Einführung in die Programmierung mit Java

Teil 5: Syntax

Martin Hofmann Steffen Jost

LFE Theoretische Informatik, Institut für Informatik,
Ludwig-Maximilians Universität, München

14. November 2017

LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Teil 5: Syntax

Syntax

Syntax: Festlegung des Satzbaus.

Beispiele syntaktisch falscher *deutscher Sätze*:

- *Kai liest eine Buch.*
- *Buch lesen Kai.*
- *Kai pr1&.*
- *Kai liest ein Buch, wenn ihr ist langweilig.*

Beispiele syntaktisch falscher *Java-Phrasen*:

```
{{x=1;};
if x==1 y=2;
```

Semantik

Semantik: Festlegung der Bedeutung eines Satzes.

Beispiele Semantische Fragen im *Deutschen*:

- *Worauf bezieht sich ein Relativpronomen?*
- *Welchen Einfluss haben Fragepartikel wie "eigentlich", "denn"?*
- *Wann verwendet man welche Zeitform?*

Beispiele Semantische Fragen bei *Java*:

- *In welcher Reihenfolge werden Ausdrücke ausgewertet?*
- *Wie werden Instanzvariablen initialisiert?*
- *Was ist ein Objekt?*

Grundfrage der Semantik von Programmiersprachen:

Welche Wirkung hat ein syntaktisch korrektes Programm?

Fragen der **Typüberprüfung** werden aus historischen Gründen ebenfalls der Semantik zugerechnet.

Formale Syntax

- Ein **Alphabet**, oft bezeichnet mit Σ, ist eine endliche Menge, deren Elemente **Symbole** genannt werden.
- Eine **Zeichenkette** (auch **Wort**) über Alphabet Σ ist eine endliche Folge von Elementen $\sigma_1, \ldots, \sigma_n$ aus Σ, mit $n \geq 0$.
  Man schreibt ein Wort als $\sigma_1\sigma_2 \ldots \sigma_n$.
  Der Fall $n = 0$ bezeichnet das **leere Wort** geschrieben ε.
- Die Menge aller Wörter über Σ wird mit Σ^* bezeichnet. Zu zwei Wörtern $w = \sigma_1 \ldots \sigma_n$ und $w' = \sigma'_1 \ldots \sigma'_m$ bildet man die **Verkettung** (Konkatenation) $ww' = \sigma_1 \ldots \sigma_n\sigma'_1 \ldots \sigma'_m$.
  Es gilt $\varepsilon w = w\varepsilon = w$ und $(ww')w'' = w(w'w'')$.
- Eine **formale Sprache** ist eine Teilmenge von Σ^*.

Beispiele

1. $\Sigma = \{a, b\}$.
   Wörter über Σ: $aaba$, $baab$, $bababa$, $baba$, ε, $bbbb$.
   Sprachen über Σ: $\{\}$, $\{a, b\}$, $\{a^n b^n \mid n \in \mathbb{N}\}$.

   Beachte: a^n bedeutet *hier*: $\underbrace{aa\ldots a}_{n\text{-Stück}}$.

2. $\Sigma = \{0, 1, \ldots, 9, \texttt{e}, \texttt{-}, \texttt{+}, \texttt{.}, \texttt{E}\}$.
   Wörter über Σ: `-1E98`, `--2e--`, `32.e`
   Sprachen über Σ: Syntaktisch korrekte `double` Konstanten, $\{\texttt{e}^n \mid n \text{ gerade}\}$.

3. $\Sigma = \{0, \ldots, 9, \texttt{if}, \texttt{while}, \texttt{;}, \texttt{\{}, \texttt{\}}, \ldots\}$
   Sprache über Σ: alle syntaktisch korrekten Java Programme.

