFA → DFA: Grundidee

- Zustand im DFA  $\approx$  Menge von Zuständen im NFA



$\{0\}$      $-a \rightarrow$      $\{0,1\}$



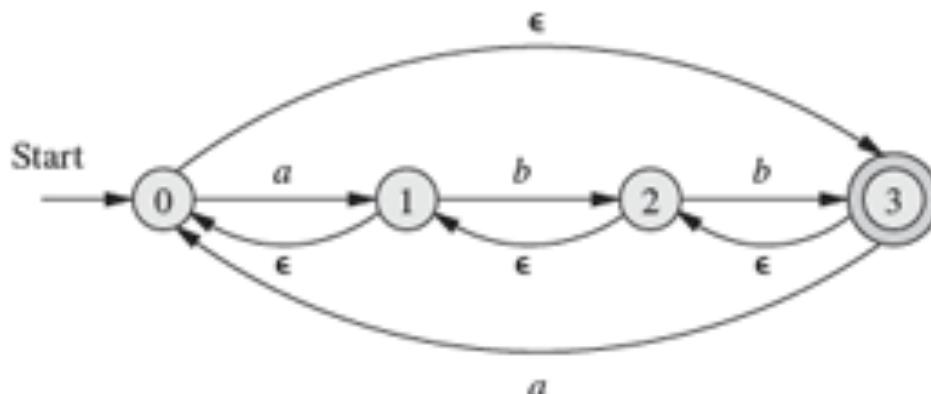
# DFA → NFA: Hilfsfunktionen

Compiler

## Lexikalische Analyse

Operation	Beschreibung
$\epsilon\text{-closure}(s)$	Menge von NFA-Zuständen, die von einem NFA-Zustand $s$ allein durch $\epsilon$ -Übergänge erreicht werden können
$\epsilon\text{-closure}(T)$	Menge von NFA-Zuständen die von einem NFA-Zustand $s$ in Menge $T$ allein durch $\epsilon$ -Übergänge erreicht werden können; $\cup_{s \in T} \epsilon\text{-closure}(s)$ .
$\text{move}(T, a)$	Menge von NFA-Zuständen, zu denen es einen Übergang über das Eingabesymbol $a$ von einem Zustand $s$ in $T$ aus gibt.

Abbildung 3.31: Operationen für NFS-Zustände



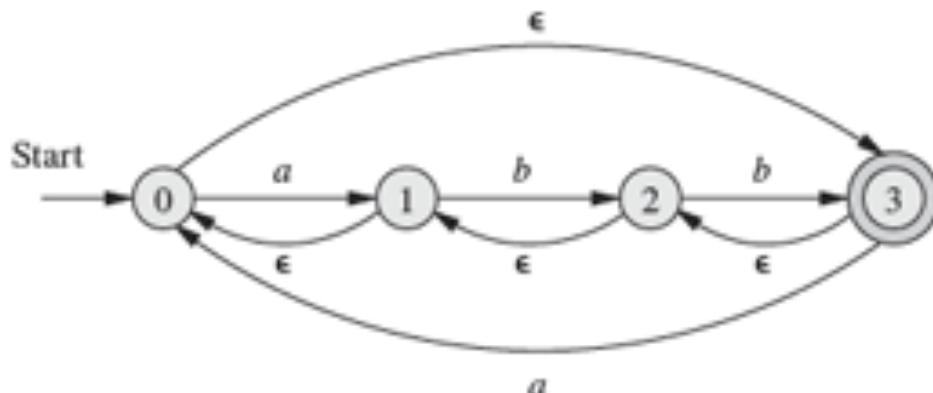


# DFA → NFA: $\epsilon$ -closure

Compiler

Lexikalische Analyse

```
schiebe alle Zustände von  $T$  auf den stack;  
initialisiere  $\epsilon$ -closure( $T$ ) mit  $T$ ;  
while ( stack ist nicht leer ) {  
    entferne  $t$ , das oberste Element, vom stack;  
    for ( jeder Zustand  $u$  mit einer Kante von  $t$  zu  $u$  mit Bezeichnung  $\epsilon$  )  
        if (  $u$  ist nicht in  $\epsilon$ -closure( $T$ ) ) {  
            füge  $u$  zu  $\epsilon$ -closure( $T$ ) hinzu;  
            schiebe  $u$  auf stack;  
        }  
    }  
}
```





# Beispiel: $\epsilon$ -closure

Was ist  $\epsilon$ -closure(0) ?

