# 2 Automaten und formale Sprachen

Jeder weiß, was eine Sprache ist, auch wenn sich dieser Begriff nur schwierig definieren lässt. Zum einen dient eine Sprache der Kommunikation. Zum anderen ist eine gesprochene oder geschriebene Sprache ein System, das aus Zeichen und Regeln besteht. Diese Regeln beschreiben, wie aus den Zeichen ein Satz der Sprache entsteht.

Auch eine formale Sprache dient der Kommunikation. Programmiersprachen sind formale Sprachen, die der Kommunikation des Programmierers mit dem Computer oder anderen Programmierern dienen. Ebenso gibt es Regeln, mit denen sich entscheiden lässt, ob ein Programmtext wohlgeformt ist oder nicht.

Im allgemeinsten Fall ist eine formalen Sprache eine Menge von Wörtern. Für praktische Zwecke nützlich sind jedoch nur formale Sprachen, die sich durch endlich viele Regeln beschreiben lassen. Abhängig von der Art dieser Regeln werden die formalen Sprachen in unterschiedliche Klassen eingeteilt. Die einfachsten formalen Sprachen sind die regulären Sprachen (Abschnitt 2.2), gefolgt von den kontextfreien Sprachen (Abschnitt 2.3), gefolgt von den Typ-0-Sprachen (Abschnitt 2.5).

Zur Beschreibung formaler Sprachen gibt es zwei grundlegende Möglichkeiten:

- Eine Sprache wird aus Regeln *erzeugt*. Ein entsprechender Formalismus sind die Grammatiken. Diese beschreiben, wie Wörter der Sprache durch das Anwenden von Regeln aus einem Startsymbol entstehen. Die Sprache ist definiert durch die Menge aller Wörter, die sich dadurch erzeugen lassen.

- Ein Wort wird analysiert und als der Sprache zugehörig *akzeptiert* oder nicht akzeptiert. Ein entsprechender Formalismus sind die Automaten. Ein Automat verarbeitet ein Wort Zeichen für Zeichen und entscheidet, ob das Wort der Sprache angehört. Die Sprache ist definiert durch die Menge aller Wörter, die der Automat akzeptiert.

Beide Formalismen sind gleichwertig. Unterschiedlich ist ihr Zweck: Eine Grammatik wird verwendet, um eine formale Sprache zu konstruieren, der zugehörige Automat, um diese Sprache zu verarbeiten. Die Automatentheorie ist daher die Grundlage zur Verarbeitung formaler Sprachen auf dem Computer und damit insbesondere der Konstruktion von Compilern und Interpretern.

## 2.1 Formale Sprachen als Wortmenge

Zunächst definieren wir den Zeichenvorrat einer formalen Sprache.

**Definition 2.1.1.** Ein *Alphabet* $\Sigma$ ist eine endliche, nicht leere Menge. Jedes $a \in \Sigma$ heißt *Terminalsymbol* oder *Buchstabe*.

B. Hollas, *Grundkurs Theoretische Informatik,*
DOI 10.1007/978-3-662-47278-1_2, 

Terminalsymbole betrachten wir als kleinste Einheiten, die nicht weiter unterteilt werden sollen. Was diese kleinsten Einheiten und damit das Alphabet sind, hängt von der jeweiligen Anwendung ab.

**Beispiel 2.1.2.** Wir betrachten Alphabete, die für verschiedene Zwecke geeignet sind:

- Mit dem Alphabet {a,...,z} lassen sich die Schlüsselwörter einer Programmiersprache darstellen.
- Zur Darstellung von DNA-Sequenzen ist das Alphabet $\{a, c, g, t\}$ geeignet.
- Das Alphabet $\{0, \ldots, 9, +, -, *, /\ \mathrm{sqr}, \exp, \log\}$ enthält Symbole, die in arithmetische Ausdrücken vorkommen. ◁

Wörter erhalten wir, indem Terminalsymbole aneinandergefügt werden.

