Theoretische Informatik

Friedhelm Seutter
Institut für Angewandte Informatik
Fachhochschule Braunschweig/Wolfenbüttel

1. September 2016

Inhaltsverzeichnis

Einleitung

Die *Theoretische Informatik* ist einer der wesentlichen Bestandteile eines Informatikstudiums. Sie liefert eine mathematisch-formale Grundlage zum Entwurf und zur Modellierung von Hardware- und Softwaresystemen und den damit verbundenen Algorithmen. Die klassischen Teilgebiete der Theoretischen Informatik sind:

- Berechenbarkeit (Turing 1936, Kleene 1936, Church 1936)
- Automatentheorie (Mealy 1954, Moore 1955, Kleene 1956)
- Formale Sprachen (Chomsky 1959)
- Komplexitätstheorie (Hartmanis, Stearns 1965, Cook 1969)

Die Namen und Jahreszahlen verweisen dabei auf die Hauptvertreter, auf die die Etablierung der jeweiligen Teilgebiete zurückzuführen ist.

Die zentrale Frage, die diese Teilgebiete miteinander verbindet und die durch diese beantwortet werden soll, ist die Folgende:

> *Welches sind die grundsätzlichen Fähigkeiten und Beschränkungen von Computern und Softwaresystemen?*

Die Automatentheorie und die Formalen Sprachen werden dabei meist zusammen behandelt, da bestimmte Automatentypen mit bestimmten Sprachtypen in einer sehr engen Beziehung zueinander stehen. Die Automatentheorie behandelt die Definitionen und Eigenschaften diverser Berechnungsmodelle. Ein solcher Automat wird dann zur Berechnung einer Funktion oder zur Analyse der Struktur einer Sprache eingesetzt. Die Definition von formalen Sprachen erfolgt über Grammatiken. Unterschiedliche Sprachtypen ergeben sich durch unterschiedliche Grammatiktypen.

Zur Beantwortung der Frage

> *Ist ein Problem durch ein Computer oder ein Softwaresystemen lösbar?*

wird ein spezieller Automat eingeführt, eine so genannte *Turingmaschine*. Der Begriff „lösbar“ wird hier im Sinne von „algorithmisch berechenbar“ verstanden. Mittels Turingmaschinen können Algorithmen formalisiert werden, so dass diese mit mathematischen Methoden analysiert und damit viele Ergebnisse auch im mathematischen Sinne bewiesen werden können. Zentrale Ergebnisse sind die *Church-Turing-These* und die *Unentscheidbarkeit des Halteproblems*.

Auch die Komplexität von Problemen und sie lösenden Algorithmen wird mittels Turingmaschinen analysiert.

> *Warum sind einige Probleme schwer (aufwändig) zu berechnen und andere leicht?*

Hier werden die Zeit- und Platzbedarfe bestimmt, die zur Lösung von Problemen im schlechtesten Fall notwending sind. Die Einteilung in *effizient* lösbare und *nicht-effizient* lösbare Probleme führt zu dem so genannten $\mathcal{P}$-$\mathcal{NP}$-*Problem* und den NP-*vollständigen* Problemen, die eine Problemklasse von in einem gewissen Sinne schwersten Problemen bilden.

Das Lernziel der Beschäftigung mit diesen Fragestellungen ist es, einen Überblick über die grundlegenden Modelle und Methoden der Theoretischen Informatik zu bekommen. Diese sollen kennen gelernt und verstanden, in ihren fachlichen Kontext eingeordnet und in einfachen Beispielen angewendet werden können. Im Einzelnen werden in den jeweiligen Kapiteln die folgenden Themen behandelt:

- Alphabete, Wörter und formale Sprachen
- Endliche Automaten und Nichtdeterminismus
- Reguläre Ausdrücke und Sprachen
- Kontextfreie Grammatiken und Sprachen
- Kellerautomaten
- Turingmaschinen, Berechenbarkeit und Entscheidbarkeit

Die Begriffe werden teils informell erläutert, teils formal definiert. Für das Studium – insbesondere die Programmierausbildung – und die Praxis können diese theoretischen Modelle grundlegende Erkenntnisse und Hinweise zur Lösung diverser Probleme liefern.

Auf eine Einführung in die Komplextitätstheorie, die – wie oben aufgeführt – auch ein Teilgebiet der Theoretischen Informatik ist, soll hier verzichtet werden. Dies würde den zeitlich zur Verfügung stehenden Rahmen sprengen oder es müßten andere Themen gestrichen werden. Da aber die Komplexitätsanalysen von Problemen und Algorithmen auch in anderen Lehrveranstaltungen behandelt werden, die anderen Themen der Theoretischen Informatik aber eher nicht, ist hier diese Auswahl getroffen worden.

Kapitel 1

Formale Sprachen

1.1 Alphabete, Wörter und Sprachen

Die Arbeitsweise von Rechnern besteht im Wesentlichen darin, Zeichenfolgen eines bestimmten Zeichenvorrats zu be- und verarbeiten. Die Verarbeitungsanweisungen und die zu bearbeitenden Werte werden in Form von Programmen und Daten an einen Rechner gegeben. Programme und Daten werden durch Zeichenfolgen repräsentiert. Die Kommunikation mit und unter Rechnern erfolgt also über Zeichenfolgen.

Auch unabhängig von Rechnern bedienen wir uns – zumindest in schriftlicher Form – Zeichenfolgen zur Kommunikation. Der Zeichenvorrat und die Regeln, nach denen die Zeichenfolgen zu konstruieren sind, definieren eine *Sprache*. Der Zeichenvorrat wird auch *Alphabet* genannt. Allgemein kann eine Sprache wie folgt beschrieben werden:

> *Sprache ist der Gebrauch gleichbleibender Zeichen zur Verständigung, ... Sprache ist ein Vorrat von sinnlich wahrnehmbaren Zeichen, die allein oder nach bestimmmten Kombinationsregeln untereinander verbunden der Kommunikation dienen. Von der ... Lautsprache leiten sich ... Zeichensysteme ab: Schreib- und Druckschrift, Morse- und Blindenschrift, Flaggen- und Verkehrszeichen, die Trommelsprache, mathematische und chemische Formelsprache u. a.* [1]

So basieren z. B. die deutsche und die englische Schriftsprache auf dem lateinischen Alphabet. Aus den Zeichen des Alphabets werden Wörter und daraus dann Sätze gebildet. Wörter und Sätze haben eine Bedeutung, eine *Semantik*. Während der griechischen Sprache nur ein anderes Alphabet zugrunde liegt, hat die chinesische Sprache einen vollständig anderen Aufbau. Alle solche Sprachen heißen in diesem Kontext *natürliche Sprachen*.

Hier werden im Folgenden *formale Sprachen* betrachtet. Zu den formalen Sprachen gehören u. a. die mathematische und chemische Formelsprache und insbesondere auch die Programmiersprachen, wie z. B. Pascal oder Java. Hier stehen allerdings formale Sprachen und ihre Eigenschaften auf einer noch abstrakteren Ebene im Mittelpunkt der Betrachtung. Programmiersprachen werden dann lediglich zur beispielhaften Konkretisierung herangezogen.

Eine *formale Sprache* – oder kurz eine *Sprache* – wird aufbauend auf den Begriffen *Alphabet* und *Wort* definiert.

Definition 1.1 (Alphabet)
Ein Alphabet ist eine endliche Menge von Zeichen.

Beispiele 1.1
$\{a, b, c, \ldots, z\}$
$\{\alpha, \beta, \gamma, \ldots, \omega\}$
$\{0, 1\}$

□

[1] dtv Brockhaus Lexikon, 1982

Definition 1.2 (Wort)
Ein Wort über einem Alphabet ist eine endliche Zeichenfolge, die aus keinem oder endlich vielen Zeichen des Alphabets zusammengesetzt ist. Die Zeichen werden unmittelbar hintereinander geschrieben.
Die Zeichenfolge, die aus keinem Zeichen besteht, heißt leeres Wort und wird mit ϵ bezeichnet.

Beispiele 1.2

$informatik$	ist Wort über $\{a, b, c, \ldots, z\}$
$rxyzzwzyq$	ist Wort über $\{a, b, c, \ldots, z\}$
0110010	ist Wort über $\{0, 1\}$
0	ist Wort über $\{0, 1\}$
ϵ	ist Wort über jedem Alphabet

□

Definition 1.3 (Länge eines Wortes)
Die Länge eines Wortes w über einem Alphabet ist die Anzahl seiner Zeichen, geschrieben $|w|$.

Beispiele 1.3

$\lvert informatik\rvert$	=	10
$\lvert rxyzzwzyq\rvert$	=	9
$\lvert 0110010\rvert$	=	7
$\lvert 0\rvert$	=	1
$\lvert\epsilon\rvert$	=	0

□

Die Menge der natürlichen Zahlen wird wie folgt definiert verwendet:

$$\begin{aligned} \mathbb{N} &:= \{1, 2, 3, \ldots\} \\ \mathbb{N}_0 &:= \{0, 1, 2, \ldots\} \end{aligned}$$

Definition 1.4
Es sei Σ ein Alphabet. Dann wird Σ^n wie folgt definiert:

$$\Sigma^n := \{w \mid w \text{ ist Wort über } \Sigma, |w| = n, n \in \mathbb{N}_0\}$$

Beispiele 1.4
Sei $\Sigma = \{0, 1\}$ Alphabet.

Σ^0	=	$\{\epsilon\}$
Σ^1	=	$\{0, 1\}$
Σ^2	=	$\{00, 01, 10, 11\}$
Σ^3	=	$\{000, 001, 010, 011, 100, 101, 110, 111\}$

□

Zu beachten ist in dem Beispiel, dass $\{0, 1\}$ einmal das Alphabet und einmal die Menge der Wörter der Länge 1 über dem Alphabet $\{0, 1\}$ ist.

Definition 1.5 (Menge aller Wörter)
Die Menge aller Wörter über dem Alphabet Σ wird mit Σ^* bezeichnet und wie folgt definiert:

$$\Sigma^* := \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \dots$$

Die ϵ-freie Menge aller Wörter über Σ wird mit Σ^+ bezeichnet und wie folgt definiert:

$$\Sigma^+ := \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \dots$$

Beispiele 1.5
Sei $\Sigma = \{0, 1\}$ Alphabet.

$$\begin{array}{lcl} \Sigma^* & = & \{\epsilon, 0, 1, 00, 01, 10, 11, 000, 001, \dots\} \\ \Sigma^+ & = & \{0, 1, 00, 01, 10, 11, 000, 001, \dots\} \end{array}$$

□

Es gilt offensichtlich $\Sigma^* = \Sigma^+ \cup \{\epsilon\}$.

Ist auf einem Alphabet eine lineare Ordnung definiert, so lassen sich die Wörter über diesem Alphabet der Länge nach aufsteigend und bei gleicher Länge lexikographisch anordnen. Eine solche Ordnung wird *Wortordnung* genannt.

Definition 1.6 (Wortordnung)
Es sei Σ ein Alphabet, auf dem eine lineare Ordnung $\preceq_\Sigma$ definiert ist. Weiterhin seien $w, w' \in \Sigma^*$.
Auf Σ^* wird dann eine lineare Ordnung $\preceq_{\Sigma^*}$ wie folgt definiert:
Es gilt $w \preceq_{\Sigma^*} w'$ genau dann, wenn

- *$w = w'$ oder*
- *$|w| < |w'|$ oder*
- *$|w| = |w'| > 0$ und*
  $w = uav, w' = ua'v', u, v, v' \in \Sigma^*, a, a' \in \Sigma$ mit $a \neq a', a \preceq_\Sigma a'$.

$\preceq_{\Sigma^*}$ oder kurz $\preceq$ heißt Wortordnung auf Σ^*.

Beispiel 1.6
Sei $\Sigma = \{a, b, c, \dots, z\}$ Alphabet mit einer darauf definierten linearen Ordnung $a \preceq_\Sigma b \preceq_\Sigma \dots \preceq_\Sigma z$.
Auf Σ^* ergibt sich damit eine Wortordnung wie folgt:

$$\epsilon \preceq a \preceq b \preceq \dots \preceq z \preceq aa \preceq ab \preceq \dots \preceq az \preceq ba \preceq bb \preceq \dots \preceq zz \preceq aaa \preceq \dots$$

□

Auf Σ^* wird nun eine Verknüpfung definiert, die *Konkatenation* oder auch *Verkettung*. Dabei werden Wörter ohne ein Verknüpfungssymbol hintereinander geschrieben bzw. miteinander verkettet.

Definition 1.7 (Konkatenation)
Es seien zwei Wörter v und w über dem Alphabet Σ gegeben. Die Verknüpfung Konkatenation $\cdot$ auf Σ^* ist dann wie folgt definiert:

$$v \cdot w := vw.$$

Das Wort vw heißt die Konkatenation von v und w und ist wieder ein Wort über dem Alphabet Σ.
Das leere Wort ist das neutrale Element bzgl. der Konkatenation, d. h. für beliebige Wörter w gilt $\epsilon w = w\epsilon = w$.

Beispiel 1.7

$theoretische$	ist ein Wort über $\{a, b, c, \ldots, z\}$
$informatik$	ist ein Wort über $\{a, b, c, \ldots, z\}$
$theoretischeinformatik$	ist dann auch ein Wort über $\{a, b, c, \ldots, z\}$

□

Das Konkatenationssymbol $\cdot$ wird meist nicht geschrieben. Um die Identifizierbarkeit der Wörter einer Konkatenation zu erhalten, kann dann geklammert werden. Gleiches gilt zur Festlegung der Auswertungsreihenfolge bei Konkatenation von mehr als zwei Wörtern. Als Klammersymbole sind Zeichen zu wählen, die nicht im Alphabet enthalten sind.

Wird die Menge aller Wörter Σ^* zusammen mit der darauf definierten Verknüpfung Konkatenation $\cdot$ und dem leeren Wort ϵ als eine algebraische Struktur betrachtet, so ist festzustellen, dass $(\Sigma^*, \cdot)$ ein Monoid mit ϵ als neutralem Element ist. Es gilt also das Assoziativgesetz

$$u(vw) = (uv)w \text{ für alle } u, v, w \in \Sigma^*.$$

Das Kommutativgesetz gilt dagegen i. a. nicht, also

$$vw \neq wv \text{ für einige } v, w \in \Sigma^*.$$

Definition 1.8 (Teilwort, Präfix, Suffix)
Es seien u und w Wörter über dem Alphabet Σ.

- *Ein Wort u heißt Teilwort von w, wenn es $t, v \in \Sigma^*$ gibt, so dass $w = tuv$.*
- *Ein Wort u heißt Präfix von w, wenn es $v \in \Sigma^*$ gibt, so dass $w = uv$.*
- *Ein Wort u heißt Suffix von w, wenn es $t \in \Sigma^*$ gibt, so dass $w = tu$.*

Gilt jeweils zusätzlich $u \neq w$, so heißt u echtes Teilwort, Präfix bzw. Suffix von w.
Das leere Wort ist Teilwort, Präfix bzw. Suffix eines jeden Wortes.

Beispiele 1.8

$form$	ist Teilwort von $informatik$
100	ist Teilwort von 0110010
in	ist Präfix von $informatik$
10	ist Suffix von 0110010

□

Primär zur Schreibvereinfachung von Wortwiederholungen werden so genannte *Wortpotenzen* als mehrfache Konkatenationen mit sich selbst eingeführt. Ggf. sind die jeweiligen Teilwörter zu klammern.

Definition 1.9
Es sei w ein Wort über dem Alphabet Σ.

$$\begin{aligned} w^0 &:= \epsilon \\ w^n &:= ww^{n-1} \text{ für } n \in \mathbb{N} \end{aligned}$$

Beispiele 1.9
Sei $\Sigma = \{0, 1\}$ Alphabet.

$$\begin{aligned} 0^5 &= 00000 \\ 10^4 &= 10000 \\ 1(001)^3 1 &= 10010010011 \end{aligned}$$

□

Neben der Konkatenation sei hier noch die Spiegelung eines Wortes eingeführt.

Definition 1.10 (Spiegelung eines Wortes)
Es sei ein Wort w über dem Alphabet Σ gegeben. Die Spiegelung von w ist dann wie folgt definiert:

$$w^R := \begin{cases} \epsilon, & \text{falls } w = \epsilon \\ a_n a_{n-1} \dots a_1, & \text{falls } w = a_1 a_2 \dots a_n \end{cases}$$

Dabei ist $a_i \in \Sigma$ für $1 \leq i \leq n$ und $n \in \mathbb{N}$.

Beispiele 1.10

$$\begin{aligned} \epsilon^R &= \epsilon \\ a^R &= a \\ (abb)^R &= bba \\ (abba)^R &= abba \\ (babbaaa)^R &= aaabbab \end{aligned}$$

□

Wörter mit der Eigenschaft $w = w^R$ wie z. B. *abba*, *otto* oder $reliefpfeiler$ heißen *Palindrom*.

Damit sind nun alle Begriffe eingeführt, um eine *formale Sprache* zu definieren.

Definition 1.11 (Formale Sprache)
Eine (formale) Sprache L über einem Alphabet Σ ist eine Menge von Wörtern über Σ, also $L \subseteq \Sigma^*$.

Beispiele 1.11
Sei $\Sigma = \{0, 1\}$ Alphabet.
$L_1 = \emptyset$
$L_2 = \{\epsilon\}$
$L_3 = \{\epsilon, 00, 01, 1001\}$
$L_4 = \{0, 00, 000, 0000, \ldots\} = \{0^i \mid i \in \mathbb{N}\}$
$L_5 = \{001, 00001, 0000001, \ldots\} = \{0^{2i}1 \mid i \in \mathbb{N}\}$
$L_6 = \Sigma^*$

□

Eine formale Sprache ist also nichts anderes als eine Menge, deren Elemente Wörter sind. Die üblichen Mengenrelationen und -operationen können also auch auf formale Sprachen angewendet werden. Eine Semantik ist für Wörter in diesem Kontext im Gegensatz zu natürlichen Sprachen aber auch im Gegensatz zu Programmiersprachen nicht definiert.

Zu beachten ist der Unterschied zwischen $\emptyset$ und $\{\epsilon\}$. $\emptyset$ ist die leere Sprache, die kein Wort enthält, und $\{\epsilon\}$ ist eine Sprache, die ein Wort enthält, das leere Wort. Ansonsten gibt endliche Sprachen, die endlich viele Wörter enthalten, und unendliche Sprachen, die unendlich viele Wörter enthalten. Das zentrale Problem im Zusammenhang mit formalen Sprachen ist die Entscheidung bzw. die Entscheidbarkeit, ob ein Wort zu einer Sprache gehört oder nicht.

1.2 Zusammenhang mit Programmiersprachen

Es bleiben die im Abschnitt 1.1 eingeführten Begriffe auf konkrete Programmiersprachen zu übertragen. Dies erfolge hier am Beispiel der Programmiersprache *Java*.

Beispiel 1.12
Die folgende Menge von Zeichen stelle das Alphabet Σ der Sprache *Java* dar:

$$\Sigma = \{a, b, c, \ldots, z, A, B, C, \ldots, Z, 0, 1, \ldots, 9, +, -, *, \ldots, \cdot\}$$

Das Alphabet Σ entspreche dem auf der Tastatur eines Rechners zur Verfügung stehenden Zeichensatzes. Darunter sind auch Zeichen, die in der Anzeige für einen Leerraum, eine Einrückung oder einen Zeilenumbruch sorgen. Wörter über Σ sind dann z. B.:

$ax11$ $= w_1$
$a = b + 1$ $= w_2$
$class$ $= w_3$
for $= w_4$
$ax11\ for\ b\ class;$ $= w_5$

```
public class Hello {
    public static void main(String[ ] args) {
        System.out.println("Hello World");
    }
}
```
$= w_6$

Es bezeichne L_{Java} die formale Sprache, die alle diejenigen Wörter über Σ enthält, die syntaktisch korrekte *Java*-Programme darstellen. Es gilt dann:

$L_{Java} \subset \Sigma^*$
$w_i \in \Sigma^*$ $i = 1, 2, \ldots, 6$
$w_i \notin L_{Java}$ $i = 1, 2, \ldots, 5$
$w_6 \in L_{Java}$

□

Die Übersetzung eines Programms – in diesem Kontext eines Wortes über dem zugrunde liegenden Alphabet – besteht aus drei Analysephasen, bevor die eigentliche Zielcode-Generierung erfolgt:

- lexikalische Analyse
- syntaktische Analyse
- semantische Analyse

Die lexikalische Analyse hat die Aufgabe, die elementaren Einheiten (Symbole, Token) zu erkennen, z. B. Identifikatoren, Zahlen, Schlüsselwörter, Operatoren, Begrenzer. Diese elementaren Einheiten setzen sich aus unmittelbar nebeneinander stehenden Zeichen des zugrunde liegenden Alphabets zusammen. Die Einheiten lassen sich mittels *regulärer Ausdrücke* beschreiben. Ihre Analyse basiert auf dem Modell der *endlichen Automaten*. Aus der Zeichenfolge wird eine Folge von elementaren Einheiten erzeugt. Diese Symbole bilden das Alphabet für die nächste Analysephase.