Compiler

Lexikalische Analyse

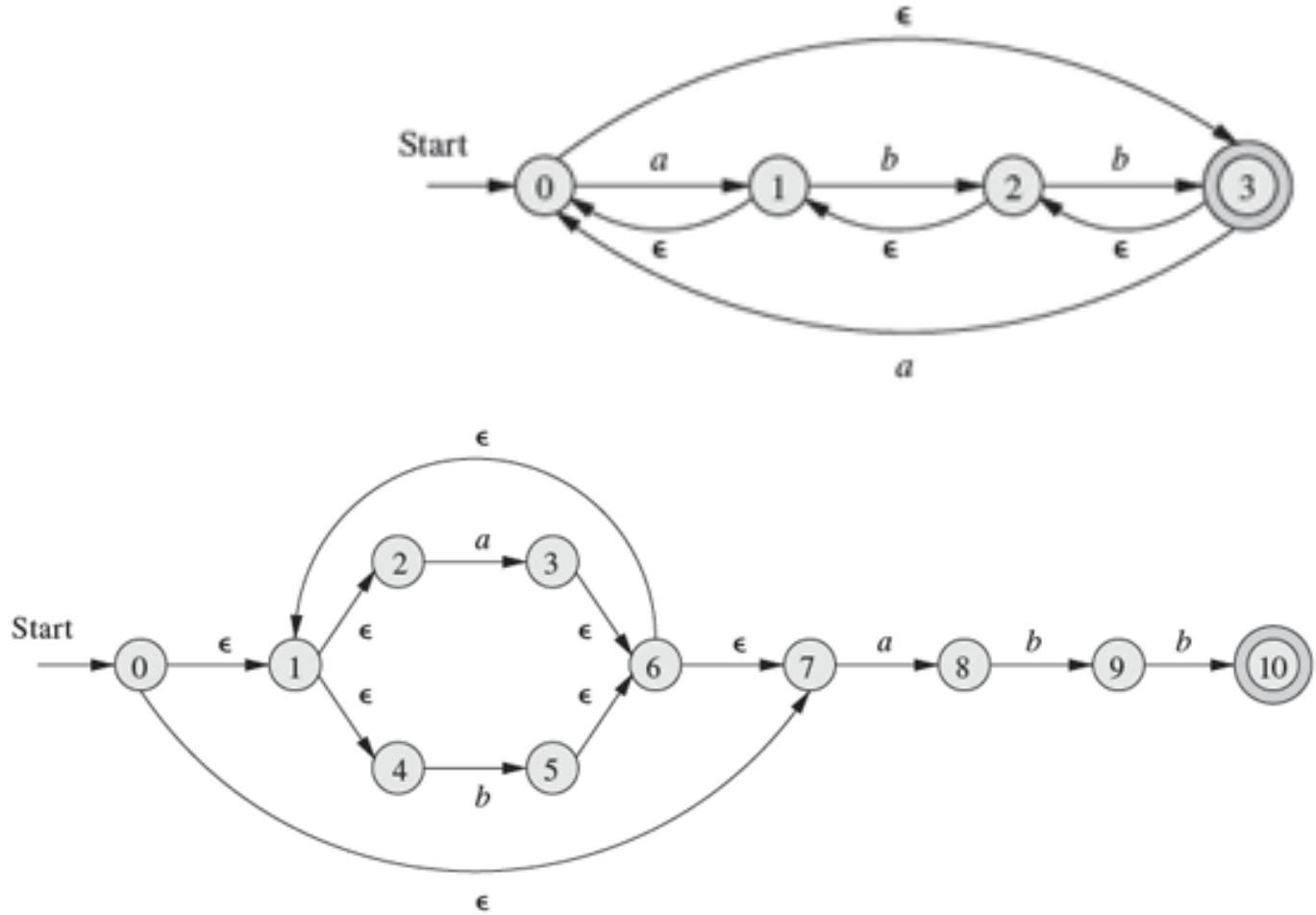


Abbildung 3.34: NFA  $N$  für  $(a|b)^*abb$

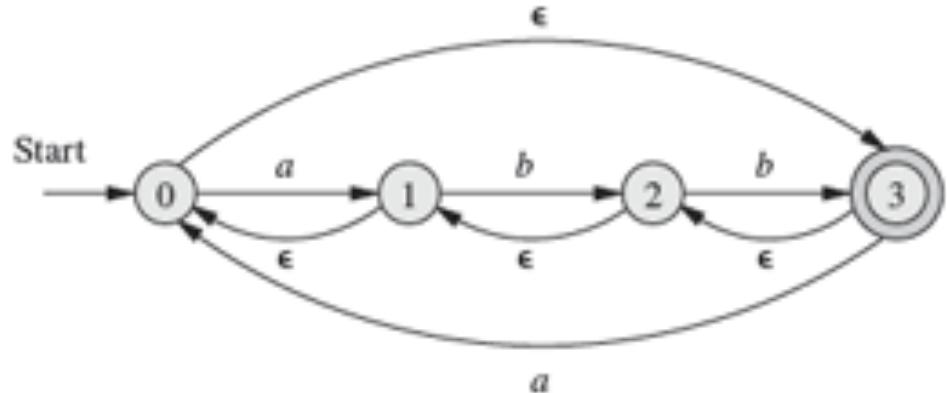


Compiler

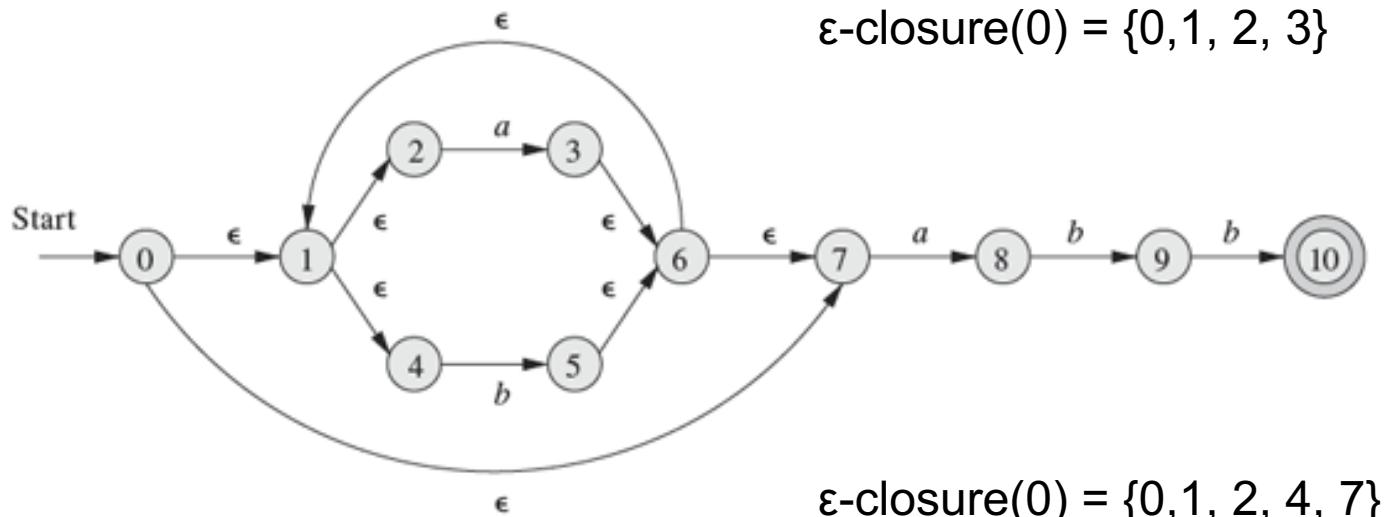
Lexikalische Analyse

# Beispiel: $\epsilon$ -closure

Was ist  $\epsilon$ -closure(0) ?



