

# **Theoretische Informatik 3. Semester**

Prof. Dr. Martin Plümicke

2. November 2023

# Inhaltsverzeichnis

<b>1</b>	<b>Berechenbarkeit und rekursive Funktionen</b>	<b>4</b>
1.1	Primitive rekursive Funktionen . . . . .	4
1.2	LOOP-Programme . . . . .	8
1.3	$\mu$ -Rekursive Funktionen . . . . .	13
1.4	WHILE-Programme . . . . .	15
1.5	Einführung Turingmaschine . . . . .	17
1.6	Turingmaschine als Automat zur Berechnung von rekursiven Funktionen .	19
<b>2</b>	<b>Grundlagen der Theorie der formalen Sprache</b>	<b>25</b>
<b>3</b>	<b>Reguläre Sprachen</b>	<b>30</b>
3.1	Reguläre Ausdrücke . . . . .	30
3.2	Endliche Automaten . . . . .	31
3.3	Von regulären Sprachen zu Scannern . . . . .	40
3.4	Der Scannergenerator JLex . . . . .	46
3.4.1	Die JLex - Spezifikation . . . . .	46
3.4.2	JLex-Direktiven (Anweisungen) . . . . .	46
3.4.3	Java-Klasse erzeugen . . . . .	47
3.4.4	Beispiel: html-Lexeme . . . . .	47

# 3 Reguläre Sprachen

## 3.1 Reguläre Ausdrücke

**Definition 3.1** (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)  $\alpha \in R(\Sigma) \wedge \beta \in R(\Sigma) \Rightarrow \alpha\beta \in R(\Sigma)$   
 $\alpha|\beta \in R(\Sigma)$   
 $\alpha^* \in R(\Sigma)$
- d)  $\alpha \in R(\Sigma) \Rightarrow (\alpha) \in R(\Sigma)$

**Beispiel.**  $\Sigma = \{a, b, c\}$

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

**Definition 3.2** (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.  $\mathcal{L}(\varepsilon) = \{\varepsilon(='''')\}$
2.  $\alpha \in \Sigma \Rightarrow \mathcal{L}(\alpha) = \{\alpha\}$
3. a)  $\alpha = \beta\gamma \Rightarrow \mathcal{L}(\alpha) = \{ww' | w \in \mathcal{L}(\beta), w' \in \mathcal{L}(\gamma)\}$   
b)  $\alpha = \beta|\gamma \Rightarrow \mathcal{L}(\alpha) = \mathcal{L}(\beta) \cup \mathcal{L}(\gamma)$   
c)  $\alpha = \beta^* \Rightarrow \mathcal{L}(\alpha) = \{\varepsilon\} \cup \{ww' | w \in \mathcal{L}(\beta), w' \in \mathcal{L}(\beta^*)\}$
4.  $\alpha = (\beta) \Rightarrow \mathcal{L}(\alpha) = \mathcal{L}(\beta)$

### Beispiel.

1.  $\Sigma = \{a, b\}$

a)  $\alpha = a \Rightarrow \mathcal{L}(\alpha) = \{a\}$

b)  $\alpha = a|b \Rightarrow \mathcal{L}(\alpha) = \{a, b\}$

c)  $\alpha = a^* \Rightarrow \mathcal{L}(\alpha) = \{\varepsilon, a, aa, aaa, aaaa, aaaaa, \dots\}$

2.  $\Sigma = \{a, b, c\}$

$\alpha = \underbrace{(a^*|b^*)}_{\beta} \underbrace{c}_{\gamma} \Rightarrow \mathcal{L}(\alpha) = \{\varepsilon c, ac, aac, aaac, \dots, bc, bbc, bbcc, \dots\}$

**Satz 3.3.** Die Menge der regulären Sprachen entspricht den Sprachen vom Typ *Chomsky-3*.

*Beweis.* Übungsaufgabe

□

## 3.2 Endliche Automaten

Endliche Automaten prüfen, ob ein Eingabestring in einer vorgegebenen regulären Sprache liegt.

**Definition 3.4** (Deterministische endliche Automat (DEA)). Unter einem DEA versteht man

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

mit

$Q \hat{=}$  Zustandsmenge (endlich)


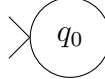

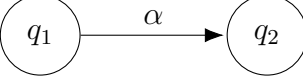
$\Sigma \hat{=}$  Alphabet

$\delta \hat{=}$  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:** 

Unter einer *Konfiguration* versteht man ein Element der Relation  $Q \times \Sigma^*$

**Algorithmus 3.5** (DEA-Erkennungsalgorithmus).

**Eingabe:**  $w \in \Sigma^*$ ,  $A = (Q, \Sigma, \delta, s, F)$

**Ausgabe:**  $erg \in \{True, False\}$

**Nachbedingung:**  $w$  wird von  $A$  erkannt.

1. Man startet mit der Konfiguration  $(q, \bar{w}) = (s, w) \in Q \times \Sigma^*$
2. Wenn  $\bar{w} \neq \varepsilon$ , sei  $\bar{w} = aw'$  mit  $a \in \Sigma$ . Falls  $\delta(q, a) \neq \perp$ :  $q = \delta(q, a)$  und  $\bar{w} = w'$ . Weiter bei Schritt 2.
3. Wenn  $\bar{w} = \varepsilon$  und der letzte Zustand  $q$  ein Finalzustand ist, dann  $erg = True$ , sonst  $erg = False$

**Beispiel.** Sei  $A$  ein DEA (Abbildung 3.1) mit

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

$Q = \{q_0, q_1\}$

$\Sigma = \{a, b\}$

$\delta = \{((q_0, a), q_1), ((q_1, b), q_0)\}$

$s = q_0$

$F = \{q_1\}$

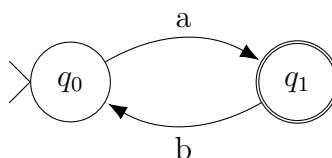


Abbildung 3.1: Graphische Darstellung des DEAs

Anwendung des Algorithmus:

**(a)**  $w = aba$

- $(q_0, aba) \xrightarrow{2} (q_1, ba) \xrightarrow{2} (q_0, a) \xrightarrow{2} (q_1, \varepsilon)$
- $w' = \varepsilon$
- $\text{erg} = \text{True}$ , da  $q_1 \in F$

**(b)**  $w = abb$

- $(q_0, abb) \xrightarrow{2} (q_1, bb) \xrightarrow{2} (q_0, b)$
- $\delta(q_0, b) = \perp$
- $\text{erg} = \text{False}$

**(c)**  $w = ab$

- $(q_0, ab) \xrightarrow{2} (q_1, b) \xrightarrow{2} (q_0, \varepsilon)$
- $q_0 \notin F$
- $\text{erg} = \text{False}$

**Satz 3.6.** Wenn  $w \in \Sigma^*$  von einem Automaten  $A$  akzeptiert wird, so sind die Konfigurationsübergänge eindeutig bestimmt.

**Beweisidee:** Ursache ist die Eigenschaft dass  $\delta$  eine Funktion ist.

**Anmerkung.** Die Eindeutigkeit der Übergänge nennt man *deterministisch*.

Offen ist noch die Frage, wie man zum DEA  $A$  im 1. Schritt des Algorithmus (Def.3.4) kommt. Wir werden nun in mehreren Schritten zeigen, wie man aus einem gegebenen regulären Ausdruck einen DEA macht, der die Sprache des regulären Ausdrucks akzeptiert. Dazu definieren wir zunächst den nichtdeterministischen endlichen Automaten.

**Definition 3.7** (Nichtdeterministische endliche Automat (NEA)). Unter einem *Nichtdeterministischen endlichen Automaten (NEA)* versteht man

$$A = (Q, \Sigma, \Delta, s, F) \text{ mit}$$

$Q \hat{=}$  der Menge der Zustände (endlich).

$\Sigma \hat{=}$  Alphabet

$\Delta \subseteq Q \times \Sigma^* \times Q \hat{=}$  endliche Übergangsrelation

$s \in Q \hat{=} \text{Anfangszustand}$

$F \subseteq Q \hat{=} \text{Menge der Finalzustände}$

Die grafische Darstellung von NEAs und der zugehörige Algorithmus sind analog zum DEA (Definition 3.4) definiert.

**Beispiel.**  $A = (Q, \Sigma, \Delta, s, F)$  mit

$$Q = \{q_0, q_1, q_2\}$$

$$\Sigma = \{a, b\}$$

$$\Delta = \{(q_0, a, q_1), (q_1, b, q_0), (q_1, b, q_2), (q_2, a, q_1)\}$$

$$s = q_0$$

$$F = \{q_1\}$$

Der NEA  $A$  ist in Abbildung 3.2 graphisch dargestellt.

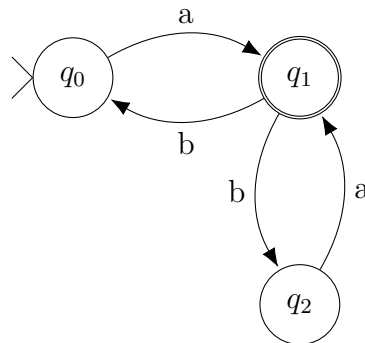


Abbildung 3.2: Nichtdeterministischer endlicher Automat

$A$  ist kein DEA, da man von  $q_1$  mit 'b' sowohl nach  $q_0$  als auch nach  $q_2$  kommen kann.

**Anmerkung.** Diese Uneindeutigkeit nennt man *nicht-deterministisch*.

Nun stellt sich die Frage, ob man jeden NEA in einen äquivalenten DEA überführen kann.

**Satz 3.8.** Zu jedem NEA existiert ein DEA, der die gleiche Sprache akzeptiert.

*Beweis.* Algorithmus der einen NEA in einen DEA umwandelt.

**Algorithmus 3.9** (NEA2DEA).

**Eingabe:** NEA  $A = (Q, \Sigma, \Delta, s, F)$

**Ausgabe:** DEA  $A' = (Q', \Sigma, \delta, s', F')$

**Nachbedingung:** Für eine Eingabe  $w \in \Sigma^*$  liefern die Algorithmen zu  $A$  und  $A'$  das gleiche Ergebnis.

Zunächst definieren wir die folgenden Hilfsfunktionen:

$\varepsilon - closure : \mathcal{P}(Q) \rightarrow \mathcal{P}(Q)$  mit

$\varepsilon - closure(T) = \mathbf{Abgeschlossene\ H\u00fclle}(T \cup \{q' \in Q \mid (q, \varepsilon, q') \in \Delta, q \in T\}), (T \subseteq Q)$

$move : \mathcal{P}(Q) \times \Sigma \rightarrow \mathcal{P}(Q)$  mit

$$move(T, a) = \begin{cases} \perp & \nexists q \in Q \text{ mit } (q', a, q) \in \Delta, q' \in T \\ \{q \in Q \mid (q', a, q) \in \Delta, q' \in T\} & \text{sonst} \end{cases}$$

$A'$  wird wie folgt berechnet:

$Q' = \{\varepsilon - closure(\{s\})\}$

**while** (es gibt einen unmarkierten Zustand  $T$  in  $Q'$ ) {

markiere  $T$ ;