**Definition 2.1.3.** Sei $\Sigma$ ein Alphabet.

- Eine endliche Folge $w = w_1 w_2 \ldots w_n$ von Terminalsymbolen $w_1, w_2, \ldots, w_n \in \Sigma$ heißt *Wort* (über dem Alphabet $\Sigma$).
- Die *Länge* $|w|$ eines Wortes $w$ ist die Anzahl Terminalsymbole, aus denen $w$ besteht.
- Das Wort $\varepsilon$ der Länge null heißt *leeres Wort*.
- $\Sigma^*$ ist die Menge aller Wörter über $\Sigma$, einschließlich $\varepsilon$.
- $\Sigma^+ = \Sigma^* - \{\varepsilon\}$ ist die Menge aller Wörter mit positiver Länge.

**Beispiel 2.1.4.**

- Jedes Schlüsselwort einer Programmiersprache ist ein Wort über dem Alphabet {a,...,z}.
- Die Folge **if** $(x = 0)\ y := 1$ ist ein Wort über dem Alphabet {**if**, (, ), **:=**, **=**, $x$, $y$, 0, 1}.
- Der Satz „die Katzen schlafen“ ist formal ein Wort über dem Alphabet {die, Katzen, schlafen}. Die Bedeutung des Begriffs „Wort“ in Definition 2.1.3 ist hier eine andere als im Deutschen. ◁

Das leere Wort $\varepsilon$ ist nach Definition 2.1.3 eine Folge von null Terminalsymbolen. Es entspricht dem Leerstring in einer Programmiersprache. Das leere Wort werden wir weiter unten nochmals betrachten.

Beachten Sie, dass $\varepsilon$, $\{\varepsilon\}$, $\emptyset$ drei verschiedene Dinge sind: $\varepsilon$ ist ein Wort, $\{\varepsilon\}$ eine Menge, die nur aus dem leeren Wort besteht und $\emptyset$ ist die leere Menge.

Die Mengen $\Sigma^*$ und $\Sigma^+$ enthalten unendlich viele Wörter, da $\Sigma$ nach Definition 2.1.1 nicht leer ist:

**Beispiel 2.1.5.** Für das Alphabet $\Sigma = \{a, b\}$ ist $\Sigma^* = \{\varepsilon, a, b, aa, ab, ba, bb, aaa, \ldots\}$. ◁

Nun können wir eine formale Sprache definieren als Menge von Wörtern über einem Alphabet:

**Definition 2.1.6.** Eine *formale Sprache* über einem Alphabet $\Sigma$ ist eine Teilmenge von $\Sigma^*$.

Diese Definition ist jedoch sehr allgemein. Auch eine unstrukturierte, „unsinnige" Wortmenge ist damit eine formale Sprache. In der Informatik sind formale Sprachen mit einer besonderen Struktur von Interesse. In den folgenden Abschnitten werden Formalismen vorgestellt, mit denen sich formale Sprachen, die im Allgemeinen unendlich sind, endlich beschreiben lassen.

**Beispiel 2.1.7.** Folgende Mengen sind formale Sprachen (vgl. Beispiel 2.1.2):

- Die Menge $\{a, c, g, t\}^+$ aller Bruchstücke von DNA-Sequenzen über dem Alphabet $\Sigma = \{a, c, g, t\}$.
- Die Menge aller arithmetischen Ausdrücke über dem Alphabet $\{0, \ldots, 9, +, -, *, /, \text{sqr}, \exp, \log\}$. ◁

Wenn man Wörter aneinanderfügt, entstehen neue Wörter. Wir nennen dies Konkatenation (dies entspricht dem String-Operator „+" in Java).

**Definition 2.1.8.** Seien $A, B$ Sprachen und $v, w$ Wörter.

- Das Wort $vw$ heißt *Konkatenation* der Wörter $v, w$.
- Das Wort $w^n$ ist die $n$-fache Konkatenation von $w$. Dabei ist $w^0 = \varepsilon$.
- Die *Konkatenation* (oder das *Produkt*) der Sprachen $A, B$ ist $AB = \{vw \mid v \in A, w \in B\}$.
- Die Sprache $A^n$ ist die $n$-fache Konkatenation von $A$. Dabei ist $A^0 = \{\varepsilon\}$.