Beispiel: Bezeichner

- **Bezeichner** sind Zeichenketten über dem Alphabet $\Sigma = \{\texttt{A}, \ldots, \texttt{Z}, \texttt{a}, \ldots, \texttt{z}, \texttt{0}, \ldots, \texttt{9}, \texttt{\_}\}$. Bezeichner müssen mit einem Buchstaben oder dem Zeichen `_` beginnen.
- Ein **Buchstabe** ist ein **Kleinbuchstabe** oder ein **Großbuchstabe**
- Ein **Kleinbuchstabe** ist ein Zeichen $\texttt{a}, \ldots, \texttt{z}$.
- Ein **Großbuchstabe** ist ein Zeichen $\texttt{A}, \ldots, \texttt{Z}$.
- Eine **Ziffer** ist ein Zeichen $\texttt{0}, \ldots, \texttt{9}$.

Backus-Naur Form

Backus Naur Form (**BNF**) für formelle Beschreibung von Sprachen:

$\langle name \rangle$	bezeichnet Menge von Sprachelementen
$\ldots \mid \ldots$	trennt mehrere Alternativen voneinander
`Courier`-Text	bezeichnet wörtliche Anteile
$[\ldots]$	bedeutet "optional"
$(\ldots)^+$	bedeutet "mindestens eins oder mehrere"
$(\ldots)^*$	bedeutet "keins oder mehrere"

Beispiel: Java-Statements

$$
\begin{array}{rcl}
\langle \mathit{statement} \rangle & ::= & \langle \mathit{type\_expression} \rangle\ \langle \mathit{ident} \rangle\ [\texttt{=}\ \langle \mathit{expression} \rangle]\texttt{;} \\
& \mid & \texttt{\{}\ (\langle \mathit{statement} \rangle)^+\ \texttt{\}} \\
& \mid & \texttt{if (}\ \langle \mathit{expression} \rangle\texttt{)}\ \langle \mathit{statement} \rangle\ [\texttt{else}\ \langle \mathit{statement} \rangle] \\
& \mid & \ldots
\end{array}
$$

Bemerkung: Strenggenommen verwenden wir hier die Erweiterte BNF (EBNF). Siehe z.B. Wikipedia.

BNF für gültige Bezeichner

Weiteres Beispiel: BNF für gültige Bezeichner

$$
\begin{array}{lcl}
\langle \textit{Bezeichner} \rangle & ::= & (\langle \textit{Buchst} \rangle \mid \texttt{\_})(\langle \textit{Buchst} \rangle \mid \langle \textit{Ziffer} \rangle \mid \texttt{\_})^* \\
\langle \textit{Buchst} \rangle & ::= & \langle \textit{KlBuchst} \rangle \mid \langle \textit{GrBuchst} \rangle \\
\langle \textit{KlBuchst} \rangle & ::= & \texttt{a} \mid \texttt{b} \mid \texttt{c} \mid \texttt{d} \mid \texttt{e} \mid \texttt{f} \mid \texttt{g} \mid \texttt{h} \mid \texttt{i} \mid \texttt{j} \mid \texttt{k} \mid \texttt{l} \mid \texttt{m} \mid \texttt{n} \mid \texttt{o} \\
 & & \mid \texttt{p} \mid \texttt{q} \mid \texttt{r} \mid \texttt{s} \mid \texttt{t} \mid \texttt{u} \mid \texttt{v} \mid \texttt{w} \mid \texttt{x} \mid \texttt{y} \mid \texttt{z} \\
\langle \textit{GrBuchst} \rangle & ::= & \texttt{A} \mid \texttt{B} \mid \texttt{C} \mid \texttt{D} \mid \texttt{E} \mid \texttt{F} \mid \texttt{G} \mid \texttt{H} \mid \texttt{I} \mid \texttt{J} \mid \texttt{K} \mid \texttt{L} \mid \texttt{M} \mid \texttt{N} \mid \texttt{O} \\
 & & \mid \texttt{P} \mid \texttt{Q} \mid \texttt{R} \mid \texttt{S} \mid \texttt{T} \mid \texttt{U} \mid \texttt{V} \mid \texttt{W} \mid \texttt{X} \mid \texttt{Y} \mid \texttt{Z} \\
\langle \textit{Ziffer} \rangle & ::= & \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9}
\end{array}
$$

double-Literale

Literal = Konstante in einer Programmiersprache.