**for** jedes  $a \in \Sigma$  {

$U = \varepsilon - closure(move(T, a));$

if( $U \notin Q'$ ) {füge  $U$  zu  $Q'$  hinzu}

$\delta(T, a) := U$

} //for

} //while

$s' = \varepsilon - closure(\{s\});$

$F' = \{T \mid T \in Q' \wedge T \cap F \neq \emptyset\}$

□

**Beispiel.** Sei folgender Automat gegeben:

$A = (\{q_0, q_1, q_2, q_3, q_4\} (= Q),$

$\{a, b\} (= \Sigma),$

$\{(q_0, a, q_1), (q_1, b, q_0), (q_1, b, q_2), (q_2, \varepsilon, q_4), (q_4, \varepsilon, q_0), (q_2, a, q_3)\} (= \Delta),$

$q_0 (= s), \{q_1, q_3\} (= F))$

**Ablauf des Algorithmus:**

**Start:**  $Q' = \{\{q_0\}\}$

**while-Schleife 1.Schritt:**  $T = \{q_0\}$

$Q' = \{\boxed{\{q_0\}}\}$



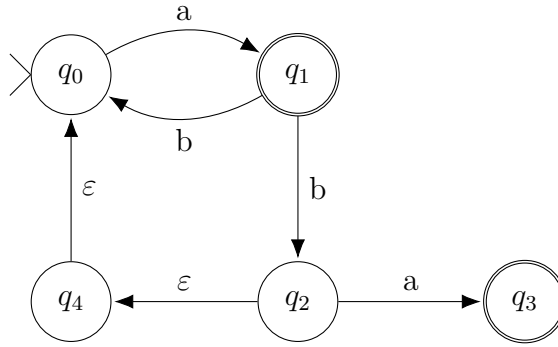


Abbildung 3.3: Graphische Darstellung von  $A$

$a \in \Sigma$ :

$$U = \varepsilon - \text{closure}(\text{move}(T, a)) = \{q_1\}$$

$$Q' = \left\{ \boxed{\{q_0\}}, \boxed{\{q_1\}} \right\}$$

$$\delta(\boxed{\{q_0\}}, a) := \{q_1\}$$

$b \in \Sigma$ :

$$\text{move}(T, b) = \perp$$

**while–Schleife 2.Schritt:**  $T = \{q_1\}$

$$Q' = \left\{ \boxed{\{q_0\}}, \boxed{\{q_1\}} \right\}$$

$a \in \Sigma$ :

$$\text{move}(T, a) = \perp$$

$b \in \Sigma$ :

$$U = \varepsilon - \text{closure}(\text{move}(T, b)) = \{q_0, q_2, q_4\}$$

$$Q' = \left\{ \boxed{\{q_0\}}, \boxed{\{q_1\}}, \boxed{\{q_0, q_2, q_4\}} \right\}$$

$$\delta(T, b) := \{q_0, q_2, q_4\}$$

**while–Schleife 3.Schritt:**  $T = \{q_0, q_2, q_4\}$

$$Q' = \left\{ \boxed{\{q_0\}}, \boxed{\{q_1\}}, \boxed{\{q_0, q_2, q_4\}} \right\}$$

$a \in \Sigma$ :

$$U = \varepsilon - \text{closure}(\text{move}(T, a)) = \{q_3, q_1\}$$

$$Q' = \left\{ \boxed{\{q_0\}}, \boxed{\{q_1\}}, \boxed{\{q_0, q_2, q_4\}}, \boxed{\{q_3, q_1\}} \right\}$$

$$\delta(T, a) := \{q_3, q_1\}$$

$b \in \Sigma$ :

$$\text{move}(T, b) = \perp$$

**while-Schleife 4.Schritt:**  $T = \{q_3, q_1\}$

$$Q' = \left\{ \boxed{\{q_0\}}, \boxed{\{q_1\}}, \boxed{\{q_0, q_2, q_4\}}, \boxed{\{q_3, q_1\}} \right\}$$

$a \in \Sigma$ :

$$\text{move}(T, a) = \perp$$

$b \in \Sigma$ :

$$U = \varepsilon - \text{closure}(\text{move}(T, b)) = \{q_2, q_0, q_4\}$$

$$Q' = \left\{ \boxed{\{q_0\}}, \boxed{\{q_1\}}, \boxed{\{q_0, q_2, q_4\}}, \boxed{\{q_3, q_1\}} \right\}$$

$$\delta(T, b) := \{q_2, q_0, q_4\}$$

$$s' = \{q_0\}$$

$$F' = \{\{q_1\}, \{q_1, q_3\}\}$$

Die graphische Darstellung des berechneten DEAs  $A' = (Q', \Sigma, \delta, s', F')$  ist in Figure 3.4 gegeben:

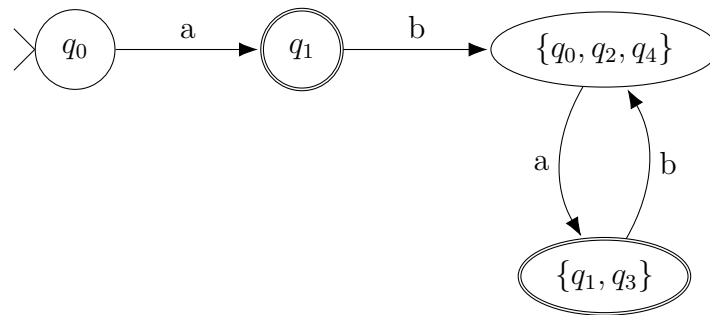


Abbildung 3.4: Graphische Darstellung von  $A'$

Es bleibt noch zu klären wie man aus einem regulären Ausdruck einen *NEA* konstruiert, der die zugehörige Sprache erkennt.

**Algorithmus 3.10** (*reg2auto*).

**Eingabe:**  $r \in R(\Sigma)$  (regulärer Ausdruck)

**Ausgabe:**  $A = (Q, \Sigma, \Delta, s, F)$  (NEA)

**Nachbedingung:**  $w \in \mathcal{L}(r) \Leftrightarrow \exists f \in F$ , so dass sich aus der Konfiguration  $(s, w)$  die Konfiguration  $(f, \varepsilon)$  ableitbar ist.