Für das leere Wort $\varepsilon$ und alle Wörter $w$ gilt

$$\varepsilon w = w\varepsilon = w$$

Das leere Wort $\varepsilon$ ist damit das neutrale Element bezüglich der Konkatenation (vergleichbar mit der 1 für die Multiplikation). Ebenso gilt

$$\{\varepsilon\}A = A\{\varepsilon\} = A$$

für alle Sprachen $A$.

Mit diesen Operatoren lassen sich auf einfache Weise regelmässig aufgebaute Mengen konstruieren. Diese Idee wird in Abschnitt 2.2.6 mit den regulären Ausdrücken weitergeführt.

**Beispiel 2.1.9.** Sei $N_0 = \{0, 1, \ldots, 9, 10, \ldots\}$ die Sprache aller Dezimalzahlen aus $\mathbb{N}_0$. Dann ist $\{-\}N_0$ die Sprache aller negativen Dezimalzahlen und der Null und $Z = N_0 \cup \{-\}N_0$ die Sprache aller Dezimalzahlen aus $\mathbb{Z}$. Für $n \geq 0$ ist $Z(\{+\}Z)^n$ die Sprache aller Summen von Dezimalzahlen mit $n + 1$ Summanden. ◁

Mit Definition 2.1.8 erhalten wir eine äquivalente Darstellung der Mengen $\Sigma^*$ und $\Sigma^+$. Da $\Sigma^n$ die Menge aller Wörter der Länge $n$ ist, können wir $\Sigma^*$ und $\Sigma^+$ schreiben als Vereinigung:

$$\Sigma^* = \bigcup_{n \geq 0} \Sigma^n \qquad \text{sowie} \qquad \Sigma^+ = \bigcup_{n \geq 1} \Sigma^n$$

Ferner ist $\Sigma^*$ abgeschlossen unter Konkatenation: Für Wörter $w_1, w_2 \in \Sigma^*$ ist auch $w_1 w_2 \in \Sigma^*$. Daher heißt $\Sigma^*$ auch *Konkatenationsabschluss* der Menge $\Sigma$.

### Aufgaben

**2.1.1**① Bestimmen Sie $\emptyset^*$, $\{\varepsilon\}^*$, $\{a\}^*$.

**2.1.2**① Sei $\Sigma = \{a, b, c\}$ und $V = \{S, T, U\}$. Geben Sie für jede der folgenden Mengen drei Elemente mit verschiedener Länge an, die in der Menge liegen, und drei Elemente mit verschiedener Länge, die nicht in der Menge liegen:

a) $\Sigma \cup \{\varepsilon\} \cup \Sigma V$

b) $(V \cup \Sigma)^* V (V \cup \Sigma)^*$

**2.1.3**① Seien $S, T, U$ Mengen. Zeigen Sie: $S(T \cup U) = ST \cup SU$.

**2.1.4**② Zeigen Sie, dass $\Sigma^*$ abzählbar unendlich ist.

**2.1.5**② Zeigen Sie, dass die Menge aller Sprachen überabzählbar ist.

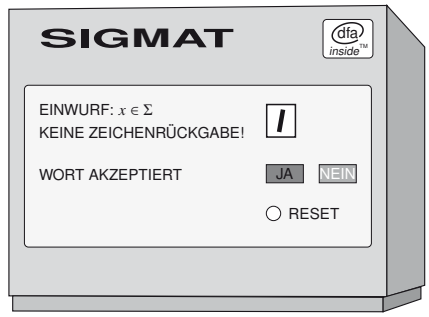
**2.1.6**④ Zeigen Sie formal: $(\Sigma^*)^* = \Sigma^*$. Verwenden Sie dabei $\Sigma^* = \bigcup_{n \geq 0} \Sigma^n$.

## 2.2 Reguläre Sprachen

Reguläre Grammatiken, endliche Automaten sowie reguläre Ausdrücke (Abschnitt 2.2.6) beschreiben genau die regulären (Typ-3) Sprachen. Die endlichen Automaten umfassen die deterministischen und die nichtdeterministischen endlichen Automaten. Ein Automat ist ein Formalismus, der einen Zwischenschritt auf dem Weg zur maschinellen Verarbeitung einer formalen Sprache darstellt. Zwischen diesen Formalismen bestehen enge Zusammenhänge, die im Folgenden dargestellt werden.