$$
\begin{array}{lcl}
\langle \textit{double-Literal} \rangle & ::= & [\langle \textit{Vorzeichen} \rangle]\langle \textit{Mantisse} \rangle[\langle \textit{Exponent} \rangle] \\
\langle \textit{Vorzeichen} \rangle & ::= & \texttt{-} \mid \texttt{+} \\
\langle \textit{Mantisse} \rangle & ::= & (\langle \textit{Ziffer} \rangle)^+[\texttt{.}(\langle \textit{Ziffer} \rangle)^*] \mid \texttt{.}(\langle \textit{Ziffer} \rangle)^+ \\
\langle \textit{Exponent} \rangle & ::= & (\texttt{e} \mid \texttt{E})[\langle \textit{Vorzeichen} \rangle](\langle \textit{Ziffer} \rangle)^+
\end{array}
$$

Zusätzliche **Kontextbedingung**: Entweder ein Dezimalpunkt, oder ein e oder ein E muss vorhanden sein.

Übung: man verbessere die BNF Darstellung so, dass diese Kontextbedingung wegfallen kann.

Syntax der BNF

Eine **BNF-Grammatik** ist ein Quadrupel $G = (\Sigma, V, S, P)$.

- Σ ist die Menge der Terminalsymbole, meist in `Courier` gesetzt.
- V ist die Menge der Nichtterminalsymbole, meist in spitze Klammern gesetzt. Im Beispiel:
  $V = \{\langle \textit{double-Literal} \rangle, \langle \textit{Mantisse} \rangle, \langle \textit{Vorzeichen} \rangle, \langle \textit{Exponent} \rangle\}$.
- $S \in V$ ist ein ausgezeichnetes Nichtterminalsymbol, das **Startsymbol**. Im Beispiel: $S = \langle \textit{double-Literal} \rangle$.
- P ist eine endliche Menge von **Produktionen** der Form $X ::= \delta$, wobei δ eine **BNF-Satzform**, s.u., ist.

BNF-Satzformen

- Jedes Symbol in $V \cup \Sigma$ ist eine BNF Satzform.
- Sind $\gamma_1, \ldots, \gamma_n$ BNF-Satzformen, so auch $\gamma_1 \mid \cdots \mid \gamma_n$ (**Auswahl**).
- Sind $\gamma_1, \ldots, \gamma_n$ BNF-Satzformen, so auch $\gamma_1 \ldots \gamma_n$ (**Verkettung**).
- Ist γ eine BNF Satzform, so auch (γ) (**Klammerung**).
- Ist γ eine BNF Satzform, so auch $(\gamma)^*$ (**Iteration**).
- Ist γ eine BNF Satzform, so auch $(\gamma)^+$ (**nichtleere Iteration**).
- Ist γ eine BNF Satzform, so auch $[\gamma]$ (**Option**).

Eine Satzform der Gestalt $\gamma_1 \mid \cdots \mid \gamma_n$ muss immer geklammert werden, es sei denn, sie tritt unmittelbar als rechte Seite einer Produktion auf.

Semantik der BNF

Sei $G = (\Sigma, V, S, P)$ und $X \in V$ ein Nichtterminalsymbol. Ein Wort w ist aus X herleitbar ("ist ein X", "ist in $L(X)$"), wenn man es aus X durch die folgenden Ersetzungsoperationen erhalten kann:

- Falls $X ::= \gamma_1 \mid \cdots \mid \gamma_n$ eine Produktion ist, so darf man ein Vorkommen von X durch eines der γ_i ersetzen.
- Ein Vorkommen von $(\gamma_1 \mid \cdots \mid \gamma_n)$ darf man durch eines der γ_i ersetzen.
- Ein Vorkommen von $(\gamma)^*$ darf man durch $\overbrace{(\gamma)(\gamma)\ldots(\gamma)}^{n\text{-mal}}$ mit $n \geq 0$ ersetzen.
- Ein Vorkommen von $(\gamma)^+$ darf man durch $\overbrace{(\gamma)(\gamma)\ldots(\gamma)}^{n\text{-mal}}$ mit $n > 0$ ersetzen.
- Ein Vorkommen von (γ) darf man durch γ ersetzen, wenn γ nicht von der Form $\gamma_1 \mid \cdots \mid \gamma_n$ ist.
- Ein Vorkommen von $[\gamma]$ darf man durch (γ) ersetzen, oder ersatzlos streichen.