Wir geben den Algorithmus *reg2auto* durch rekursive Regeln an:

$\varepsilon$ :

$$\text{reg2auto}(\varepsilon) = (\{q_0, q_1\}, \Sigma, \{(q_0, \varepsilon, q_1)\}, q_0, \{q_1\})$$

$a \in \Sigma$ :

$$\text{reg2auto}(a) = (\{q_0, q_1\}, \Sigma, \{(q_0, a, q_1)\}, q_0, \{q_1\})$$

$\alpha, \beta \in R(\Sigma)$ :

$\text{reg2auto}(\alpha\beta) = \mathbf{let}$

$$(Q_1, \Sigma, \Delta_1, s_1, \{f_1\}) = \text{reg2auto}(\alpha)$$

$$(Q_2, \Sigma, \Delta_2, s_2, \{f_2\}) = \text{reg2auto}(\beta)$$

**in**

$$(Q_1 \dot{\cup} Q_2, \Sigma, (\Delta_1 \dot{\cup} \Delta_2 \dot{\cup} \{(f_1, \varepsilon, s_2)\}), s_1, \{f_2\})$$

**end**

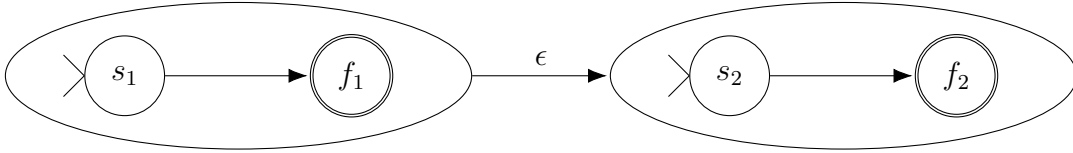


Abbildung 3.5: *reg2auto* Sequenz

$\text{reg2auto}(\alpha|\beta) = \mathbf{let}$

$$(Q_1, \Sigma, \Delta_1, s_1, \{f_1\}) = \text{reg2auto}(\alpha)$$

$$(Q_2, \Sigma, \Delta_2, s_2, \{f_2\}) = \text{reg2auto}(\beta)$$

**in**

$$((Q_1 \dot{\cup} Q_2 \dot{\cup} \{s_0, f_3\}, \Sigma, \Delta_1 \dot{\cup} \Delta_2 \dot{\cup} \{(s_0, \varepsilon, s_1), (s_0, \varepsilon, s_2), (f_1, \varepsilon, f_3), (f_2, \varepsilon, f_3)\}), s_0, \{f_3\})$$

**end**

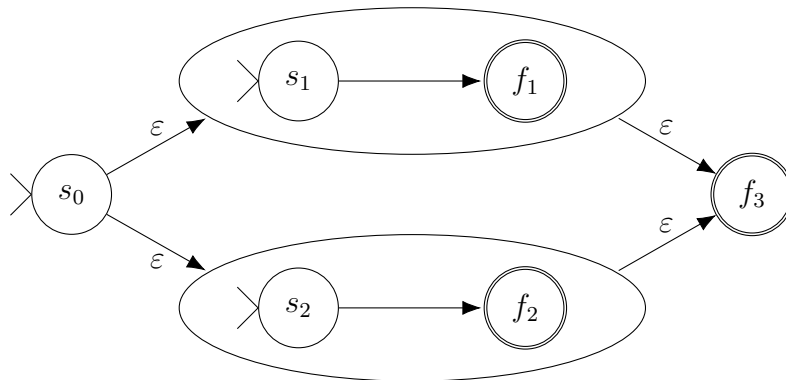


Abbildung 3.6: *reg2auto* Alternative

$\text{reg2auto}(\alpha^*) = \mathbf{let}$

$$(Q_1, \Sigma, \Delta_1, s_1, \{f_1\}) = \text{reg2auto}(\alpha)$$

**in**

$(Q_1, \Sigma, \Delta_1 \cup \{(s_1, \varepsilon, f_1), (f_1, \varepsilon, s_1)\}, s_1, \{f_1\})$   
**end**

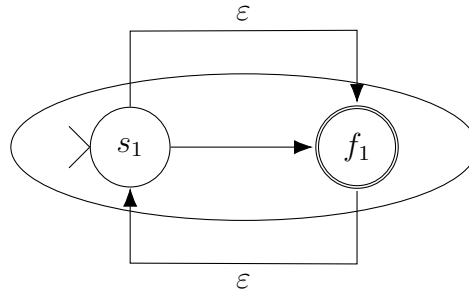


Abbildung 3.7: *reg2auto* Stern

**Beispiel.**  $r = a(ba)^*$

$reg2auto(r)$  :

$$reg2auto(a) = (\{q_0, q_1\}, \Sigma, \{(q_0, a, q_1)\}, q_0, \{q_1\})$$

$reg2auto((ba)^*)$  :

$reg2auto(ba)$  :

$$reg2auto(b) = (\{q_2, q_3\}, \Sigma, \{(q_2, b, q_3)\}, q_2, \{q_3\})$$

$$reg2auto(a) = (\{q_4, q_5\}, \Sigma, \{(q_4, a, q_5)\}, q_4, \{q_5\})$$

$$= (\{q_2, q_3, q_4, q_5\}, \Sigma, \{(q_2, b, q_3), (q_4, a, q_5), (q_3, \varepsilon, q_4)\}, q_2, \{q_5\})$$

$$= (\{q_2, q_3, q_4, q_5\}, \Sigma, \{(q_2, b, q_3), (q_4, a, q_5), (q_3, \varepsilon, q_4), (q_2, \varepsilon, q_5), (q_5, \varepsilon, q_2)\}, q_2, \{q_5\})$$

$$= (\{q_0, q_1, q_2, q_3, q_4, q_5\}, \Sigma,$$

$$\{(q_0, a, q_1), (q_2, b, q_3), (q_4, a, q_5), (q_3, \varepsilon, q_4), (q_2, \varepsilon, q_5), (q_5, \varepsilon, q_2), (q_1, \varepsilon, q_2)\}, q_0, \{q_5\})$$

Der berechnete Automat ist in Abbildung 3.8 dargestellt.