### 2.2.1 Deterministische endliche Automaten (DFAs)

Während sich mit einer Grammatik ein Wort in einer Folge von Ableitungsschritten erzeugen lässt, verarbeitet ein Automat ein Wort und akzeptiert dieses oder nicht. Die Eingabe des Automaten ist eine Folge von Zeichen, die schrittweise verarbeitet werden. Nach jedem verarbeiteten Zeichen kann der Automat in einen beliebigen Zustand übergehen und die Folge der bisher verarbeiteten Zeichen entweder (i) akzeptieren oder (ii) nicht akzeptieren. Jeder Zustand, in dem sich der Automat im Fall (i) befindet, heißt *Endzustand*. Vor der Eingabe des ersten Zeichens befindet sich der Automat im *Startzustand*. Die einzige Möglichkeit, sich Dinge zu merken, besteht für den Automaten darin, in einen von endlich vielen Zuständen überzugehen.




Die Funktionsweise eines endlichen Automaten lässt sich am einfachsten durch einen gerichteten Graphen beschreiben, dessen Kanten mit Zeichen aus $\Sigma$ beschriftet sind. Jeder Knoten des Graphen entspricht einem Zustand des Automaten. Der Startzustand ist durch einen eingehenden Pfeil gekennzeichnet, Endzustände durch zwei Kreise.

**Beispiel 2.2.1.** Der Automat $M_1$ in Abbildung 2.1 akzeptiert alle Wörter über $\Sigma = \{a, b, c\}$, die *abc* enthalten. Für jede Eingabe aus $\Sigma^*$ startet $M_1$ im Startzustand $z_0$ und liest nacheinander alle Zeichen der Eingabe, wobei $M_1$ entsprechend der Kantenbeschriftung Zustände wechselt. Der Folgezustand kann dabei auch der selbe Zustand sein. Sobald $M_1$ den Endzustand $z_E$ erreicht, hat $M_1$ die Eingabe akzeptiert. Für die

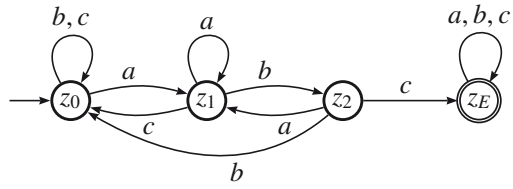



**Abb. 2.1** DFA $M_1$

Eingabe $w = cbaababcc$ durchläuft $M_1$, beginnend mit dem Startzustand $z_0$, die Zustände $z_0, z_0, z_0, z_1, z_1, z_2, z_1, z_2, z_E, z_E$. Da der letzte Zustand ein Endzustand ist, akzeptiert $M_1$ das Wort $w$. ◁

Die Erkennung von Zeichenketten ist eine der wichtigsten Anwendungen endlicher Automaten. Im Compilerbau werden endliche Automaten für die lexikalische Analyse (Abschnitt 2.2.9) verwendet.

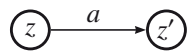



**Abb. 2.2** Überführungsfunktion $\delta$ mit $\delta(z, a) = z'$

Der Automat aus dem eben betrachteten Beispiel ist ein DFA *(Deterministic Finite Automaton)*, was bedeutet, dass der Automat nach jedem gelesenen Zeichen in genau einen Folgezustand wechselt. Für die graphische Darstellung heißt das, dass von keinem Knoten zwei mit dem gleichen Zeichen beschriftete Kanten ausgehen.

Ein DFA $M$ besitzt genau einen Startzustand und mindestens einen Endzustand. Die von $M$ akzeptierte Sprache $L(M)$ besteht aus allen Eingaben $w \in \Sigma^*$, so dass $M$ nach dem Lesen von $w$ einen Endzustand erreicht.