$$\epsilon\text{-closure}(0) = \{0, 1, 2, 3\}$$



$$\epsilon\text{-closure}(0) = \{0, 1, 2, 4, 7\}$$

Abbildung 3.34: NFA  $N$  für  $(a|b)^*$  abb



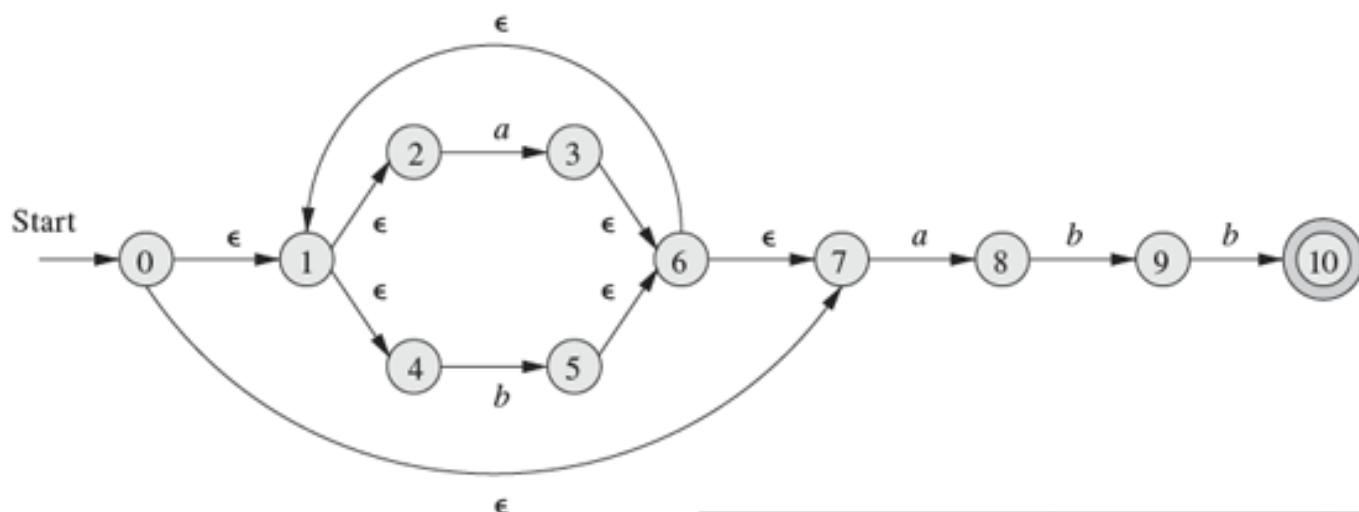
## Compiler

ursprünglich ist  $\epsilon$ -closure( $s_0$ ) der einzige Zustand in  $Dstates$ , und er ist unmarkiert;  
**while** ( es gibt einen unmarkierten Zustand  $T$  in  $Dstates$  ) {  
    markiere  $T$ ;  
    **for** ( jedes Eingabesymbol  $a$  ) {  
         $U = \epsilon$ -closure( $move(T, a)$ );  
        **if** (  $U$  ist nicht in  $Dstates$  )  
            füge  $U$  als unmarkierten Zustand in  $Dstates$  hinzu;  
         $Dtran[T, a] = U$ ;  
    }  
}

Abbildung 3.32: Die Teilmengenbildung



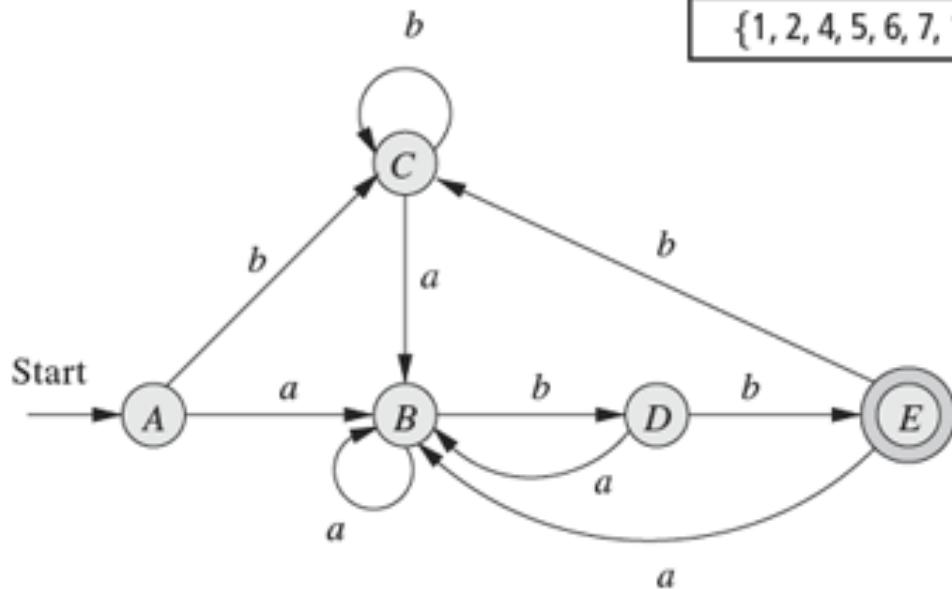
Compiler



## Lexikalische Analyse

Abbildung 3.34: NFA  $N$  für  $(a|b)^*abb$

NFA-Zustand	DFA-Zustand	a	b
$\{0, 1, 2, 4, 7\}$	A	B	C
$\{1, 2, 3, 4, 6, 7, 8\}$	B	B	D
$\{1, 2, 4, 5, 6, 7\}$	C	B	C
$\{1, 2, 4, 5, 6, 7, 9\}$	D	B	E
$\{1, 2, 4, 5, 6, 7, 10\}$	E	B	C





# Minimierung: Algorithmus

Compiler

## METHODE:

### Lexikalische Analyse

1. Beginnen Sie mit einer Anfangspartition  $\Pi$  mit zwei Gruppen,  $F$  und  $S - F$ , den akzeptierenden und nicht akzeptierenden Zuständen von  $D$ .

```
ursprünglich sei  $\Pi_{new} = \Pi$ ;
for ( jede Gruppe  $G$  von  $\Pi$  ) {
    partitioniere  $G$  in Untergruppen, sodass zwei Zustände  $s$  und  $t$ 
    dann und nur dann in derselben Untergruppe sind, wenn für alle
    Eingabesymbole  $a$  die Zustände  $s$  und  $t$  für  $a$  Übergänge
    zu Zuständen in derselben Gruppe von  $\Pi$  haben;
    /* Schlimmstenfalls befindet sich ein Zustand alleine in einer Untergruppe */
    ersetze  $G$  in  $\Pi_{new}$  durch die Menge aller gebildeten Untergruppen;
}
```