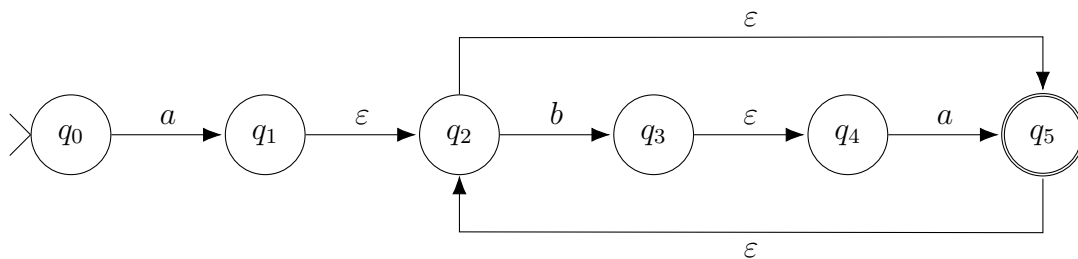


Abbildung 3.8: Ergebnis Beispiel zum Algorithmus *reg2auto*

**Definition 3.11** (Automat akzeptiert Sprache). Sei  $r \in R(\Sigma)$  ein regulärer Ausdruck und  $A$  ein Automat. Der Automat  $A$  *akzeptiert* die Sprache  $\mathcal{L}(r)$ , falls für alle  $w \in \mathcal{L}(r)$  aus der Startkonfiguration  $(s, w)$  eine Finalkonfiguration  $(f, \varepsilon)$  ableitbar ist, wobei  $s$  der Startzustand und  $f$  ein Finalzustand ist.

**Satz 3.12** (Korrektheit von *reg2auto*). Sei  $r \in R(\Sigma)$  und  $A = \text{reg2auto}(r)$ . Genau dann akzeptiert  $A$  die Sprache  $\mathcal{L}(r)$ .

*Beweis.* Der Beweis wird per Induktion geführt. □

**Satz 3.13.** Folgende Aussagen sind äquivalent:

- a) Es gibt einen regulären Ausdruck  $\alpha$ .
- b) Es gibt eine reguläre Grammatik  $G$  mit  $\mathcal{L}(G) = \mathcal{L}(\alpha)$ .
- c) Es gibt einen nichtdeterministischen endlichen Automaten der  $\mathcal{L}(\alpha)$  akzeptiert.
- d) Es gibt einen deterministischen endlichen Automaten der  $\mathcal{L}(\alpha)$  akzeptiert.

*Beweis.* Der Beweis wurde teilweise erbracht.

**a)  $\Rightarrow$  b):** Übungsaufgabe 2.6

**a)  $\Rightarrow$  c):** Algorithmus *reg2auto*, Satz 3.12

**c)  $\Rightarrow$  d):** Satz 3.8

Für die restlichen Fälle wird auf die Literatur verwiesen. □

### 3.3 Von regulären Sprachen zu Scannern

Im vorhergehenden Abschnitt haben wir nun kennengelernt, wie man reguläre Sprachen durch reguläre Ausdrücke definiert. Wir werden im Folgenden (Kapitel ??) Sprachen über Alphabete definieren, deren Symbole aus *Tokens* (reguläre Sprachen) bestehen. Elemente der Tokens nennt man *Lexeme*.

Wenn Textfiles, die mehrere Lexeme enthalten, gelesen werden sollen und für jedes Token ein regulärer Ausdruck existiert, für den ein NEA gebildet wurde, so ist nicht klar, welcher NEA zu Erkennung des nächsten Lexems benötigt wird. Zur Lösung dieses Problems benötigt man einen Steuerungsalgorithmus, der im DEA-Simulator (vgl. Abbildung 3.9) enthalten ist.

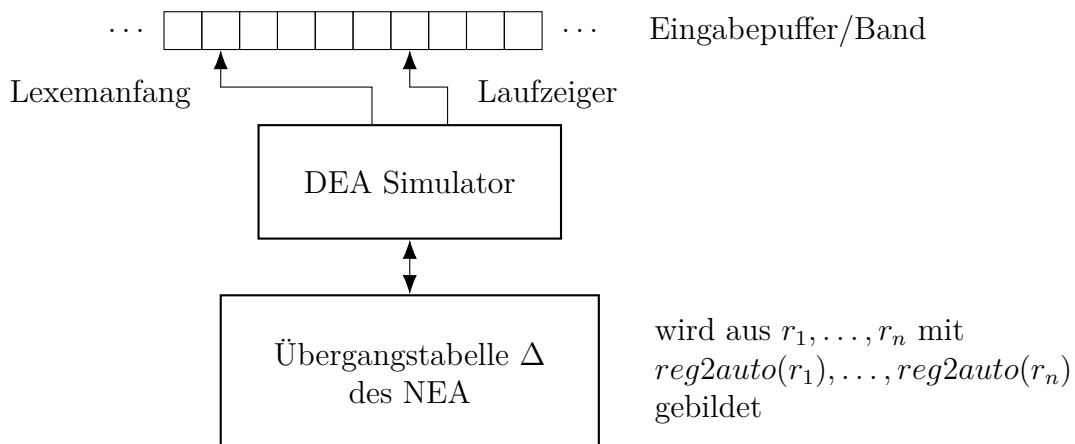


Abbildung 3.9: Scanvorgang eines Textfiles

Man definiert die Tokens einer Programmiersprache in einer *Lex-Spezifikation*:

$$\begin{aligned}
 & r_1 \{ \text{action 1} \} \\
 & \quad \vdots \\
 & r_n \{ \text{action n} \}
 \end{aligned}$$

$r_1 \dots r_n$  sind reguläre Ausdrücke über dem Alphabet  $\Sigma$  z.B. der ASCII-Zeichen. action 1, ..., action n sind Aktionen (programmiert in einer Programmiersprache), die ausgeführt werden, wenn ein gelesenes Lexem zum jeweiligen regulären Ausdruck passt.