Beispiel

$$
\begin{aligned}
\langle \textit{double-Literal} \rangle &\to [\langle \textit{Vorzeichen} \rangle]\langle \textit{Mantisse} \rangle[\langle \textit{Exponent} \rangle] \\
&\to \langle \textit{Mantisse} \rangle[\langle \textit{Exponent} \rangle] \\
&\to (\langle \textit{Ziffer} \rangle)^+[\texttt{.}(\langle \textit{Ziffer} \rangle)^*][\langle \textit{Exponent} \rangle] \\
&\to (\langle \textit{Ziffer} \rangle)^+\,\texttt{.}(\langle \textit{Ziffer} \rangle)^*[\langle \textit{Exponent} \rangle] \\
&\to (\langle \textit{Ziffer} \rangle)^+\,\texttt{.}(\langle \textit{Ziffer} \rangle)(\langle \textit{Ziffer} \rangle)[\langle \textit{Exponent} \rangle] \\
&\to (\langle \textit{Ziffer} \rangle)\,\texttt{.}(\langle \textit{Ziffer} \rangle)(\langle \textit{Ziffer} \rangle)[\langle \textit{Exponent} \rangle] \\
&\to \texttt{2.71}\langle \textit{Exponent} \rangle \\
&\to \texttt{2.71}(\texttt{e} \mid \texttt{E})[\langle \textit{Vorzeichen} \rangle](\langle \textit{Ziffer} \rangle)^+ \\
&\to \texttt{2.71E}\langle \textit{Vorzeichen} \rangle(\langle \textit{Ziffer} \rangle)(\langle \textit{Ziffer} \rangle)(\langle \textit{Ziffer} \rangle) \\
&\to \texttt{2.71E}(\texttt{-} \mid \texttt{+})\texttt{001} \to \texttt{2.71E-001}
\end{aligned}
$$

Also $\texttt{2.71E-001} \in L(\langle \textit{double-Literal} \rangle)$.

Kontextbedingungen

Manche syntaktische Bedingungen lassen sich mit BNF nur schwer oder gar nicht formulieren.
Man gibt daher manchmal zusätzliche **Kontextbedingungen** an, denen die syntaktisch korrekten Wörter zusätzlich genügen müssen.
Beispiele:

- Bezeichner dürfen nicht zu den Schlüsselwörtern gehören wie z.B. `let`, `if`, etc.
- double-Literale müssen `.`, `e`, oder `E` enthalten.

Andere Bedingungen, wie korrekte Typisierung oder rechtzeitige Definition von Bezeichnern werden, wie schon gesagt, der Semantik zugerechnet.

Varianten

- Oft (und in der Java-Sprachdefinition) wird statt $(\gamma)^*$ auch $\{\gamma\}$ geschrieben und $(\gamma)^+$ wird nicht verwendet.
- Die spitzen Klammern zur Kennzeichnung der Nichtterminalsymbole werden oft weggelassen.
- Steht kein `Courier` Zeichensatz zur Verfügung, so schließt man die Terminalsymbole in "Anführungszeichen" ein.
- Statt des Produktionssymbols ::= wird manchmal, speziell auch in der Java-Sprachdefinition, ein einfacher Doppelpunkt verwendet.
- Die Java Sprachdefinition setzt Alternativen durch separate Zeilen ab anstatt durch $|$.
- BNF Grammatiken lassen sich auch grafisch in Form von **Syntaxdiagrammen** darstellen.

Ableitungsbäume

Ableitungen in einer BNF lassen sich grafisch durch **Ableitungsbäume** darstellen.
Diese Ableitungsbäume sind für die Festlegung der Semantik von Bedeutung.
Ein **Parser** berechnet zu einem vorgegebenen Wort einen Ableitungsbaum, falls das Wort in $L(S)$ ist, und erzeugt eine Fehlermeldung, falls nicht.
Diese Aufgabe bezeichnet man als **Syntaxanalyse**.