In der syntaktischen Analyse wird das zu übersetzende Programm auf seine Struktur hin untersucht. Es geht dabei um Beziehungen unter den Symbolen, die auch über unmittelbares Nebeneinanderstehen hinausgehen, z.B. Blockstruktur und Klammerstruktur von Ausdrücken. Diese Struktur wird mittels *kontextfreier Grammatiken* beschrieben, die Analyse basiert auf dem Modell der *Kellerautomaten*. Die sematische Analyse umfasst z.B. Typüberprüfungen, die mittels anderer Konzepte und Methoden realisiert werden.

Allgemein ist in diesen Analysephasen die Problematik enthalten, dass aus der unendlichen Menge aller Wörter über dem Alphabet diejenigen – i. Allg. auch unendlich vielen – Wörter erkannt werden müssen, die zu der betrachteten formalen Sprache gehören. Es sind also für die Spezifikation und Analyse unendlicher Sprachen endliche Repräsentationen zu finden. Mit dem Konzept der Grammatiken und den Automatenmodellen sind diese auch über die Übersetzung von Programmen einer Programmiersprache hinaus gegeben. Die Grammatiken stellen eine *generierende* Sprachrepräsentation und die Automaten eine *analysierende* Sprachrepräsentation dar. Allerdings gibt es auch formale Sprachen, die sich nicht mittels solcher Repräsentationen darstellen lassen.

Kapitel 2

Endliche Automaten

2.1 Deterministische endliche Automaten

Die *endlichen Automaten* stellen ein sehr einfaches Berechnungsmodell dar. In Abhängigkeit von einer Eingabe werden gewisse Fallunterscheidungen vorgenommen und unterschiedlich bearbeitet. Für die gesamte Berechnung wird nur ein begrenzter Speicherbereich benötigt, und die Berechnungsdauer hängt nur linear von der Länge der Eingabe ab. Zu den wichtigsten Anwendungsbereichen gehören der Schaltungsentwurf und die Sprachanalyse.

Beispiel 2.1
Ein sehr einfaches Beispiel ist ein Ein-/Aus-Druckschalter zum Schalten irgendeines Gerätes. Das Gerät erinnert sich, ob es „aus“ oder „an“ ist. Ist es „aus“ und der Schalter wird gedrückt, so wird das Gerät eingeschaltet, ist es „an“ und der Schalter wird gedrückt, so wird das Gerät ausgeschaltet. Das Gerät merkt sich also seinen Zustand. Seine Eingaben bestehen aus dem Drücken des Schalters. Graphisch läßt sich ein solches Modell eines endlichen Automaten in einem so genannten Zustandsgraph darstellen (Abbildung 2.1).

□

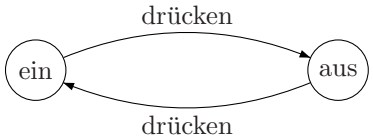


Abbildung 2.1: Zustandsgraph eines Ein-/Ausschalters

Als Weiteres wird ein Beispiel aus dem Schaltungsentwurf vorgestellt. Dieses führt zu endlichen Automaten, die auch eine Ausgabe erzeugen. Gegenüber den endlichen Automaten ohne Ausgabe stellen diese einen allgemeineren Ansatz dar. Detailliert werden dann aber die Modelle und Verfahren aus der Sprachanalyse behandelt, die wie Beispiel 2.1 keine explizite Ausgabe erzeugen.

Beispiel 2.2
Ein so genannter *Volladdierer* addiert zwei nicht-negative ganze Zahlen in binärer Darstellung seriell. Die n-stelligen Summanden $a = (a_{n-1} \dots a_1 a_0)_2$ und $a' = (a'_{n-1} \dots a'_1 a'_0)_2$ werden synchron getaktet, ziffernweise an den Volladdierer gegeben. Dieser addiert die jeweiligen Ziffern und gibt die berechneten Summenziffern entsprechend getaktet aus. Ein ggf. auftretender Übertrag wird intern bis zum nächsten Takt zwischengespeichert und dann berücksichtigt. Das Verfahren entspricht dem Vorgehen bei der so genannten *Papier-und-Bleistift-Methode* zur Addition von Zahlen.

Ein Volladdierer hat taktweise die folgende *Schaltfunktion* zu berechnen:

$$f : \{0,1\}^3 \to \{0,1\}^2 \text{ mit } f(a_k, a'_k, c_k) = (s, c)$$

und

$$\begin{aligned} s &: \{0,1\}^3 \to \{0,1\} \text{ mit } s(a_k, a'_k, c_k) = s_k \\ c &: \{0,1\}^3 \to \{0,1\} \text{ mit } c(a_k, a'_k, c_k) = c_{k+1}, \end{aligned}$$

wobei die Summenziffer s_k und der Übertrag für den nächsten Takt c_{k+1} durch die nachfolgende Tabelle definiert werden:

a_k	0	0	1	1	0	0	1	1
a'_k	0	1	0	1	0	1	0	1
c_k	0	0	0	0	1	1	1	1
s_k	0	1	1	0	1	0	0	1
c_{k+1}	0	0	0	1	0	1	1	1

Der Volladdierer hat also während einer Berechnung zwei verschiedene interne *Zustände* zu berücksichtigen: Beim vorangehenden Takt ist kein Übertrag aufgetreten, $c_k = 0$, oder beim vorangehenden Takt ist ein Übertrag aufgetreten, $c_k = 1$. Zu berücksichtigen ist die Initialisierung des Zustands mit $c_0 = 0$ und der für die Gültigkeit der Berechnung notwendige Schlusswert $c_n = 0$.

Die Berechnung des Volladdierers in Abhängigkeit von den Summandenziffern und seinem aktuellen Zustand kann durch einen so genannten *Zustandsgraph* dargestellt werden. Die Knoten des Graphen repräsentieren die Zustände, die gerichteten Kanten die Zustandsüberführungen. Die jeweiligen Ein- und Ausgaben werden den Kanten in der Form $(a_k, a'_k)/s_k$ angefügt. Ein Pfeil markiert den initialen Zustand (Abbildung 2.2).

□

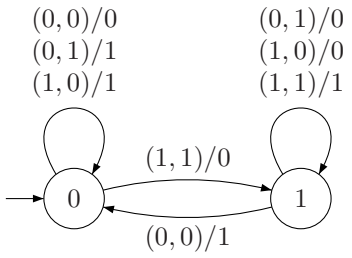


Abbildung 2.2: Zustandsgraph eines Volladdierers

Die Abstrahierung dieses Beispiels zu einem Automatenmodell führt zu den *Mealy-Automaten* und *Moore-Automaten*. Das sind endliche Automaten, die taktweise eine auf einem Eingabeband befindliche Eingabe verarbeiten und auf einem Ausgabeband eine Ausgabe erzeugen. Dabei durchlaufen sie eine von der Eingabe abhängige Zustandsfolge. Während bei einem Mealy-Automaten die Ausgabe von dem aktuellen Zustand und dem jeweiligen Eingabezeichen abhängt, hängt die Ausgabe beim Moore-Automaten ausschließlich vom aktuellen Zustand ab. Das Beispiel in Abbildung 2.2 stellt einen Mealy-Automaten dar.

Der Einsatz von endlichen Automaten zur Analyse und Erkennung formaler Sprachen – als *Sprachakzeptor* – kennt als Ausgabe aber nur die beiden Alternativen *Ja* und *Nein*, und dazu wird kein Ausgabeband benötigt. Solche Automaten sollen im Folgenden beschrieben und bzgl. ihrer Eigenschaften analysiert werden.

Ein *endlicher Automat* (EA) besteht aus einer Steuereinheit und einem Eingabeband mit Lesekopf. Die Steuereinheit enthält das Steuerungsprogramm des Automaten und einen endlichen Speicher, in dem die jeweiligen Zustände gespeichert werden. Mittels der *endlich* vielen Zustände kann sich der Automat *endlich* viel merken. Das Eingabeband enthält das zu analysierende Eingabewort, die Eingabe. Das Band ist in Felder unterteilt, wovon jedes Feld genau ein Eingabezeichen aufnimmt. Über den Lesekopf werden die Zeichen von der Steuereinheit gelesen (Abbildung 2.3).

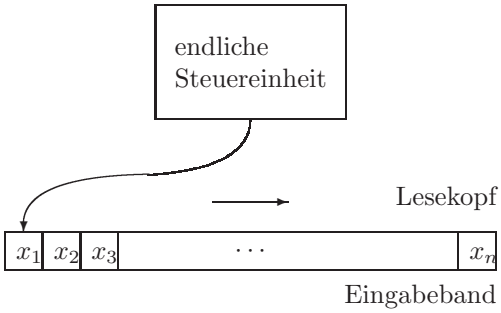


Abbildung 2.3: Endlicher Automat (EA)

Initial befindet sich der Automat in einem besonderen Zustand, dem Startzustand. Auf dem Eingabeband befindet sich die Eingabe. Das Band hat jeweils genau die Länge der Eingabe, also die Anzahl der Felder auf dem Band entspricht genau der Zeichenanzahl des Eingabewortes. Das Eingabewort wird getaktet von links nach rechts abgearbeitet, der Lesekopf befindet sich also initial auf dem Feld ganz links. Der Automat liest in einem Takt das Zeichen unter dem Lesekopf und geht in Abhängigkeit von diesem Zeichen und dem aktuell gespeicherten Zustand in einen Folgezustand über. Der Lesekopf rückt ein Feld nach rechts weiter. Es folgt der nächste Takt. Der Automat beendet die Analyse, wenn das letzte Zeichen der Eingabe verarbeitet ist.

Entsprechend dem Schema eines endlichen Automaten sind Mealy- und Moore-Automaten im Wesentlichen nur um ein Ausgabeband mit Schreibkopf erweitert (Abbildung 2.4). Die Ausgabe wird synchron mit dem Verarbeitungsfortschritt der Eingabe erzeugt.

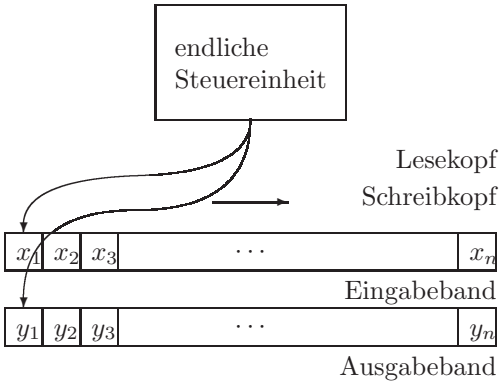


Abbildung 2.4: Endlicher Automat mit Ausgabeband

Formal wird ein endlicher Automat durch ein 5-Tupel beschrieben (Definition 2.1).

Definition 2.1 (Deterministischer endlicher Automat)
Ein 5-Tupel $E = (Q, \Sigma, \delta, q_0, F)$ heißt deterministischer endlicher Automat (DEA) oder kurz endlicher Automat, falls gilt:

- *Q ist eine endliche nicht-leere Menge, die Zustandsmenge*
- *Σ ist ein Alphabet, das Eingabealphabet*
- *$\delta : Q \times \Sigma \to Q$ ist eine Funktion, die Zustandsüberführungsfunktion oder auch Überführungsfunktion*
- *$q_0 \in Q$ ist der Startzustand*
- *$F \subseteq Q$ ist die Menge der Endzustände oder der akzeptierenden Zustände*

Hier wird zunächst die *deterministische* Variante eines endlichen Automaten betrachtet. Das sind Automaten, für die zu einem Paar aus Eingabezeichen und Zustand genau ein Folgezustand definiert ist. In das Konzept des *Nichtdeterminismus* wird später eingeführt.

Die Überführungsfunktion δ definiert das Programm des endlichen Automaten. Es ist üblich die Überführungsfunktion in einer so genannten *Überführungstabelle* zu notieren. Tabelle 2.1 spezifiziert dabei die Bedeutung der einzelnen Spalten.

δ :	Q	Σ	Q
	aktueller Zustand	gelesenes Zeichen	Folge-zustand

Tabelle 2.1: Überführungstabelle eines DEA

Eine solche Überführungstabelle besteht aus endlich vielen Zeilen. Für jedes Argumentpaar $(q, x) \in Q \times \Sigma$ von δ muss genau eine Zeile in der Tabelle enthalten sein. Dem Startzustand wird ein Pfeil ($\to$) vorangestellt, den Endzuständen ein Stern ($*$).

Die Überführungstabelle kann auch in Form einer Matrix angeordnet werden. Den Zeilen sind die Zustände zugeordnet, den Spalten die Eingabezeichen. Die jeweiligen Folgezustände befinden sich dann als Matrixelemente an den Schnittlinien (Tabelle 2.2).

δ : Q \ Σ	$\cdots$	x_j	$\cdots$
$\vdots$		$\vdots$	
q_i	$\cdots$	$\delta(q_i, x_j)$	$\cdots$
$\vdots$		$\vdots$	

Tabelle 2.2: Überführungstabelle eines DEA als Matrix angeordnet

Ein endlicher Automat kann durch einen bezeichneten gerichteten Graphen beschrieben werden, einem *Zustandsgraph*, *Zustandsdiagramm* oder auch *Überführungsgraph*. Jedem Zustand entspricht ein Knoten. Zwei Knoten q, q' sind genau dann durch eine gerichtete Kante von q nach q' verbunden, wenn der Automat bei einer Eingabe x von dem aktuellen Zustand q in den Folgezustand q' übergeht. Die Kan-

te wird mit der Eingabe x bezeichnet. Der Startzustand wird mit einem auf diesen zeigenden Pfeil gekennzeichnet, die Endzustände mit einem Doppelkreis.

Beispiel 2.3
Es sei $E = (\{0, 1, 2\}, \{a, b\}, \delta, 0, \{0\})$ ein deterministischer endlicher Automat mit der Überführungsfunktion δ. In Abbildung 2.5 ist E als Zustandsgraph dargestellt.

δ :

	a	b
$\rightarrow * 0$	1	0
1	2	1
2	0	2

□

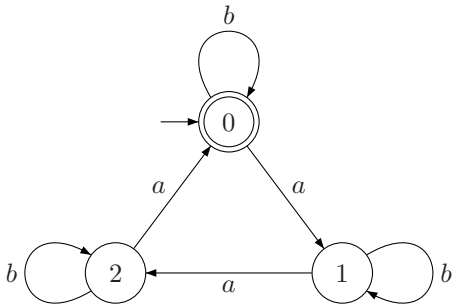


Abbildung 2.5: Endlicher Automat E

Die Arbeitsweise und das Ergebnis einer Berechnung eines endlichen Automaten ist informell bereits oben beschrieben worden. Einen formalen Ansatz dafür liefert Definition 2.2.

Definition 2.2 (Berechnung eines endlichen Automaten)
Sei $E = (Q, \Sigma, \delta, q_0, F)$ ein endlicher Automat und sei $w = x_1 x_2 \dots x_n$ ein Wort, bei dem jedes $x_i \in \Sigma$, $1 \leq i \leq n$. E akzeptiert dann w, falls eine Zustandsfolge $r_0, r_1, \dots, r_n$ in Q unter drei Bedingungen existiert:

1. $r_0 = q_0$,
2. $\delta(r_i, x_{i+1}) = r_{i+1}$, *für* $i = 0, 1, \dots, n-1$,
3. $r_n \in F$.

Man sagt dann, dass E die Sprache L erkennt, falls $L = \{w \mid E$ akzeptiert $w\}$, geschrieben $L(E) = L$.

Sei abschließend nochmals der endliche Automat aus dem vorangehenden Beispiel aufgegriffen, um sein dynamisches Verhalten mit einer konkreten Eingabe zu skizzieren.

Beispiel 2.4
Es sei der endliche Automat E aus Beispiel 2.3 gegeben. Die Berechnung von E mit den Eingabewörtern $w_1 = baabab$ und $w_2 = baababa$ kann dann wie folgt dargestellt werden. Das Teilwort $x_{i+1} \dots x_n \in \Sigma^*$ ist dabei ab dem Takt i noch abzuarbeiten, der Lesekopf steht auf dem Zeichen x_{i+1} ,$i = 0, 1, \dots, n-1$.

i	Q	$x_{i+1} \dots x_n \in \Sigma^*$
0	0	$baabab$
1	0	$aabab$
2	1	$abab$
3	2	bab
4	2	ab
5	0	b
6	0	

i	Q	$x_{i+1} \dots x_n \in \Sigma^*$
0	0	$baababa$
1	0	$aababa$
2	1	$ababa$
3	2	$baba$
4	2	aba
5	0	ba
6	0	a
7	1	

Für w_1 gibt also eine Zustandsfolge vom Startzustand bis zu einem Endzustand, $0, 0, 1, 2, 2, 0, 0$, d.h. $w_1 \in L(E)$. Für w_2 führt die Zustandsfolge dagegen vom Startzustand nicht zu einem Endzustand, $0, 0, 1, 2, 2, 0, 0, 1$, d.h. $w_2 \notin L(E)$.

Als von E erkannte Sprache $L(E)$ ergibt sich:

$$L(E) = \{w \mid w \in \{a, b\}^*,\ |w|_a \bmod 3 = 0\}$$

Dabei ist $|w|_a$ die Anzahl der a in w.

Auch wenn die Sprache $L(E)$ leicht zu spezifizieren war, so ist eine solche Aussage im Allgemeinen zu beweisen. Dies ist meist recht aufwändig und soll hier nicht durchgeführt werden. Zu zeigen wären die folgenden Eigenschaften der Zustände von E und $w \in \{a, b\}^*$:

I. Die Berechnung von E und Eingabe w endet genau dann im Zustand 0, wenn $|w|_a \bmod 3 = 0$.

II. Die Berechnung von E und Eingabe w endet genau dann im Zustand 1, wenn $|w|_a \bmod 3 = 1$.

III. Die Berechnung von E und Eingabe w endet genau dann im Zustand 2, wenn $|w|_a \bmod 3 = 2$.

Diese Behauptungen sind durch vollständige Induktion über die Länge eines Wortes zu beweisen. Aus diesen Eigenschaften und $0 \in F$ und $1, 2 \notin F$ ergibt sich nun insgesamt

$$L(E) \quad = \quad \{w \mid w \in \{a, b\}^*,\ |w|_a \bmod 3 = 0\}.$$

□

Wie in Abschnitt 1.2 erwähnt lassen sich die elementaren Einheiten eines Programms durch das Modell eines endlichen Automaten analysieren und erkennen. Das folgende Beispiel zeigt den Ansatz für den Aufbau eines lexikalischen Analysators (Scanners). Das Beispiel wird dann im nächsten Abschnitt fortgeführt.

Beispiel 2.5
Sei das zugrunde liegende Alphabet $\Sigma = \{a, b, \dots, z, 0, 1, \dots, 9\}$.

1. Ein Identifikator bzw. Bezeichner besteht aus einem Wort w, beginnend mit einem Buchstaben und gefolgt von keinem oder endlich vielen Buchstaben oder Ziffern:
$$w \in \{a, b, \dots, z\}\{a, b, \dots, z, 0, 1, \dots, 9\}^*$$
Der Einfachheit halber sei auf Großbuchstaben verzichtet.
Ein endlicher Automat E_I, der Identifikatoren akzeptiert, ist in Abbildung 2.6 dargestellt. E_I akzeptiert aber auch die reservierten Wörter als Identifikatoren.

2. Ganze Zahlen ohne Vorzeichen bestehen aus einem Wort w aus endlich vielen Ziffern:
$$w \in \{0, 1, 2, \ldots, 9\}^+$$
Ein endlicher Automat E_Z, der solche Zahlen akzeptiert, ist in Abbildung 2.7 dargestellt.

3. Das reservierte Wort bzw. Schlüsselwort for besteht aus der Konkatenation der Zeichen $f \cdot o \cdot r$.
Ein endlicher Automat E_{for}, der dieses Wort akzeptiert, ist in Abbildung 2.8 dargestellt.