### 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)

```

Problem: Oftmals ist es nicht eindeutig durch welche reguläre Ausdrücke ein erkanntes Lexem akzeptiert wird. (Bsp. `while` (man weiß nicht, ob `while` durch das Token `WHILE` oder das Token `IDENTIFIER` erkannt wird))

## Regel: Principle of Longest match

Es wird immer so weit gelesen, dass nach dem nächsten Zeichen kein regulärer Ausdruck der Lex-Spezifikation mehr passen würde.

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

### Beispiel.

whilei  $\Rightarrow$  Token *Identifler*

while  $\rightarrow$  Token *WHILE*

Wir entwickeln den DEA-Simulator in 2 Schritten:

Zunächst geben wir einen Simulator an, der nur 1 Lexem akzeptiert. Dieser wird dann zum vollständigen DEA-Simulator erweitert.

**Algorithmus 3.14.** Sei  $r \in R(\Sigma)$  ein regulärer Ausdruck und  $A = (Q, \Sigma, \Delta, s_0, F) = \text{reg2auto}(r)$ . Seien weiter  $\varepsilon$ -closure und  $move$  wie im Beweis zu Satz 3.8 gegeben.

**Eingabe:**  $w \in \Sigma^*$

**Ausgabe:**  $erg \in \{True, False\}$

**Nachbedingung:**  $erg = (w \text{ wird von } \text{reg2auto}(r) \text{ akzeptiert})$

```
s =  $\varepsilon$ -closure(s0)
a = nextchar() //liest nächstes Zeichen von Eingabestring
while(a  $\neq$  "") {
    s =  $\varepsilon$ -closure(move(s, a))
    a = nextchar()
}
if((s  $\cap$  F)  $\neq$   $\emptyset$ ) {return true;}
else {return false}
```

Es ist offensichtlich, dass dieser Algorithmus den Algorithmus aus dem Beweis zu Satz 3.8 simuliert. Der Algorithmus liefert true, wenn das Wort akzeptiert wird. Wenn  $move(s, a)$  undefiniert ist, liefert der Algorithmus eine Exception (wird als false interpretiert). Wenn der letzte Zustand kein Finalzustand ist, wird false zurückgegeben.

Nächster Schritt: Übergang von einem regulären Ausdruck zu endlich vielen regulären Ausdrücken.

**Algorithmus 3.15.** Seien in einer Lex-Spezifikation die regulären Ausdrücke  $r_1, \dots, r_n$  gegeben und seien:  $A_i = \text{reg2auto}(r_i)$  mit  $A_i = (Q_i, \Sigma, \Delta_i, s_i, F_i)$ . Wir fassen  $A_1, \dots, A_n$  zu einem NEA  $A$  zusammen:

$$A = ((\bigcup_i Q_i) \cup \{s_0\}, \Sigma, (\bigcup_i \Delta_i) \cup \Delta', s_0, (\bigcup_i F_i))$$

wobei  $\Delta' = \{(s_0, \varepsilon, s_i) \mid s_i, \text{Startzustand von } A_i\}$

Eine Veranschaulichung ist in Abbildung 3.10 zu sehen.

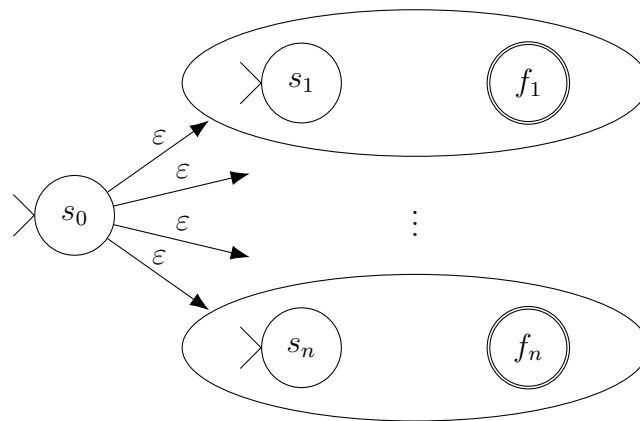


Abbildung 3.10: NEA für den Scanner

**Eingabe:**  $w$  (Eingabestring, z.B. Java - Datei) und  $A$  (konstruierter Automat)

**Vorbedingung:**  $w$  enthält nur Lexeme aus den Tokens der Lex-Spezifikation<sup>1</sup>

**Ausgabe:**  $((lex_1, f_1^*), \dots, (lex_m, f_m^*))$

**Nachbedingung:**  $w = lex_1 + lex_2 + \dots + lex_m$

$$s_1 = w$$

$$s_{i+1} = lex_{i+1} + \dots + lex_m$$

$lex_i$  ist das Lexem, das bei der Eingabe  $s_i$  erkannt wird (Principle of longest match)

$f_i^*$  ist der Finalzustand, der nach dem Lesen von  $lex_i$  erreicht wird.

$start = 0$  //Pointer auf das 1. Zeichen von  $w$

$a = \text{nextchar}()$

$pointer = 1$  //Pointer auf das 2. Zeichen von  $w$ .

$i = 1$  // Laufindex der die Lexeme zählt

**while** ( $a \neq eof$ ) { // End - Of - File

$\bar{s} = \varepsilon - \text{closure}(s_0)$

<sup>1</sup>Mögliche Fehleingaben können über ein Token der Fehleingaben abgefangen werden