**Definition 2.2.2.** Ein *DFA* ist ein Tupel $M = (Z, \Sigma, \delta, z_0, E)$, wobei gilt

- $Z$ ist die Menge der Zustände.
- $\Sigma$ ist das Eingabealphabet.
- $\delta : Z \times \Sigma \to Z$ ist die *Überführungsfunktion*. Dabei bedeutet $\delta(z, a) = z'$, dass $M$ im Zustand $z$ für die Eingabe $a$ in den Zustand $z'$ wechselt (Abbildung 2.2).
- $z_0 \in Z$ ist der Startzustand.
- $E \subseteq Z$ ist die Menge der Endzustände.

**Beispiel (Fortsetzung).** Der DFA $M_1$ lässt sich formal beschreiben durch $M_1 = (Z, \Sigma, \delta, z_0, E)$ mit $Z = \{z_0, z_1, z_2, z_E\}$, $\Sigma = \{a, b, c\}$, $E = \{z_E\}$ und $\delta$ wie in folgender Tabelle angeben:

| $\delta$ | $z_0$ | $z_1$ | $z_2$ | $z_E$ |
|---|---|---|---|---|
| $a$ | $z_1$ | $z_1$ | $z_1$ | $z_E$ |
| $b$ | $z_0$ | $z_2$ | $z_0$ | $z_E$ |
| $c$ | $z_0$ | $z_0$ | $z_E$ | $z_E$ |

◁

Wir definieren nun die Sprache $L(M)$ als Menge aller Wörter $w$, für die der DFA $M$, gestartet im Zustand $z_0$ mit Eingabe $w$, einen Endzustand erreicht. Dazu definieren wir die erweiterte Überführungsfunktion $\hat{\delta}$. $\hat{\delta}(z, w)$ gibt an, in welchen Zustand sich $M$ befindet, nachdem $M$, ausgehend vom Zustand $z$, das Wort $w$ gelesen hat.

**Definition 2.2.3.** Sei $M = (Z, \Sigma, \delta, z_0, E)$ ein DFA.

- Die *erweiterte Überführungsfunktion* $\hat{\delta} : Z \times \Sigma^* \to Z$ von $M$ ist definiert durch

$$\hat{\delta}(z, w) = \begin{cases} z & \text{für } w = \varepsilon \\ \hat{\delta}(\delta(z, a), x) & \text{für } w = ax \text{ mit } a \in \Sigma, x \in \Sigma^* \end{cases}$$

- Die von $M$ *akzeptierte Sprache* ist $L(M) = \{w \in \Sigma^* \mid \hat{\delta}(z_0, w) \in E\}$.

Wenn wir die rekursive Definition der erweiterte Überführungsfunktion $\hat{\delta}$ expandieren, erhalten wir $\hat{\delta}(z, a_1a_2 \ldots a_n) = \delta(\ldots\delta(\delta(z, a_1), a_2)\ldots, a_n)$.

**Beispiel (Fortsetzung).** Mit $\delta$ wie in obiger Tabelle angegeben erhalten wir

$$\begin{aligned}
\hat{\delta}(z_0, cbaababcc) &= \hat{\delta}(\delta(z_0, c), baababcc) = \\
\hat{\delta}(z_0, baababcc) &= \hat{\delta}(\delta(z_0, b), aababcc) = \\
\hat{\delta}(z_0, aababcc) &= \hat{\delta}(\delta(z_0, a), ababcc) = \\
\hat{\delta}(z_1, ababcc) &= \hat{\delta}(\delta(z_1, a), babcc) = \\
\hat{\delta}(z_1, babcc) &= \hat{\delta}(\delta(z_1, b), abcc) = \\
\hat{\delta}(z_2, abcc) &= \hat{\delta}(\delta(z_2, a), bcc) = \\
\hat{\delta}(z_1, bcc) &= \hat{\delta}(\delta(z_1, b), cc) = \\
\hat{\delta}(z_2, cc) &= \hat{\delta}(\delta(z_2, c), c) = \\
\hat{\delta}(z_E, c) &= \hat{\delta}(\delta(z_E, c), \varepsilon) = \\
\hat{\delta}(z_E, \varepsilon) &= z_E
\end{aligned}$$

und damit $cbaababcc \in L(M_1)$. In dieser Ableitung erhalten wir genau die Zustandsfolge, die der DFA für die Eingabe *cbaababcc* durchläuft. ◁

