

Compilerbau

Wintersemester 2010 / 2011

Dr. Heiko Falk

Technische Universität Dortmund

Lehrstuhl Informatik 12

Entwurfsautomatisierung für Eingebettete Systeme

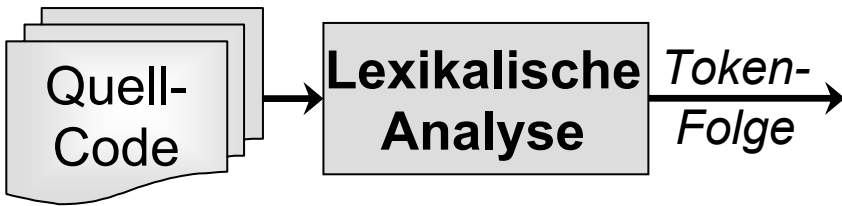
Kapitel 3

Lexikalische Analyse (Scanner)

Gliederung der Vorlesung

- **Kapitel 1: Compiler – Abhängigkeiten und Anforderungen**
- **Kapitel 2: Interner Aufbau von Compilern**
- **Kapitel 3: Lexikalische Analyse (Scanner)**
 - Von Regulären Ausdrücken zu Nichtdeterministischen Automaten
 - Von Nichtdeterministischen zu Deterministischen Automaten
 - Automaten-Minimierung
 - Praktische Umsetzung: FLEX
- **Kapitel 4: Syntaktische Analyse (Parser)**
- **Kapitel 5: Semantische Analyse**
- **Kapitel 6: Instruktionsauswahl**
- **Kapitel 7: Register-Allokation**
- **Kapitel 8: Code-Optimierung**
- **Kapitel 9: Ausblick**

Aufgabe der Lexikalischen Analyse (1)

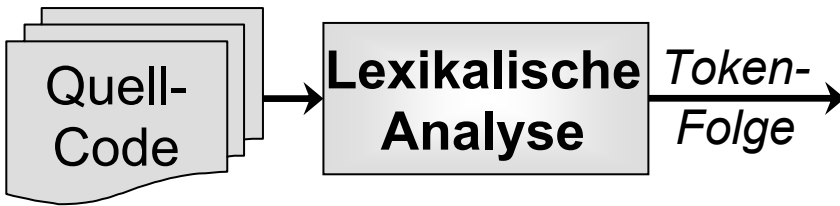


- Zeichenweises Einlesen des Quellprogramms vom Start der Datei bis zum Ende
- Verwaltung eines Zeigers auf die aktuelle Position in der Quell-Datei

```
int main()  
{  
    int val = 41;  
  
    return ++val;  
}
```

Four vertical arrows point downwards from the top of the code block. The first arrow is white, the second is light green, the third is medium green, and the fourth is dark green. They are positioned above the first four characters of the first line: 'i', 'n', 't', and ' '.

Aufgabe der Lexikalischen Analyse (2)



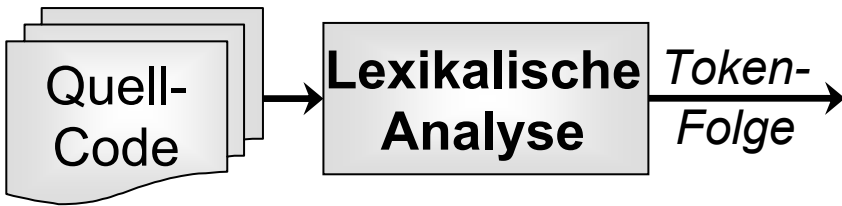
- Bestimmen des längsten Präfix der restlichen Quelldatei ab der aktuellen Position, der ein Wort besonderer Bedeutung der Quellsprache ist.

```

int main()
{
    int val = 41;
    ↓
    return ++val;
}
  
```

Beispiel: Ab aktueller Position muss für C, C++, Java, ... das Wort '++' erkannt werden, und nicht bloß '+'.
☞ Längste Präfixe!

Aufgabe der Lexikalischen Analyse (3)



- Umwandeln des gefundenen Wortes von einer (unhandlichen) Zeichenkette in effizientere Codierung (Token)

```

int main()
{
    int val = 41;
    ↓
    return ++val;
}
  
```

Beispiel: Erzeugung des Tokens

- (id, va1) für den Bezeichner (*Identifizier*) va1, oder
- (1, 17) falls 1 der Code für Bezeichner ist und va1 der 17. gefundene Bezeichner ist.

Worte einer Sprache

- Eine endliche, nichtleere Menge von Zeichen Σ heißt *Alphabet*.
- Ein *Wort* x über Σ der Länge n ist eine endliche Folge von Zeichen aus Σ : $x : x_1x_2\dots x_n$ mit $x_i \in \Sigma$

Spezielle Worte und Mengen von Worten:

- | | | |
|-----------------|---|--|
| ■ ε | | <i>Leeres Wort ohne Zeichen</i> |
| ■ Σ^0 | $\{ x \mid x : \emptyset \rightarrow \Sigma \} = \{ \varepsilon \}$ | <i>Menge aller Worte ohne Zeichen</i> |
| ■ Σ^n | $\{ x \mid x : \{1, \dots, n\} \rightarrow \Sigma \}$ | <i>Menge aller Worte der Länge n über Σ</i> |
| ■ Σ^* | $\bigcup_{n \geq 0} \Sigma^n$ | <i>Menge aller Worte über Σ</i> |
| ■ Σ^+ | $\bigcup_{n \geq 1} \Sigma^n$ | <i>Menge aller nichtleeren Worte über Σ</i> |
| ■ $x.y$ | $x_1\dots x_n y_1\dots y_m$ | <i>Konkatenation von $x = x_1\dots x_n$ und $y = y_1\dots y_m$</i> |

Formale Sprachen

- *Formale Sprachen* über Σ sind Teilmengen von Σ^* , d.h. Mengen irgendwelcher Worte über Σ .

Operationen auf formalen Sprachen L, L_1, L_2 :

- | | | |
|------------------|---|-----------------------------------|
| ■ $L_1 \cup L_2$ | | <i>Vereinigung von Sprachen</i> |
| ■ $L_1 L_2$ | $\{ x.y \mid x \in L_1, y \in L_2 \}$ | <i>Konkatenation von Sprachen</i> |
| ■ \bar{L} | $\Sigma^* - L$ | <i>Komplement einer Sprache</i> |
| ■ L^n | $\{ x_1 \dots x_n \mid x_i \in L, 1 \leq i \leq n \}$ | <i>n-fache Konkatenation</i> |
| ■ L^* | $\bigcup_{n \geq 0} L^n$ | <i>Abschluss einer Sprache</i> |
| ■ L^+ | $\bigcup_{n \geq 1} L^n$ | <i>Nichtleerer Abschluss</i> |

Zusammenhang Scanner ↔ Sprachen

- Die vom Scanner erkannten Worte der Quellsprache bilden eine nichtleere reguläre Sprache.
- Reguläre Sprachen sind formale Sprachen, die sich mit Hilfe *regulärer Ausdrücke* beschreiben lassen.
- Reguläre Sprachen können durch *endliche Automaten* erkannt werden.

Grundlegender Ablauf zum Entwurf eines Scanners:

- ☞ Beschreibung der Worte der Quellsprache mit Hilfe regulärer Ausdrücke.
- ☞ (Automatisierte) Überführung der regulären Ausdrücke in endliche Automaten, die die Worte der Quellsprache erkennen.

Reguläre Sprachen

Definition (*Reguläre Sprache*):

Sei Σ ein Alphabet.

- \emptyset und $\{\varepsilon\}$ sind reguläre Sprachen über Σ .
- Für alle $a \in \Sigma$ ist $\{a\}$ eine reguläre Sprache.
- Sind R_1 und R_2 reguläre Sprachen über Σ , so auch
 - $R_1 \cup R_2$
 - $R_1 R_2$ und
 - R_1^*

Reguläre Ausdrücke

Definition (*Regulärer Ausdruck*):

Sei Σ ein Alphabet.

- \emptyset ist ein regulärer Ausdruck für die reguläre Sprache \emptyset
- ε ist ein regulärer Ausdruck für die reguläre Sprache $\{ \varepsilon \}$
- a (für $a \in \Sigma$) ist ein regulärer Ausdruck für die reguläre Sprache $\{ a \}$
- Sind r_1 und r_2 reguläre Ausdrücke für die regulären Sprachen R_1 und R_2 , so ist
 - $r_1|r_2$ ein regulärer Ausdruck für die reguläre Sprache $R_1 \cup R_2$,
 - r_1r_2 ein regulärer Ausdruck für die reguläre Sprache R_1R_2 ,
 - r_1^* ein regulärer Ausdruck für die reguläre Sprache R_1^* .

Reguläre Ausdrücke und Sprachen

Beispiel:

Sei $\Sigma = \{ a, b \}$ ein Alphabet.