In den Abbildungen 2.6 bis 2.8 zu diesem Beispiel gelten die folgenden abkürzenden Schreibweisen:

$$\begin{aligned} \mathit{buchstabe} &\in \{a, b, c, \ldots, z\} \\ \mathit{ziffer} &\in \{0, 1, 2, \ldots, 9\} \\ x_f &\in \Sigma \setminus \{f\} \\ x_o &\in \Sigma \setminus \{o\} \\ x_r &\in \Sigma \setminus \{r\} \end{aligned}$$

□

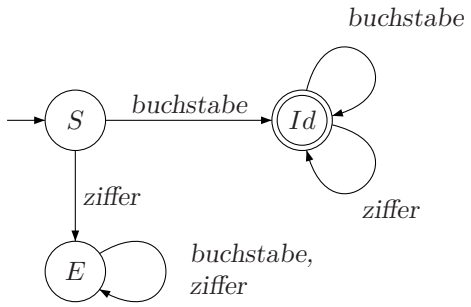


Abbildung 2.6: Endlicher Automat E_I

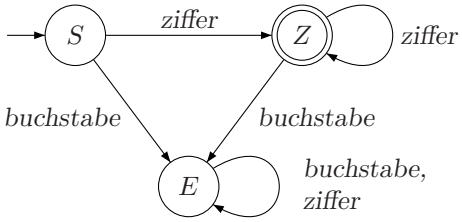


Abbildung 2.7: Endlicher Automat E_Z

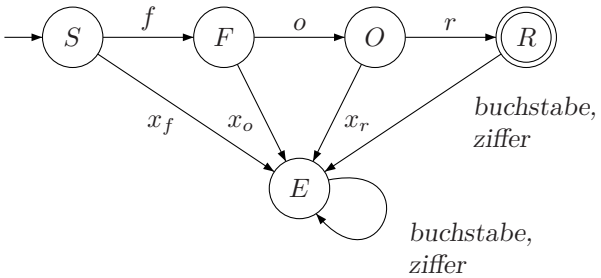


Abbildung 2.8: Endlicher Automat E_{for}

2.2 Nichtdeterministische endliche Automaten

Der *Nichtdeterminismus* erweitert das Konzept der deterministischen Verarbeitungsweise. Bezogen auf endliche Automaten bedeutet die deterministische Verarbeitungsweise, dass je Zustand und gelesenem Zeichen genau ein Folgezustand definiert ist. Ein nichtdeterministischer endlicher Automat lässt dagegen je Zustand und gelesenem Zeichen keinen bis mehrere Folgezustände zu. Des weiteren kann es auch Zustandsüberführungen geben, ohne dass ein Zeichen des Eingabebandes gelesen wird, so genannte ϵ-Überführungen. Betrachte dazu das Beispiel 2.6.

Beispiel 2.6
Es sei der nichtdeterministische endliche Automat N durch den Zustandsgraph in Abbildung 2.9 definiert.

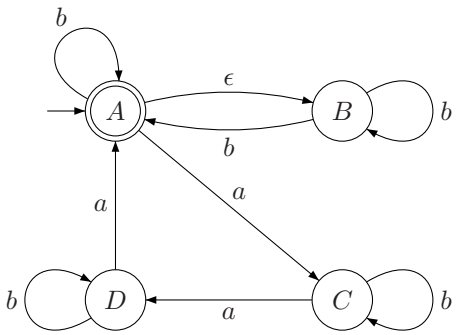


Abbildung 2.9: Nichtdeterministischer endlicher Automat N

Es handelt sich um einen nichtdeterministischen endlichen Automaten, weil der Zustand A eine ϵ-Überführung hat und der Zustand B für die Eingabe a keinen Folgezustand und für die Eingabe b zwei Folgezustände hat.

□

Neben der Einführung einer formalen Notation für nichtdeterministische endliche Automaten ist seine Arbeitsweise im Hinblick auf die Zustandsübergänge zu spezifizieren. Darüber hinaus ist die Beziehung zwischen deterministischen und nichtdeterministischen endlichen Automaten bzgl. der von ihnen erkennbaren Sprachen und Sprachfamilien zu analysieren.

Gegenüber der Definition von deterministischen endlichen Automaten sind zwei Erweiterungen für nichtdeterministische endliche Automaten in der Überführungsfunktion notwendig. Die Möglichkeit von ϵ-Überführungen wird formal dadurch ausgedrückt, dass im Argument auch das leere Wort als Eingabezeichen zugelassen wird, $\Sigma_\epsilon := \Sigma \cup \{\epsilon\}$. Die Potenzmenge $\mathcal{P}(Q)$ als Wertebereich erlaubt es weiterhin, keinen oder mehrere Folgezustände zu spezifizieren. Die Folgezustände werden zu der jeweiligen Teilmenge von Q zusammengefasst.

Definition 2.3 (Nichtdeterministischer endlicher Automat)
Ein 5-Tupel $N = (Q, \Sigma, \delta, q_0, F)$ heißt nichtdeterministischer endlicher Automat (NEA), falls gilt:

- *Q ist eine endliche nicht-leere Menge, die Zustandsmenge*
- *Σ ist ein Alphabet, das Eingabealphabet*
- *$\delta : Q \times \Sigma_\epsilon \to \mathcal{P}(Q)$ ist eine Funktion, die Zustandsüberführungsfunktion oder auch Überführungsfunktion*
- *$q_0 \in Q$ ist der Startzustand*
- *$F \subseteq Q$ ist die Menge der Endzustände*

Auch die Überführungsfunktion eines nichtdeterministischen endlichen Automaten kann in Form einer Überführungstabelle dargestellt werden (Tabelle 2.3). Um auszudrücken, dass eine Zustandsüberführung bzgl. eines Zeichens nicht definiert ist, wird als Funktionswert die leere Menge spezifiziert. Auch eine Darstellung der Überführungstabelle als Matrix ist wieder üblich. Die Überführungstabelle des Beispiels 2.6 ist noch einmal in Beispiel 2.7 dargestellt.

δ :	Q	Σ_ϵ	$\mathcal{P}(Q)$
	aktueller Zustand	gelesenes Zeichen	Folgezustandsmenge

Tabelle 2.3: Überführungstabelle eines NEA

Beispiel 2.7
$N = (\{A, B, C, D\}, \{a, b\}, \delta, A, \{A\})$ sei ein nichtdeterministischer endlicher Automat mit der Überführungsfunktion δ.

δ :	a	b	ϵ
$\to * A$	$\{C\}$	$\{A\}$	$\{B\}$
B	$\emptyset$	$\{A, B\}$	$\emptyset$
C	$\{D\}$	$\{C\}$	$\emptyset$
D	$\{A\}$	$\{D\}$	$\emptyset$

□

Wie aber arbeitet nun ein nichtdeterministischer endlicher Automat seine Eingabe ab, wenn er mehrere Möglichkeiten für einen Zustandübergang hat? Er spaltet sich in alle Alternativen auf und verfolgt diese Berechnungspfade unabhängig voneinander weiter. Ist für einen Zustand und eine Eingabe kein Folgezustand definiert, so terminiert dieser Berechnungspfad. Bei ϵ-Überführungen wird kein Eingabezeichen gelesen, der Lesekopf verbleibt auf dem aktuellen Zeichen, und der Automat geht lediglich in einen Folgezustand über. Führt mindestens einer von diesen Berechnungspfaden in einen Endzustand und ist dann das Eingabewort auf diesem Berechnungspfad auch vollständig abgearbeitet, so akzeptiert ein nichtdeterministischer endlicher Automat die Eingabe. Die Berechnungspfade des NEA N aus Beispiel 2.7 für eine Eingabe ist in Abbildung 2.10 dargestellt.

Ein deterministischer endlicher Automat dagegen durchläuft für jede Eingabe nur genau einen Berechnungspfad. Im akzeptierenden Fall endet dieser in einem Endzustand, andernfalls nicht.

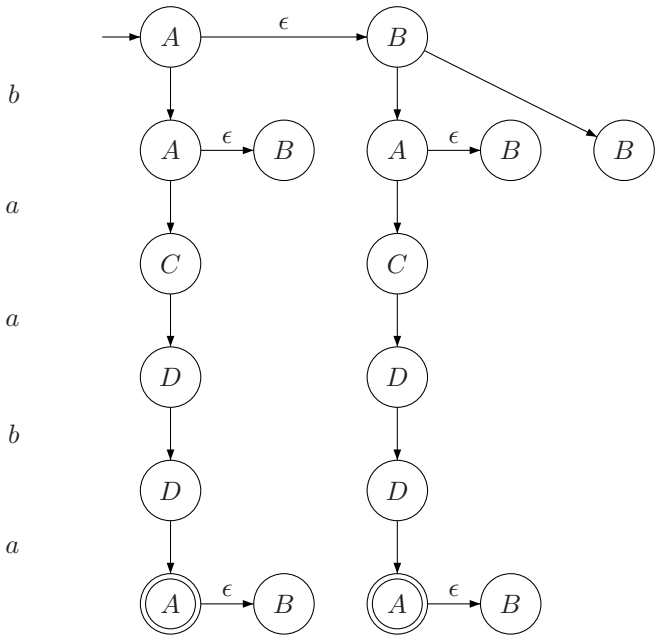


Abbildung 2.10: Berechnung des NEA N für die Eingabe *baaba*

Formal kann die Berechnung und das Ergebnis eines nichtdeterministischen endlichen Automaten mittels Definition 2.4 beschrieben werden.

Definition 2.4 (Berechnung eines NEA)
Sei $N = (Q, \Sigma, \delta, q_0, F)$ ein nichtdeterministischer endlicher Automat. Sei w ein Wort über dem Alphabet Σ, und es könne w als $w = y_1 y_2 \dots y_m$ geschrieben werden, wobei jedes $y_i \in \Sigma_\epsilon$, $1 \leq i \leq m$. Dann akzeptiert N das Wort w, falls eine Zustandsfolge $r_0, r_1, \dots, r_m$ in Q unter den folgenden drei Bedingungen existiert:

1. *$r_0 = q_0$,*
2. *$r_{i+1} \in \delta(r_i, y_{i+1})$, für $i = 0, 1, \dots, m-1$,*
3. *$r_m \in F$.*

Man sagt dann, dass N die Sprache L erkennt, falls $L = \{w \mid N \text{ akzeptiert } w\}$, geschrieben $L(N) = L$.

In Definition 2.4 wird das zu analysierende Wort w als $w = y_1 \dots y_m$ geschrieben, mit $y_i \in \Sigma_\epsilon$, $1 \leq i \leq m$. An einigen Positionen kann hier nun ein ϵ auftreten, obwohl sonst doch gesagt war, dass in der Eingabe keine ϵ auftreten. Das gilt auch weiterhin. Die hier eingefügten ϵ markieren nur die Positionen der ϵ-Überführungen.

Beispiel 2.8
Werde weiterhin das Beispiel 2.7 für die Eingabe $w_1 = baaba$ betrachtet. Es gibt zwei Möglichkeiten w_1 über dem Alphabet Σ_ϵ zu schreiben, so dass die Bedingungen aus Definition 2.4 erfüllt sind:

i	Q	$y_{i+1} \dots y_m \in \Sigma_\epsilon^*$
0	A	$baaba$
1	A	$aaba$
2	C	aba
3	D	ba
4	D	a
5	A	

i	Q	$y_{i+1} \dots y_m \in \Sigma_\epsilon^*$
0	A	$\epsilon baaba$
1	B	$baaba$
2	A	$aaba$
3	C	aba
4	D	ba
5	D	a
6	A	

Also ist $w_1 \in L(N)$.

Das Wort $w_2 = baabaa$ kann aber nicht über dem Alphabet Σ_ϵ geschrieben werden, so dass die Bedingungen aus Definition 2.4 erfüllt sind. Die Berechnungen terminieren niemals in einem Endzustand, also ist $w_2 \notin L(N)$.

Insgesamt bleibt ohne Beweis festzuhalten, dass N die Sprache

$$L(N) = \{w \mid w \in \{a,b\}^*,\ |w|_a \bmod 3 = 0\}$$

erkennt.

□

Die Beispiele 2.4 und 2.8 haben nun gezeigt, dass die Sprache $L = \{w \mid w \in \{a,b\}^*,\ |w|_a \bmod 3 = 0\}$ sowohl von deterministischen als auch von nichtdeterministischen endlichen Automaten erkannt werden kann. Es ist darüber hinaus leicht möglich weitere endliche Automaten – deterministische oder nichtdeterministische – zu konstruieren, die die gleiche Sprache erkennen, auch über die triviale Modifikation einer Umbenennung der Zustandsnamen hinaus. Zwei endliche Automaten sollen als äquivalent bezeichnet werden, wenn die von ihnen erkannten Sprachen gleich sind.

Definition 2.5 (Äquivalenz)
Seien E_1 und E_2 deterministische oder nichtdeterministische endliche Automaten. E_1 und E_2 heißen äquivalent, wenn die von ihnen erkannten Sprachen gleich sind, also wenn $L(E_1) = L(E_2)$ gilt.

In welcher Beziehung stehen nun aber die deterministischen endlichen Automaten und die nichtdeterministischen endlichen Automaten bzgl. der von ihnen jeweils erkennbaren Sprachen? Eine Beziehung ist trivial. Ein deterministischer endlicher Automat ist ein spezieller nichtdeterministischer endlicher Automat (Satz 2.1).

Satz 2.1
Es sei E ein deterministischer endlicher Automat. Dann gibt es einen äquivalenten nichtdeterministischen endlichen Automat N.

Beweis:
Es sei $E = (Q, \Sigma, \delta, q_0, F)$ ein deterministischer endlicher Automat.

Konstruktion eines zu E äquivalenten nichtdeterministischen endlichen Automaten $N = (Q', \Sigma', \delta', q_0', F')$:

Zustandsmenge:	Q'	$:=$	Q
Eingabealphabet:	Σ'	$:=$	Σ
Startzustand:	q_0'	$:=$	q_0
Endzustände:	F'	$:=$	F

Überführungsfunktion:

$$\delta' : Q' \times \Sigma'_\epsilon \to \mathcal{P}(Q')$$

$$\delta'(q,x) := \begin{cases} \{\delta(q,x)\}, & \text{für alle } (q,x) \in Q' \times \Sigma' \\ \emptyset, & \text{für alle } (q,x) \in Q' \times \{\epsilon\} \end{cases}$$

□

Aber gilt auch die Umkehrung dieser Beziehung? Sie ergibt sich aus dem Satz 2.2. Damit erweisen sich die Konzepte des Determinismus und des Nichtdeterminismus bei endlichen Automaten als äquivalent.

Der Beweis des Satzes 2.2 erfolgt in zwei Schritten:

- Konstruktion des gesuchten Automaten aus dem gegebenen Automaten
- Nachweis der Äquivalenz der zwei Automaten

Im ersten Teil, der Konstruktion, steckt die grundlegende Idee zum Beweis der Aussage des Satzes. Der zweite Teil verifiziert, dass die Konstruktion wirklich die Anforderungen erfüllt. Die Konstruktion liefert implizit auch einen Algorithmus zur Transformation eines Automatenmodells in ein anderes. Nach einem solchen konstruktiven Beweisprinzip sind viele weitere Aussagen der Automatentheorie bewiesen.

Die Idee in dem vorliegenden Beweis ist die *Teilmengen-Konstruktion*, bei der der deterministische Automat alle Berechnungspfade des gegebenen nichtdeterministischen endlichen Automaten parallel verfolgt. Dazu erhält er alle Zustandsteilmengen des nichtdeterministischen Automaten als Zustände. Die Zustände, in denen sich der nichtdeterministische Automat auf den unterschiedlichen Berechnungspfaden in einem Zeittakt befindet, bilden dann die Zustandsteilmenge, in der sich der deterministische Automat in diesem Zeittakt befindet. Bezogen auf diese Zustandsteilmengen sind die weiteren Komponenten des Automaten zu spezifizieren. Nach Abschluss der Konstruktion können die Zustandsteilmengen in beliebige andere Zustandsnamen umbenannt werden.

Satz 2.2 (Teilmengen-Konstruktion)
Es sei N ein nichtdeterministischer endlicher Automat. Dann gibt es einen äquivalenten deterministischen endlichen Automat E.

Beweis:
Es sei $N = (Q, \Sigma, \delta, q_0, F)$ ein nichtdeterministischer endlicher Automat.

Die Konstruktion eines zu N äquivalenten deterministischen endlichen Automaten $E = (Q', \Sigma', \delta', q_0', F')$ erfolgt in zwei Schritten, ohne Berücksichtigung von ϵ-Überführungen (I) und unter Berücksichtigung von ϵ-Überführungen (II).

I. Ohne Berücksichtigung von ϵ-Überführungen:

Zustandsmenge:	Q'	$:=$	$\mathcal{P}(Q)$
Eingabealphabet:	Σ'	$:=$	Σ
Startzustand:	q_0''	$:=$	$\{q_0\}$
Endzustände:	F'	$:=$	$\{R \mid R \in Q', R \cap F \neq \emptyset\}$
Überführungsfunktion:	δ''	$:$	$Q' \times \Sigma' \to Q'$
	$\delta''(R,x)$	$:=$	$\bigcup_{r \in R} \delta(r,x)$

II. Berücksichtigung von ϵ-Überführungen:

Zunächst ist zu jeder Zustandsteilmenge $R \in Q'$ die Menge zu spezifizieren, die daraus durch Verfolgen von keiner oder mehreren ϵ-Überführungen erreicht werden kann.

$E(R) := \{q \mid q$ ist von R über keine oder mehrere ϵ-Überführungen erreichbar$\}$

Dann wird die Überführungsfunktion entsprechend modifiziert. Für die in (I.) durch δ'' bestimmten Folgezustände werden die durch ϵ-Überführungen erreichbaren Zustände bestimmt.

Überführungsfunktion:	δ'	$:$	$Q' \times \Sigma' \to Q'$
	$\delta'(R,x)$	$:=$	$\bigcup_{r \in R} E(\delta(r,x)) = E(\delta''(R,x))$

Als Startzustand ist nun $q_0' = E(\{q_0\})$ zu wählen.

Insgesamt werden nach dieser Konstruktion also erst die Folgezustände entsprechend der gelesenen Eingabezeichen bestimmt und dann die ϵ-Überführungen verfolgt. Diese Reihenfolge könnte auch umgedreht werden.

Durch vollständige Induktion über die Länge eines Wortes w bleibt zu zeigen, dass für den so konstruierten Automaten E $w \in L(N) \Leftrightarrow w \in L(E)$ gilt. Dies soll hier aber nicht erfolgen. □

Beispiel 2.9
Gegeben sei nochmals der nichtdeterministische endliche Automat N aus Beispiel 2.7. Es werde dazu nach der Teilmengen-Konstruktion (Satz 2.2) ein äquivalenter deterministischer endlicher Automat $E = (Q', \Sigma', \delta', q_0', F')$ konstruiert.

I. Konstruktion ohne ϵ-Überführungen:

Zustandsmenge:	Q'	$:=$	$\{\emptyset, \{A\}, \{B\}, \{C\}, \{D\}, \{A,B\},$
			$\{A,C\}, \{A,D\}, \{B,C\}, \{B,D\}, \{C,D\},$
			$\{A,B,C\}, \{A,B,D\}, \{A,C,D\},$
			$\{B,C,D\}, \{A,B,C,D\}\}$
Eingabealphabet:	Σ'	$:=$	$\{a,b\}$
Startzustand:	q_0''	$:=$	$\{A\}$
Endzustände:	F'	$:=$	$\{\{A\}, \{A,B\}, \{A,C\}, \{A,D\},$
			$\{A,B,C\}, \{A,B,D\}, \{A,C,D\},$
			$\{A,B,C,D\}\}$

Überführungsfunktion $\delta'' : Q' \times \Sigma' \to Q'$ mit

$$\delta''(R,x) = \bigcup_{r \in R} \delta(r,x)$$

δ'':

	a	b
$\emptyset$	$\emptyset$	$\emptyset$
$\{A\}$	$\{C\}$	$\{A\}$
$\{B\}$	$\emptyset$	$\{A, B\}$
$\{C\}$	$\{D\}$	$\{C\}$
$\{D\}$	$\{A\}$	$\{D\}$
$\{A, B\}$	$\{C\}$	$\{A, B\}$
$\{A, C\}$	$\{C, D\}$	$\{A, C\}$
$\{A, D\}$	$\{A, C\}$	$\{A, D\}$
$\{B, C\}$	$\{D\}$	$\{A, B, C\}$
$\{B, D\}$	$\{A\}$	$\{A, B, D\}$
$\{C, D\}$	$\{A, D\}$	$\{C, D\}$
$\{A, B, C\}$	$\{C, D\}$	$\{A, B, C\}$
$\{A, B, D\}$	$\{A, C\}$	$\{A, B, D\}$
$\{A, C, D\}$	$\{A, C, D\}$	$\{A, C, D\}$
$\{B, C, D\}$	$\{A, D\}$	$\{A, B, C, D\}$
$\{A, B, C, D\}$	$\{A, C, D\}$	$\{A, B, C, D\}$

Betrachte z.B.:

$$\begin{aligned}
\delta''(\{A\}, a) &= \delta(A, a) = \{C\} \\
\delta''(\{A\}, b) &= \delta(A, b) = \{A\}
\end{aligned}$$

$$\begin{aligned}
\delta''(\{B\}, a) &= \delta(B, a) = \emptyset \\
\delta''(\{B\}, b) &= \delta(B, b) = \{A, B\}
\end{aligned}$$

$$\begin{aligned}
\delta''(\{A, B, C\}, a) &= \delta(A, a) \cup \delta(B, a) \cup \delta(C, a) \\
&= \{C\} \cup \emptyset \cup \{D\} \\
&= \{C, D\} \\
\delta''(\{A, B, C\}, b) &= \delta(A, b) \cup \delta(B, b) \cup \delta(C, b) \\
&= \{A\} \cup \{A, B\} \cup \{C\} \\
&= \{A, B, C\}
\end{aligned}$$

II. Berücksichtigung der ϵ-Überführungen:

Die Zustandsmengen $E(R)$ und die Überführungsfunktion $\delta'(R,x)$ sind für jedes $R \in Q'$ und $x \in \{a,b\}$ zu konstruieren.