Abbildung 3.64: Aufbau von  $\Pi_{new}$

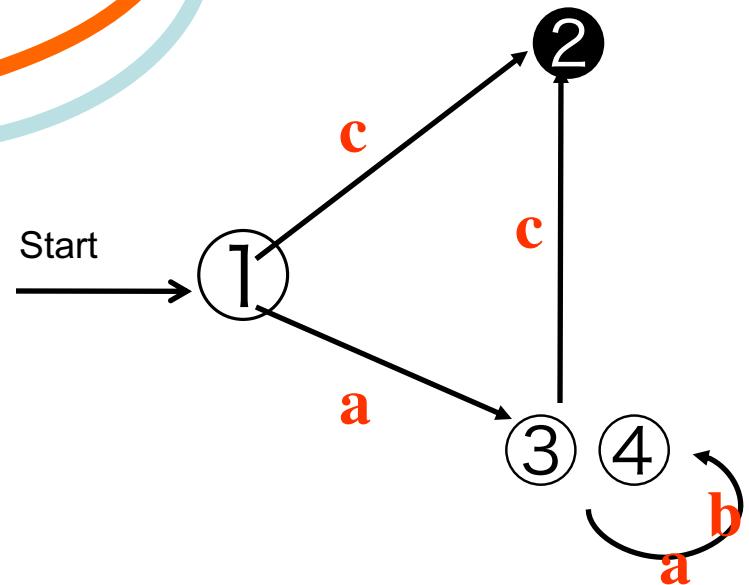
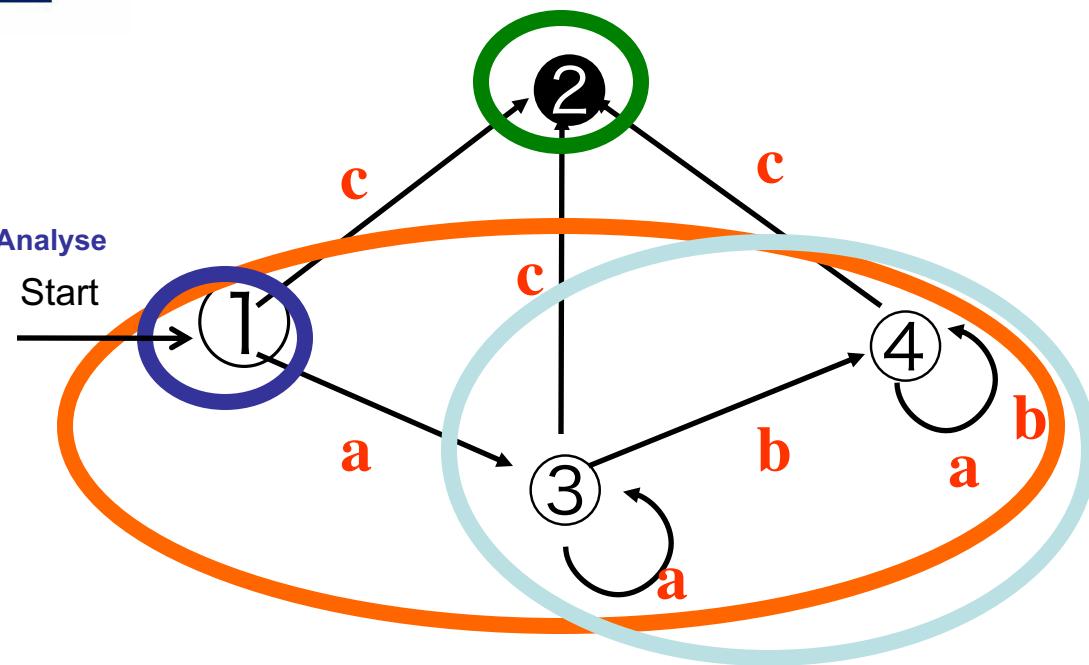
Wiederholen bis  $\Pi_{new} = \Pi$



# Minimierung: ein einfaches Beispiel

Compiler

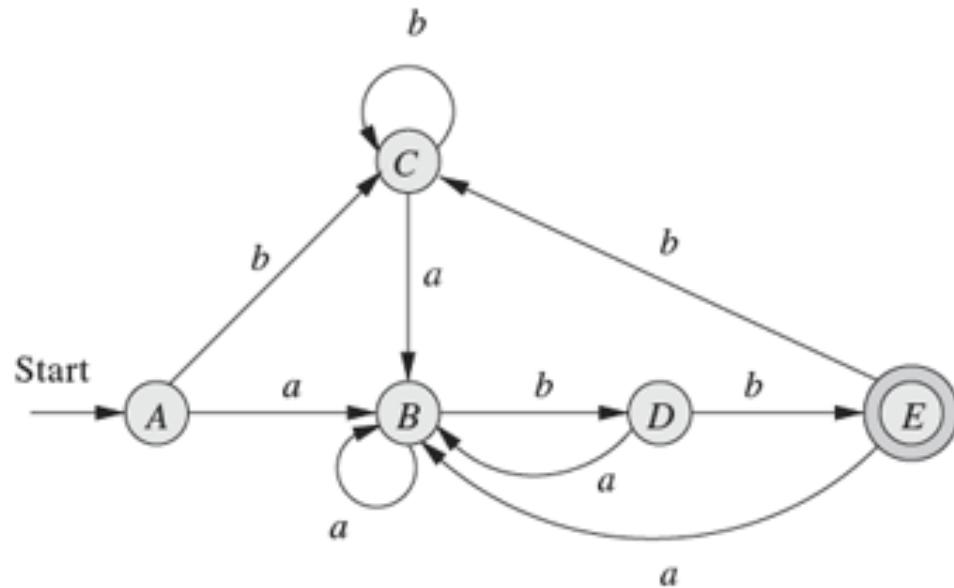
Lexikalische Analyse





Compiler

Lexikalische Analyse



Anfangspartition:  
 $\{A, B, C, D\}$   $\{E\}$

Zustand	a	b
A	B	A
B	B	D
D	B	E
E	B	A

Abbildung 3.65: Übergangstabelle des Minimalzustands-DFA



Compiler

Autor:

Folie: 69

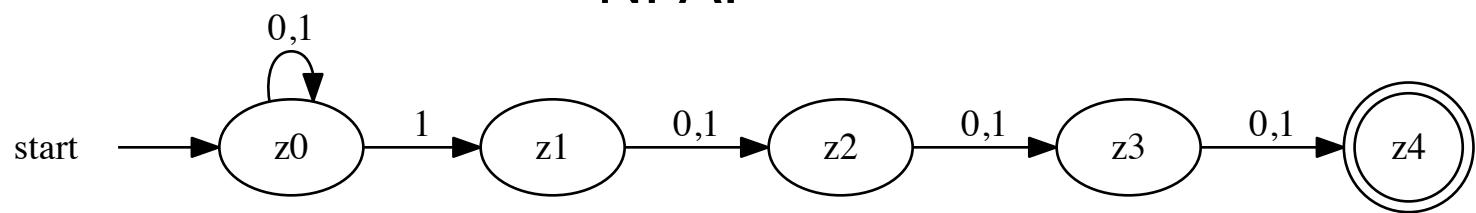
# Komplexität 复杂性

- Anzahl Zustände DFA
  - schlimmster Fall:  $2^n$  (NFA: n Zustände)
  - Praxis  $\approx n$
- Laufzeit zur Erkennung eines Tokens:
  - linear: jedes Symbol vom Input maximal einmal überprüft
  - Siehe aber später (maximal munching)