Regulärer Ausdruck	Reguläre Sprache	Worte
<ul style="list-style-type: none"> ■ $a b$ 	$\{ a, b \}$	a, b
<ul style="list-style-type: none"> ■ ab^*a 	$\{a\}\{b\}^*\{a\}$	$aa, aba, abba, abbba, \dots$
<ul style="list-style-type: none"> ■ $(ab)^*$ 	$\{ ab \}^*$	$\varepsilon, ab, abab, \dots$
<ul style="list-style-type: none"> ■ $abba$ 	$\{a\}\{b\}\{b\}\{a\}$	$abba$

Nichtdeterministische Endliche Automaten

Definition (*Nichtdeterministischer Endlicher Automat, NEA*):

Ein *nichtdeterministischer endlicher Automat* ist ein Tupel

$M = (\Sigma, Q, q_0, F, \Delta)$ mit

- Σ : endliches **Eingabealphabet**
- Q : endliche Menge von **Zuständen**
- $q_0 \in Q$: **Anfangszustand**
- $F \subseteq Q$: Menge der **akzeptierenden Endzustände**
- $\Delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times Q$: Relation der **Zustandsübergänge**

Schritte eines NEAs

Definition (*Schrittrelation*):

Sei $M = (\Sigma, Q, q_0, F, \Delta)$ ein NEA. Ein Paar (q, w) mit $q \in Q$ und $w \in \Sigma^*$ heißt *Konfiguration*. Die *Schrittrelation* \xrightarrow{M} setzt Konfigurationen in Beziehung:

$$(q, ew) \xrightarrow{M} (p, w) \Leftrightarrow (q, e, p) \in \Delta \text{ und } e \in \Sigma \cup \{ \varepsilon \}$$

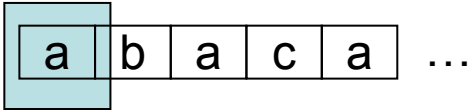
\xrightarrow{M}^* bezeichnet die transitive Hülle der Schrittrelation.

In Worten: Die Schrittrelation \xrightarrow{M} modelliert einen einzelnen Zustandsübergang von M von aktuellem Zustand q in Folgezustand p für ein Eingabezeichen e bzw. ε und ein Rest-Wort w .

\xrightarrow{M}^* modelliert eine ganze Folge einzelner Schritte für ein Wort w .

Arbeitsweise von NEAs

Eingabeband:



Start des Automaten:

- Lesekopf steht auf erstem Zeichen der Eingabe
- Speicher des Automaten enthält Anfangszustand q_0

NEA M:

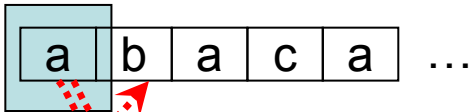
Akt. Zustand: q_0



Kontrolle Δ :

Arbeitsweise von NEAs

Eingabeband:



NEA M :

Akt. Zustand: q_0

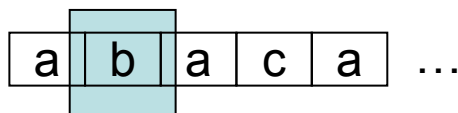
Kontrolle Δ : q_0 a
 q_1

Pro Verarbeitungsschritt:

- „Raten“ der aktuellen Eingabe e : entweder aktuelles Zeichen unter Lesekopf oder ‘ ε ’.
 (☞ *Nichtdeterminismus*)
- Berechnen des Folgezustands q_{i+1} anhand von aktuellem Zustand q_i und Eingabe e : $q_{i+1} \in \Delta(q_i, e)$
- Abspeichern von q_{i+1}
- Nur falls Zeichen von Eingabeband gelesen wurde: Lesekopf um eine Position nach rechts bewegen.

Arbeitsweise von NEAs

Eingabeband:



Pro Verarbeitungsschritt:

- „Raten“ der aktuellen Eingabe e : entweder aktuelles Zeichen unter Lesekopf oder ‘ ε ’.
(☞ *Nichtdeterminismus*)
- Berechnen des Folgezustands q_{i+1} anhand von aktuellem Zustand q_i und Eingabe e : $q_{i+1} \in \Delta(q_i, e)$
- Abspeichern von q_{i+1}
- Nur falls Zeichen von Eingabeband gelesen wurde: Lesekopf um eine Position nach rechts bewegen.

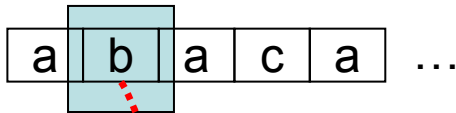
NEA M :

Akt. Zustand: q_1

Kontrolle Δ : $q_1 \quad \varepsilon$
 $\underbrace{\hspace{10em}}_{q_2}$

Arbeitsweise von NEAs

Eingabeband:



NEA M:

Akt. Zustand: q_2

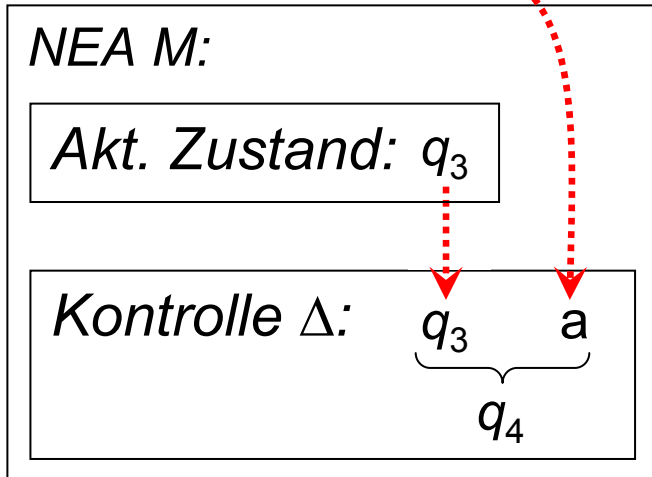
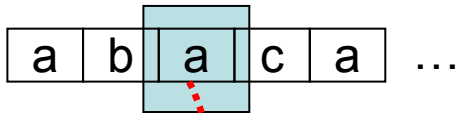
Kontrolle Δ : q_2 b
 q_3

Pro Verarbeitungsschritt:

- „Raten“ der aktuellen Eingabe e : entweder aktuelles Zeichen unter Lesekopf oder ‘ ε ’.
 (☞ *Nichtdeterminismus*)
- Berechnen des Folgezustands q_{i+1} anhand von aktuellem Zustand q_i und Eingabe e : $q_{i+1} \in \Delta(q_i, e)$
- Abspeichern von q_{i+1}
- Nur falls Zeichen von Eingabeband gelesen wurde: Lesekopf um eine Position nach rechts bewegen.

Arbeitsweise von NEAs

Eingabeband:

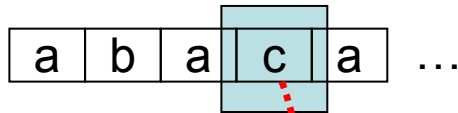


Pro Verarbeitungsschritt:

- „Raten“ der aktuellen Eingabe e : entweder aktuelles Zeichen unter Lesekopf oder ‘ ε ’.
(☞ *Nichtdeterminismus*)
- Berechnen des Folgezustands q_{i+1} anhand von aktuellem Zustand q_i und Eingabe e : $q_{i+1} \in \Delta(q_i, e)$
- Abspeichern von q_{i+1}
- Nur falls Zeichen von Eingabeband gelesen wurde: Lesekopf um eine Position nach rechts bewegen.

Ein NEA als Scanner (1)

Eingabeband:



NEA M :

Akt. Zustand: q_4

Kontrolle Δ :

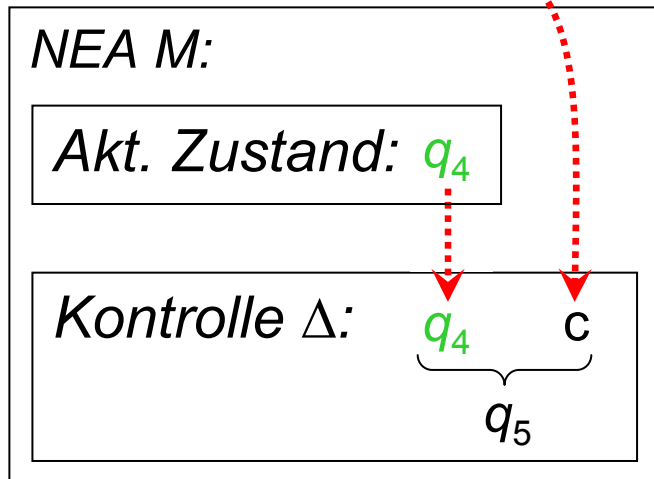
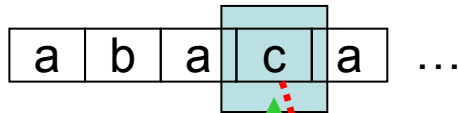
q_4	c
$\underbrace{\hspace{10em}}_{\emptyset}$	

Gültiges Wort gefunden, wenn

- aktueller Zustand q_i ein Endzustand ist, d.h. $q_i \in F$, und
- für aktuelle Eingabe e kein weiterer Übergang möglich ist, d.h. $\Delta(q_i, e) = \emptyset$
- Gefundenes Wort: aba

Ein NEA als Scanner (2)

Eingabeband:

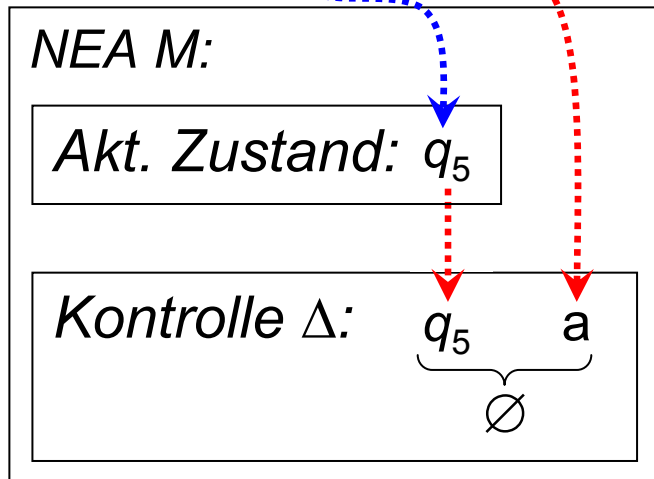
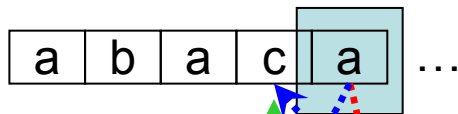


Gültiges Wort evtl. nicht-maximaler Länge gefunden, wenn

- aktueller Zustand q_i ein Endzustand ist, d.h. $q_i \in F$, und
 - für aktuelle Eingabe e noch weitere Übergänge möglich sind, d.h. $\Delta(q_i, e) = q_{i+1} \neq \emptyset$
- ☞ Speichere aktuellen Zustand und Position des Lesekopfes in separatem Speicher zwischen.
- ☞ Fahre mit Folgezustand und Eingabe e wie gewohnt fort.

Ein NEA als Scanner (3)

Eingabeband:

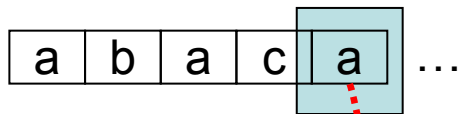


Entdeckung fehlerhafter Worte:

- Ist $\Delta(q_i, e) = \emptyset$, aber q_i kein akzeptierender Endzustand ($q_i \notin F$):
 - lade letzten Endzustand und Position des Lesekopfes aus separatem Speicher;
 - akzeptiere Wort, das nach diesem „Backtracking“ erkannt wurde.
- Gefundenes Wort: *aba*

Ein NEA als Scanner (4)

Eingabeband:



NEA M :

Akt. Zustand: q_5

Kontrolle Δ :

q_5	a
$\underbrace{\hspace{10em}}_{\emptyset}$	

Entdeckung fehlerhafter Worte:

- Ist $\Delta(q_i, e) = \emptyset$, aber q_i kein akzeptierender Endzustand ($q_i \notin F$), und
- enthält separater Speicher noch keinen Endzustand:
- ☞ Die komplette bisher gelesene Eingabe repräsentiert kein Wort der Sprache.
- ☞ Abbruch mit Fehlermeldung

Beispiel: NEA für ganze und reale Konstanten

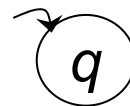
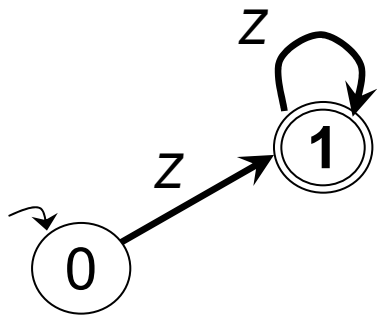
Zu erkennende Worte:

- Ganzzahlige Konstanten: alle Zahlen bestehend aus Ziffern 0 – 9
z.B. **042**
- Realzahlige Konstanten:
 - Bestehen aus Vorkommateil, einem ‘.’ als Dezimalpunkt und einem Nachkommateil
 - Vorkommateil ist eine ganzzahlige Konstante wie oben
 - Nachkommateil ist eine ganzzahlige Konstante wie oben, optional gefolgt von ‘E’ und einem zweistelligen dezimalen Exponenten

z.B. **42.08E15**

Beispiel: NEA für ganze und reale Konstanten

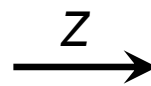
Zustandsübergangsdiagramm:



Startzustand



Akzeptierender Endzustand

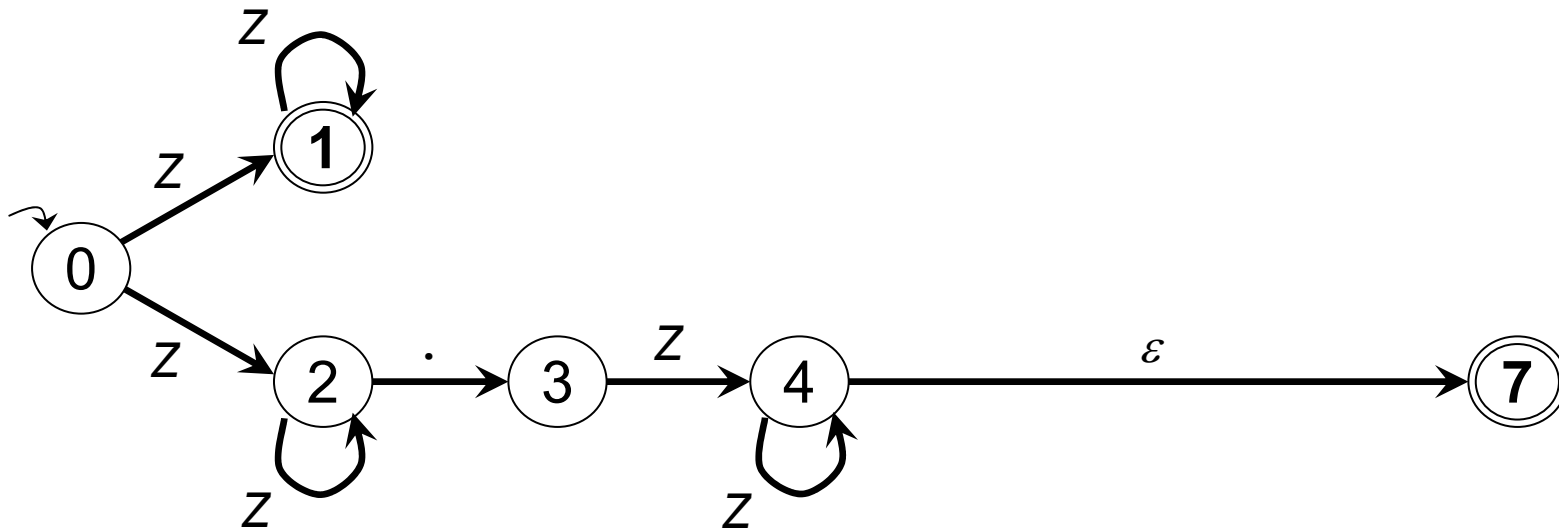


Zustandsübergang für eine der Ziffern aus 0 – 9

- Dieser Teilautomat akzeptiert beliebige ganzzahlige Konstanten, die aus mindestens einer Ziffer bestehen.
- Regulärer Ausdruck für diesen Teilautomaten: ZZ^*

Beispiel: NEA für ganze und reale Konstanten

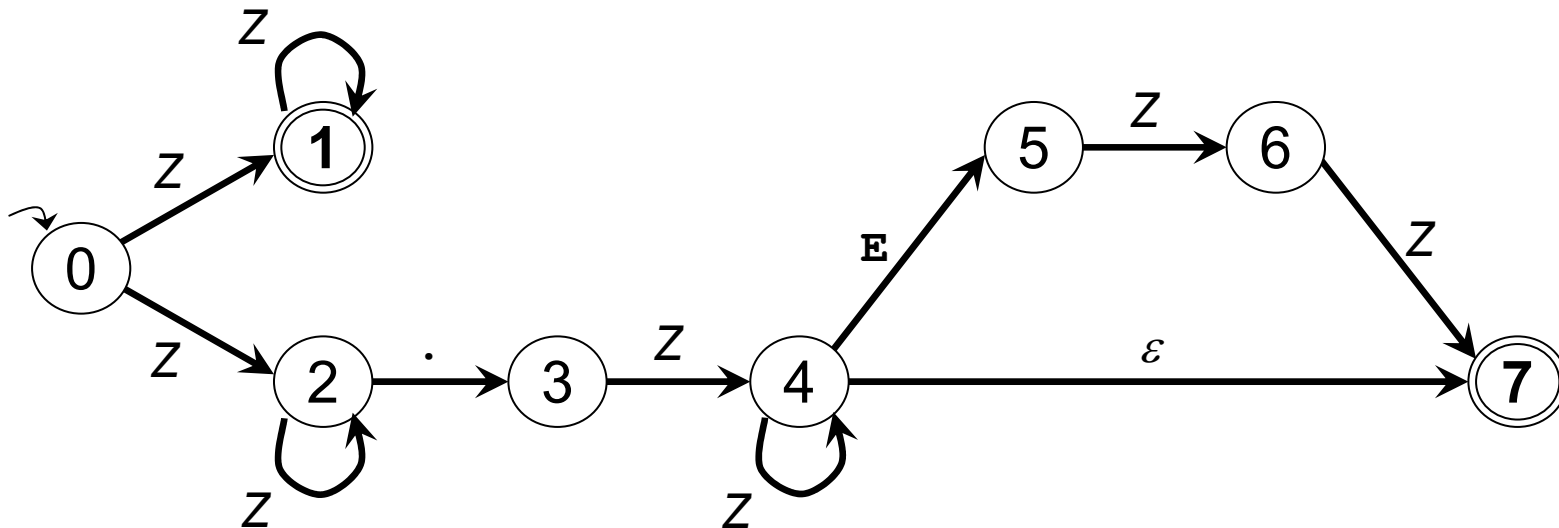
Zustandsübergangsdiagramm:



- Zusätzlich: Teilautomat akzeptiert beliebige reale Konstanten, deren Vor- & Nachkommateile aus mindestens einer Ziffer bestehen, getrennt durch '.'.
- Regulärer Ausdruck für diesen Teilautomaten: $ZZ^*(\underline{\epsilon} \mid .ZZ^*)$

Beispiel: NEA für ganze und reale Konstanten

Zustandsübergangsdiagramm:



- Zusätzlich: Automat akzeptiert Nachkommateile mit optionalem zweistelligem Exponenten
- Regulärer Ausdruck für den kompletten Automaten:
 $ZZ^*(\underline{\varepsilon} | (\cdot ZZ^*(\underline{\varepsilon} | \mathbf{E}ZZ)))$

NEA-Erzeugung aus Regulären Ausdrücken

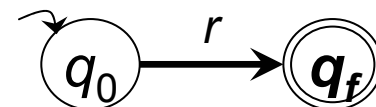
Algorithmus:

Eingabe: Endliches nichtleeres Alphabet Σ ,
 regulärer Ausdruck r über Σ

Ausgabe: Übergangsdigramm eines NEAs M , der die von r
 beschriebene reguläre Sprache R akzeptiert.

- **Initialisierung:**

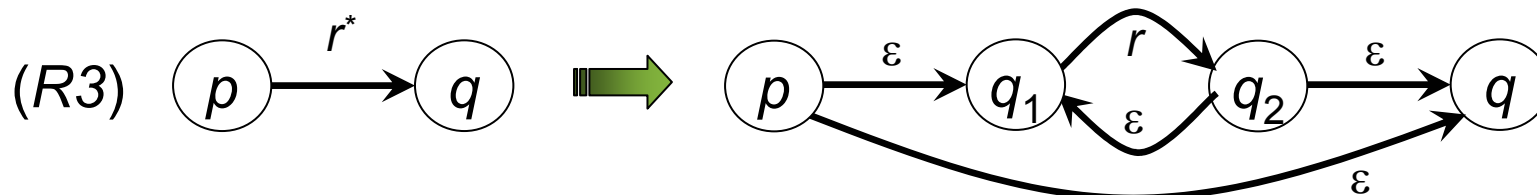
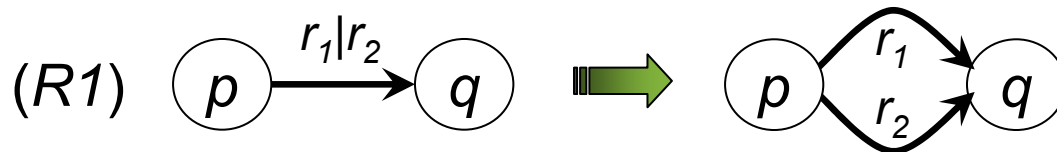
Zustände:	$Q = \{ q_0, q_f \}$
Anfangszustand:	q_0
Endzustand:	q_f
Zustandsübergang:	Für Eingabe r



NEA-Erzeugung aus Regulären Ausdrücken

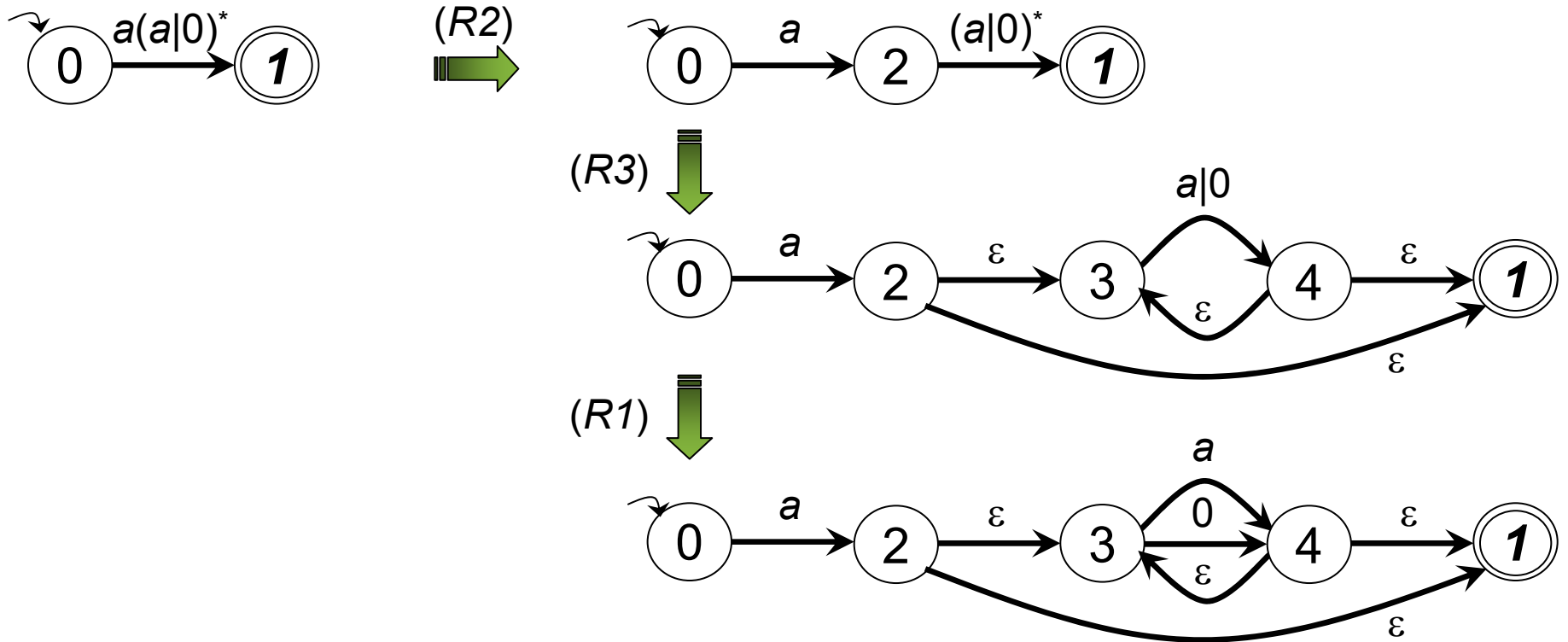
Algorithmus (Fortsetzung):

- **Schrittweise Zerlegung** von r gemäß syntaktischer Struktur und Verfeinerung des aktuellen Übergangendiagramms. Anwendung folgender Regeln, bis alle Kanten von M mit Zeichen aus Σ oder ε markiert sind:



Erzeugung eines NEAs

Beispiel: Sei $\Sigma = \{ a, 0 \}$ ein Alphabet. $r = a(a|0)^*$ beschreibt die Menge der Worte über Σ , die mit einem a beginnen.



Gliederung der Vorlesung

- **Kapitel 1: Compiler – Abhängigkeiten und Anforderungen**
- **Kapitel 2: Interner Aufbau von Compilern**
- **Kapitel 3: Lexikalische Analyse (Scanner)**
 - Von Regulären Ausdrücken zu Nichtdeterministischen Automaten
 - Von Nichtdeterministischen zu Deterministischen Automaten
 - Automaten-Minimierung
 - Praktische Umsetzung: FLEX
- **Kapitel 4: Syntaktische Analyse (Parser)**
- **Kapitel 5: Semantische Analyse**
- **Kapitel 6: Instruktionsauswahl**
- **Kapitel 7: Register-Allokation**
- **Kapitel 8: Code-Optimierung**
- **Kapitel 9: Ausblick**

Problem eines Scanners mit NEAs

Nichtdeterminismus:

Der NEA, der gemäß vorigem Abschnitt als Scanner für einen regulären Ausdruck r erzeugt wird, müsste an zwei Stellen *raten*:

- Er müsste zuerst entscheiden, ob er ein Zeichen der Eingabe liest oder einen ε -Übergang macht, falls $\Delta(q_i, \varepsilon)$ definiert ist.
- Da Δ eine Übergangsrelation ist, kann es für einen aktuellen Zustand q_i und eine Eingabe e mehrere Folgezustände q_{i+1} geben:

$$\Delta(q_i, e) = \{ q', q'', q''', \dots \}$$

Der NEA muss entscheiden, welchen der möglichen Folgezustände er annimmt.

Deterministische Endliche Automaten

Ausweg:

Überführen des für regulären Ausdruck r generierten NEAs M in deterministischen endlichen Automaten M' :

Für jeden NEA M , der eine Sprache L akzeptiert, gibt es einen DEA M' , der L akzeptiert.