```

while (move( $\bar{s}, a) \neq \perp$ ){
     $\bar{s} = \varepsilon - \text{closure}(\text{move}, (\bar{s}, a))$ 
     $a = \text{nextchar}()$ 
     $\text{pointer}++$ 
}
 $\text{lex}_i = w.\text{substring}(\text{start}, \text{pointer}-1)$ 
 $f_i^*$  = Finalzustand in  $\bar{s}$  mit kleinsten Index
 $\text{start} = \text{pointer} - 1$ 
 $i++$ 
}

```

**Beispiel.** Lex-Spezifikation:

```

(" |\t|\n) {;}
while {;}
(a|...|z)(a|...|z)* {;}

```

Aus der Lex-Spezifikation ergibt sich der NEA  $A$  in Abbildung 3.11.

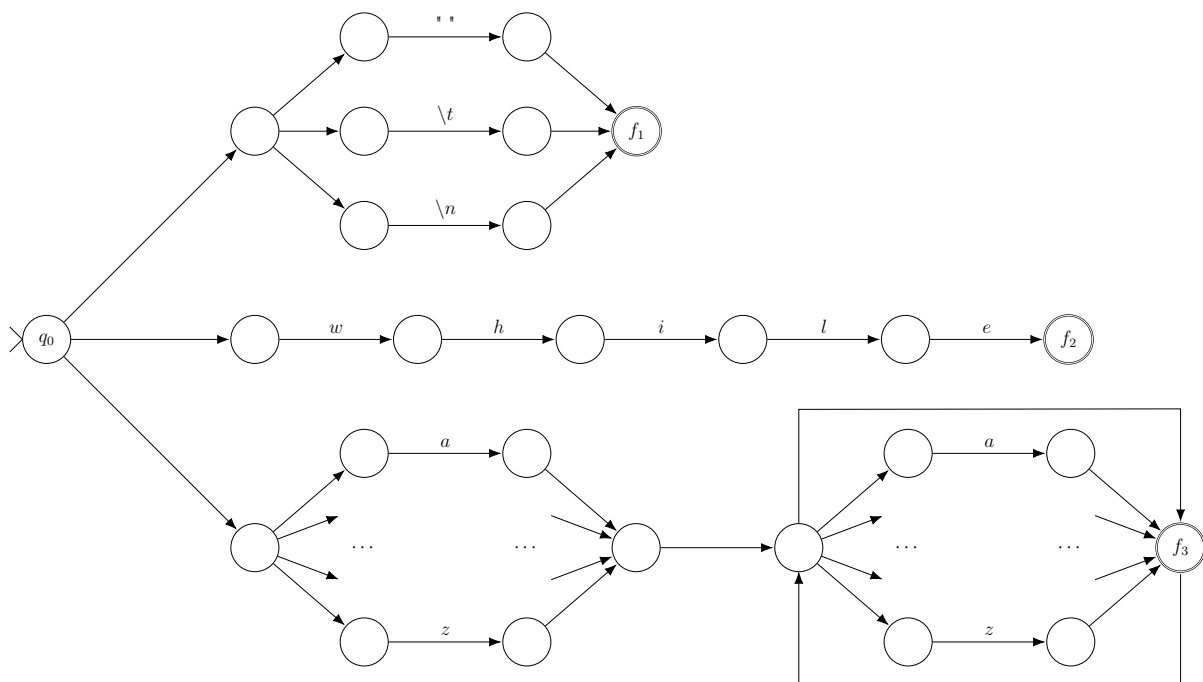


Abbildung 3.11: NEA der Lex-Spezifikation

$w = \text{"while whilez"}$

**Ablauf des Algorithmus:**

### 1. Lexem:

$start = 0$   
 $a = "w"$   
 $pointer = 1$   
 $i = 1$   
 $a = "h"$   
 $pointer = 2$   
 $a = "i"$   
 $pointer = 3$   
 $a = "l"$   
 $pointer = 4$   
 $a = "e"$   
 $pointer = 5$   
 $a = " "$   
 $pointer = 6$   
 $move(\bar{s}, " ") = \perp$   
 $lex_1 = w.substring(0, 5) = "while"$   
 $f_1^* = f_2$   
 $start = 5$   
 $i = 2$

### 2. Lexem:

$a = "w"$   
 $pointer = 7$   
 $move(\bar{s}, "w") = \perp$   
 $lex_2 = w.substring(5, 6) = " "$   
 $f_2^* = f_1$   
 $start = 6$   
 $i = 3$

### 3. Lexem:

$a = "h"$   
 $pointer = 8$   
 $a = "i"$   
 $pointer = 9$   
 $a = "l"$   
 $pointer = 10$   
 $a = "e"$   
 $pointer = 11$   
 $a = "z"$

```

pointer = 12
a = eof
pointer = 13
move( $\bar{s}$ , eof) =  $\perp$ 
lex3 = w.substring(6, 12) = "whilez"
f3* = f3

```

**Ergebnis:** ((*"while"*, f<sub>2</sub>), (" ", f<sub>1</sub>), (*"whilez"*, f<sub>3</sub>))

## 3.4 Der Scannergenerator JLex

JLex ist ein Tool, welches aus einer Lex-Spezifikation ein Java-Programm generiert, das den DEA-Simulator implementiert.

Die Actions der Lex-Spezifikation werden in Java ausprogrammiert.

### 3.4.1 Die JLex - Spezifikation

User Code (Java)

%%

JLex Direktive

%%

Lex-Spezifikation

### 3.4.2 JLex-Direktiven (Anweisungen)

**%class** *name* : definiert den Klassennamen *name* der erzeugten Klasse (default: Yylex)

**%function** *name* : definiert den Methodennamen *name*, die das nächste Lexem liest  
(default: yylex())

**%type** *name* : definiert den Rückgabebetyp von der Methode, die das nächste Lexem liest  
(default: Ytoken)

<pre> %{     Java - Code %} </pre>	: <i>Java - Code</i> wird an den Anfang der erzeugten Klasse kopiert
------------------------------------	--

```
%init{
    Java - Code
%init}
```

*Java - Code* : *Java - Code* wird in den Konstruktor der erzeugten Klasse kopiert.