$E(R) := \{q \mid q$ ist von R über keine oder mehrere ϵ-Überfürungen erreichbar$\}$

R	$E(R)$
$\emptyset$	$\emptyset$
$\{A\}$	$\{A,B\}$
$\{B\}$	$\{B\}$
$\{C\}$	$\{C\}$
$\{D\}$	$\{D\}$
$\{A,B\}$	$\{A,B\}$
$\{A,C\}$	$\{A,B,C\}$
$\{A,D\}$	$\{A,B,D\}$
$\{B,C\}$	$\{B,C\}$
$\{B,D\}$	$\{B,D\}$
$\{C,D\}$	$\{C,D\}$
$\{A,B,C\}$	$\{A,B,C\}$
$\{A,B,D\}$	$\{A,B,D\}$
$\{A,C,D\}$	$\{A,B,C,D\}$
$\{B,C,D\}$	$\{B,C,D\}$
$\{A,B,C,D\}$	$\{A,B,C,D\}$

Startzustand:

$$q_0' = E(\{A\}) = \{A,B\}$$

Überführungsfunktion $\delta' : Q' \times \Sigma' \to Q'$ mit

$$\delta'(R,x) = \bigcup_{r \in R} E(\delta(r,x)) = E(\delta''(R,x))$$

δ' : R	a			b		
$\emptyset$	$E(\emptyset)$	=	$\emptyset$	$E(\emptyset)$	=	$\emptyset$
$\{A\}$	$E(\{C\})$	=	$\{C\}$	$E(\{A\})$	=	$\{A,B\}$
$\{B\}$	$E(\emptyset)$	=	$\emptyset$	$E(\{A,B\})$	=	$\{A,B\}$
$\{C\}$	$E(\{D\})$	=	$\{D\}$	$E(\{C\})$	=	$\{C\}$
$\{D\}$	$E(\{A\})$	=	$\{A,B\}$	$E(\{D\})$	=	$\{D\}$
$\{A,B\}$	$E(\{C\})$	=	$\{C\}$	$E(\{A,B\})$	=	$\{A,B\}$
$\{A,C\}$	$E(\{C,D\})$	=	$\{C,D\}$	$E(\{A,C\})$	=	$\{A,B,C\}$
$\{A,D\}$	$E(\{A,C\})$	=	$\{A,B,C\}$	$E(\{A,D\})$	=	$\{A,B,D\}$
$\{B,C\}$	$E(\{D\})$	=	$\{D\}$	$E(\{A,B,C\})$	=	$\{A,B,C\}$
$\{B,D\}$	$E(\{A\})$	=	$\{A,B\}$	$E(\{A,B,D\})$	=	$\{A,B,D\}$
$\{C,D\}$	$E(\{A,D\})$	=	$\{A,B,D\}$	$E(\{C,D\})$	=	$\{C,D\}$
$\{A,B,C\}$	$E(\{C,D\})$	=	$\{C,D\}$	$E(\{A,B,C\})$	=	$\{A,B,C\}$
$\{A,B,D\}$	$E(\{A,C\})$	=	$\{A,B,C\}$	$E(\{A,B,D\})$	=	$\{A,B,D\}$
$\{A,C,D\}$	$E(\{A,C,D\})$	=	$\{A,B,C,D\}$	$E(\{A,C,D\})$	=	$\{A,B,C,D\}$
$\{B,C,D\}$	$E(\{A,D\})$	=	$\{A,B,D\}$	$E(\{A,B,C,D\})$	=	$\{A,B,C,D\}$
$\{A,B,C,D\}$	$E(\{A,C,D\})$	=	$\{A,B,C,D\}$	$E(\{A,B,C,D\})$	=	$\{A,B,C,D\}$

Betrachte z.B.:

$$\begin{aligned}
\delta'(\{A\},a) &= E(\delta''(\{A\},a)) = E(\{C\}) = \{C\}\\
\delta'(\{A\},b) &= E(\delta''(\{A\},b)) = E(\{A\}) = \{A,B\}\\
\\
\delta'(\{B\},a) &= E(\delta''(\{B\},a)) = E(\emptyset) = \emptyset\\
\delta'(\{B\},b) &= E(\delta''(\{B\},b)) = E(\{A,B\}) = \{A,B\}\\
\\
\delta'(\{D\},a) &= E(\delta''(\{D\},a)) = E(\{A\}) = \{A,B\}\\
\delta'(\{D\},b) &= E(\delta''(\{D\},b)) = E(\{D\}) = \{D\}\\
\\
\delta'(\{A,D\},a) &= E(\delta''(\{A,D\},a)) = E(\{A,C\}) = \{A,B,C\}\\
\delta'(\{A,D\},b) &= E(\delta''(\{A,D\},b)) = E(\{A,D\}) = \{A,B,D\}\\
\\
\delta'(\{A,B,C\},a) &= E(\delta''(\{A,B,C\},a)) = E(\{C,D\}) = \{C,D\}\\
\delta'(\{A,B,C\},b) &= E(\delta''(\{A,B,C\},b)) = E(\{A,B,C\}) = \{A,B,C\}
\end{aligned}$$

Eine Umbenennung der Zustände werde hier nicht vorgenommen. In Abbildung 2.11 ist der Zustandsgraph von E dargestellt.

Die Berechnungen von E für die Eingabewörter $w_1 = baaba$ und $w_2 = baabaa$ ergeben sich wie folgt:

i	Q'	$x_{i+1}\dots x_n \in \Sigma'^*$
0	$\{A,B\}$	$baaba$
1	$\{A,B\}$	$aaba$
2	$\{C\}$	aba
3	$\{D\}$	ba
4	$\{D\}$	a
5	$\{A,B\}$	

i	Q'	$x_{i+1}\dots x_n \in \Sigma'^*$
0	$\{A,B\}$	$baabaa$
1	$\{A,B\}$	$aabaa$
2	$\{C\}$	$abaa$
3	$\{D\}$	baa
4	$\{D\}$	aa
5	$\{A,B\}$	a
6	$\{C\}$	

Die Berechnung von w_1 endet in einem Endzustand, die Berechnung von w_2 nicht. Also ist $w_1 \in L(E)$ und $w_2 \notin L(E)$.

□

Die Teilmengen-Konstruktion erzeugt aus n Zuständen des nichtdeterministischen endlichen Automaten 2^n Zustände für den deterministischen endlichen Automaten. Die Zustandsanzahl vergrößert sich also exponentiell. An dem vorangehenden Beispiel kann aber auch leicht erkannt werden, dass sehr viele Zustände entstehen können, die dann gar nicht benötigt werden, weil sie ausgehend vom Startzustand *unerreichbar* sind. In dem Beispiel sind ausgehend von dem Startzustand $\{A,B\}$ nur die Zustände $\{C\}$ und $\{D\}$ erreichbar. Es gibt allerdings NEAs deren äquivalente DEAs tatsächlich alle 2^n Zustände benötigen.

Aufgrund der Äquivalenz von deterministischen und nichtdeterministischen endlichen Automaten müssen diese nunmehr nicht weiter unterschieden werden. Was bringt dann aber der Nichtdeterminismus bei endlichen Automaten? Nun ist zwar die Implementierung von gegebenen deterministischen endlichen Automaten im Gegensatz zu nichtdeterministischen endlichen Automaten direkt möglich, aber ihr Entwurf ist häufig nicht so einfach. Vor allem ist er wegen der größeren Zustandsanzahl wesentlich aufwändiger, auch wenn nicht in allen Fällen die gegenüber der nichtdeterministischen Variante exponentielle Zustandsanzahl benötigt wird.

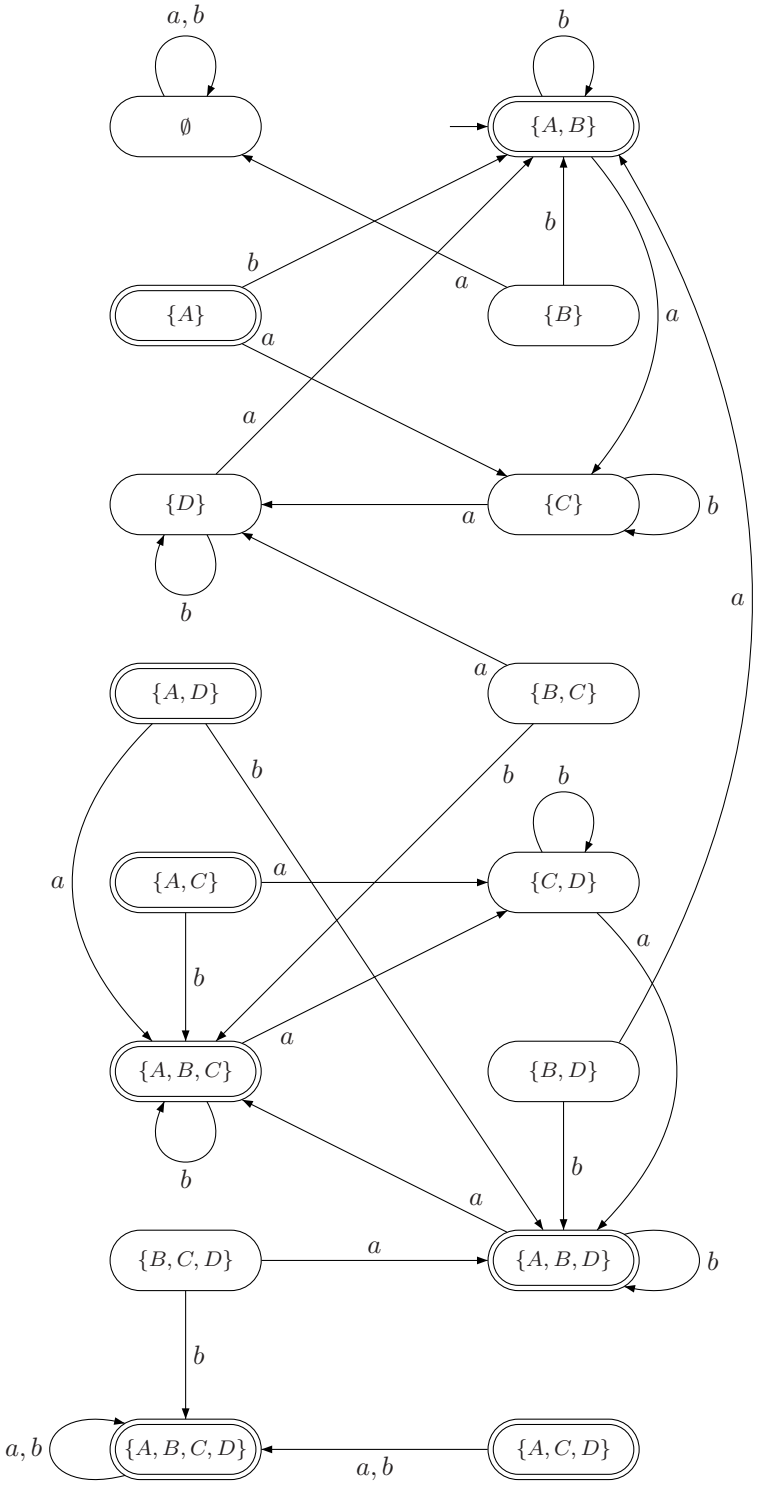


Abbildung 2.11: Zustandsgraph des DEA E aus Beispiel 2.9

Werde hier nochmals das Beispiel 2.5 zur Analyse und Erkennung der elementaren Einheiten eines Programms durch das Modell eines endlichen Automaten aufgegriffen. Der Entwurf der einzelnen Automaten und ihr Zusammenfügen mittels nichtdeterministischer endlicher Automaten ist deutlich einfacher.

Die einzelnen Automaten werden zunächst als so genannte *unvollständige* endliche Automaten – es gibt nicht zu jedem Paar aus Zustand und Eingabezeichen einen Folgezustand – konstruiert und danach werden diese nichtdeterministisch zusammengefügt (Beispiel 2.10). Darauf würde der Algorithmus *Teilmengen-Konstruktion* angewendet werden, um zu einem DEA zu gelangen. Die große Zustandsanzahl könnte ggf. durch einen Algorithmus zur Minimisierung des DEA verringert werden. Darauf soll hier aber nicht weiter eingegangen werden.

Beispiel 2.10
Sei das zugrunde liegende Alphabet $\Sigma = \{a, b, \ldots, z, 0, 1, \ldots, 9\}$.

1. Ein Identifikator bzw. Bezeichner besteht aus einem Wort w, beginnend mit einem Buchstaben und gefolgt von keinem oder endlich vielen Buchstaben oder Ziffern:
$$w \in \{a, b, \ldots, z\}\{a, b, \ldots, z, 0, 1, \ldots, 9\}^*$$
Der Einfachheit halber sei auf Großbuchstaben verzichtet.

   Ein unvollständiger endlicher Automat N_I, der Identifikatoren akzeptiert, ist in Abbildung 2.12 dargestellt. N_I akzeptiert aber auch die reservierten Wörter als Identifikatoren.

2. Ganze Zahlen ohne Vorzeichen bestehen aus einem Wort w aus endlich vielen Ziffern:
$$w \in \{0, 1, 2, \ldots, 9\}^+$$

   Ein unvollständiger endlicher Automat N_Z, der solche Zahlen akzeptiert, ist in Abbildung 2.13 dargestellt.

3. Das reservierte Wort for besteht aus der Konkatenation der Zeichen $f \cdot o \cdot r$.

   Ein unvollständiger endlicher Automat N_{for}, der dieses Wort akzeptiert, ist in Abbildung 2.14 dargestellt.

4. Ein lexikalischer Analysator (Scanner) besteht nun aus einer Kombination aller dieser Automaten. Ein Programm wird zeichenweise gelesen. Je nach analysiertem Präfix verzweigt ein solcher Automat in die entsprechende Alternative. Hat er eine elementare Einheit des Programms erkannt, so übersetzt er diese in ein entsprechendes Symbol, geht wieder in seinen Startzustand und beginnt mit der Analyse der verbleibenden Zeichen des Programms.

   Der Zustandsgraph in Abbildung 2.15 beschreibt einen endlichen erkennenden Automaten N_{Lex}, der eine Eingabe auf ein Identifikator, eine ganze Zahl ohne Vorzeichen oder das reservierte Wort *end* analysieren kann. Der Zustand, in dem der Automat anhält, signalisiert die erkannte Alternative.

   Ein zu N_{Lex} äquivalenter deterministischer endlicher Automat E_{Lex} ist in Abbildung 2.16 dargestellt. Dieser ist allerdings nicht mittels der Teilmengen-Konstruktion konstruiert worden. Hier gelten die abkürzenden Schreibweisen wie in Beispiel 2.5 mit der Änderung $x_f \in \{a, b, \ldots, z\} \setminus \{f\}$.

□

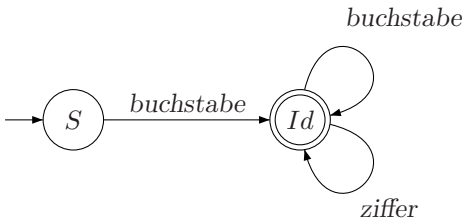


Abbildung 2.12: Unvollständiger endlicher Automat N_I

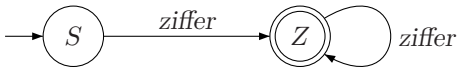


Abbildung 2.13: Unvollständiger endlicher Automat N_Z

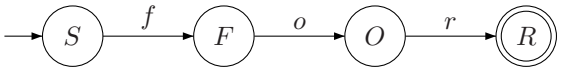


Abbildung 2.14: Unvollständiger endlicher Automat N_{for}

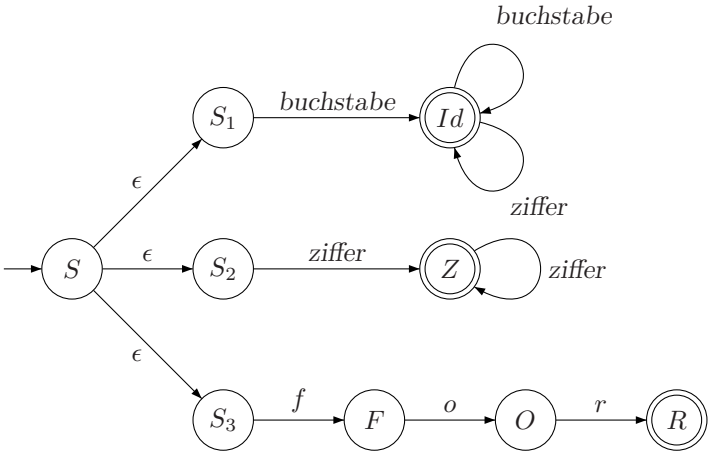


Abbildung 2.15: Nichtdeterministischer endlicher Automat N_{Lex}

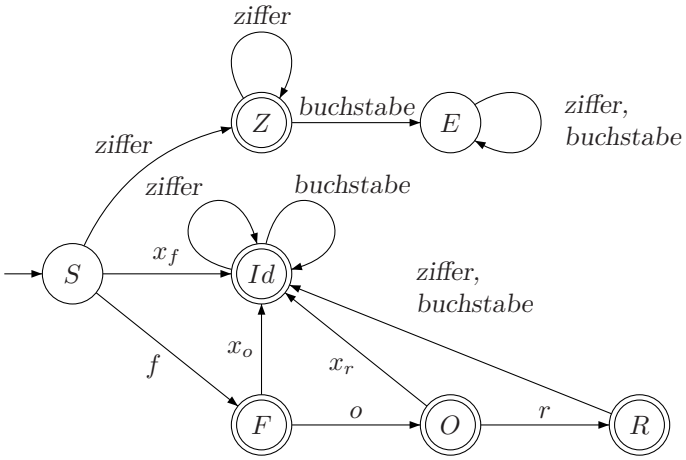


Abbildung 2.16: Deterministischer endlicher Automat E_{Lex}

Kapitel 3

Reguläre Sprachen

3.1 Reguläre Sprachen und Operationen

Formale Sprachen – selbst Mengen – lassen sich wieder zu Mengen zusammenfassen, so genannten *Sprachfamilien* bzw. *Klassen* von Sprachen. Die von endlichen Automaten erkennbaren Sprachen bilden somit eine Sprachfamilie, die Sprachfamilie der *regulären Sprachen*. Dieser Name bekommt über die durch *reguläre Ausdrücke* definierten Sprachen seine Rechtfertigung. Zunächst aber wird die Sprachfamilie der regulären Sprachen mittels endlicher Automaten definiert und es werden weiterhin die *regulären Operationen* eingeführt und diskutiert.

Definition 3.1 (Reguläre Sprachen)
Eine Sprache heißt reguläre Sprache, falls es einen endlichen Automaten gibt, der diese Sprache erkennt. Die Sprachfamilie oder Klasse der regulären Sprachen wird mit $\mathcal{L}_{reg}$ bezeichnet.

Auf den regulären Sprachen werden nun einige elementare Operationen eingeführt und weiterhin gezeigt, dass diese auf den regulären Sprachen auch abgeschlossen sind. Die damit verbundenen Verfahren erweisen sich für die Konstruktion von endlichen Automaten zur Lösung gewisser Probleme als sehr hilfreich. Die Probleme werden in Teilprobleme zerlegt, einzeln gelöst und die Teillösungen dann zur Gesamtlösung zusammengefügt.

Definition 3.2 (Reguläre Operationen)
Seien A und B Sprachen. Die regulären Operationen Vereinigung, Konkatenation und Stern (Kleeneabschluss) werden dann wie folgt definiert:

- ***Vereinigung***: $A \cup B = \{x \mid (x \in A) \vee (x \in B)\}$.
- ***Konkatenation***: $A \cdot B = AB = \{xy \mid (x \in A) \wedge (y \in B)\}$.
- ***Stern***: $A^* = \{x_1 x_2 \dots x_k \mid k \geq 0,\ x_i \in A,\ 1 \leq i \leq k\}$.

Beispiel 3.1
Seien $A = \{b, ab, aba\}$ und $B = \{\epsilon, a, bb\}$ Sprachen über $\{a, b\}$.

$$\begin{aligned} A \cup B &= \{\epsilon, a, b, ab, bb, aba\} \\ AB &= \{b, ab, aba, ba, abaa, bbb, abbb, ababb\} \\ B^* &= \{\epsilon, a, aa, bb, aaa, abb, bba, aaaa, aabb, abba, bbaa, bbbb, aaaaa, \dots\} \end{aligned}$$

□

Beispiel 3.2
Seien $A = \{a^i \mid i \in \mathbb{N}_0\}$ und $B = \{b^i \mid i \in \mathbb{N}_0\}$ Sprachen über $\{a, b\}$.

$$\begin{aligned} A \cup B &= \{w \mid w = a^i \text{ oder } w = b^i, i \in \mathbb{N}_0\} \\ AB &= \{a^i b^j \mid i, j \in \mathbb{N}_0\} \\ B^* &= B \end{aligned}$$

□

Die Beweisideen zum Nachweis der Abschlusseigenschaften der regulären Operationen (Sätze 3.1, 3.2 und 3.3) sind in den Abbildungen 3.1, 3.2 und 3.3 dargestellt.