Definition (**Deterministischer Endlicher Automat, DEA**):

Sei $M = (\Sigma, Q, q_0, F, \Delta)$ ein NEA. M heißt *deterministischer endlicher Automat*, wenn Δ eine partielle Funktion $\delta: Q \times \Sigma \rightarrow Q$ ist.

In Worten: In einem DEA gibt es keine ε -Übergänge, und für jeden aktuellen Zustand q_i und jede Eingabe e gibt es höchstens einen Folgezustand $q_{i+1} = \delta(q_i, e)$.

Umwandlung NEA \rightarrow DEA (1)

Grundsätzliche Idee:

Ein Zustand q' des DEA M' repräsentiert alle diejenigen Zustände q von M , die M vom Startzustand q_0 aus für ein Wort w nach q überführen – einschließlich aller möglichen ε -Übergänge.

Definition (ε -Folgezustände):

Sei $M = (\Sigma, Q, q_0, F, \Delta)$ ein NEA und $q \in Q$. Die ε -Folgezustände von q sind:

$$\varepsilon\text{-FZ}(q) = \{ p \in Q \mid (q, \varepsilon) \xrightarrow{*}_M (p, \varepsilon) \}$$

d.h. alle Folgezustände p von q (inkl. q), die allein durch ε -Übergänge angenommen werden können. Für Zustandsmengen $S \subseteq Q$ gilt:

$$\varepsilon\text{-FZ}(S) = \bigcup_{q \in S} \varepsilon\text{-FZ}(q)$$

Umwandlung NEA \rightarrow DEA (2)

Algorithmus:

Eingabe: NEA $M = (\Sigma, Q, q_0, F, \Delta)$

Ausgabe: DEA $M' = (\Sigma, Q', q_0', F', \delta)$

- **Initialisierung:**

Anfangszustand:	$q_0' = \varepsilon\text{-FZ}(q_0)$
Zustände:	$Q' = \{q_0'\}$
Endzustände:	$F' = \emptyset$
Zustandsübergänge:	$\delta = \emptyset$
Hilfsmarkierung:	$\text{marked}[q_0'] = \text{false};$

Anfangs enthält M' den einzigen Zustand q_0' . q_0' modelliert den Startzustand q_0 von M und alle weiteren Zustände, die von q_0 aus mit fortgesetzten ε -Übergängen angenommen werden können.

Umwandlung NEA \rightarrow DEA (3)

Algorithmus (*Fortsetzung*):

- for (<alle Zustände $S = \{ q_{S,1}, q_{S,2}, q_{S,3}, \dots \} \in Q'$ mit
marked[S] == false >)
 - // Nur Zustände bearbeiten, die noch nicht betrachtet wurden.
marked[S] = true;

 - // Aktueller Zustand S des DEA ist akzeptierend, wenn er
// irgend einen akzeptierenden Zustand des NEA enthält.
if (<S enthält Zustand $q_{S,i} \in Q$ mit $q_{S,i} \in F$ >)
 - $F' = F' \cup S$;

...

Umwandlung NEA \rightarrow DEA (4)

Algorithmus (Fortsetzung):

- for (<alle Zustände $S = \{ q_{S,1}, q_{S,2}, q_{S,3}, \dots \} \in Q'$ mit
 marked[S] == false >)
- ...
- for (<alle Buchstaben $a \in \Sigma$ >)
 - // Alle ε -Folgezustände ermitteln, die M für Eingabe a an-
 // nehmen kann, ausgehend von jedem NEA-Zustand $q_{S,i}$
 $T = \varepsilon\text{-FZ}(\{ p \in Q \mid (q_{S,i}, a, p) \in \Delta \text{ und } q_{S,i} \in S \})$;
 - if ($T \notin Q'$)
 - $Q' = Q' \cup T$; // Füge Menge T aller möglichen
 - marked[T] = false; // Folgezustände zu Q' hinzu.
- ...

Umwandlung NEA \rightarrow DEA (5)

Algorithmus (*Fortsetzung*):

- for (<alle Zustände $S = \{ q_{S,1}, q_{S,2}, q_{S,3}, \dots \} \in Q'$ mit
marked[S] == false >)

...

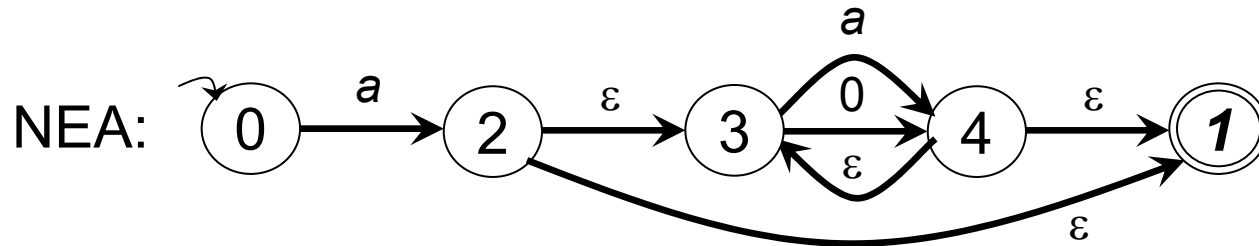
- for (<alle Buchstaben $a \in \Sigma$ >)

...

- // Füge neuen Zustandsübergang von aktuellem Zustand S
// zu Folgezustand T unter Eingabe a hinzu.
 $\delta = \delta \cup \{ (S, a) \rightarrow T \};$

Erzeugung eines DEAs (1)

Beispiel: Sei $\Sigma = \{ a, 0 \}$ ein Alphabet, $r = a(a|0)^*$.

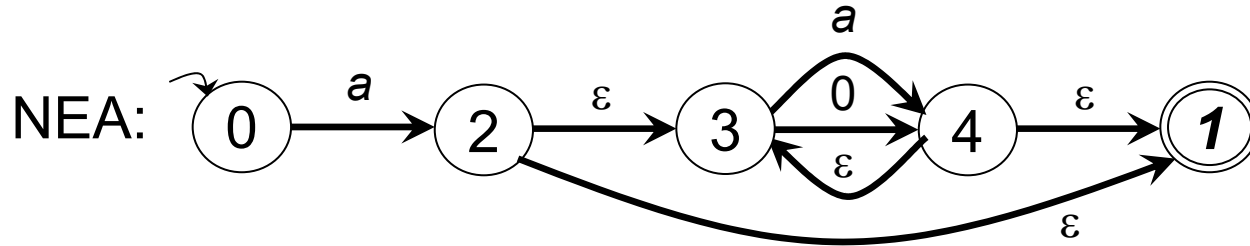


Startzustand: $0' = \{ 0 \}$

Initiale Zustandsmenge: $Q' = \{ 0' \}$

Erzeugung eines DEAs (2)

Beispiel:



Akt. Zustand S:

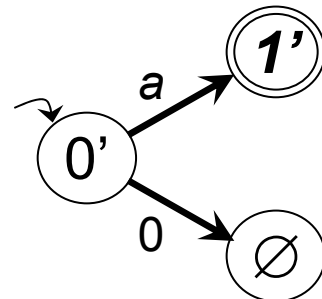
$$0' = \{ 0 \}$$

Neues Q':

$$\{ 0', 1', \emptyset \} \text{ mit}$$

$$1' = \{ 2, 3, 1 \}$$

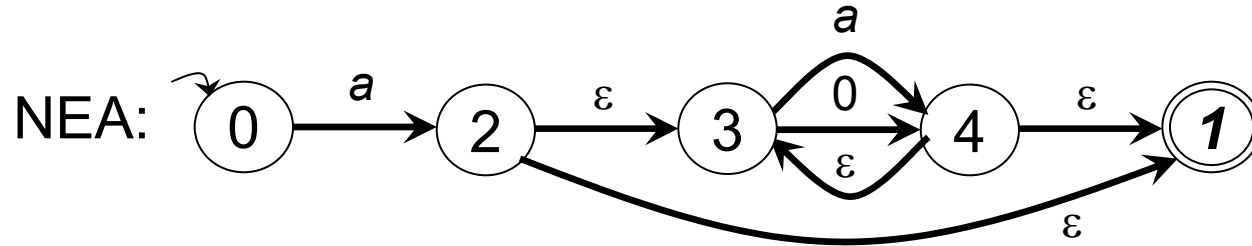
(Teil-) DEA:



← Spezieller Fehler-Zustand

Erzeugung eines DEAs (3)

Beispiel:



Akt. Zustand S:

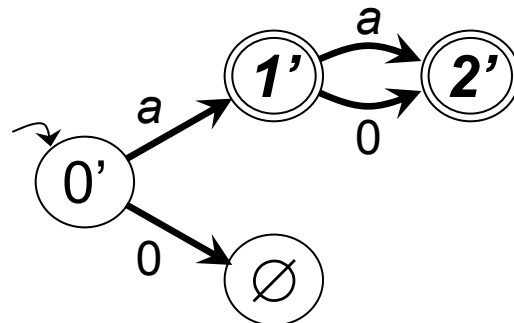
$$\textcircled{1'} = \{ \textcircled{2}, \textcircled{3}, \textcircled{1} \}$$

Neues Q':

$$\{ \textcircled{0'}, \textcircled{1'}, \textcircled{2'}, \emptyset \} \text{ mit}$$