Struktur eines Compilers

Vereinfacht!

- Lexikalische Analyse:
  - Einlesen der Dateien
  - Zerlegung in **Tokens**
    (Bezeichner, Literale, Schlüsselworte, Operatorsymbole,...)
- Syntaxanalyse: Erstellen eines Syntaxbaumes
- Semantische Analyse (Typüberprüfung, Platzzuweisung)
- Optimierung
- Codeerzeugung

Java-Sprachdefinition (Fragment)

```
Statement:
        StatementWithoutTrailingSubstatement
        IfThenStatement
        IfThenElseStatement
        WhileStatement
        ForStatement

StatementWithoutTrailingSubstatement:
        Block
        EmptyStatement
        ExpressionStatement
        ReturnStatement
```

```
StatementNoShortIf:
        StatementWithoutTrailingSubstatement
        IfThenElseStatementNoShortIf
        WhileStatementNoShortIf
        ForStatementNoShortIf

IfThenStatement:
        if ( Expression ) Statement

IfThenElseStatement:
        if ( Expression ) StatementNoShortIf else Statement

IfThenElseStatementNoShortIf:
        if ( Expression ) StatementNoShortIf else
                                        StatementNoShortIf
```

```
WhileStatement:
        while ( Expression ) Statement

WhileStatementNoShortIf:
        while ( Expression ) StatementNoShortIf

Block:
        { [BlockStatements] }

BlockStatements:
        BlockStatement
        BlockStatements BlockStatement

BlockStatement:
        LocalVariableDeclarationStatement
        ClassDeclaration
        Statement
```

Parsergeneratoren

Erinnerung: *Parser* : $\Sigma^* \to$ *Ableitungsbäume* $\cup$ *Fehlermeldungen*.
Ein **Parsergenerator** erzeugt aus einer BNF-Grammatik automatisch einen Parser.

Der bekannteste Parsergenerator heißt "yacc" (*yet another compiler-compiler*). Er erzeugt aus einer BNF-Grammatik einen in der Programmiersprache C geschriebenen Parser.

Für Java gibt es "JavaCUP", "AntLR", u.a. Es wird ein in Java geschriebener Parser erzeugt.

Geschichte

Grammatikformalismen, die syntaktisch korrekte Sätze durch einen Erzeugungsprozess definieren (wie die BNF) heißen **generative Grammatiken**.
Sie gehen auf den Sprachforscher Noam Chomsky (1928–) zurück.

Eine **kontextfreie Chomsky-Grammatik** ist eine BNF-Grammatik ohne die Konstrukte $[\ ], |, (\ )^+, (\ )^*$. Man kann jede BNF-Grammatik durch eine kontextfreie Chomsky-Grammatik simulieren.

Chomsky betrachtete u.a. auch kontextsensitive Grammatiken.
Die Backus-Naur-Form wurde von den Informatikern John Backus und Peter Naur entwickelt, die die Bedeutung von Chomskys generativen Grammatiken für Programmiersprachensyntax erkannten.

Beispiel einer Chomsky Grammatik

$$
\begin{array}{lcl}
\langle \mathit{Bezeichner} \rangle & ::= & \langle \mathit{KlBuchst} \rangle \langle \mathit{Folge} \rangle \\
\langle \mathit{Bezeichner} \rangle & ::= & \_\ \langle \mathit{Folge} \rangle \\
\langle \mathit{Folge} \rangle & ::= & \langle \mathit{Zeichen} \rangle \langle \mathit{Folge} \rangle \\
\langle \mathit{Folge} \rangle & ::= & \varepsilon \\
\langle \mathit{Zeichen} \rangle & ::= & \text{'} \\
\langle \mathit{Zeichen} \rangle & ::= & \_ \\
\langle \mathit{Zeichen} \rangle & ::= & \langle \mathit{Buchst} \rangle \\
\langle \mathit{Zeichen} \rangle & ::= & \langle \mathit{Ziffer} \rangle \\
\langle \mathit{Buchst} \rangle & ::= & \langle \mathit{KlBuchst} \rangle \\
\langle \mathit{Buchst} \rangle & ::= & \langle \mathit{GrBuchst} \rangle
\end{array}
$$

Reguläre Ausdrücke

Eine rechte Seite einer BNF ohne Nichtterminalsymbole ist ein **regulärer Ausdruck**.