Satz 3.1
Die regulären Sprachen sind unter der Operation Vereinigung abgeschlossen.

Beweis:
Erkenne der NEA $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ die Sprache A_1 und der NEA $N_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ die Sprache A_2. Seien Q_1 und Q_2 disjunkt.

Konstruktion eines NEA $N = (Q, \Sigma, \delta, q_0, F)$, der die Sprache $A_1 \cup A_2$ erkennt:

1. $Q = \{q_0\} \cup Q_1 \cup Q_2$ mit $q_0 \notin Q_1 \cup Q_2$
2. q_0 ist Startzustand von N
3. $F = F_1 \cup F_2$
4. Für jedes $q \in Q$ und jedes $a \in \Sigma_\epsilon$ definiere

$$\delta(q,a) = \begin{cases} \delta_1(q,a) & \text{für } q \in Q_1 \\ \delta_2(q,a) & \text{für } q \in Q_2 \\ \{q_1, q_2\} & \text{für } q = q_0,\ a = \epsilon \\ \emptyset & \text{für } q = q_0,\ a \neq \epsilon \end{cases}$$

□

Satz 3.2
Die regulären Sprachen sind unter der Operation Konkatenation abgeschlossen.

Beweis:
Erkenne der NEA $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ die Sprache A_1 und der NEA $N_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ die Sprache A_2. Seien Q_1 und Q_2 disjunkt.

Konstruktion eines NEA $N = (Q, \Sigma, \delta, q_1, F_2)$, der die Sprache $A_1 \circ A_2$ erkennt:

1. $Q = Q_1 \cup Q_2$
2. q_1 ist Startzustand von N
3. F_2 ist die Menge von Endzuständen von N
4. Für jedes $q \in Q$ und jedes $a \in \Sigma_\epsilon$ definiere

$$\delta(q,a) = \begin{cases} \delta_1(q,a) & \text{für } q \in Q_1,\ q \notin F_1 \\ \delta_1(q,a) & \text{für } q \in F_1,\ a \neq \epsilon \\ \delta_1(q,a) \cup \{q_2\} & \text{für } q \in F_1,\ a = \epsilon \\ \delta_2(q,a) & \text{für } q \in Q_2 \end{cases}$$

□

Satz 3.3
Die regulären Sprachen sind unter der Operation Stern abgeschlossen.

Beweis:
Erkenne der NEA $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ die Sprache A_1.

Konstruktion eines NEA $N = (Q, \Sigma, \delta, q_0, F)$, der die Sprache A_1^* erkennt:

1. $Q = \{q_0\} \cup Q_1$ mit $q_0 \notin Q_1$
2. q_0 ist Startzustand von N
3. $F = \{q_0\} \cup F_1$
4. Für jedes $q \in Q$ und jedes $a \in \Sigma_\epsilon$ definiere

$$\delta(q,a) = \begin{cases} \delta_1(q,a) & \text{für } q \in Q_1,\ q \notin F_1 \\ \delta_1(q,a) & \text{für } q \in F_1,\ a \neq \epsilon \\ \delta_1(q,a) \cup \{q_1\} & \text{für } q \in F_1,\ a = \epsilon \\ \{q_1\} & \text{für } q = q_0,\ a = \epsilon \\ \emptyset & \text{für } q = q_0,\ a \neq \epsilon \end{cases}$$

□

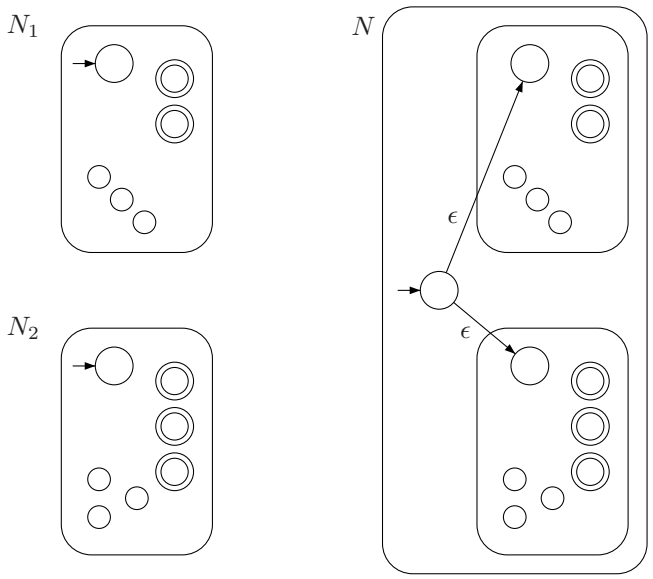


Abbildung 3.1: Konstruktion eines NEA N mit $L(N) = A_1 \cup A_2$

Beispiel 3.3
Es seien zwei nichtdeterministische endliche Automaten N_1 und N_2 gegeben (Abbildung 3.4a). Nach den Beweisen zu den Sätzen 3.1, 3.2 und 3.3 sind dann die nichtdeterministischen Automaten N_V mit $L(N_V) = L(N_1) \cup L(N_2)$, N_K mit $L(N_K) = L(N_1)L(N_2)$ und N_S mit $L(N_S) = (L(N_1))^*$ konstruiert (Abbildungen 3.4b, 3.4c und 3.4d).

□

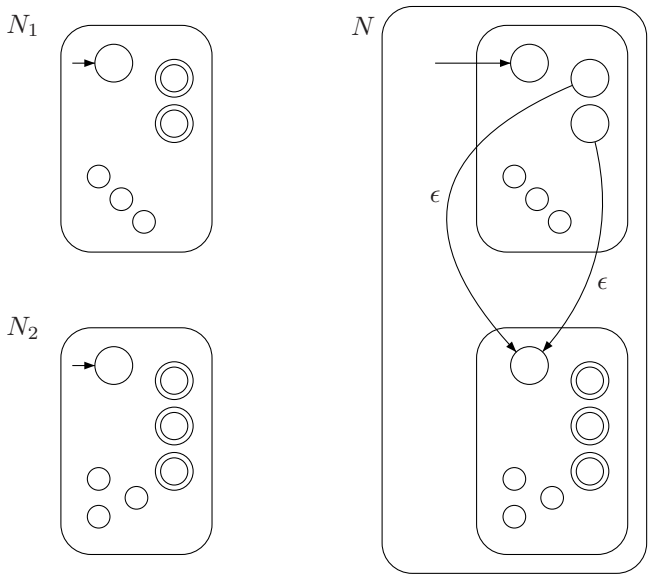


Abbildung 3.2: Konstruktion eines NEA N mit $L(N) = A_1 \cdot A_2$

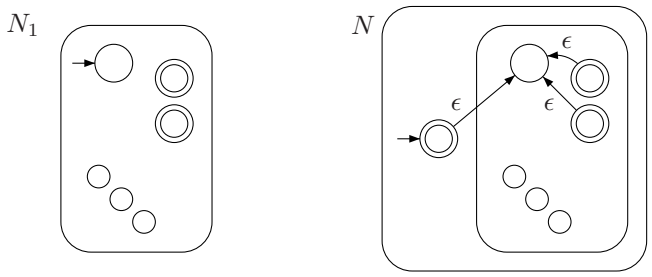


Abbildung 3.3: Konstruktion eines NEA N mit $L(N) = A_1^*$

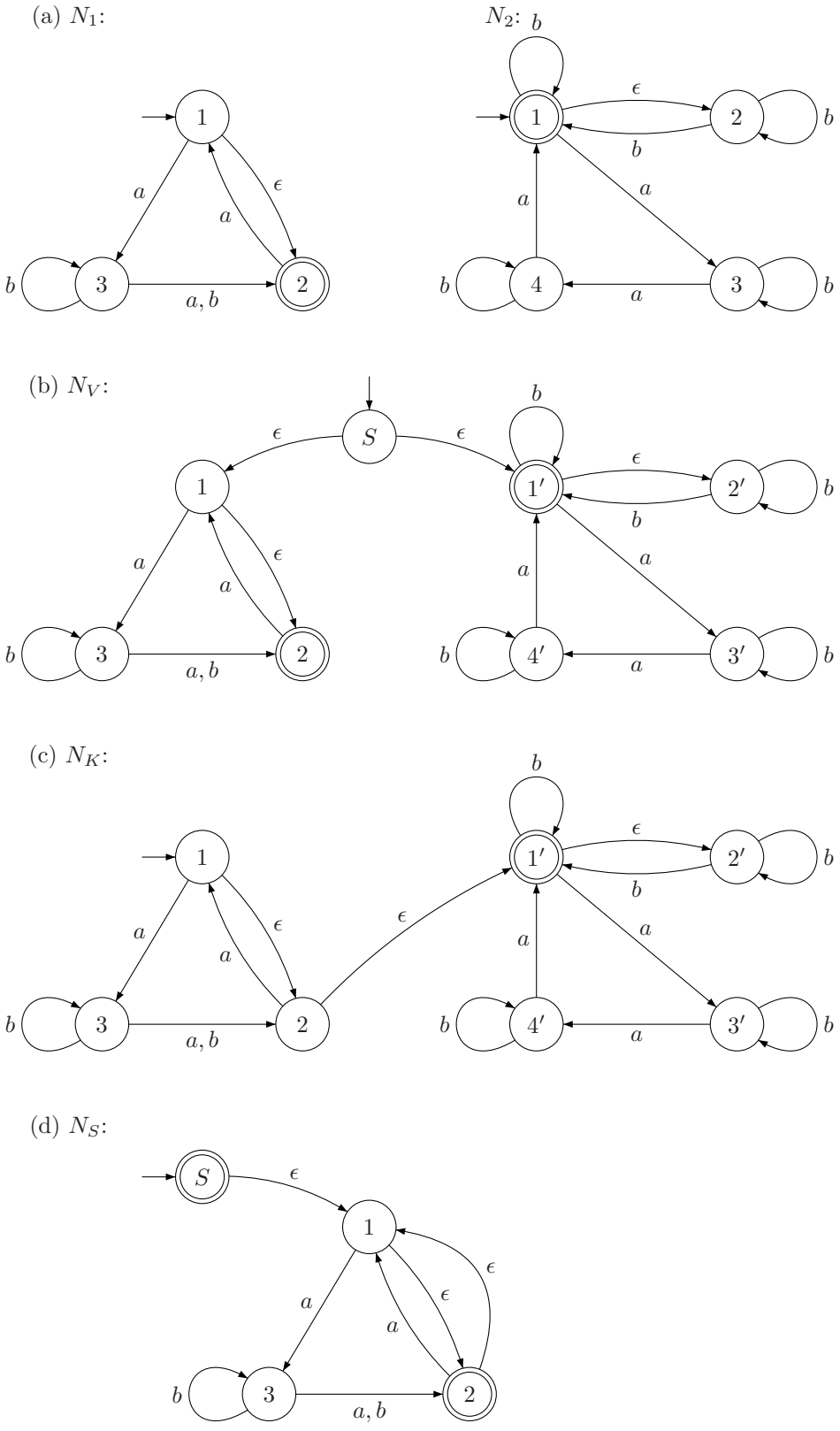


Abbildung 3.4: (a) NEA N_1 und NEA N_2, (b) NEA N_V mit $L(N_V) = L(N_1) \cup L(N_2)$, (c) NEA N_K mit $L(N_K) = L(N_1)L(N_2)$ und (d) NEA N_S mit $L(N_S) = (L(N_1))^*$

3.2 Reguläre Ausdrücke

Die regulären Sprachen sind bisher mittels endlicher Automaten definiert worden. Ein Wort gehört genau dann zu einer Sprache, wenn es von einem endlichen Automaten akzeptiert wird. In diesem Abschnitt werden nun Sprachen durch einen algebraischen Ansatz beschrieben, den *regulären Ausdrücken*. Ein regulärer Ausdruck definiert auch wieder eine Sprache, und es wird sich zeigen, dass dies genau die regulären Sprachen sind.

Reguläre Ausdrücke haben eine Vielzahl von Anwendungen in der Informatik, insbesondere in der Textverarbeitung, wenn es darum geht Zeichenfolgen zu finden oder diese zu ändern. Das Betriebssystem UNIX und die Programmiersprache PERL zum Beispiel stellen solche Methoden zur Verfügung. Auch Suchmaschinen im World Wide Web benutzen diese Techniken, um intelligente Suchanfragen zu ermöglichen.

Eine andere Anwendung kann in der Definition von Programmiersprachen und beim Entwurf von Compilern gefunden werden. Die elementaren Einheiten (Token) werden mittels regulärer Ausdrücke definiert. Es kann dann daraus ein Teil eines Compilers automatisch erzeugt werden, der Scanner oder auch der lexikalische Analysator. Die regulären Ausdrücke werden dazu in äquivalente endliche Automaten überführt, die genau diese elementaren Einheiten erkennen.

Definition 3.3 (Reguläre Ausdrücke)
Ein Wort R ist ein regulärer Ausdruck über dem Alphabet Σ, falls R von einer der folgenden Formen ist und $\{\epsilon,\ \emptyset,\ |,\ \cdot,\ ^*,\ (,\ )\} \cap \Sigma = \emptyset$:

1. *a, wobei $a \in \Sigma$,*
2. *ϵ,*
3. *$\emptyset$,*
4. *$(R_1 \mid R_2)$, wobei R_1 und R_2 reguläre Ausdrücke sind,*
5. *$(R_1 \cdot R_2) = (R_1 R_2)$, wobei R_1 und R_2 reguläre Ausdrücke sind, oder*
6. *(R_1^*), wobei R_1 ein regulärer Ausdruck ist.*

Damit ist die syntaktische Form eines regulären Ausdrucks definiert. Seine semantische Definition bestimmt die durch ihn repräsentierte formale Sprache.

Definition 3.4 (Reguläre Sprache)
Ein regulärer Ausdruck R über Σ repräsentiert eine formale Sprache. Ist es notwendig zwischen einem regulären Ausdruck R und der Sprache zu unterscheiden, die dieser beschreibt, so bezeichnet $L(R)$ die Sprache von R.

1. *$a \in \Sigma$ repräsentiert $\{a\}$,*
2. *ϵ repräsentiert $\{\epsilon\}$, die Sprache, die das leere Wort enthält,*
3. *$\emptyset$ repräsentiert $\emptyset$, die Sprache, die kein Wort enthält,*
4. *$(R_1 \mid R_2)$ repräsentiert $L(R_1) \cup L(R_2)$,*
5. *$(R_1 \cdot R_2)$ repräsentiert $L(R_1) \cdot L(R_2) = L(R_1)L(R_2)$,*
6. *(R_1^*) repräsentiert $(L(R_1))^*$.*

Die Auswertungsreihenfolge der auf reguläre Ausdrücke angewandten Operatoren orientiert sich an der Auswertungsreihenfolge der auf Sprachen angewandten Operatoren. Als erstes wird der Stern ausgewertet, dann die Konkatenation und als letztes die Vereinigung, wobei der Operator | auch als *oder* gesprochen wird. Soll von dieser Reihenfolge bei der Auswertung abgewichen werden, so ist ein regulärer Ausdruck entsprechend zu klammern. Äußere Klammern können auch entfallen. Der Operator | wird häufig auch als + oder als $\cup$ geschrieben.

Definition 3.5 (Äquivalenz)
Zwei reguläre Ausdrücke R_1 und R_2 über Σ heißen äquivalent, geschrieben $R_1 \equiv R_2$, falls $L(R_1) = L(R_2)$.

Auf die explizite Definition von ϵ als regulärem Ausdruck könnte verzichtet werden, da es sich implizit aus $\emptyset^*$ ergibt. Es gilt

$$L(\emptyset^*) = (L(\emptyset))^* = \emptyset^* = \{\epsilon\} = L(\epsilon),$$

also $\emptyset^* \equiv \epsilon$.

Zur Schreibvereinfachung werden die folgenden Notationen eingeführt. Seien dabei R ein regulärer Ausdruck über $\Sigma = \{x_1, x_2, \ldots, x_n\}$ und $k \in \mathbb{N}$ eine Konstante.

$$\begin{aligned} \Sigma &\equiv x_1 \mid x_2 \mid \ldots \mid x_n \\ R^+ &\equiv RR^* \\ R^0 &\equiv \epsilon \\ R^k &\equiv \underbrace{RR\ldots R}_{k\text{-mal}} \end{aligned}$$

Beispiel 3.4
Es sei $\Sigma = \{0, 1\}$ ein Alphabet.

Die folgenden Wörter R sind dann reguläre Ausdrücke über Σ und repräsentieren die jeweiligen regulären Sprachen $L(R)$ über Σ:

R	$L(R)$		
$\emptyset$	$L(\emptyset)$	$=$	$\emptyset$
ϵ	$L(\epsilon)$	$=$	$\{\epsilon\}$
0	$L(0)$	$=$	$\{0\}$
1	$L(1)$	$=$	$\{1\}$
$(\emptyset \mid \epsilon)$	$L((\emptyset \mid \epsilon))$	$=$	$\emptyset \cup \{\epsilon\} = \{\epsilon\}$
$(\epsilon \mid 0)$	$L((\epsilon \mid 0))$	$=$	$\{\epsilon\} \cup \{0\} = \{\epsilon, 0\}$
$(0 \mid 1)$	$L((0 \mid 1))$	$=$	$\{0\} \cup \{1\} = \{0, 1\}$
$(\epsilon 0)$	$L((\epsilon 0))$	$=$	$\{\epsilon\}\{0\} = \{\epsilon 0\} = \{0\}$
(01)	$L((01))$	$=$	$\{0\}\{1\} = \{01\}$
(0^*)	$L((0^*))$	$=$	$\{0\}^* = \{0^i \mid i = 0, 1, 2, \ldots\}$
$((0 \mid 1)^*)$	$L(((0 \mid 1)^*))$	$=$	$\{0, 1\}^* = \Sigma^*$
$((0 \mid 1)(0^*))$	$L(((0 \mid 1)(0^*)))$	$=$	$\{0, 1\}\{0^i \mid i = 0, 1, 2, \ldots\}$
		$=$	$\{00^i, 10^i \mid i = 0, 1, 2, \ldots\}$
$(((0 \mid 1)(0^*))^*)$	$L((((0 \mid 1)(0^*))^*))$	$=$	$\{00^i, 10^i \mid i = 0, 1, 2, \ldots\}^* = \{0, 1\}^* = \Sigma^*$

□

Beispiel 3.5
Betrachte nochmals das Beispiel 2.5, das den Ansatz einer Modellierung eines lexikalischen Analysators durch endliche Automaten skizziert. Die elementaren Symbole, die diese endlichen Automaten analysieren, lassen sich durch reguläre Ausdrücke definieren.

Sei $\Sigma = \{a, b, c, \ldots, z, 0, 1, 2, \ldots, 9\}$ ein Alphabet. Mit den regulären Ausdrücken

$$\begin{aligned} \mathit{buchstabe} &\equiv a \mid b \mid c \mid \ldots \mid z \text{ und} \\ \mathit{ziffer} &\equiv 0 \mid 1 \mid 2 \mid \ldots \mid 9 \end{aligned}$$

können dann die elementaren Symbole einer Programmiersprache wie folgt definiert werden:

1. Identifikator bzw. Bezeichner: $\mathit{buchstabe}(\mathit{buchstabe} \mid \mathit{ziffer})^*$
2. ganze Zahl ohne Vorzeichen: ziffer^+
3. reserviertes Wort for: for

Ein Identifikator beginnt mit einem Buchstaben – auf Großbuchstaben sei hier verzichtet – und wird von keinem oder endlich vielen Buchstaben oder Ziffern gefolgt. Eine Längenbeschränkung von Identifikatoren gibt es bei dieser Definition nicht.

Eine ganze Zahl ohne Vorzeichen besteht aus einer oder endlich vielen Ziffern. Führende Nullen sind bei dieser Definition erlaubt. Eine Beschränkung des Wertebereichs – wie in Programmiersprachen im Allgemeinen üblich – ist mit dieser Definition nicht gegeben.

Das reservierte Wort for besteht aus der Konkatenation der Zeichen f, o und r.

□

Beispiel 3.6
Eine weitere Anwendung von regulären Ausdrücken ist zur Formulierung von Suchanfragen nach Stichwörtern in Texten gegeben.

Werden z. B. über eine Suchmaschine Informationen zu regulären Ausdrücken gesucht. Unter Berücksichtigung, dass diese Begriffe im Singular oder im Plural vorkommen können, kann die Suchanfrage wie folgt formuliert werden:

$$\mathit{regulär}(er|e|en)\ \ \mathit{Ausdr}(u|ü)ck(\epsilon|e|en)$$

Das zugrunde liegende Alphabet umfasst hier mindestens die Zeichen

$$\Sigma = \{\sqcup, a, b, \ldots, z, A, B, \ldots, Z\}.$$

Das erste Zeichen in Σ ist dabei das Leerzeichen ($Blank$), das bei Textverarbeitung immer benötigt wird. Zu beachten ist, dass es verschieden von dem leeren Wort ist.