```
%eof{
    Java - Code
%eof}
```

*Java - Code* : *Java - Code* wird ausgeführt bei Erreichen von dem Dateiende.

### 3.4.3 Java-Klasse erzeugen

Das Packet JLex2.jar muss in den CLASSPATH der Java-Umgebung aufgenommen werden (tcsh-Shell: setenv CLASSPATH .:Jlex2.jar).

Aufruf des Scannergenerators:

```
java -cp JLex2.jar JLex2.Main Datei (JLex-Spezifikation)
```

Dadurch wird die Klasse Datei.java erzeugt, die den DEA-Simulator für die JLex-Spezifikation Datei enthält

### 3.4.4 Beispiel: html-Lexeme

Als Beispiel geben wir eine JLex-Spezifikation für einen kleinen Ausschnitt der Sprache html an.

```
%%

%public
%class browserlexer
%type int
%eofval{
    System.out.println("EOF reached");
    return -1;
%eofval}

a = (a|A)
b = (b|B)
// ... hier noch weitere Abkürzungen

ws = [ \t\r\n\b\015]+

%%
```

```

"<(h|H)(t|T)(m|M)(l|L)">" { System.out.println(yytext()); }
"</(h|H)(t|T)(m|M)(l|L)">" { System.out.println(yytext()); }

"<{b}">"          { System.out.println(yytext()); }
"</{b}">"         { System.out.println(yytext()); }

[^\<]+            { System.out.println(yytext()); }
{ws}              { System.out.println(yytext()); }
.                 { System.out.println("FEHLER: "+yytext()); }

```

Nach den Direktiven werden durch `a`, `b` und `ws` Abkürzungen für reguläre Ausdrücke definiert. Als reguläre Ausdrücke werden die Tags `<html>`, `</html>` und `<b>`, `</b>` (jeweils alle Buchstaben klein und groß geschrieben) festgelegt. Weiterhin gibt es einen regulären Ausdruck, der alle Strings erkennt, die nicht mit `<` anfangen. Schließlich gibt es einen regulären Ausdruck `ws` für alle `Whitespaces` (Leerzeichen, Tabulatoren, ...) und einen regulären Ausdruck „.“ für alle Strings (außer *Newline*), die sonst nicht erfasst wurden.

```

class Main {
    public static void main(String args[]) throws java.io.
        IOException {
        browserlexer b = new browserlexer (new java.io.
            InputStreamReader(System.in));
        while (b.yylex() != -1);
    }
}

```

In der Methode `main` der Klasse `Main` wird eine Instanz der Klasse `browserlexer`, die aus der `JLex`-Spezifikation generiert wurde, erzeugt. Dann werden durch eine Schleife alle Lexeme nach dem *Principle of longest match* der Datei gelesen.

Mit folgendem `Makefile` kann man den Scanner generieren und compilieren.

```

Main.class: browserlexer.class Main.java
    javac Main.java

browserlexer.class: browserlexer.java
    javac browserlexer.java

browserlexer.java: browserlexer
    java -cp JLex2.jar JLex2.Main browserlexer

clean:
    rm *.class browserlexer.java

```

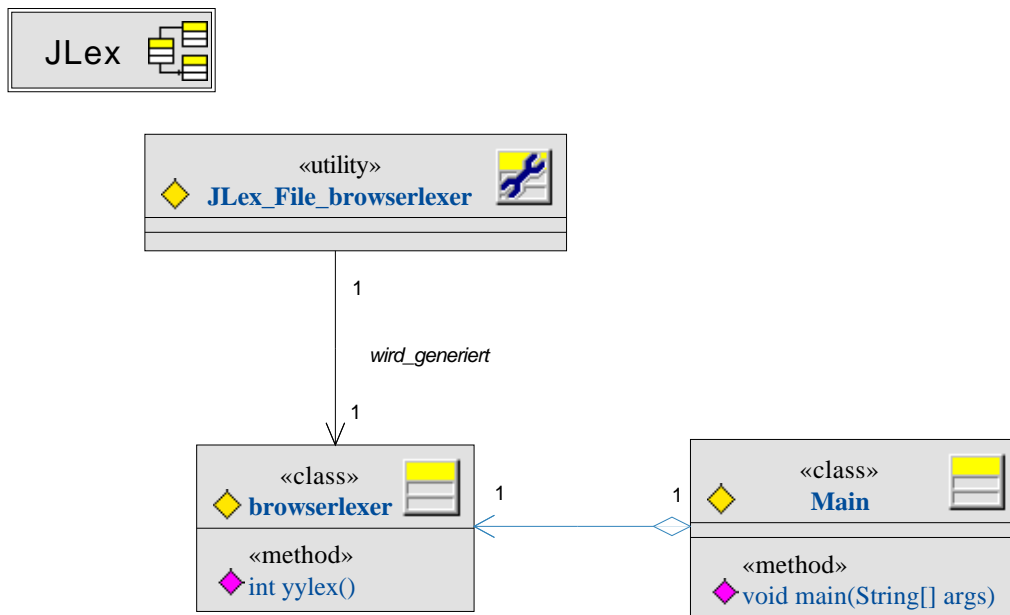


Abbildung 3.12: Klassen-Diagramm der Klassen des Scanners