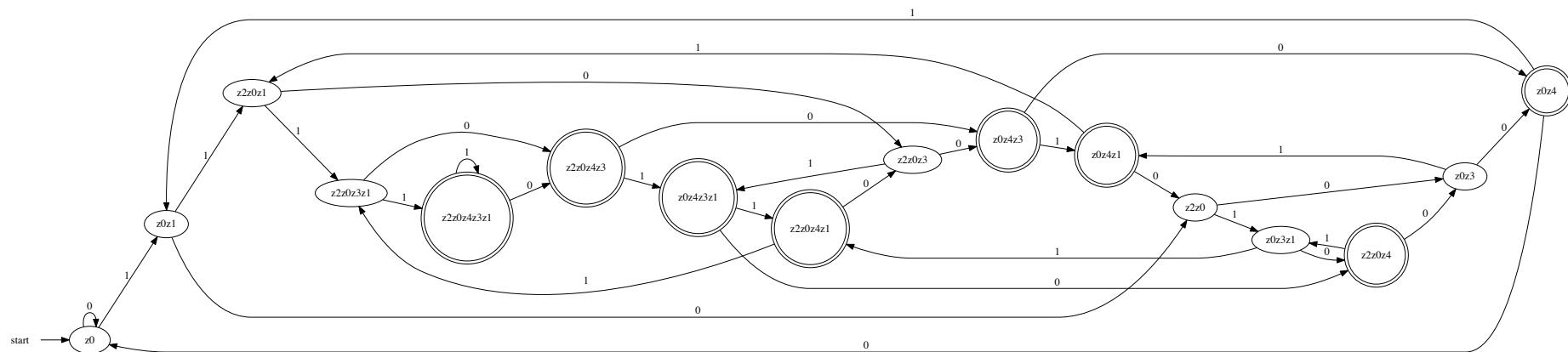
Compiler

NFA:



Lexikalische Analyse

DFA:

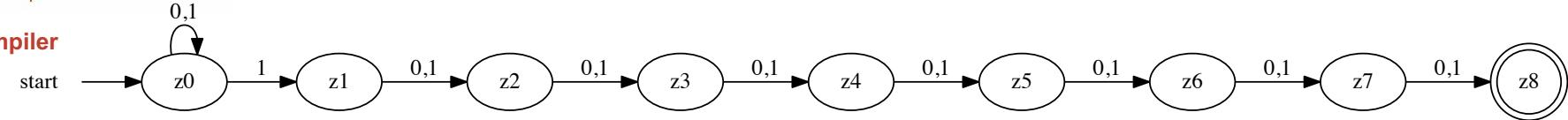


Autor:  
Aho et al.