In der Praxis wird in solchen Anwendungen aber häufig eine hiervon abweichende Darstellung von regulären Ausdrücken verwendet.

□

Aus der Definition der Äquivalenz zweier regulärer Ausdrücke (Definition 3.5) ergeben sich Rechenregeln für reguläre Ausdrücke. Einige davon sind in dem Satz 3.4 zusammengefasst.

Satz 3.4 (Rechenregeln)
Es seien R, S und T reguläre Ausdrücke über Σ. Dann gelten:

1.	$(S) \equiv S$	*(Klammerung)*
2.	$S \mid T \equiv T \mid S$	*(Kommutativität von $\mid$)*
3.	$R \mid (S \mid T) \equiv (R \mid S) \mid T$	*(Assoziativität von $\mid$)*
4.	$R(ST) \equiv (RS)T$	*(Assoziativität von $\cdot$)*
5.	$R(S \mid T) \equiv (RS) \mid (RT)$ $(S \mid T)R \equiv (SR) \mid (TR)$	*(Distributivität)*
6.	$S \mid S \equiv S$	*(Idempotenz von $\mid$)*
7.	$S \mid \emptyset \equiv S$	*(Einselement bzgl. $\mid$)*
8.	$S\epsilon \equiv \epsilon S \equiv S$	*(Einselement bzgl. $\cdot$)*
9.	$S\emptyset \equiv \emptyset S \equiv \emptyset$	*(Nullelement bzgl. $\cdot$)*

Beweis:

1. $L((S)) = L(S)$
2. $L(S \mid T) = L(S) \cup L(T) = L(T) \cup L(S) = L(T \mid S)$
3. $L(R \mid (S \mid T)) = L(R) \cup (L(S) \cup L(T)) = (L(R) \cup L(S)) \cup L(T) = L((R \mid T) \mid S)$
4. $L(R(ST)) = L(R)(L(S)L(T)) = (L(R)L(S))L(T) = L((RT)S)$
5. $L(R(S \mid T)) = L(R)(L(S) \cup L(T)) = L(R)L(S) \cup L(R)L(T) = L((RS) \mid (RT))$
   $L((S \mid T)R) = (L(S) \cup L(T))L(R) = L(S)L(R) \cup L(T)L(R) = L((SR) \mid (TR))$
6. $L(S \mid S) = L(S) \cup L(S) = L(S)$
7. $L(S \mid \emptyset) = L(S) \cup \emptyset = L(S)$
8. $L(S\epsilon) = L(S)\{\epsilon\} = L(S)$
   $L(\epsilon S) = \{\epsilon\}L(S) = L(S)$
9. $L(S\emptyset) = L(S)\emptyset = \emptyset$
   $L(\emptyset S) = \emptyset L(S) = \emptyset$

□

Beispiele 3.7
Sei $\Sigma = \{0, 1\}$ ein Alphabet. Dann sind die folgenden regulären Ausdrücke über Σ äquivalent bzw. nicht äquivalent:

1. $00 \mid 10 \equiv 10 \mid 00$
2. $(00 \mid 10)0 \equiv 000 \mid 100$
3. $10 \not\equiv 01$

□

Das letzte Beispiel zeigt insbesondere, dass die Konkatenation regulärer Ausdrücke nicht kommutativ ist.

Nachfolgend wird nun gezeigt, dass jede Sprache, die von einem endlichen Automaten erkannt werden kann, auch durch einen regulären Ausdruck dargestellt werden

kann, und dass es zu jeder durch einen regulären Ausdruck repräsentierten Sprache auch einen endlichen Automaten gibt, der diese Sprache erkennt. Deterministische endliche Automaten, nichtdeterministische endliche Automaten und reguläre Ausdrücke definieren somit ein und dieselbe Sprachfamilie, die Klasse der regulären Sprachen.

Satz 3.5 (Äquivalenz mit endlichen Automaten)
Eine Sprache ist genau dann regulär, wenn sie durch einen regulären Ausdruck repräsentierbar ist.

Zum Beweis dieses Satzes sind zwei Richtungen zu zeigen. Dies erfolgt über die Lemmata 3.6 und 3.7.

Lemma 3.6
Eine Sprache ist regulär, wenn sie durch einen regulären Ausdruck repräsentierbar ist.

Beweis:
Sei ein regulärer Ausdruck R über dem Alphabet Σ gegeben. Es sind sechs Fälle zu betrachten, da ein regulärer Ausdruck nach Definition aus sechs verschiedenen Fällen aufgebaut sein kann. Für jeden Fall wird ein äquivalenter NEA konstruiert. Insgesamt ergibt sich damit ein NEA N, für den $L(N) = L(R)$ gilt. Eine von einem NEA erkennbare Sprache ist auch von einem DEA erkennbar und somit regulär.

1. $R = a$ für $a \in \Sigma$. Dann ist $L(R) = \{a\}$ und der NEA
$$N = (\{q_1, q_2\}, \Sigma, \delta, q_1, \{q_2\})$$
$$\delta(q, x) = \begin{cases} \{q_2\} & \text{für } q = q_1,\ x = a \\ \emptyset & \text{für } q \neq q_1 \text{ oder } x \neq a \end{cases}$$
erkennt $\{a\}$.
2. $R = \epsilon$. Dann ist $L(R) = \{\epsilon\}$ und der NEA
$$N = (\{q_1\}, \Sigma, \delta, q_1, \{q_1\})$$
$$\delta(q, x) = \emptyset \text{ für jedes } q \text{ und } x$$
erkennt $\{\epsilon\}$.
3. $R = \emptyset$. Dann ist $L(R) = \emptyset$ und der NEA
$$N = (\{q_1\}, \Sigma, \delta, q_1, \emptyset)$$
$$\delta(q, x) = \emptyset \text{ für jedes } q \text{ und } x$$
erkennt $\{\emptyset\}$.
4. $R = R_1 \mid R_2$.
5. $R = R_1 \cdot R_2$.
6. $R = R_1^*$.

Die Zustandsgraphen der konstruierten nichtdeterministischen endlichen Automaten der ersten drei Fälle sind in Abbildung 3.5 dargestellt.

Zum Nachweis der letzten drei Fälle kann auf die Beweise der Sätze 3.1, 3.2, 3.3 zurück gegriffen werden. Dort war die Abgeschlossenheit der regulären Sprachen bzgl. der Operationen Vereinigung, Konkatenation und Stern mittels NEAs bewiesen worden. Siehe dazu auch die Abbildungen 3.1, 3.2 und 3.3. □

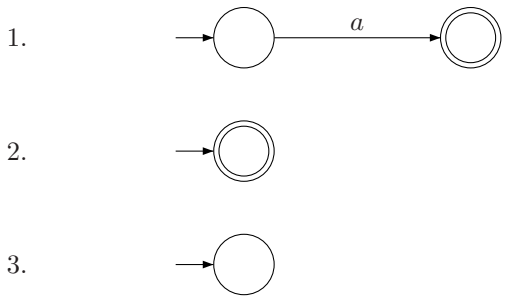
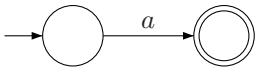


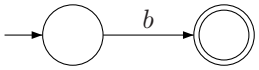
Abbildung 3.5: NEAs zum Beweis von Lemma 3.6, Fälle 1, 2, und 3

Beispiel 3.8
Sei der reguläre Ausdruck $a(a|b)^*$ über dem Alphabet $\{a, b\}$ gegeben.
Konstruktion eines NEA N mit $L(N) = L(a(a|b)^*)$:

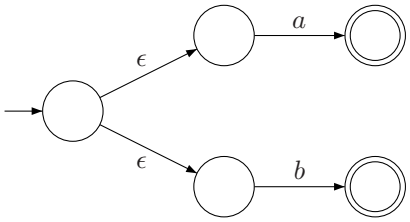
a:



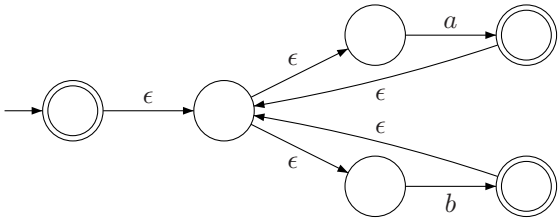
b:



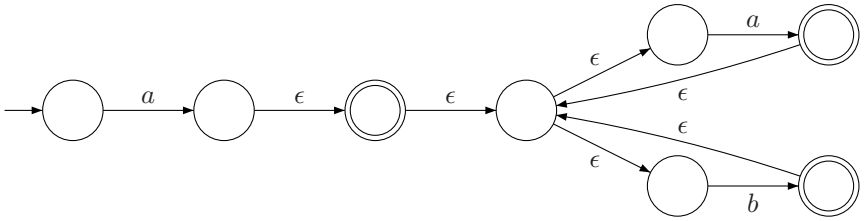
$a|b$:



$(a|b)^*$:



$a(a|b)^*$:



□

Lemma 3.7
Eine Sprache ist durch einen regulären Ausdruck repräsentierbar, wenn sie regulär ist.

Beweis:
Sei eine reguläre Sprache durch einen DEA gegeben. Die Konstruktion eines äquivalenten regulären Ausdrucks könnte dann z.B. durch sukzessive Elimination von Zuständen über so genannte *Generalisierte nichtdeterministische endliche Automaten* erfolgen. Auf diese Konstruktion sei hier aber verzichtet. Sie kann in [4] nachgelesen werden. □

Zum Abschluss dieses Abschnitts soll noch einmal auf das Eingangsbeispiel zu den deterministischen endlichen Automaten eingegangen werden. Die durch diesen Automaten definierte Sprache ist natürlich auch durch einen regulären Ausdruck darstellbar.

Beispiel 3.9
Sei der deterministische endlichen Automaten aus Beispiel 2.3 gegeben. Er erkennt die reguläre Sprache $L = \{w \mid w \in \{a, b\}^*,\ |w|_a \bmod 3 = 0\}$.

Der reguläre Ausdruck $b^*(ab^*ab^*ab^*)^*$ repräsentiert die Sprache L.

Anmerkung: Dieser reguläre Ausdruck ist unabhängig von dem gegebenen DEA und ohne ein Konstruktionsverfahren erstellt worden.

□

3.3 Eigenschaften regulärer Sprachen

Nach der Einführung und Definition der regulären Sprachen gilt es, einige ihrer Eigenschaften zu untersuchen. So werden hier Abschlusseigenschaften einiger Operationen und eine notwendige, aber leider nicht hinreichende, Eigenschaft betrachtet, die „Pump“-Eigenschaft. Im Abschnitt 3.1 ist bereits die Abgeschlossenheit der regulären Operationen Vereinigung, Konkatenation und Stern (Kleeneabschluss) gezeigt worden. Insbesondere diese hätten auch zur Definition der regulären Sprachen herangezogen werden können. Nach dem so genannten *Satz von Kleene* ist die kleinste Sprachfamilie über Σ, die unter Vereinigung, Konkatenation und Stern abgeschlossen ist und die Sprachen $\emptyset$, $\{\epsilon\}$ und $\{a\}$ mit $a \in \Sigma$ enthält, die der regulären Sprachen.

Die weiteren hier betrachteten Operationen und Probleme werden alle abgeschlossen sein. Aber es gibt auch nichtreguläre Sprachen, für die diese Aussagen nicht mehr bzw. nur noch eingeschränkt gelten. Zum Nachweis, dass eine Sprache nicht regulär ist, kann das so genannte *Pumping Lemma* herangezogen werden. Es charkterisiert eine Eigenschaft regulärer Sprachen, die dann nicht mehr vorhanden ist.

Für Sprachen sind neben den regulären Operationen (Definition 3.2) auch die Operationen Komplement und Durchschnitt von Interesse (Definition 3.6).

Definition 3.6 (Weitere Operationen)
Seien A und B Sprachen über dem Alphabet Σ. Die Operationen Durchschnitt und Komplement werden dann wie folgt definiert:

- ***Durchschnitt**: $A \cap B = \{x \mid (x \in A) \wedge (x \in B)\}$.*
- ***Komplement**: $\overline{A} = \Sigma^* \setminus A$.*

Satz 3.8 (Komplement)
Es sei L eine reguläre Sprache über dem Alphabet Σ. Dann ist auch $\overline{L}$ eine reguläre Sprache.

Beweis:
Da L eine reguläre Sprache über Σ, gibt es einen deterministischen endlichen Automaten $E = (Q, \Sigma, \delta, q_0, F)$ mit $L(E) = L$.

Der endliche Automat $E' = (Q, \Sigma, \delta, q_0, F')$ mit $F' = Q \backslash F$ erkennt dann die Sprache $L(E') = \overline{L}$, denn E' akzeptiert ein Wort $w \in \Sigma^*$ genau dann, wenn E das Wort w nicht akzeptiert. □

Satz 3.9 (Durchschnitt)
Es seien L_1 und L_2 reguläre Sprachen. Dann ist auch $L_1 \cap L_2$ eine reguläre Sprache.

Beweis:
Wegen

$$L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$$

und den Sätzen 3.1 und 3.8 ist auch der Durchschnitt zweier regulärer Sprachen eine reguläre Sprache. □

Der Satz 3.9 kann natürlich auch konstruktiv über endliche Automaten bewiesen werden. Dies soll hier aber nicht erfolgen.

Zum Abschluss dieses Abschnitts wird eine Eigenschaft regulärer Sprachen betrachtet, deren Fehlen zum Nachweis für die Nichtregularität einer Sprache verwendet wird (Satz 3.10). In dem sich daran anschließenden Beispiel (Beispiel 3.11) wird dann eine nichtreguläre Sprache spezifiziert. Es gibt also Sprachen, die nicht regulär sind.

Satz 3.10 (Pumping Lemma)
Es sei L eine reguläre Sprache. Dann gibt es eine Konstante $n \in \mathbb{N}$, so dass jedes Wort $w \in L$ mit $|w| \geq n$ in drei Teilwörter $w = xyz$ mit

1. $y \neq \epsilon$
2. $|xy| \leq n$
3. $\forall\, k \in \mathbb{N}_0 : xy^k z \in L$

zerlegt werden kann.

Beweis:
Es sei L eine reguläre Sprache über Σ. Dann gibt es einen endlichen Automaten $E = (Q, \Sigma, \delta, q_0, F)$ mit $L = L(E)$. E habe n Zustände.

Betrachte nun ein Wort $w \in L$ mit $|w| = m \geq n$ und $w = a_1 a_2 \ldots a_m$ mit $a_i \in \Sigma$, $1 \leq i \leq m$. Die Berechnung von w durch E kann dann durch die folgende Zustandsfolge beschrieben werden:

$$q_0, \ldots, q_i, \ldots, q_m$$

$q_i \in Q$ ist also der Zustand, in dem sich E nach Abarbeitung des Präfixes $a_1 a_2 \ldots a_i$ von w befindet und $q_m \in F$. Einschließlich dem Startzustand durchläuft E $m+1$ Zustände. Da aber $|Q| = n < m+1$, muss es nach dem Schubfachprinzip bereits bei Abarbeitung des Präfixes $a_1 a_2 \ldots a_n$ zwei Zustände q_r und q_s geben, so dass $q_r = q_s$, $0 \leq r < s \leq n$:

$$q_0, \ldots, q_r \ldots, q_s, \ldots, q_m$$

Es kann also $w = xyz$ wie folgt in Teilwörter zerlegt werden:

$$\begin{aligned} x &= a_1 \ldots a_r \\ y &= a_{r+1} \ldots a_s \\ z &= a_{s+1} \ldots a_m \end{aligned}$$

Vom Startzustand q_0 wird über x der Zustand q_r erreicht, über y wieder q_r, denn $q_r = q_s$, und über z schließlich q_m.

Wegen $0 \leq r < s \leq n$ ist $y \neq \epsilon$ und $|xy| \leq n$.

Betrachte das Wort $xy^k z$ für $k \in \mathbb{N}_0$:

I. Für $k = 0$ gibt es die Berechnung mit der Zustandsfolge

$$q_0, \ldots, q_r, \ldots, q_m$$

also $xz \in L$.

II. Für $k > 0$ gibt es die Berechnung mit der Zustandsfolge

$$q_0, \ldots, q_r \ldots, q_r, \ldots, q_r, \ldots, q_r, \ldots, q_m$$

also $xy^k z \in L$.

Veranschaulichen lassen sich diese Berechnungspfade über Abbildung 3.6. □

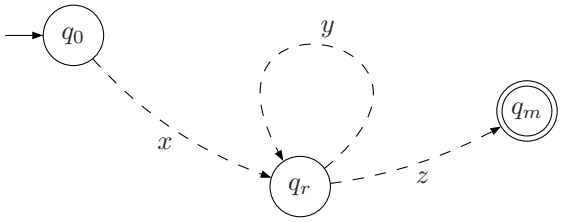


Abbildung 3.6: Berechnung von $xy^k z \in L$ mit $k \geq 0$

Beispiel 3.10
Betrachte die reguläre Sprache $L = \{a^i b^j \mid i, j \in \mathbb{N}_0\}$, die durch den regulären Ausdruck $a^* b^*$ definiert wird.

Wähle $n = 1$ und betrachte die Wörter $a^p b^m$, $p \in \mathbb{N}$, $m \in \mathbb{N}_0$. Es gilt $|a^p b^m| = p + m \geq 1$. $a^p b^m$ kann dann in die Teilwörter $x = \epsilon$, $y = a$ und $z = a^{p-1} b^m$ zerlegt werden. Für $a^p b^m = xyz$ gilt dann

1. $y = a \neq \epsilon$,
2. $|xy| = |a| \leq 1$,
3. $\forall\, k \in \mathbb{N}_0 : xy^k z = a^k a^{p-1} b^m = a^{k+p-1} b^m \in L$.

Die Wörter $a^0 b^m$, $m \geq 1$, können in die Teilwörter $x = \epsilon$, $y = b$ und $z = b^{m-1}$ zerlegt werden. Die Eigenschaften 1. bis 3. sind dann auch erfüllt.

□

Beispiel 3.11
Betrachte die Sprache $L = \{a^i b^i \mid i \in \mathbb{N}_0\}$.

Annahme: L sei regulär.

Dann muss es eine Konstante $n \in \mathbb{N}$ geben, so dass jedes Wort $w \in L$ mit $|w| \geq n$ in drei Teilwörter $w = xyz$ zerlegt werden kann, für die die Bedingungen des Pumping Lemmas erfüllt sind.

Betrachte das Wort $w = a^m b^m$ mit $m \geq n$:

Es gilt $|w| = |a^m b^m| = 2m > m \geq n$. Wegen $|xy| \leq n \leq m$, besteht dann xy ausschließlich aus a's. Daraus ergibt sich die folgende Zerlegung von $w = xyz$:

I. $x = a^r$ mit $r < n$, da $y \neq \epsilon$, d. h. $|y| > 0$.

II. $y = a^s$ mit $s > 0$ und $r + s \leq n$.

III. $z = a^t b^m$ mit $r + s + t = m$.

Nach dem Pumping Lemma muss nun für jedes $k \in \mathbb{N}_0$ $xy^kz \in L$ sein.

Mit $k = 0$ gilt

$$xy^0z = a^r(a^s)^0a^tb^m = a^ra^tb^m = a^{r+t}b^m.$$

Da aber $r + t < m$, ist $xy^0z \notin L$. Es ergibt sich also ein Widerspruch zum Pumping Lemma.

Die Annahme, L sei regulär, muss also falsch sein.

□

Satz 3.11 (Existenz nichtregulärer Sprachen)
Es gibt Sprachen, die nicht regulär sind.

Beweis:
Die Sprache $\{a^ib^i \mid i \in \mathbb{N}_0\}$ ist nicht regulär (Beispiel 3.11). □

Kapitel 4

Kontextfreie Sprachen

4.1 Kontextfreie Grammatiken

Bisher sind zwei Konzepte zur Definition regulärer Sprachen betrachtet worden, ein die Wörter der Sprache *analysierendes* Konzept durch endliche Automaten und ein die Wörter *beschreibendes* Konzept durch reguläre Ausdrücke. In diesem Abschnitt soll dem ein weiteres grundlegendes Konzept hinzugefügt werden, ein *generierendes* Konzept. Die Wörter einer Sprache werden mittels *Grammatiken* generiert.

Weiterhin wurde gezeigt, dass es Sprachen gibt, die nicht reguär sind. Die *kontextfreien Grammatiken* – es handelt sich dabei um einen speziellen Grammatiktyp – sind in der Lage, auch rekursive Strukturen in Sprachen zu beschreiben. Sie sind also mächtiger als die bisher betrachteten Verfahren und werden deshalb in einer Vielzahl von Anwendungen verwandt. Sprachen, die über kontextfreie Grammatiken definiert werden, werden *kontextfreie Sprachen* genannt.