Aus Definition 2.2.3 erhält man eine Methode, um einen DFA zu implementieren. In folgendem Code muss die Überführungsfunktion $\delta$ als zweidimensionales Array `delta` und die Menge der Endzustände als Array `finalStates` gegeben sein. Damit lässt sich $\hat{\delta}$ als rekursive Funktion darstellen. Der Aufruf `w.substring(1)` liefert das Wort `w` ohne das erste Zeichen. Wenn der Startzustand des DFA 0 ist, kann damit festgestellt werden, ob ein Wort zu der vom DFA akzeptierten Sprache gehört. Dieses Verfahren besitzt eine Laufzeit in $O(|w|)$, da in jedem Schritt ein Zeichen von $w$ verarbeitet wird.

```
int deltaHat(int z, String w) {
  if(w.length() == 0) return z;
  else return deltaHat(delta[z][w.charAt(0)], w.substring(1));
}

boolean isFinalState(int z) {
  return finalStates.asList.contains(z);
}

boolean accepted(String w) {
  int startState = 0;
  return isFinalState(deltaHat(startState, w));
}
```

Wenn ein Unicode-Zeichensatz verwendet wird, enthält das Array `delta` allerdings $2^{16}$ Spalten. Es lassen sich Zeit und Platz bei der Konstruktion des DFA sparen, wenn die Zeichen auf ein kleineres Alphabet abgebildet werden.

## Aufgaben

**2.2.1**① Geben Sie einen DFA an, der alle Wörter über dem Alphabet $\{a, n, s, x\}$ akzeptiert, die auf ananas enden.

**2.2.2**② Geben Sie alle Sprachen an, die in $\{L(M) \mid L(M)$ wird akzeptiert von einem DFA mit genau einem Zustand$\}$ liegen.

**2.2.3**② Sei $M = (Z, \Sigma, \delta, z_0, E)$ ein DFA und $M' = (Z, \Sigma, \delta, z_0, Z)$ . Zeigen Sie: $L(M') = \Sigma^*$.

**2.2.4**② Konstruieren Sie einen DFA über $\Sigma = \{a, b, c\}$, der alle Strings akzeptiert, die *aba* oder *acb* enthalten.

Welche Zeit braucht der DFA, um *aba* oder *acb* in einem String der Länge $n$ zu finden?

**2.2.5**④ Sei $L = \{x \in \{0, 1\}^+ \mid$ Die Binärzahl $x$ ist durch 3 teilbar$\}$. Geben Sie einen DFA $M$ an mit $L(M) = L$.

## 2.2.2 Nichtdeterministische endliche Automaten (NFAs)

Nichtdeterminismus ist ein wichtiges Konzept in der Automatentheorie, das auch bei den Kellerautomaten (Abschnitt 2.3.1) und den Turing-Maschinen (Abschnitt 2.5.5) verwendet wird. Von zentraler Bedeutung ist der Unterschied zwischen deterministischen und nichtdeterministischen Turing-Maschinen in der Komplexitätstheorie (Abschnitt 3.3).

In Beispiel 2.2.1 oder in Aufgabe 2.2.1 besteht eine Schwierigkeit darin, die Rückwärtskanten des DFA zu konstruieren. Die Rückwärtskanten sind die Übergänge, die zu einem bereits durchlaufenen Zustand führen, wenn mit dem gelesenen Zeichen ein Präfix des gesuchten Wortes nicht verlängert werden kann. Bei der Konstruktion eines nichtdeterministischen Automaten besteht dieses Problem nicht. Für viele Probleme ist es einfacher, einen nichtdeterministische Automaten anzugeben als einen deterministischen Automaten.