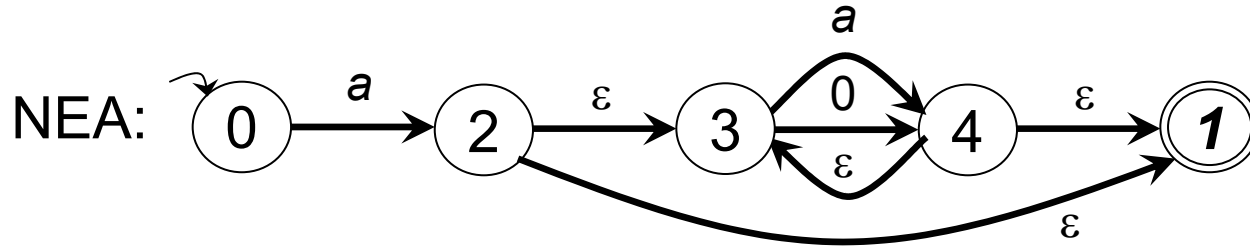
$$\textcircled{2'} = \{ \textcircled{4}, \textcircled{3}, \textcircled{1} \}$$

(Teil-) DEA:



Erzeugung eines DEAs (4)

Beispiel:



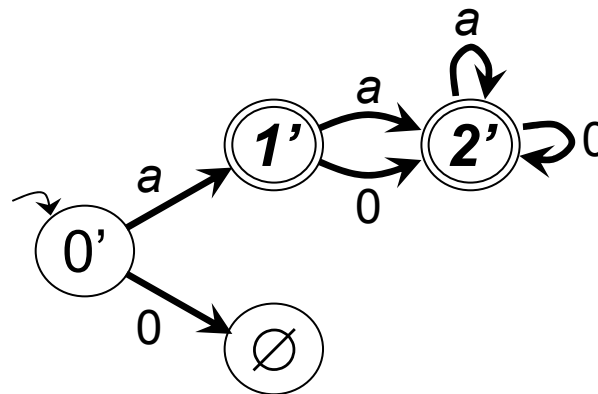
Akt. Zustand S:

$$\textcircled{2'} = \{ \textcircled{4}, \textcircled{3}, \textcircled{1} \}$$

Neues Q':

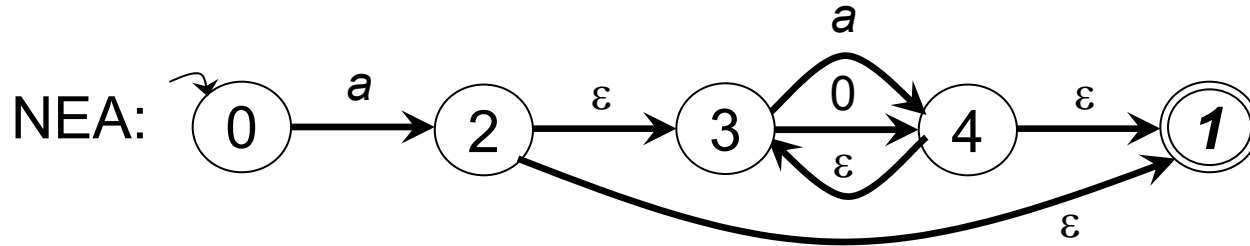
$$\{ \textcircled{0'}, \textcircled{1'}, \textcircled{2'}, \textcircled{\emptyset} \}$$

(Teil-) DEA:

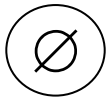


Erzeugung eines DEAs (5)

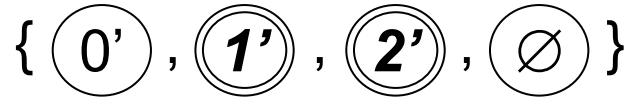
Beispiel:



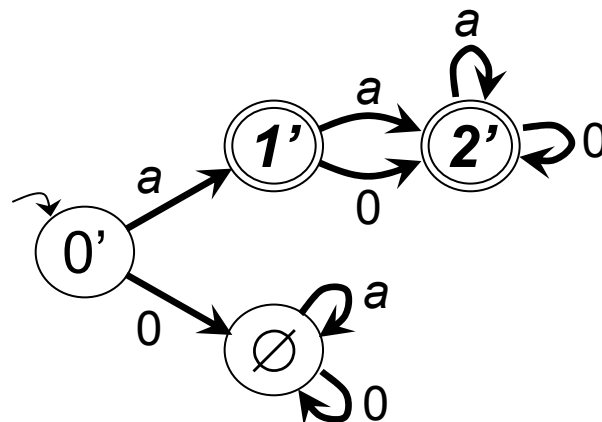
Akt. Zustand S:



Neues Q':



(Teil-) DEA:



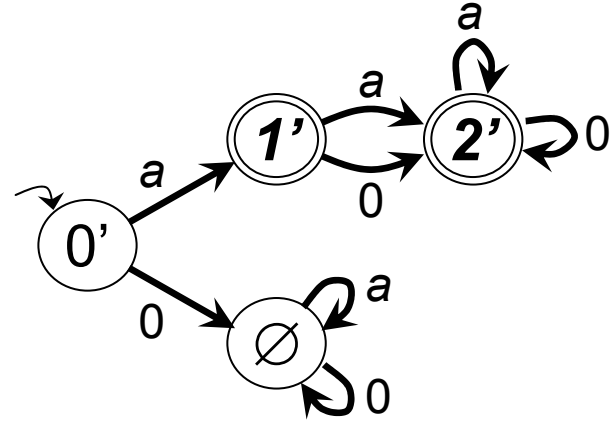
Gliederung der Vorlesung

- **Kapitel 1: Compiler – Abhängigkeiten und Anforderungen**
- **Kapitel 2: Interner Aufbau von Compilern**
- **Kapitel 3: Lexikalische Analyse (Scanner)**
 - Von Regulären Ausdrücken zu Nichtdeterministischen Automaten
 - Von Nichtdeterministischen zu Deterministischen Automaten
 - Automaten-Minimierung
 - Praktische Umsetzung: FLEX
- **Kapitel 4: Syntaktische Analyse (Parser)**
- **Kapitel 5: Semantische Analyse**
- **Kapitel 6: Instruktionsauswahl**
- **Kapitel 7: Register-Allokation**
- **Kapitel 8: Code-Optimierung**
- **Kapitel 9: Ausblick**

Motivation

Redundante Zustände:

DEAs, die gemäß vorigem Abschnitt erzeugt werden, enthalten in der Regel viele redundante Zustände:

- Ist DEA in Zustand 1' oder 2', so akzeptiert er.
 - Ist DEA in Zustand 1', so wechselt er für alle weitere Eingaben von einem akzeptierenden in einen weiteren akzeptierenden Zustand (dito für 2').
- 
- ☞ 1' und 2' sind also bzgl. der Akzeptanz des DEAs äquivalent – der DEA verhält sich stets gleich, egal ob er in 1' oder 2' ist.
 - ☞ 1' und 2' können zu einem Zustand zusammengefasst werden.

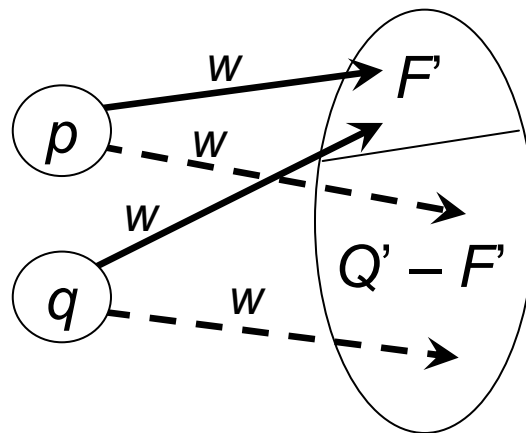
Äquivalenz von Zuständen

Definition (Äquivalenz):

Sei $M' = (\Sigma, Q', q_0', F', \delta)$ ein DEA. Zwei Zustände p und q aus Q' sind *äquivalent*, genau dann wenn für jedes Wort $w \in \Sigma^*$ gilt: $(p, w) \xrightarrow[M']{*}$ und $(q, w) \xrightarrow[M']{*}$ führen beide entweder ausschließlich in Zustände aus F' oder ausschließlich in Zustände aus $Q' - F'$.

Anschaulich:

- entweder nur Übergänge \rightarrow
- oder nur Übergänge \dashrightarrow



Folgerungen... (1)

...für einen Algorithmus zur Automaten-Minimierung:

- Der Algorithmus zerlegt die Zustandsmenge Q' des DEA M' in eine Partition P : $Q' = P_1 \cup P_2 \cup P_3 \cup \dots = P$
- Eine Teilmenge $P_i \in P$ enthält Zustände, von denen *angenommen* wird, dass sie äquivalent sind.
- Der Algorithmus startet mit der Partition $P = F' \cup \{Q' - F'\}$, d.h. es wird optimistisch angenommen, dass jeweils alle akzeptierenden und nicht akzeptierenden Zustände äquivalent sind.
- Wird für eine Teilmenge $P_i \in P$ festgestellt, dass sie inäquivalente Zustände enthält, so wird P_i wiederum zerlegt in $P_{i,1}$ und $P_{i,2}$.
- Am Ende enthalten alle Teilmengen P_i nur Zustände, die auch tatsächlich äquivalent sind.