Formal werden reguläre Ausdrücke (über einem Alphabet Σ) wie folgt gebildet:

- Jedes Terminalsymbol $a \in \Sigma$ ist ein regulärer Ausdruck.
- ϵ ist ein regulärer Ausdruck.
- Sind $E_1, E_2, \ldots, E_n$ reguläre Ausdrücke, so auch $E_1 E_2 \ldots E_n$ und $E_1 \mid E_2 \mid \cdots \mid E_n$.
- Ist E regulärer Ausdruck, so auch E^* und E^+ und $[E]$

Zu jedem regulären Ausdruck E bezeichnet man mit $L(E)$ die durch ihn definierte Sprache über Σ.

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

- Wenn $E = (a|b)^* bbb$ so ist
  $L(E) = \{w \in \Sigma^* \mid w \text{ hört mit } bbb \text{ auf}\}$.
- Wenn $E = (ab \mid aabb \mid aab(ab)^* b)^*$ so besteht $L(E)$ aus allen Wörtern w, sodass gilt:
  - w enthält genausoviele a's wie b's.
  - Ist u ein Anfangsstück von w, so enthält u mindestens soviele a's wie b's und die Zahl der a's übersteigt die der b's um höchstens zwei.
- Dieselbe Sprache wird auch durch $(a(ab)^* b)^*$ definiert.

Bemerkung zur Tradition

Traditionell erlaubt man Konkatenation und Alternative nur für $n = 2$ und fügt ε (leeres Wort) $\{\}$ (leere Sprache) explizit hinzu.

Außerdem sind E^+ und $[E]$ "ublicherweise nicht Teil der offiziellen Syntax für reguläre Ausdrücke und werden durch EE^* und $E \mid \epsilon$ wiedergegeben.

Deterministische endliche Automaten

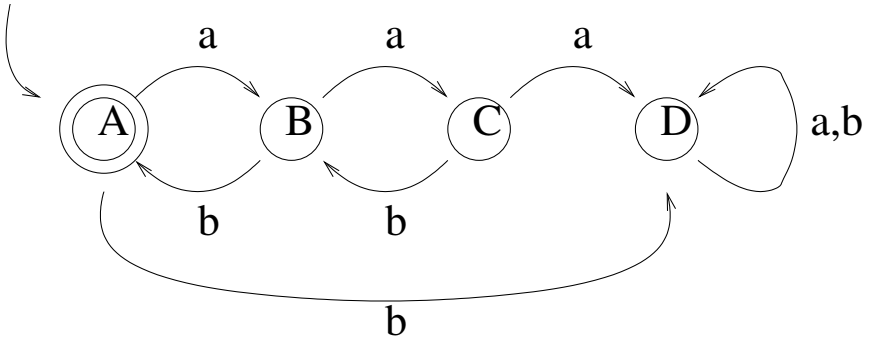
Ein **deterministischer endlicher Automat (DEA)** über einem Alphabet Σ besteht aus

- Einer endlichen Menge Q von **Zuständen**
- Für jeden Zustand $q \in Q$ und Zeichen $\sigma \in \Sigma$ ein Folgezustand $\delta(q, \sigma) \in Q$. Die Zuordnung $(q, \sigma) \mapsto \delta(q, \sigma)$ ist die Zustandsüberführungsfunktion.
- Einem Anfangszustand $q_0 \in Q$
- Einer Teilmenge $F \subseteq Q$ von **akzeptierenden Zuständen**.

Formal ist ein DEA also ein Tupel $A = (\Sigma, Q, \delta, q_0, F)$.