# NFA:

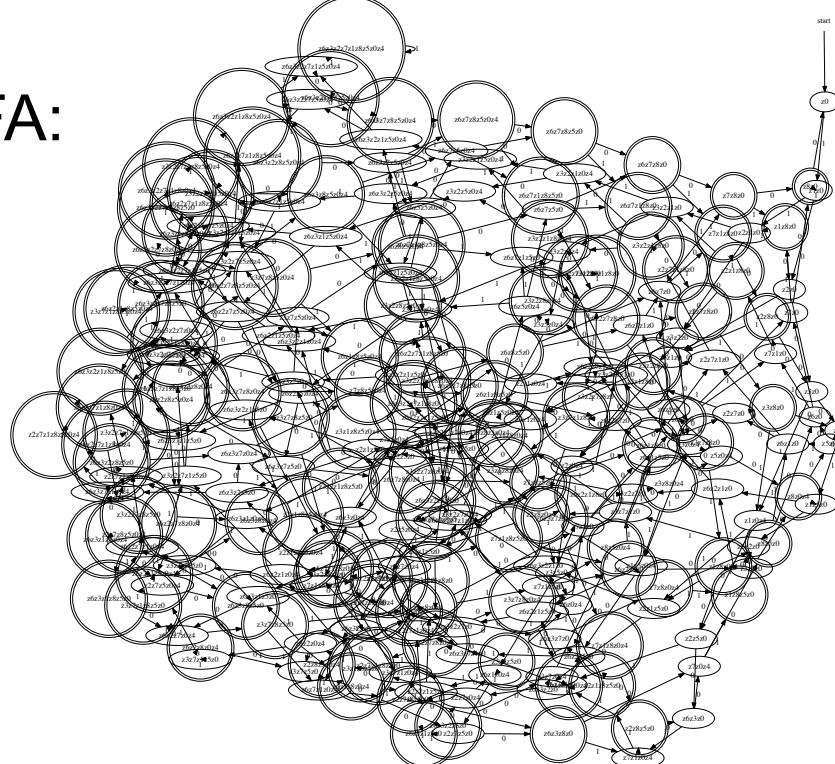
Compiler



Lexikalische Analyse

# DFA:

Solche  
Fälle  
tauchen  
in der  
Compilerbaupraxis  
nicht auf





## Compiler

## Lexikalische Analyse

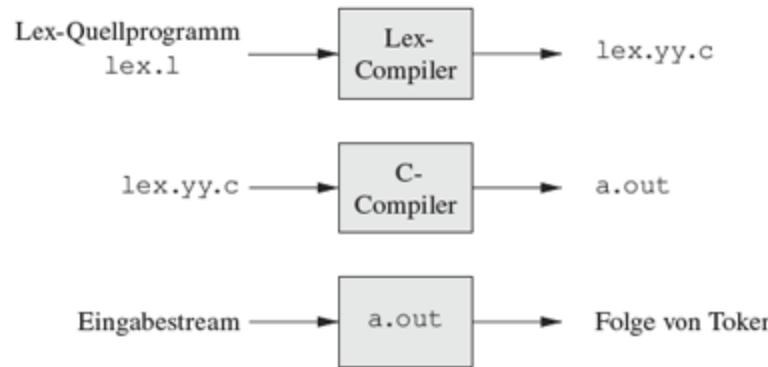


Abbildung 3.22: Erstellen eines Lexers mit Lex

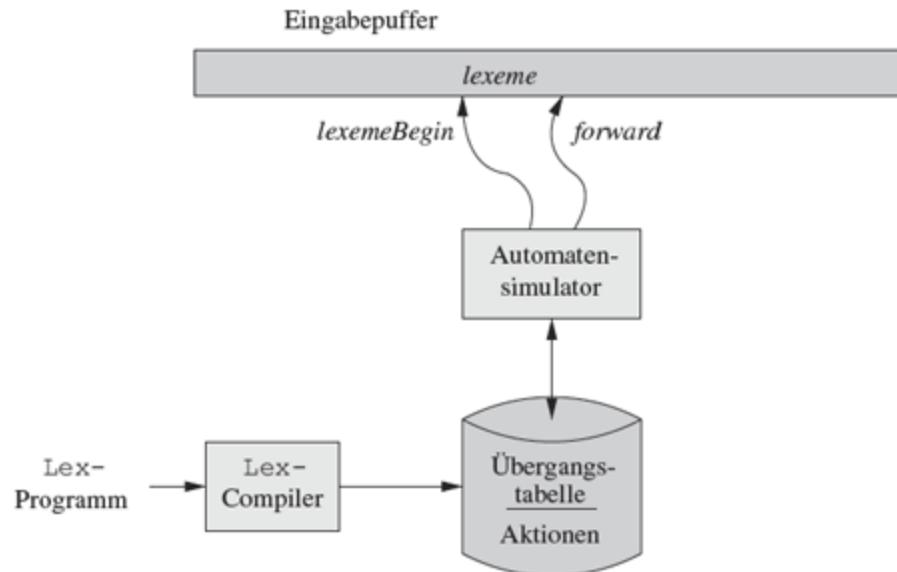
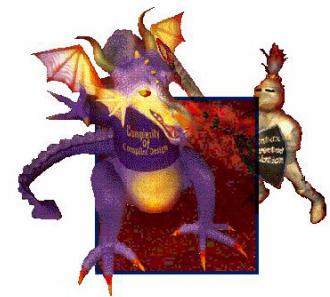


Abbildung 3.49: Ein Lex-Programm wird in eine Übergangstabelle und Aktionen umgewandelt, die von einem Simulator für endliche Automaten verwendet werden



# Kombination mehrerer R.E.

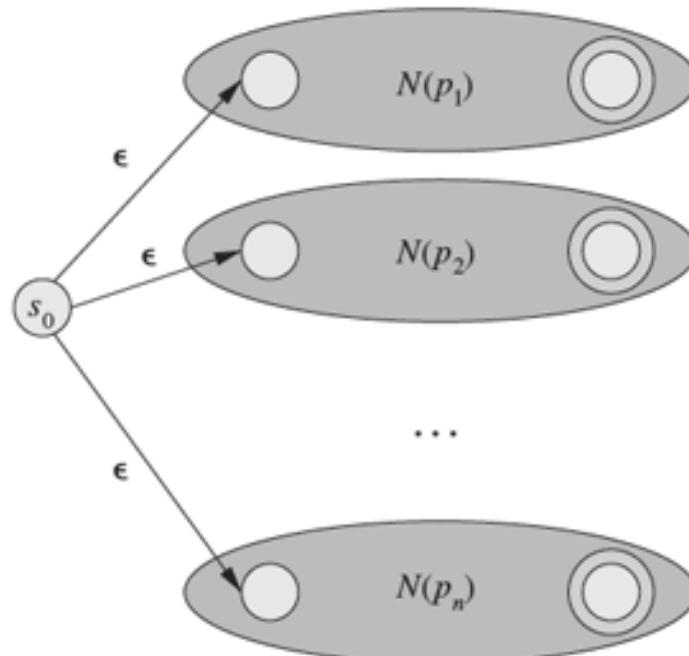
Compiler

a { Aktion  $A_1$  für Muster  $p_1$  }

abb { Aktion  $A_2$  für Muster  $p_2$  }

$a^*b^+$  { Aktion  $A_3$  für Muster  $p_3$  }

Lexikalische Analyse





## Compiler

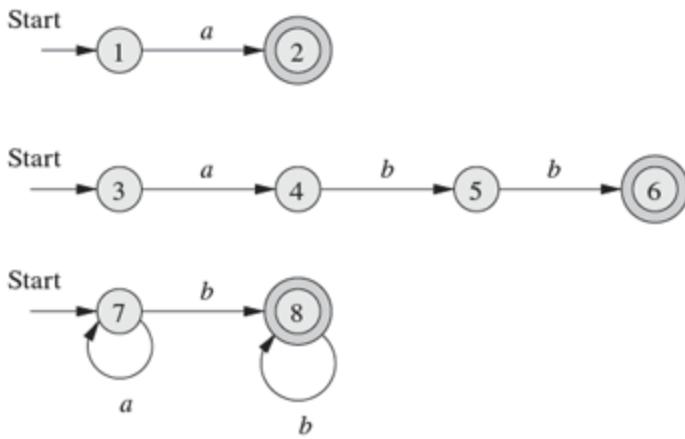


Abbildung 3.51: NFAs für  $a$ ,  $abb$  und  $a^*b^+$

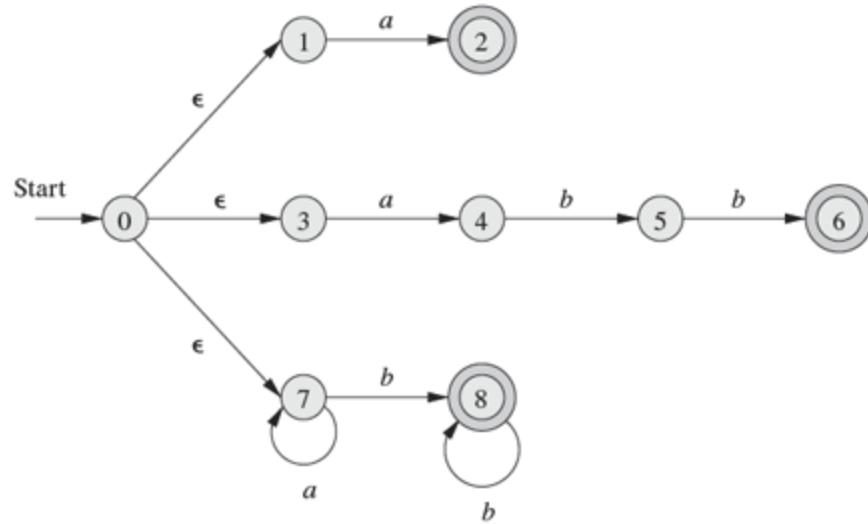


Abbildung 3.52: Kombinierter NFA

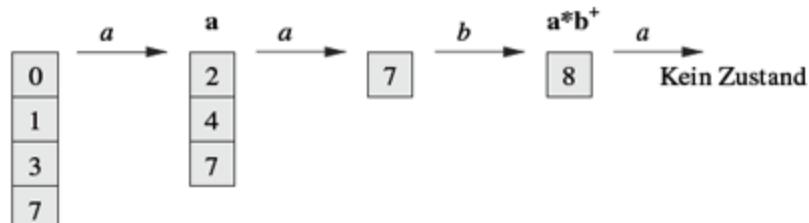


Abbildung 3.53: Sequenz der Mengen eingetreterener Zustände bei der Verarbeitung der Eingabe aaba



a { Aktion  $A_1$  für Muster  $p_1$  }  
abb { Aktion  $A_2$  für Muster  $p_2$  }  
 $a^*b^+$  { Aktion  $A_3$  für Muster  $p_3$  }