Ein nichtdeterministischer endlicher Automat oder *NFA (Nondeterministic Finite Automaton)* ist eine Verallgemeinerung eines deterministischen endlichen Automaten (DFA). Während ein DFA für jedes Paar aus Zustand und gelesenem Zeichen genau einen Folgezustand besitzt, besitzt ein NFA beliebig viele Folgezustände. In der graphischen Darstellung wird dies durch ebenso viele mit dem gleichen Zeichen beschriftete Kanten ausgedrückt, die von einem Knoten (Zustand) ausgehen (Abbildung 2.3). Die Anzahl der möglichen Folgezustände für ein gelesenes Zeichen kann auch null sein.

**Beispiel 2.2.4.** Der NFA $M_2$ in Abbildung 2.4 akzeptiert die gleiche Sprache wie der DFA $M_1$ aus Beispiel 2.2.1. Verglichen mit dem DFA $M_1$ ist der NFA $M_2$ wesentlich einfacher konstruiert, da keine Rückwärtskanten notwendig sind. ◁

Wie ist dieser Nichtdeterminismus zu verstehen? Eine Möglichkeit besteht darin, einen NFA als ein Modell zur Beschreibung zulässiger Zustandsfolgen zu betrachten.

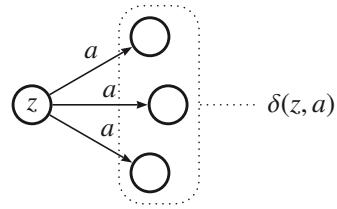



**Abb. 2.3** Überführungsfunktion eines NFA

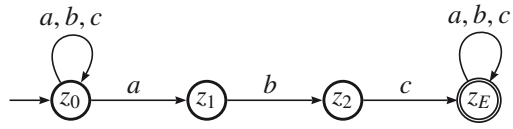



**Abb. 2.4** NFA $M_2$

So wie eine Straßenkarte ein Modell für die möglichen Wege durch eine Stadt ist, so ist ein NFA ein Modell, das mögliche Wege im NFA bzw. Zustandsfolgen bei der Verarbeitung eines Wortes beschreibt. Der NFA akzeptiert eine Eingabe, wenn es dabei eine Zustandsfolge gibt, die zu einem Endzustand führt. So wie auch eine Straßenkarte nicht „weiß“, welche Abzweigung an einer Kreuzung zu nehmen ist, so „weiß“ auch ein NFA nicht, welcher Folgezustand auszuwählen ist. Wir können lediglich ablesen, welche Übergänge möglich sind und welche nicht.

⚠ Ein NFA „weiß“ nicht, welcher Folgezustand auszuwählen ist. Ein NFA ist ein abstraktes Modell, das mögliche Zustandsfolgen beschreibt. Ein NFA lässt sich nicht unmittelbar als Programm implementieren.

**Beispiel (Fortsetzung).** Für die Eingabe $w = cbaababcc$ kann $M_2$ die Zustandsfolge $z_0, z_0, z_0, z_0, z_0, z_0, z_1, z_2, z_E, z_E$ durchlaufen und damit einen Endzustand erreichen. Daher wird $w$ von $M_2$ akzeptiert. Für die Eingabe *abcabc* gibt es zwei Zustandsfolgen, die $M_2$ durchlaufen kann, um einen Endzustand zu erreichen ($z_0, z_0, z_0, z_0, z_1, z_2, z_E$ sowie $z_0, z_1, z_2, z_E, z_E, z_E, z_E$). Dagegen gibt es für die Eingabe *abba* keine Zustandsfolge, mit der $M_2$ einen Endzustand erreichen kann. Daher wird *abba* nicht von $M_2$ akzeptiert. ◁

Oft werden Formulierungen wie „der NFA wählt nichtdeterministisch ein Folge von Zuständen aus, die zu einem Endzustand führt.“ verwendet. Diese ist in dem Sinne zu verstehen, dass für die gegebene Eingabe eine derartige Zustandsfolge existiert. Dies ist vergleichbar mit der Auswahl von Regeln einer Grammatik, um ein gegebenes Wort abzuleiten.

Eine andere Möglichkeit zum Verständnis des Nichtdeterminismus besteht darin, diesen als parallele Berechnung zu betrachten. Wenn es mehrere Folgezustände gibt, teilt sich die Berechnung in mehrere Zweige auf, die unabhängig voneinander verfolgt werden. Wenn einer dieser Zweige einen Zustand erreicht, der keinen Folgezustand besitzt, wird die Berechnung dieses Zweigs beendet. Erreicht ein Zweig mit dem letzten Zeichen der Eingabe einen Endzustand, dann wird die Eingabe akzeptiert.