Beispiel

Grafische Darstellung eines Automaten:



Formal: $Q = \{A, B, C, D\}$, $q_0 = A$ und $F = \{A\}$,

δ	a	b
A	B	D
B	C	A
C	D	B
D	D	D

Lauf von A auf *aababaabab*:

	a	*a*	*b*	*a*	*b*	*a*	*a*	*b*	*a*	*b*
A	*B*	*C*	*B*	*C*	*B*	*C*	*D*	*D*	*D*	*D*

. . . und auf *ababaababb*:

	a	*b*	*a*	*b*	*a*	*a*	*b*	*a*	*b*	*b*
A	*B*	*A*	*B*	*A*	*B*	*C*	*B*	*C*	*B*	*A*

Akzeptierte Sprache

Gegeben ein endlicher Automat $A = (\Sigma, Q, q_0, F)$ und ein Wort $w = \sigma_0\sigma_1 \cdots \sigma_{n-1}$.
Der **Lauf** von A auf w ist die durch Abarbeiten von w gemäß δ von q_0 aus entstehende Zustandsfolge:

$$q_0, q_1, q_2, \ldots, q_n$$

wobei q_0 der Anfangszustand ist und $q_{i+1} = \delta(q_i, \sigma_i)$.
Der Automat **akzeptiert** das Wort w, wenn $q_n \in F$, wenn also nach Abarbeitung des Wortes w ein Zustand aus F erreicht wird.
Die Sprache $L(A)$ des Automaten A umfasst alle Wörter, die A akzeptiert.
Im Beispiel ist die Sprache gerade die Sprache $L\big((a(ab)^*b)^*\big)$

Automat in Java

```
import java.util.Scanner;
public class AutomatTest {
    public static void main(String[] args) {
        Scanner console     = new Scanner(System.in);
        MeinAutomat automat = new MeinAutomat();
        String eingabe      = console.nextLine();
        boolean fehler      = false;

        for (int i = 0; !fehler && i<eingabe.length(); i++) {
            if (eingabe.charAt(i) == 'a')
                automat.lies_a();
            else if (eingabe.charAt(i) == 'b')
                automat.lies_b();
            else fehler = true;
        }
        if (fehler)
            System.out.println("Falsche Eingabe.");
        else if (automat.hatAkzeptiert())
            System.out.println("Akzeptiert.");
        else
            System.out.println("Verworfen.");       }}
```

Der Automat selbst

```
public class MeinAutomat {
    private int zustand;

    private final int A = 0;
    private final int B = 1;
    private final int C = 2;
    private final int D = 3;

    public MeinAutomat() {
        zustand = A; }

    public void lies_a() {
        if (zustand == A) zustand = B;
        else if (zustand == B) zustand = C;
        else if (zustand == C) zustand = D;
        else if (zustand == D) zustand = D; }
```

```
    public void lies_b() {
        if (zustand == A) zustand = D;
        else if (zustand == B) zustand = A;
        else if (zustand == C) zustand = B;
        else if (zustand == D) zustand = D; }

    public boolean hatAkzeptiert() {
        return zustand == A;    }

    public void zuruecksetzen() {
        zustand = A; }
}
```

Nichtdeterministische endliche Automaten

Ein **nichtdeterministischer endlicher Automat (NEA)** ist ein Tupel

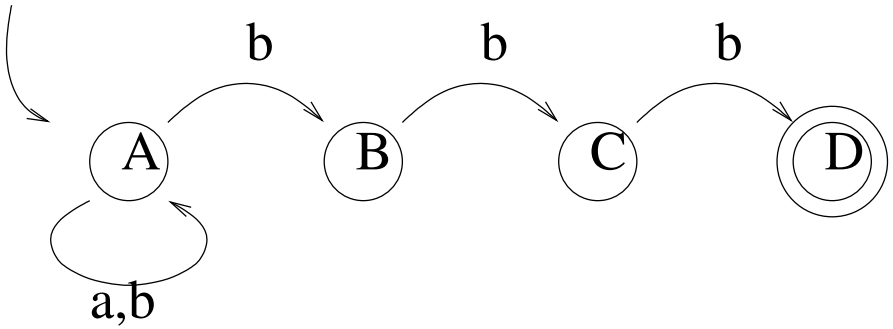
$$A = (\Sigma, Q, q_0, \delta, F)$$

wobei δ einem Zustand q und Symbol σ eine **Menge** von Zuständen $\delta(q, \sigma)$ zuordnet.