Kontextfreie Grammatiken wurden zuerst im Zusammenhang mit der Beschreibung der Struktur natürlicher Sprachen studiert, um die Beziehungen von Subjekt, Prädikat und Objekt zu beschreiben. In der Informatik werden kontextfreie Grammatiken z.B. zur

- Spezifikation von arithmetischen, logischen, relationalen Ausdrücken
- Spezifikation von Blockstrukturen in Programmiersprachen (z. B. Pascal) und Markierungssprachen (z. B. HTML, XML)
- Spezifikation der Syntax in Progammiersprachen (z. B. Pascal)

herangezogen. Ihre Notation weicht allerdings häufig von der hier eingeführten ab. So werden in Programmiersprachen als kontextfreie Produktionsregeln u.a. die Backus-Naur-Form oder auch Syntaxdiagramme verwendet. In Markierungssprachen z.B. treten die Markierungen immer paarweise als öffnende und schließende Klammer in der Form `<x>` und `</x>` auf.

Zur Analyse, Interpretation oder Übersetzung solcher mittels kontextfreier Grammatiken definierter Strukturen wird ein so genannter *Parser* oder *Syntax-Analysator* verwendet. Das dahinter stehende Berechnungsmodell ist ein *Kellerautomat*, der im nächsten Abschnitt besprochen wird. Sobald die Struktur einer Sprache durch eine kontextfreie Grammatik definiert ist, kann automatisch ein Kellerautomat generiert werden, der genau die Wörter dieser Sprache akzeptiert.

Beispiel 4.1
Die folgenden Regeln (Produktionsregeln oder auch Ersetzungsregeln) einer kontextfreien Grammatik definieren einen sehr vereinfachten Ausschnitt der deutschen Sprache. Sei das zugrunde liegende Alphabet $\Sigma = \{$ ␣, `a`, `b`, `c`, ... , `z`, `A`, `B`, `C`, ... , `Z` $\}$.

⟨Satz⟩	→	⟨Substantiv-Phrase⟩⟨Verb-Phrase⟩
⟨Substantiv-Phrase⟩	→	⟨Substantiv⟩ | ⟨Artikel⟩⟨Substantiv⟩ | ⟨Pronomen⟩
⟨Verb-Phrase⟩	→	⟨Verb⟩ | ⟨Verb⟩⟨Substantiv-Phrase⟩
⟨Artikel⟩	→	`ein` | `eine` | `der` | `die` | `das`
⟨Substantiv⟩	→	`Mann` | `Hund` | `Katze`
⟨Pronomen⟩	→	`ich` | `wir`
⟨Verb⟩	→	`sehen` | `mögen` | `beißen`

Durch fortlaufende Ersetzung von Zeichen mittels dieser Regeln kann ausgehend vom Symbol ⟨Satz⟩ der Satz „`wir sehen eine Katze` “ abgeleitet werden. Das Leerzeichen zwischen z.B. „wir“ und „sehen“ wird dadurch erhalten, dass die jeweiligen Wörter jeweils mit einem Leerzeichen abschließen.

$$
\begin{aligned}
\langle \text{Satz} \rangle &\Rightarrow \langle \text{Substantiv-Phrase} \rangle \langle \text{Verb-Phrase} \rangle \\
&\Rightarrow \langle \text{Pronomen} \rangle \langle \text{Verb-Phrase} \rangle \\
&\Rightarrow \texttt{wir}\ \langle \text{Verb-Phrase} \rangle \\
&\Rightarrow \texttt{wir}\ \langle \text{Verb} \rangle \langle \text{Substantiv-Phrase} \rangle \\
&\Rightarrow \texttt{wir sehen}\ \langle \text{Substantiv-Phrase} \rangle \\
&\Rightarrow \texttt{wir sehen}\ \langle \text{Artikel} \rangle \langle \text{Substantiv} \rangle \\
&\Rightarrow \texttt{wir sehen eine}\ \langle \text{Substantiv} \rangle \\
&\Rightarrow \texttt{wir sehen eine Katze}
\end{aligned}
$$

Der Satz „`eine Katze wir sehen`“ hat keine Ableitung mit diesen Regeln.

□

Definition 4.1 (Kontextfreie Grammatik)
Eine kontextfreie Grammatik ist ein 4-Tupel (V, Σ, R, S) mit

1. *V ist eine endliche, nichtleere Menge von Variablen bzw. Nichtterminalzeichen,*
2. *Σ ist ein Alphabet von Terminalszeichen, $V \cap \Sigma = \emptyset$,*
3. *$R = \{A \to w \mid A \in V,\ w \in (V \cup \Sigma)^*\}$ ist eine endliche Menge von Regeln oder Produktionen,*
4. *$S \in V$ ist die Startvariable.*

Regeln mit gleicher linker Seite

$$A \to w_1, A \to w_2, \ldots, A \to w_n \text{ werden kurz } A \to w_1 \mid w_2 \mid \ldots \mid w_n$$

geschrieben, $n \in \mathbb{N}$.

Charakteristisch für die kontextfreien Produktionen ist, dass die linke Seite aus genau einem Nichtterminalzeichen und die rechte Seite aus einem Wort über den Terminal- oder Nichtterminalzeichen der Grammatik besteht. Daher auch der Name, denn in einer Ableitung wird ein Nichtterminalzeichen unabhängig von seinen Nachbarzeichen – seinem Kontext – durch die rechte Seite einer anwendbaren Produktion ersetzt. Die von kontextfreien Grammatiken erzeugten Sprachen werden nun auch *kontextfreie Sprachen* genannt.

Definition 4.2 (Ableitung)
Sei $G = (V, \Sigma, R, S)$ eine kontextfreie Grammatik. Falls $u, v, w \in (V \cup \Sigma)^*$ Wörter sind und $A \to w \in R$ eine Produktion, dann wird uAv in uwv überführt, geschrieben $uAv \Rightarrow uwv$.
Existiert eine Folge solcher Überführungen $u_1, u_2, \ldots, u_k$ für $k \geq 0$ und

$$u \Rightarrow u_1 \Rightarrow u_2 \Rightarrow \ldots \Rightarrow u_k \Rightarrow v$$

oder gilt $u = v$, dann wird u zu v abgeleitet, geschrieben $u \Rightarrow^* v$.

Definition 4.3 (Sprache einer Grammatik)
Die von einer Grammatik $G = (V, \Sigma, R, S)$ erzeugte oder generierte Sprache $L(G)$ ist die Menge aller Wörter über den Terminalzeichen, die von der Startvariablen abgeleitet werden können,

$$L(G) = \{w \in \Sigma^* \mid S \Rightarrow^* w\}.$$

Definition 4.4 (Kontextfreie Sprache)
Eine Sprache wird kontextfreie Sprache genannt, wenn es eine kontextfreie Grammatik gibt, die sie erzeugt. Die Klasse oder Sprachfamilie der kontextfreien Sprachen wird mit $\mathcal{L}_{kf}$ bezeichnet.

Beispiel 4.2
Das Beispiel 4.1 ist formal somit wie folgt zu spezifizieren: $G_0 = (V, \Sigma, R, S)$ mit

- $V = \{\langle\text{Satz}\rangle, \ldots, \langle\text{Verb}\rangle\}$
- $\Sigma = \{\ \sqcup, \texttt{a}, \ldots, \texttt{Z}\}$
- R ist die Menge mit den oben aufgeführten Regeln
- $S = \langle\text{Satz}\rangle$

`wir sehen eine Katze` $\in L(G_0)$, denn `wir sehen eine Katze` $\in \Sigma^*$ und es gibt die Ableitung $\langle\text{Satz}\rangle \Rightarrow^*$ `wir sehen eine Katze` in G_0.

□

Beispiel 4.3
Sei die kontextfreie Grammatik $G_1 = (\{S\}, \{a, b\}, \{S \to aSb,\ S \to \epsilon\}, S)$ gegeben.

Betrachte einige Ableitungen:

$S \Rightarrow \epsilon$
$S \Rightarrow aSb \Rightarrow ab$
$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb$
$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb \Rightarrow aaabbb$
$\vdots$
$S \Rightarrow aSb \Rightarrow \ldots \Rightarrow a^nSb^n \Rightarrow a^nb^n$, $n \in \mathbb{N}_0$

$L(G_1) = \{a^nb^n \mid n \in \mathbb{N}_0\}$ ist eine kontextfreie Sprache.

□

Die Sprache $L(G_1)$ ist also kontextfrei. Zur Erinnerung, in Beispiel 3.11 wurde gezeigt, dass diese Sprache nicht regulär ist.

Zur Veranschaulichung von Ableitungen insbesondere in den einleitend genannten Anwendungen haben sich *Ableitungsbäume* als sehr hilfreich erwiesen. Sie sollen deshalb hier in allgemeiner Form eingeführt werden.

Definition 4.5 (Ableitungsbaum)
Es seien eine kontextfreie Grammatik $G = (V, \Sigma, R, S)$ und eine Ableitung $A \Rightarrow^* w$ mit $A \in V,\ w \in (V \cup \Sigma)^*$ gegeben. Die Darstellung der Ableitung heißt Ableitungsbaum, falls gilt:

- *Alle inneren Knoten sind mit Zeichen aus V benannt.*
- *Alle Blätter sind mit Zeichen aus $V \cup \Sigma$ oder mit dem Zeichen ϵ benannt. Ein Blatt ϵ ist der einzige Kindknoten seines Vaterknotens.*
- *Sei $w_1 \Rightarrow w_2$ ein Ableitungsschritt aus $A \Rightarrow^* w$, der auf Anwendung der Produktion $B \to X_1 \dots X_n$ resultiert, wobei $X_i \in V \cup \Sigma,\ 1 \leq i \leq n$, oder $X_1 = \epsilon$ und $n = 1$. Die Knoten X_i sind in der Reihenfolge von 1 bis n die Nachfolger des Knotens B.*

Ein Ableitungsbaum mit dem Startzeichen der Grammatik als Wurzel und Blättern ausschließlich aus Terminalzeichen, also einer vollständigen Ableitung eines von der Grammatik erzeugten Wortes, ist ein wichtiger Spezialfall. In Abbildung 4.1 ist ein Ableitungsbaum für ein Wort aus Beispiel 4.3 gegeben.

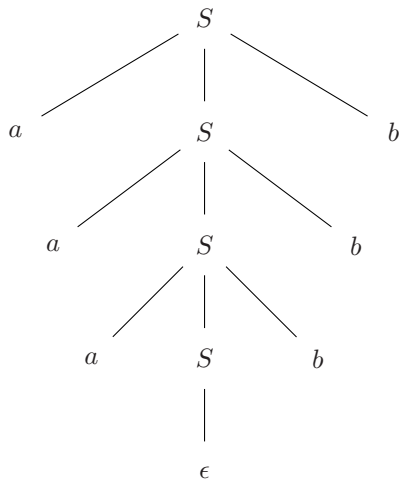


Abbildung 4.1: Ableitungsbaum für *aaabbb* bzgl. der Grammatik G_1

Zu den kontextfreien Sprachen gehören auch solche, die Spiegelstrukturen mit und ohne Mittekennung oder Klammerstrukturen enthalten. Hier sei eine kontextfreie Grammatik spezifiziert, die Klammerstrukturen generiert.

Beispiel 4.4
Sei die kontextfreie Grammatik $G_3 = (\{S\}, \{(,)\}, \{S \to (S) \mid SS \mid \epsilon\}, S)$ gegeben.

Betrachte einige Ableitungen:

$S \Rightarrow \epsilon$
$S \Rightarrow (S) \Rightarrow (\ )$
$S \Rightarrow SS \Rightarrow (S)S \Rightarrow ((S))S \Rightarrow ((\ ))S \Rightarrow ((\ ))(S) \Rightarrow ((\ ))(\ )$
$S \Rightarrow SS \Rightarrow S(S) \Rightarrow S(\ ) \Rightarrow (S)(\ ) \Rightarrow ((S))(\ ) \Rightarrow ((\ ))(\ )$

$L(G_3) = \{w \mid w \in \{(,)\}^*$ ist ein Wort einer richtig geschachtelten Klammerung$\}$ ist eine kontextfreie Sprache.

□

Das Beispiel 4.4 zeigt, dass es kontextfreie Grammatiken gibt, mit denen ein Wort über verschiedene Ableitungen erzeugt werden kann. Für diese beiden Ableitungen gibt es aber nur einen Ableitungsbaum. Er ist in Abbildung 4.2 dargestellt.

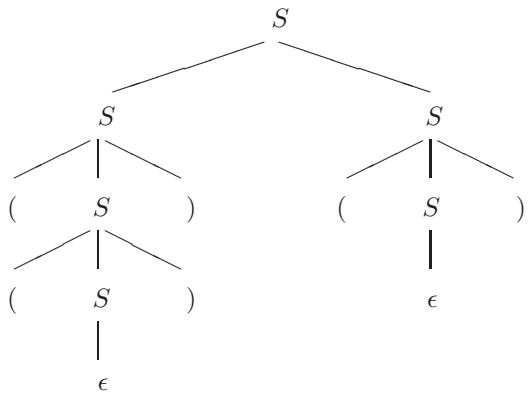


Abbildung 4.2: Ableitungsbaum des Wortes (( ))( )

Nun führen verschiedene Ableitungen eines Wortes leider nicht immer auf den gleichen Ableitungsbaum. Betrachte dazu das Beispiel 4.5.

Beispiel 4.5
Es sei $G_4 = (\{E\}, \{a, +, *, (,)\}, R, E)$ mit

$$R = \{E \to E + E \mid E * E \mid (E) \mid a\}$$

gegeben.

Betrachte dazu die folgenden Ableitungen:

I. $E \Rightarrow E + E \Rightarrow E + E * E \Rightarrow^* a + a * a$

II. $E \Rightarrow E * E \Rightarrow E + E * E \Rightarrow^* a + a * a$

III. $E \Rightarrow E * E \Rightarrow (E) * E \Rightarrow (E + E) * E \Rightarrow^* (a + a) * a$

Die von G_4 erzeugten Wörter können als arithmetische Ausdrücke interpretiert werden. Die Auswertung dieser Ausdrücke muss sich nun an ihrer Struktur orientieren, und diese wird durch den zugehörigen Ableitungsbaum bestimmt. Betrachte dazu den Ableitungsbaum der Ableitung I (Abbildungen 4.3).

Die Auswertung von $a + a * a$ muss ausgehend von den Blättern zur Wurzel hin erfolgen, denn nur bei den Blättern sind die Werte des Terminalzeichens a – in Programmiersprachen spricht man dann von Identifikatoren – bekannt. Die Wurzel hält dann den Wert des gesamten Ausdrucks. Also, es wird erst das Produkt $a * a$ gebildet und dann wird a addiert. Dies entspricht der üblichen Auswertung arithmetischer Ausdrücke, Punkt- vor Strichrechnung.

Nun wird aber der Ausdruck $a + a * a$ auch durch die Ableitung II erzeugt. Der zugehörige Ableitungsbaum ist in Abbildung 4.4 dargestellt. Hier wird erst die Summe $a + a$ gebildet und dann a multipliziert. Dies entspricht nicht der üblichen Auswertungsreihenfolge. Wäre genau diese Auswertung erwartet, dann müßte geklammert werden (Ableitung III).

□

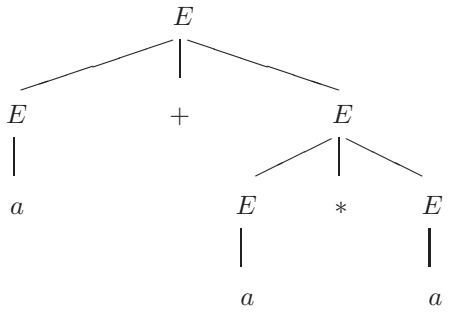


Abbildung 4.3: Ableitungsbaum für Wort $a + a * a$ (I)

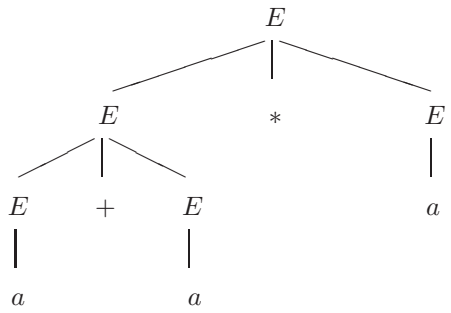


Abbildung 4.4: Ableitungsbaum für Wort $a + a * a$ (II)

Eine Grammatik, die z. B. arithmetische Ausdrücke oder auch die Syntax einer Programmiersprache definiert, darf solche strukturellen Mehrdeutigkeiten nicht enthalten. Darüber hinaus muss sie natürlich auch die richtige Struktur definieren. Die Grammatik G_4 aus Beispiel 4.5 ist also nicht geeignet, um arithmetische Ausdrücke und ihre Auswertung zu spezifizieren.

Allgemein werden die Grammatik-Eigenschaften *eindeutig* und *mehrdeutig* wie folgt definiert.

Definition 4.6 (Ein- und Mehrdeutigkeit)
Es sei G eine kontextfreie Grammatik.

- *G heißt eindeutig, wenn es zu jedem Wort $w \in L(G)$ genau einen Ableitungsbaum gibt.*
- *G heißt mehrdeutig, wenn es ein Wort $w \in L(G)$ mit mehreren Ableitungsbäumen gibt.*

Eine kontextfreie Sprache L heißt inhärent mehrdeutig, falls es keine eindeutige kontextfreie Grammatik für L gibt.

Die Grammatik G_4 ist also mehrdeutig, denn z.B. das Wort $a + a * a$ hat zwei Ableitungsbäume. Die Sprache $L(G_4)$ ist aber nicht inhärent mehrdeutig, betrachte dazu die Grammatik G_5 in Beispiel 4.6. Diese Grammatik ist äquivalent zu G_4, d.h. $L(G_4) = L(G_5)$, und eindeutig, was hier nicht bewiesen wird. Es gibt natürlich aber inhärent mehrdeutige Sprachen.

Beispiel 4.6
Sei die kontextfreie Grammatik $G_5 = (V, \Sigma, R, E)$ gegeben, mit

$$\begin{array}{lcll} V & = & \{ & E, T, F \quad \}, \\ \Sigma & = & \{ & a, +, *, (,) \quad \}, \\ R & = & \{ & E \rightarrow E + T \mid T, \\ & & & T \rightarrow T * F \mid F, \\ & & & F \rightarrow (E) \mid a \quad \}. \end{array}$$

Die Ableitungsbäume der Wörter $a + a * a$ und $(a + a) * a$ sind in den Abbildungen 4.5 und 4.6 dargestellt.

□

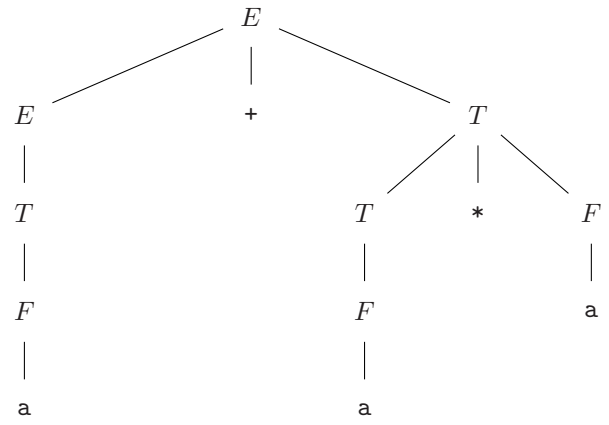


Abbildung 4.5: Ableitungsbaum für $a + a * a$ bzgl. Grammatik G_5

Abschließend in diesem Abschnitt wird noch eine Normalform für kontextfreie Grammatiken eingeführt, die *Chomsky Normalform*. Durch eine solche Normierung kann die Anwendung kontextfreier Grammatiken und die Herleitung von Eigenschaften kontextfreier Sprachen vereinfacht werden. So wird die hier definierte Normalform später zum Beweis des Pumping Lemmas für kontextfreie Sprachen benötigt.

Definition 4.7 (Chomsky Normalform)
Eine kontextfreie Grammatik (V, Σ, R, S) ist in Chomsky Normalform, wenn die Regeln von der folgenden Form sind:

$$\begin{array}{lcl} A & \rightarrow & BC \\ A & \rightarrow & a \end{array}$$

wobei $a \in \Sigma$ und $A, B, C \in V$, und B und C dürfen nicht die Startvariable sein. Zusätzlich ist die Regel $S \rightarrow \epsilon$ erlaubt.

Satz 4.1 (Chomsky Normalform)
Jede kontextfreie Sprache kann durch eine kontextfreie Grammatik in Chomsky Normalform generiert werden.