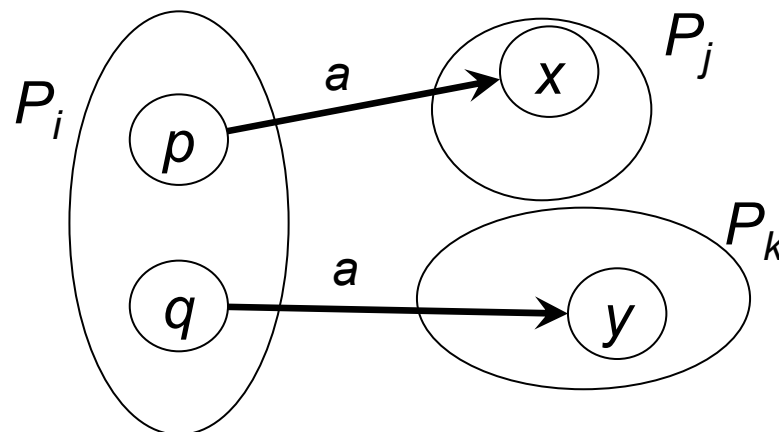
Folgerungen... (2)

...für einen Algorithmus zur Automaten-Minimierung:

- Seien p und q Zustände einer Teilmenge P_i , $a \in \Sigma$ ein Eingabezeichen, $x = \delta(p, a)$ der Folgezustand von p unter Eingabe a und $y = \delta(q, a)$ der Folgezustand von q .
 p und q sind *inäquivalent*, wenn x und y in unterschiedlichen Teilmengen liegen, d.h. $x \in P_j$, $y \in P_k$ und $P_j \neq P_k$.

Anschaulich:

☞ p und q sind inäquivalent.



Automaten-Minimierung (1)

Algorithmus:

Eingabe: DEA $M' = (\Sigma, Q', q_0', F', \delta)$

Ausgabe: DEA $M'_{\min} = (\Sigma, Q'_{\min}, q'_{0,\min}, F'_{\min}, \delta_{\min})$

- **Initialisierung:**

Partition P : $P = \{ F', Q' - F' \};$

Abbruchkriterium: `bool modified = true;`

Automaten-Minimierung (2)

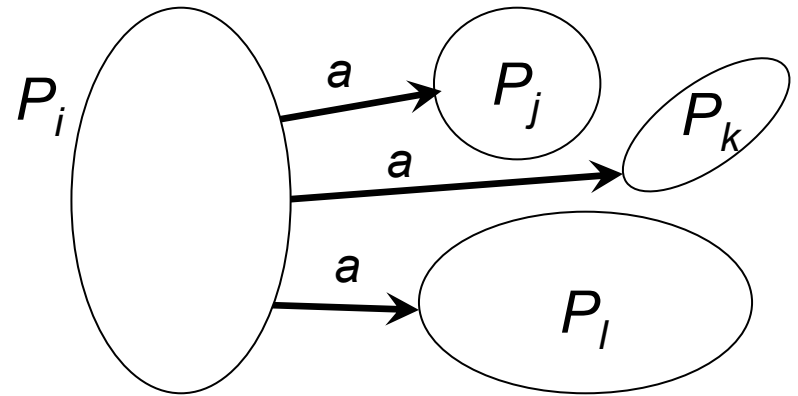
Algorithmus (Fortsetzung):

- while (modified)
 - $P' = P$; modified = false;
 - for (<alle Teilmengen $P_i \in P$ >)
 - for (<alle Buchstaben $a \in \Sigma$ >)
 - // Teste auf inäquivalente Zustände innerhalb von P_i .
 ($P_{i,1}, P_{i,2}$) = split(P_i, a);
 - // Enthält P_i inäquivalente Zustände bzgl. Eingabe a ?
 if (($P_{i,1} \neq \emptyset$) && ($P_{i,2} \neq \emptyset$))
 - $P' = (P' \setminus P_i) \cup P_{i,1} \cup P_{i,2}$; // Ersetze P_i durch $P_{i,1}$
 - modified = true; break; // und $P_{i,2}$.
- $P = P'$;

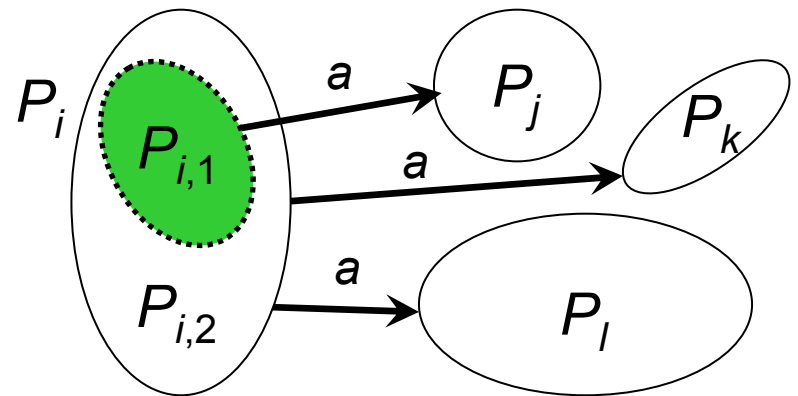
Automaten-Minimierung (3)

Zerlegung von P_i bzgl. Eingabe a – Hilfsfunktion split:

- Vor Aufruf von split:
 - P_i hat für Eingabe a Übergänge z.B. nach P_j , P_k und P_l .



- Zerlegung mit Hilfe von split:
 - Wähle eine der Ziel-Mengen aus, z.B. P_j .
 - Zerlege P_i in alle Zustände, die für a nach P_j führen, und Rest.



Automaten-Minimierung (4)

Algorithmus (*Fortsetzung*):

- Aufbau des minimierten DEAs:**

$$Q'_{\min} = P$$

$$\delta_{\min}(P_i, a) = P_j \quad \text{wenn } \delta(q, a) = p \text{ mit } a \in \Sigma \text{ und } p \in P_j \text{ für ein (und damit für alle) } q \in P_i.$$

$$q'_{0,\min} =$$

Die Teilmenge $P_k \in P$, die q_0 enthält

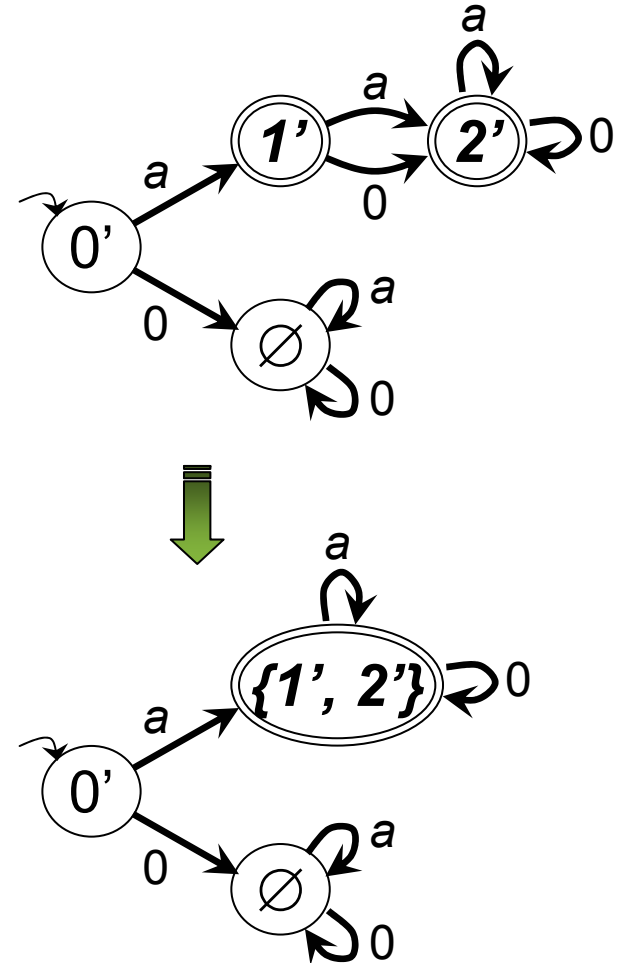
$$F'_{\min} =$$

Die Teilmengen $P_l \in P$, die einen Zustand aus F enthalten

Automaten-Minimierung (5)

Beispiel:

- Nebenstehender DEA
 $M' = (\Sigma = \{a, 0\}, Q' = \{ 0', 1', 2', \emptyset \}, \delta, q_0' = 0', F' = \{1', 2'\})$
soll minimiert werden.
- Für beide Zustände 1' und 2' und beliebige Worte $w \in \Sigma^*$ gilt:
 $(1', w) \xrightarrow{*}_M' \in F'$ und $(2', w) \xrightarrow{*}_M' \in F'$
- ☞ 1' und 2' sind äquivalent.
- Keine weiteren Zustände sind äquivalent, da sie zu jeweils anderem Akzeptanz-Verhalten führen.