Ein Lauf eines NEA auf einem Wort $w = \sigma_0\sigma_1 \ldots \sigma_{n-1}$ ist dann eine Zustandsfolge $q_0, q_1, \ldots, q_n$ sodass $q_{i+1} \in \delta(q_i, \sigma_i)$.

Zu ein und demselben Wort gibt es i.a. mehrere Läufe.
Der NEA akzeptiert ein Wort, wenn ein Lauf existiert, der in einem akzeptierenden Zustand endet.

Beispiel



Formal: $Q = \{A, B, C, D\}$, $q_0 = A$ und $F = \{D\}$,

δ	a	b
A	{A}	{A,B}
B	{ }	{C}
C	{ }	{D}
D	{ }	{ }

Akzeptierender Lauf von A auf *aabbabbb*:

	a	*a*	*b*	*b*	*a*	*b*	*b*	*b*
A	*A*	*A*	*A*	*A*	*A*	*B*	*C*	*D*

Sprache des Automaten: $L(A) = L((a|b)^* bbb)$

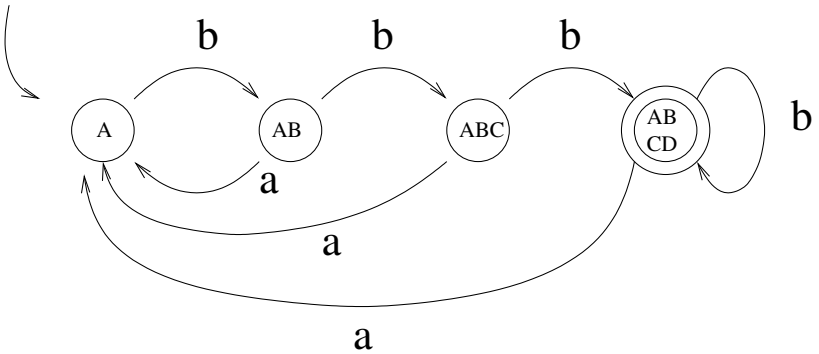
Äquivalenz von DEA, NEA, Reg.Ausdr.

Satz (ohne Beweis):

DEAs, NEAs und reguläre Ausdrücke haben die gleiche Stärke:

- Zu jedem regulären Ausdruck E existiert ein NEA A mit $L(A) = L(E)$.
- Zu jedem NEA A existiert ein DEA A' mit $L(A) = L(A')$ (allerdings hat A' im allgemeinen sehr viel mehr Zustände als A!).
- Zu jedem DEA A existiert ein regulärer Ausdruck E mit $L(E) = L(A)$.

DEA für $(a|b)^* bbb$.



Die Zustände des DEA entsprechen Mengen von Zuständen des vorher gezeigten NEA.

Reguläre Sprachen

Eine Sprache $L \subseteq \Sigma^*$ ist **regulär**, wenn ein DEA A (alternativ NEA oder regulärer Ausdruck) existiert mit $L = L(A)$.
Nicht jede durch eine BNF beschreibbare Sprache ist regulär.
Beispiel: Korrekt geklammerte arithmetische Ausdrücke.
Sprachen, die von BNF oder kontextfreien Grammatiken beschrieben werden, heißen **kontextfrei**.
Nicht jede Sprache ist kontextfrei: Beispiel $\{ww \mid w \in \Sigma^*\}$

Zusammenfassung Syntax

- Formale Sprachen sind einfach Mengen von Zeichenketten.
- BNF ist ein System zur Definition formaler Sprachen.
- Ein Parser berechnet zu gegebener BNF B und Wort w entweder einen Ableitungsbaum oder liefert die Information, dass w nicht zur von B definierten Sprache gehört.
- Reguläre Ausdrücke sind ein Spezialfall der BNF und erlauben die Definition regulärer Sprachen.
- Deterministische endliche Automaten beschreiben auch reguläre Sprachen und lassen sich unmittelbar implementieren.
- Nichtdeterministische endliche Automaten erlauben kürzere Beschreibungen und lassen sich durch Einführung von Zustandsmengen als DEA implementieren.