Compiler

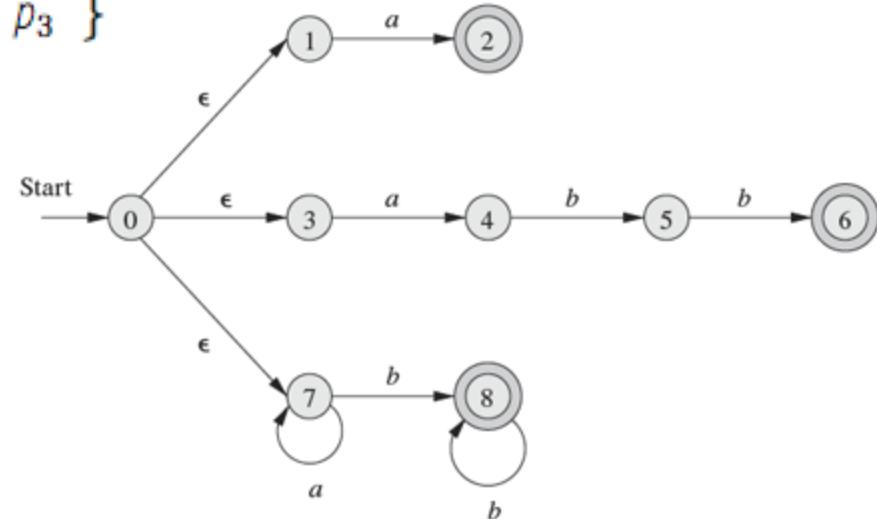


Abbildung 3.52: Kombinierter NFA

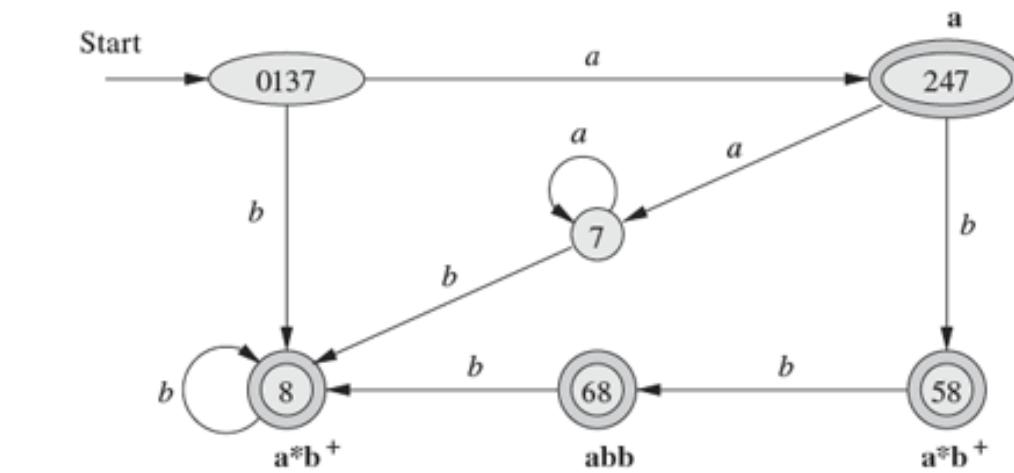


Abbildung 3.54: Übergangsgraph für den DFA, der die Muster a, abb und  $a^*b^+$  verarbeitet



Compiler

Autor:

Folie: 76

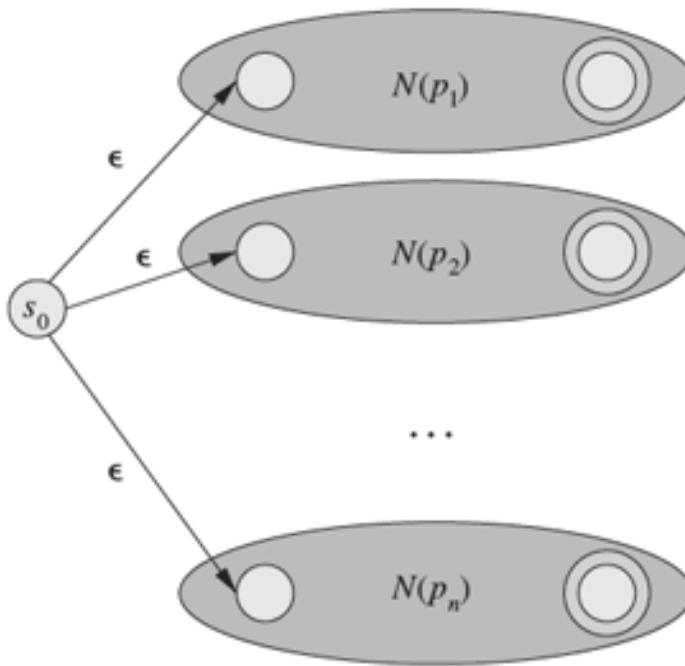
# Konfliktlösung解决冲突

- Ein längeres Präfix ist einem kürzeren vorzuziehen 长的前缀比短的前缀更合适 (Maximal Munching Rule)
- Passt das längste mögliche Präfix auf zwei oder mehr Muster, ist das zuerst im Lex-Programm aufgeführte zu wählen. 如果可能的最长序号符合两个或更多的模式，则应选择列在首位的lex方案。



Compiler

Autor:



# Implementierung von Maximal Munching?

1. Kennzeichnung der Endzustände mit erkannten Tokenklasse
2. Konvertierung nach DFA
3. Bei Besuch eines Endzustandes: merke InputPosition und Tokenklasse;
4. DFA zu Ende laufen lassen (bis Sackgasse) und dann **Backtracking** zur letzten gespeicherten Pos.+1



Compiler

Autor:

Folie: 78

# Maximal Munching: Beispiel

- Tokenklassen:
  - $a^*b$
  - a
- Beispieleingabe:
  - aabaaa



## Compiler

```
/*
 * Definition der manifesten Konstanten
 * LT, LE, EQ, NE, GT, GE,
 * IF, THEN, ELSE, ID, NUMBER, RELOP */
}
```

```
/* reguläre Definitionen */
delim [ \t\n]
ws {delim}+
letter [A-Za-z]
digit [0-9]
id {letter}{letter}|{digit})*
number {digit}+(\.{digit}+)?(E[+-]?{digit}+)?
```

```
%%
```

```
{ws} {/* keine Aktion und kein Rücksprung */}
if {return(IF);}
then {return(THEN);}
else {return(ELSE);}
{id} {yyval = (int) installID(); return(ID);}
{number} {yyval = (int) installNum(); return(NUMBER);}
"<" {yyval = LT; return(RELOP);}
"<=" {yyval = LE; return(RELOP);}
"=" {yyval = EQ; return(RELOP);}
">" {yyval = NE; return(RELOP);}
">" {yyval = GT; return(RELOP);}
">=" {yyval = GE; return(RELOP);}

%%
```

```
int installID() /* Funktion zur Installation der Lexeme, auf deren erstes Zeichen
                  yytext zeigt und deren Länge yyleng ist, in der Symbolebene
                  und Rückgabe eines Zeigers darauf */
}
int installNum() /* ähnlich wie installID, schreibt aber numerische Konstanten in
                  eine separate Tabelle */
}
```