Gliederung der Vorlesung

- **Kapitel 1: Compiler – Abhängigkeiten und Anforderungen**
- **Kapitel 2: Interner Aufbau von Compilern**
- **Kapitel 3: Lexikalische Analyse (Scanner)**
 - Von Regulären Ausdrücken zu Nichtdeterministischen Automaten
 - Von Nichtdeterministischen zu Deterministischen Automaten
 - Automaten-Minimierung
 - **Praktische Umsetzung: FLEX**
- **Kapitel 4: Syntaktische Analyse (Parser)**
- **Kapitel 5: Semantische Analyse**
- **Kapitel 6: Instruktionsauswahl**
- **Kapitel 7: Register-Allokation**
- **Kapitel 8: Code-Optimierung**
- **Kapitel 9: Ausblick**

Scanner-Generierung

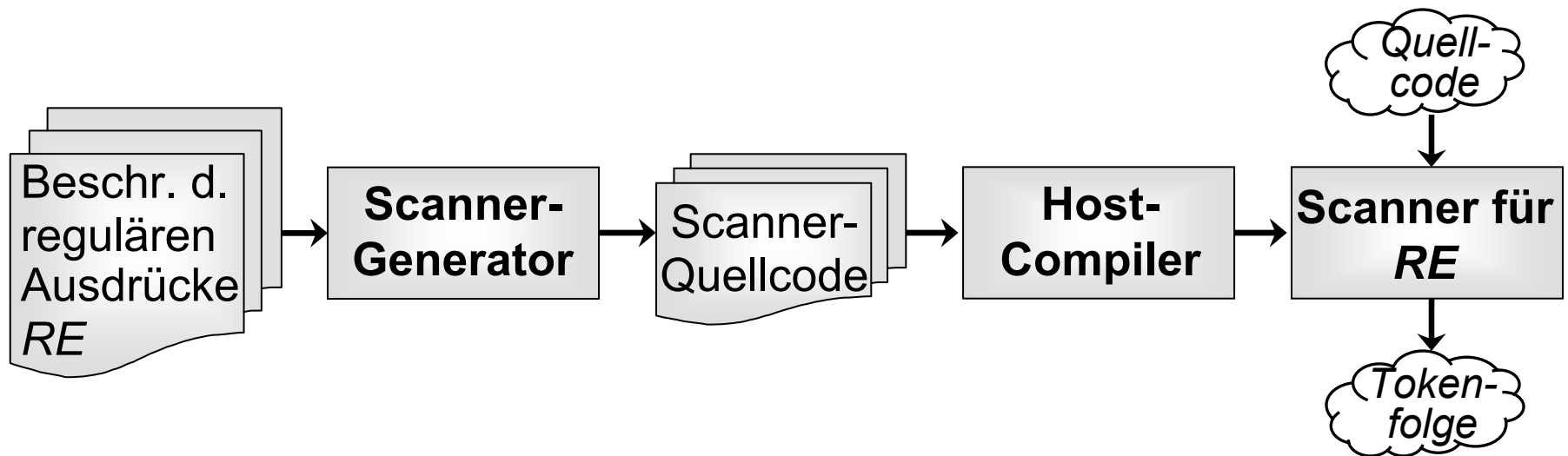
Implementierung eines Scanners:

- Stark von Quellsprache abhängige Aufgabe.
 - Theoretische Grundlagen der lexikalischen Analyse bestens erforscht, effiziente Algorithmen zur Erzeugung eines Scanners auf Basis regulärer Ausdrücke existieren.
 - Per-Hand-Implementierung eines Scanners bei vorhandenem theoretischen Unterbau nicht mehr vertretbar.
(Und auch nicht gewünscht, da diese Aufgabe schlichtweg zu technisch und „langweilig“ ist...)
- 👉 Statt dessen: Verwendung sog. *Scanner-Generatoren*



Scanner-Generatoren

- Sog. *Meta-Programme*, d.h. Programme, die Programme als Ausgabe erzeugen.
- Ein *Scanner-Generator* erhält eine Beschreibung der zu erkennenden regulären Ausdrücke als Eingabe und erzeugt daraus einen Scanner für eben diese Ausdrücke.



Scanner-Generierung mit FLEX (1)

Grundlegender Aufbau einer FLEX Scanner-Beschreibung

- FLEX: *Fast Lexical Analyzer Generator*
 Weit verbreiteter Scanner-Generator, GNU open source
- Generiert aus Beschreibung C-Code eines Scanners
- Eingabedatei für FLEX:



Scanner-Generierung mit FLEX (2)

Präambel

- Eine Präambel ist stets in `%{` und `%}` einzuschließen.
- Es können mehrere Präambeln pro FLEX-Datei vorkommen.
- Eine Präambel enthält beliebigen C-Code, der unverändert an den Beginn des von FLEX erzeugten Codes kopiert wird:

```

%{
    /* Counter, in which line a word was found. */
    unsigned int lineNumber = 1;

    /* Forward declaration of helper function. */
    void incrementLineNumber();
%}
  
```

Scanner-Generierung mit FLEX (3)

Definitionen

- Definitionen erlauben, die Beschreibung regulärer Ausdrücke zu vereinfachen.
- Eine Definition führt für einen regulären Ausdruck einen Namen ein, der an späterer Stelle im FLEX-File verwendet werden kann:

```
digit [0-9]
```

```
letter [a-zA-Z]
```

```
identifizier {letter} ({letter} | {digit}) *
```

- Zeichenklassen werden innerhalb von [] aufgeführt. Bereiche von Zeichen können mit – angegeben werden.
- Reguläre Ausdrücke können mit Hilfe der Operatoren *, +, ?, | und Konkatenation gebildet werden.

Scanner-Generierung mit FLEX (4)

Regeln

- Regeln bestehen aus einem *Muster*, d.h. einem regulären Ausdruck, und einer *Aktion*, d.h. einem Stück C-Code.
- Die Aktion muss in der selben Zeile wie das Muster starten.
- Die Aktion wird ausgeführt, wenn der Scanner ein Symbol als zum Muster gehörend erkannt hat:

```

/* Matches integer numbers.
   yytext is a pointer to the matched string.
   Rules start in column 0. Any indented parts
   are directly copied to the output file. */
{digit}+  printf( "%d: number %d\n",
                  lineNumber, atoi( yytext ) );

```

Scanner-Generierung mit FLEX (5)

Regeln (Fortsetzung)

```

/* Matches operators. Longer C code snippets for
   actions can be enclosed in { ... } */
"+"|"++"|"-"|"--"|"*"|"/"|"="|"==" {
    printf( "%d: operator %s\n",
            lineNumber, yytext );
}

/* Matches keywords of some fictive programming
   language. */
if|then|else|for|while|do|to {
    printf( "%d: keyword %s\n",
            lineNumber, yytext );
}
  
```

Scanner-Generierung mit FLEX (6)

Regeln (Fortsetzung)

```

/* Matches identifiers. Used names of
   definitions in a rule's pattern must be
   enclosed in { ... } */
{identifier} printf( "%d: identifier %s\n",
                    lineNumber, yytext );

[ \t] /* Consumes white spaces and tabs. */

/* Consumes new-line characters. Increments
   line counter. */
\n incrementLineNumber();
  
```

Scanner-Generierung mit FLEX (7)

Regeln (*Fortsetzung*)

- Werden in Regeln Namen für reguläre Ausdrücke verwendet, die im Definitionsteil deklariert wurden, so sind diese Namen in { } einzuschließen: `{digit}+`
- ☞ Dies verhindert, dass die Regel auf die Zeichenkette „`digit`“ reagiert.
- Die Zeichenkette „`while`“ bspw. wird durch die Muster der Keyword- und Identifier-Regeln erfasst.
- ☞ FLEX erkennt „`while`“ als Schlüsselwort, da die Keyword-Regel in der FLEX-Datei vor der Identifier-Regel steht.

Scanner-Generierung mit FLEX (8)

C-Funktionen

- Beliebige C-Funktionen können für Hilfszwecke angegeben werden.
- Hilfsfunktionen können innerhalb von Aktionen aufgerufen werden.

```

/* Increment global variable lineNumber by 1. */
void incrementLineNumber() { lineNumber++; }
  
```

```

/* A complete C program needs a main function. */
int main() {
    yylex();          /* Invoke scanner. */
    printf( "Read lines: %d\n", lineNumber );
    return 0; }
  
```


Literatur

- Reinhard Wilhelm, Dieter Maurer, *Übersetzerbau*, 2. Auflage, Springer, 1997. (☞ Kapitel 7)
ISBN 3-540-61692-6
- Andrew W. Appel, *Modern compiler implementation in C*, Cambridge University Press, 1998. (☞ Kapitel 2)
ISBN 0-521-58390-X
- J. Eliot B. Moss, *Compiler Techniques – Fall 2008*, 2008. (☞ Foliensatz 6)
<http://www-ali.cs.umass.edu/~moss/610.html>

Zusammenfassung

- **Reguläre Ausdrücke: Eleganter Mechanismus, um Worte einer Sprache zu spezifizieren.**
- **Nichtdeterministische Automaten können reguläre Ausdrücke maximaler Länge erkennen, sind aber wegen „Ratens“ ungeeignet für eine reale Implementierung.**
- **Aus nichtdeterministischen Automaten für reguläre Ausdrücke können deterministische Automaten erzeugt werden.**
- **Minimierung liefert deterministischen Automaten mit minimaler Zustandsmenge.**
- **FLEX: Ein Standard-Werkzeug zur Scanner-Generierung.**