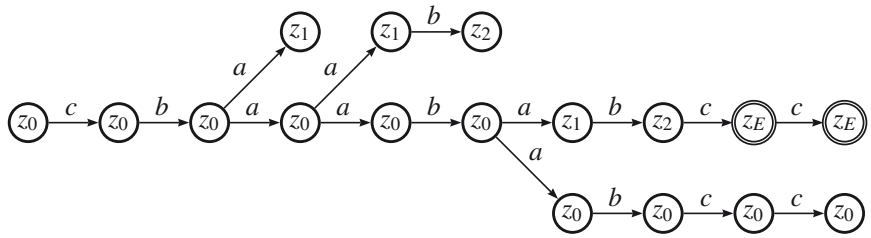



**Abb. 2.5** Berechnungsbaum des NFA $M_2$ für die Eingabe $w = cbaababcc$

**Beispiel (Fortsetzung).** Der Berechnungsbaum für die Eingabe $w = cbaababcc$ ist in Abbildung 2.5 dargestellt. Da es einen Zweig gibt, der in einem Endzustand endet und der alle Zeichen von $w$ enthält, akzeptiert $M_2$ die Eingabe $w$. ◁

Den beiden Betrachtungsweisen des Nichtdeterminismus entsprechen zwei Möglichkeiten, den Übergang des NFA in Folgezustände formal zu definieren. Dies lässt sich zum einen durch eine Überführungsrelation $\delta$ definieren, die Tripel der Form (Zustand, Zeichen, Folgezustand) enthält. Die andere Möglichkeit besteht darin, eine Überführungsfunktion $\delta$ zu definieren, in der die möglichen Folgezustände zu einer Menge zusammengefaßt sind: Für einen Zustand $z$ und ein Zeichen $a \in \Sigma$ ist $\delta(z, a)$ die Menge von Zuständen, in die $M$ übergehen kann (Abbildung 2.3). Damit ist die Überführungsfunktion eines NFA eine Abbildung in die Potenzmenge von $Z$. Überführungsrelation und Überführungsfunktion sind gleichwertige Formalismen. Wir verwenden hier die Überführungsfunktion zur Definition eines NFA.

**Definition 2.2.5.** Ein *NFA* ist ein Tupel $M = (Z, \Sigma, \delta, S, E)$, wobei gilt

- $Z$ ist die Menge der Zustände.
- $\Sigma$ ist das Eingabealphabet.
- $\delta : Z \times \Sigma \to \mathcal{P}(Z)$ ist die *Überführungsfunktion*. Dabei bedeutet $z' \in \delta(z, a)$, dass $M$ im Zustand $z$ für die Eingabe $a$ in den Zustand $z'$ wechseln kann.
- $S \subseteq Z$ ist die Menge der Startzustände.
- $E \subseteq Z$ ist die Menge der Endzustände.

Ein DFA können wir als Spezialfall eines NFA mit einem Startzustand und $|\delta(z, a)| = 1$ für alle $z \in Z$, $a \in \Sigma$ betrachteten. Für $\delta(z, a) = \emptyset$ besitzt ein NFA keinen Folgezustand. Dies lässt sich als „Sackgasse“ oder erfolgloses Ende eines Berechnungszweiges interpretieren.

**Beispiel (Fortsetzung).** Der NFA $M_2$ lässt sich formal beschreiben durch $M_2 = (Z, \Sigma, \delta, S, E)$ mit $Z = \{z_0, z_1, z_2, z_E\}$, $\Sigma = \{a, b, c\}$, $S = \{z_0\}$, $E = \{z_E\}$ und $\delta$ wie in folgender

http://www.springer.com/978-3-662-47277-4

Grundkurs Theoretische Informatik
Mit Aufgaben und Anwendungen
Hollas, B.
2015, VIII, 192 S. 40 Abb., Softcover
ISBN: 978-3-662-47277-4