Lexeme	Tokenname	Attributwert
Jedes ws	-	-
if	if	-
then	then	-
else	else	-
Jedes id	id	Zeiger auf Tabelleneintrag
Jedes number	number	Zeiger auf Tabelleneintrag
<	relop	LT
<=	relop	LE
=	relop	EQ
<>	relop	NE
>	relop	GT
>=	relop	GE

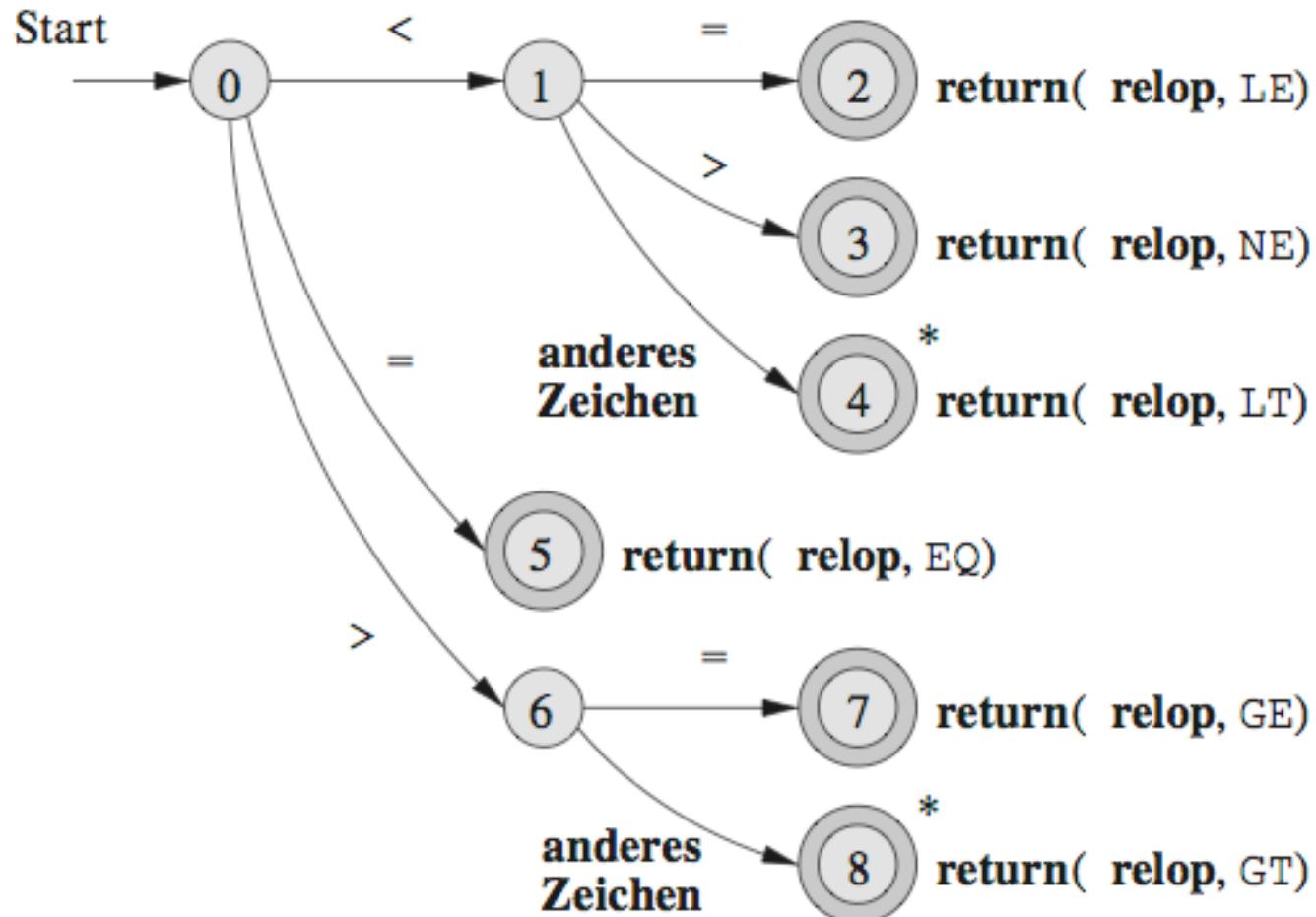


Compiler

Autor:

Folie: 80

# Teil des Automaten:





Compiler

Autor:

Folie: 81

# Laufzeit: Komplexität

- Ein Token: linear
- Stream von Tokens, mit maximal munch:
  - Im schlimmsten Fall: quadratisch !!
    - Demo: quadratic.lex
  - Taucht in der Praxis aber selten auf



Compiler

# Lookahead 瞻前顾后

r1/r2

erkenne r1, aber nur wenn es von r2 gefolgt ist

Lexikalische Analyse

Beispiel FORTRAN IF:  
IF / (.\*)

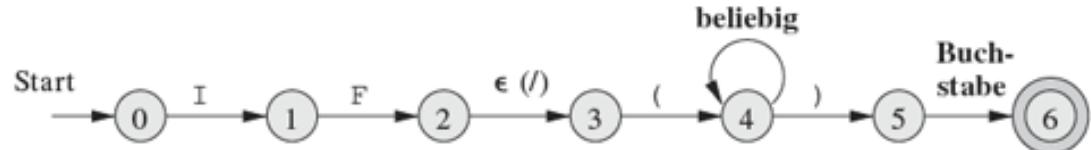


Abbildung 3.55: NFA, der das Schlüsselwort IF erkennt

Achtung: Lex zählt Lookahead bzgl.  
maximal-munch mit !  
Flex erlaubt keinen Lookahead



Compiler

Autor:

- Lexikalische Analyse: Was & Warum
- Reguläre Ausdrücke
- Endliche Automaten
  - Deterministisch vs. NFA
  - Übersetzungen RE → NFA → DFA
  - Minimizing
- Grundlagen von lex
- Maximal munching