Beweis:
Eine kontextfreie Grammatik $G = (V, \Sigma, R, S)$ wird in Chomsky Normalform umgewandelt:

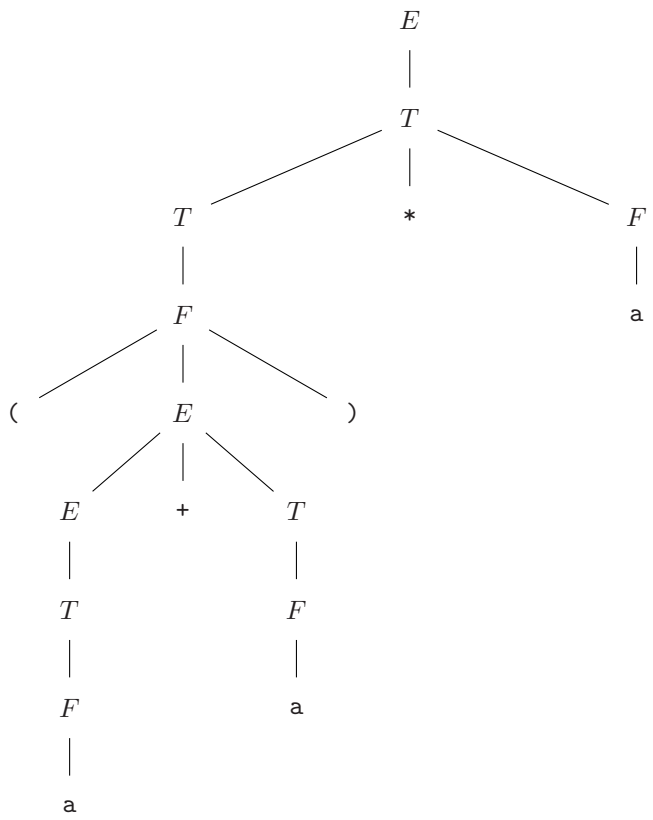


Abbildung 4.6: Ableitungsbaum für $(a + a) * a$ bzgl. Grammatik G_5

I. S_0 wird die neue Startvariable und die Regel $S_0 \to S$ wird hinzugefügt.

II. Die ϵ-Regeln $A \to \epsilon$, $A \in V \setminus \{S_0\}$, werden entfernt. Für jedes A auf der rechten Seite einer Regel $B \to uAv$, $B \in V$, $u, v \in (V \cup \Sigma)^*$, wird eine Regel $B \to uv$ hinzugefügt. Falls $B \to \epsilon$ im Verlauf hinzugefügt wird, wird das Verfahren für diese ϵ-Regel wiederholt.

III. Regeln der Form $A \to B$, $A, B \in V$, werden entfernt. Für jedes B auf der linken Seite einer Regel $B \to u$, $u \in (V \cup \Sigma)^+$ wird die Regel $A \to u$ hinzugefügt.

IV. Jede Regel $A \to u_1 u_2 \dots u_k$, mit $k \geq 3$ und $u_i \in (V \cup \Sigma)$, $1 \leq i \leq k$, wird durch die Regeln $A \to u_1 A_1$, $A_1 \to u_2 A_2$, …, $A_{k-2} \to u_{k-1} u_k$ ersetzt. $A_1, \dots, A_{k-2}$ sind neue Variable.

V. Regeln der Form $A \to u_1 u_2$, $u_1 \in \Sigma$, $u_2 \in (V \cup \Sigma)$, werden durch $A \to U_1 u_2$ and $U_1 \to u_1$ ersetzt. Regeln $A \to U_1 u_2$, $u_2 \in \Sigma$, werden durch $A \to U_1 U_2$ und $U_2 \to u_2$ ersetzt. U_1 und U_2 sind neue Variable.

Die Grammatik $G' = (V', \Sigma, R', S_0)$ ist dann in Chomsky Normalform und es gilt $L(G) = L(G')$. □

Beispiel 4.7
Gegeben sei die kontextfreie Grammatik $G = (\{A, B, C\}, \{a, b, c\}, R, A)$ mit den Regeln

$$\begin{aligned} A &\to BC\ , \\ B &\to bB \mid \epsilon\ , \\ C &\to aCc \mid \epsilon\ . \end{aligned}$$

Diese wird nach dem im Beweis zu Satz 4.1 gegebenen Verfahren in eine äquivalente Grammatik G' in Chomsky Normalform überführt.

I. Sei S die neue Startvariable. Füge Regel $S \to A$ zu R hinzu.

$$\begin{aligned} S &\to A \\ A &\to BC \\ B &\to bB \mid \epsilon \\ C &\to aCc \mid \epsilon \end{aligned}$$

II. Entferne Regel $C \to \epsilon$ von R.
Füge Regeln $A \to B$ und $C \to ac$ zu R hinzu.

$$\begin{aligned} S &\to A \\ A &\to BC \mid B \\ B &\to bB \mid \epsilon \\ C &\to aCc \mid ac \end{aligned}$$

Entferne Regel $B \to \epsilon$ von R.
Füge Regeln $A \to C \mid \epsilon$ und $B \to b$ zu R hinzu.

$$\begin{aligned} S &\to A \\ A &\to BC \mid B \mid C \mid \epsilon \\ B &\to bB \mid b \\ C &\to aCc \mid ac \end{aligned}$$

Entferne Regel $A \to \epsilon$ von R.
Füge Regel $S \to \epsilon$ zu R hinzu.

$$\begin{aligned} S &\to A \mid \epsilon \\ A &\to BC \mid B \mid C \\ B &\to bB \mid b \\ C &\to aCc \mid ac \end{aligned}$$

III. Entferne Regel $A \to C$ von R.
Füge Regeln $A \to aCc \mid ac$ zu R hinzu.

$$\begin{aligned} S &\to A \mid \epsilon \\ A &\to BC \mid B \mid aCc \mid ac \\ B &\to bB \mid b \\ C &\to aCc \mid ac \end{aligned}$$

Entferne Regel $A \to B$ von R.
Füge Regeln $A \to bB \mid b$ zu R hinzu.

$$\begin{aligned} S &\to A \mid \epsilon \\ A &\to BC \mid bB \mid b \mid aCc \mid ac \\ B &\to bB \mid b \\ C &\to aCc \mid ac \end{aligned}$$

Entferne Regel $S \to A$ von R.
Füge Regeln $S \to BC \mid bB \mid b \mid aCc \mid ac$ zu R hinzu.

$$\begin{aligned} S &\to BC \mid bB \mid b \mid aCc \mid ac \mid \epsilon \\ A &\to BC \mid bB \mid b \mid aCc \mid ac \\ B &\to bB \mid b \\ C &\to aCc \mid ac \end{aligned}$$

IV. Ersetze die Regel $S \to aCc$ durch die Regeln $S \to aS_1$ und $S_1 \to Cc$.
Verfahre entsprechend für $A \to aCc$ und $C \to aCc$.

$$\begin{aligned} S &\to BC \mid bB \mid b \mid aS_1 \mid ac \mid \epsilon \\ A &\to BC \mid bB \mid b \mid aS_1 \mid ac \\ B &\to bB \mid b \\ C &\to aS_1 \mid ac \\ S_1 &\to Cc \end{aligned}$$

V. Ersetze die Regel $S \to bB$ durch die Regeln $S \to B_1B$ und $B_1 \to b$.
Verfahre entsprechend für $A \to bB$ und $B \to bB$.

$$\begin{aligned} S &\to BC \mid B_1B \mid b \mid aS_1 \mid ac \mid \epsilon \\ A &\to BC \mid B_1B \mid b \mid aS_1 \mid ac \\ B &\to B_1B \mid b \\ C &\to aS_1 \mid ac \\ S_1 &\to Cc \\ B_1 &\to b \end{aligned}$$

Ersetze die Regel $S \to aS_1$ durch die Regeln $S \to A_1S_1$ und $A_1 \to a$.
Verfahre entsprechend für $A \to aS_1$ und $C \to aS_1$.

$$\begin{aligned}
S &\to BC \mid B_1B \mid b \mid A_1S_1 \mid ac \mid \epsilon \\
A &\to BC \mid B_1B \mid b \mid A_1S_1 \mid ac \\
B &\to B_1B \mid b \\
C &\to A_1S_1 \mid ac \\
S_1 &\to Cc \\
B_1 &\to b \\
A_1 &\to a
\end{aligned}$$

Ersetze die Regel $S \to ac$ durch die Regeln $S \to A_1C_1$, $A_1 \to a$ und $C_1 \to c$.
Verfahre entsprechend für $A \to ac$ und $C \to ac$.

$$\begin{aligned}
S &\to BC \mid B_1B \mid b \mid A_1S_1 \mid A_1C_1 \mid \epsilon \\
A &\to BC \mid B_1B \mid b \mid A_1S_1 \mid A_1C_1 \\
B &\to B_1B \mid b \\
C &\to A_1S_1 \mid A_1C_1 \\
S_1 &\to Cc \\
B_1 &\to b \\
A_1 &\to a \\
C_1 &\to c
\end{aligned}$$

Ersetze die Regel $S_1 \to Cc$ durch die Regeln $S_1 \to CC_1$ und $C_1 \to c$.

$$\begin{aligned}
S &\to BC \mid B_1B \mid b \mid A_1S_1 \mid A_1C_1 \mid \epsilon \\
A &\to BC \mid B_1B \mid b \mid A_1S_1 \mid A_1C_1 \\
B &\to B_1B \mid b \\
C &\to A_1S_1 \mid A_1C_1 \\
S_1 &\to CC_1 \\
B_1 &\to b \\
A_1 &\to a \\
C_1 &\to c
\end{aligned}$$

$G' = (\{A, B, C, S, A_1, B_1, C_1, S_1\}, \{a, b, c\}, R', S)$ ist äquivalent zu G und in Chomsky Normalform.

□

4.2 Kellerautomaten

Die kontextfreien Sprachen wurden im vorangehenden Abschnitt mittels eines generativen Verfahrens eingeführt, den kontextfreien Grammatiken. Es gibt für diese Sprachfamilie nun ebenso ein analysierendes bzw. erkennendes Verfahren wie für die regulären Sprachen. Dazu wird das Modell des endlichen Automaten erweitert, so dass diese auch rekursive Strukturen erkennen können. Diese Erweiterung führt zu den so genannten *Kellerautomaten*. Der endlichen Steuereinheit und dem Eingabeband des bisherigen Automatenmodells wird ein Stapelspeicher – auch *Keller* oder *Stack* genannt – hinzugefügt (Abbildung 4.7).

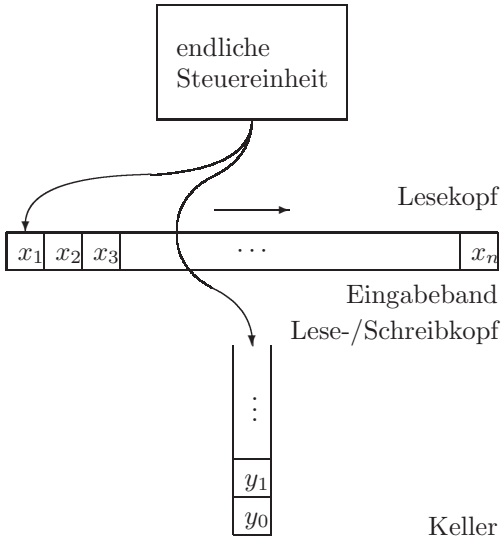


Abbildung 4.7: Kellerautomat

Ein Stapelspeicher zeichnet sich dadurch aus, dass auf diesen ausschließlich über eine Speicherzelle zugegriffen werden kann, wie z. B. bei einem Bücherstapel, nämlich oben. Ein Zeichen kann nur oben neu hineingeschrieben werden, die bisher gespeicherten Zeichen befinden sich damit jeweils eine Speicherzelle weiter unten im Stapelspeicher. Auch ein lesender Zugriff kann nur auf die oberste Speicherzelle erfolgen. Bei diesem Automatenmodell ist mit dem Lesen eines Zeichens auch seine Löschung verbunden, die darunter gespeicherten Zeichen befinden sich damit dann eine Speicherzelle weiter oben.

Der Keller eines Kellerautomaten zeichnet sich weiterhin dadurch aus, dass er eine unbeschränkte Speicherkapazität aufweist. Es können in ihm also beliebig viele Zeichen gespeichert werden. Da er aber zum Startzeitpunkt leer ist und in jedem Takt nur ein Zeichen hinzukommen kann, wird er nach endlich vielen Takten auch nur endlich viele Zeichen enthalten. Der Keller weist also eine so genannte *potentiell unendliche* Speicherkapazität auf.

Der Kellerautomat wird hier als ein nichtdeterministischer Automat eingeführt. Es gibt natürlich auch die deterministische Variante mit den entsprechnden Einschränkungen wie beim endlichen Automaten auch. Die deterministischen Kellerautomaten können aber gegenüber den nichtdeterministischen Kellerautomaten nur eine echt kleinere Sprachfamilie erkennen.

In Compilern erfolgt die Syntaxanalyse und die Steuerung des gesamten Übersetzungsprozesses eines Programmes durch einen so genannten *Parser*. Der Parser entspricht dem Berechnungsmodell eines deterministischen Kellerautomaten. Nicht

nur bei Programmiersprachen sondern auch bei Markierungssprachen wie z.B. XML oder bei natürlichen Sprachen wird die Analyse und Übersetzung entsprechend der Arbeitsweise dieses Automatentypes durchgeführt. Dennoch werden die Kellerautomaten hier nur in der allgemeineren nichtdeterministischen Variante betrachtet.

Definition 4.8 (Kellerautomat)
Ein Kellerautomat (KA) ist ein 6*-Tupel* $(Q, \Sigma, \Gamma, \delta, q_0, F)$ *mit*

1. Q *ist eine nichtleere, endlich Zustandsmenge,*
2. Σ *ist das Eingabealphabet,*
3. Γ *ist das Kelleralphabet,*
4. $\delta : Q \times \Sigma_\epsilon \times \Gamma_\epsilon \to \mathcal{P}(Q \times \Gamma_\epsilon)$ *ist die Überführungsfunktion,*
5. $q_0 \in Q$ *ist der Startzustand, und*
6. $F \subseteq Q$ *ist die Menge von End- bzw. akzeptierenden Zuständen.*

Ein Kellerautomat arbeitet nun wie ein endlicher Automat, nur dass bei seinen Überführungen der Keller zu berücksichtigen ist. Er befindet sich zu Beginn im Startzustand. Auf dem Eingabeband steht das abzuarbeitende Wort, der Lesekopf befindet sich auf dem ersten Zeichen, der Keller ist leer.

In Abhängigkeit von seinem aktuellen Zustand, dem Zeichen auf dem Eingabeband und dem obersten Zeichen im Keller geht er in einen Folgezustand über, rückt auf dem Eingabeband den Lesekopf um ein Feld weiter und ersetzt das oberste Kellerzeichen durch ein – möglicherweise auch leeres – Zeichen aus dem Kelleralphabet. Der restliche Kellerinhalt bleibt unverändert bestehen.

Ein Kellerautomat kann alternativ dazu auch eine so genannte ϵ-Überführung ausführen: Unabhängig vom aktuellen Zeichen auf dem Eingabeband, nur abhängig von seinem aktuellen Zustand und dem obersten Zeichen im Keller erfolgen die oben beschriebenen Aktionen. Der Lesekopf bleibt dann aber auf dem Feld stehen.

Ein Kellerautomat beendet seine Arbeit, wenn das Eingabeband abgearbeitet ist. Er akzeptiert oder erkennt seine Eingabe, wenn er sich dann in einem Endzustand befindet, andernfalls akzeptiert er seine Eingabe nicht. Er bleibt auch stehen, wenn sein Überführungsverhalten für ein vorliegendes Tripel aus $Q \times \Sigma_\epsilon \times \Gamma_\epsilon$ nicht definiert ist. Er akzeptiert dann die Eingabe nicht.

Bei all dem ist zu berücksichtigen, dass der Kellerautomat nichtdeterministisch arbeitet. Er kann sich also je Berechnungsschritt in mehrere Berechnungspfade aufspalten. Die Eingabe wird genau dann akzeptiert, wenn die Eingabe auf mindestens einem der Berechnungspfade akzeptiert wird.

Definition 4.9 (Berechnung eines Kellerautomaten)
Seien $K = (Q, \Sigma, \Gamma, \delta, q_0, F)$ ein Kellerautomat und w ein Wort über dem Alphabet Σ. Könne w als $w = y_1 y_2 \dots y_m$ geschrieben werden, wobei $y_i \in \Sigma_\epsilon$, $1 \leq i \leq m$.

K akzeptiert dann w, falls eine Zustandsfolge $r_0, r_1, \dots, r_m \in Q$ und eine Folge von Wörtern $s_0, s_1, \dots, s_m \in \Gamma^*$ existieren, die die folgenden drei Bedingungen erfüllen. Die Wörter s_i repräsentieren die Folge von Kellerinhalten, die K auf einem akzeptierenden Berechnungspfad durchläuft.

1. *$r_0 = q_0$ und $s_0 = \epsilon$,*
2. *$(r_{i+1}, b) \in \delta(r_i, y_{i+1}, a)$,*
   wobei $s_i = at$ und $s_{i+1} = bt$ für $a, b \in \Gamma_\epsilon$ und $t \in \Gamma^*$,
   für $i = 0, 1, \dots, m-1$.
3. *$r_m \in F$.*

Man sagt dann, dass K die Sprache L erkennt, falls $L = \{w \mid K \text{ akzeptiert } w\}$, geschrieben $L(K) = L$.

In Definition 4.9 wird das zu analysierende Wort w als $w = y_1 \dots y_m$ geschrieben, mit $y_i \in \Sigma_\epsilon$, $1 \leq i \leq m$. Analog zu den nichtdeterministischen endlichen Automaten kann hier nun wieder an einigen Positionen ein ϵ auftreten. Diese eingefügten ϵ markieren wieder nur die Positionen der ϵ-Überführungen, sie bedeuten nicht, dass an diesen Stellen in der Eingabe ein ϵ steht.

Beispiel 4.8
Sei $K_1 = (Q, \Sigma, \Gamma, \delta, q_1, F)$ ein KA mit

- $Q = \{q_1, q_2, q_3, q_4\}$,
- $\Sigma = \{a, b\}$,
- $\Gamma = \{a, \$\}$,
- $F = \{q_1, q_4\}$,
- Überführungsfunktion δ:

Q	Σ_ϵ	Γ_ϵ	$\mathcal{P}(Q \times \Gamma_\epsilon)$
q_1	ϵ	ϵ	$\{(q_2, \$)\}$
q_2	a	ϵ	$\{(q_2, a)\}$
q_2	b	a	$\{(q_3, \epsilon)\}$
q_3	b	a	$\{(q_3, \epsilon)\}$
q_3	ϵ	$\$$	$\{(q_4, \epsilon)\}$

Für alle in der Überführungstabelle nicht aufgeführten Tripel $(q, x, y) \in Q \times \Sigma_\epsilon \times \Gamma_\epsilon$ gilt $\delta(q, x, y) = \emptyset$, d. h. Überführungen sind für diese (q, x, y) nicht definiert.

Der KA K_1 speichert jedes gelesene a im Keller und löscht bei jedem gelesenen b ein a im Keller. Ist nach Abarbeitung der Eingabe der Keller leer, dann hat die Anzahl der a und b übereingestimmt. Abbildung 4.8 zeigt den Überführungsgraphen von K_1.

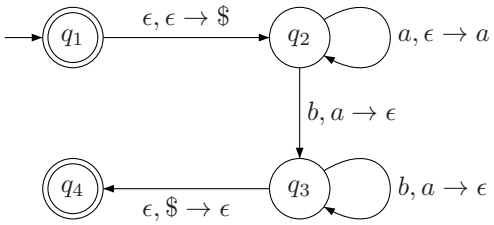


Abbildung 4.8: Kellerautomat K_1, der die Sprache $\{a^n b^n \mid n \in \mathbb{N}_0\}$ erkennt

Betrachte die folgenden Berechnungen von K_1:

I. Eingabewort *aaabbb*:

i	Q	$y_{i+1} \dots y_m \in \Sigma_\epsilon^*$	$s_i \in \Gamma_\epsilon^*$
0	q_1	$\epsilon aa\epsilon abbb\epsilon$	ϵ
1	q_2	$aa\epsilon abbb\epsilon$	\$
2	q_2	$a\epsilon abbb\epsilon$	a\$
3	q_2	$\epsilon abbb\epsilon$	aa\$

$\delta(q_2, \epsilon, a)$ ist nicht definiert. Die Eingabe ist nicht vollständig abgearbeitet, also wird die Eingabe auf diesem Berechnungspfad nicht akzeptiert. Die vorab angenommenen Positionen der ϵ-Überführungen waren vielleicht nicht korrekt.

i	Q	$y_{i+1} \dots y_m \in \Sigma_\epsilon^*$	$s_i \in \Gamma_\epsilon^*$
0	q_1	$\epsilon aaabbb\epsilon$	ϵ
1	q_2	$aaabbb\epsilon$	\$
2	q_2	$aabbb\epsilon$	a\$
3	q_2	$abbb\epsilon$	aa\$
4	q_2	$bbb\epsilon$	aaa\$
5	q_3	$bb\epsilon$	aa\$
6	q_3	$b\epsilon$	a\$
7	q_3	ϵ	\$
8	q_4		

Die Eingabe ist vollständig abgearbeitet und $q_4 \in F$, also wird die Eingabe akzeptiert.

II. Eingabewort *aaabb*:

i	Q	$y_{i+1} \dots y_m \in \Sigma_\epsilon^*$	$s_i \in \Gamma_\epsilon^*$
0	q_1	$\epsilon aaabb\epsilon$	ϵ
1	q_2	$aaabb\epsilon$	\$
2	q_2	$aabb\epsilon$	a\$
3	q_2	$abb\epsilon$	aa\$
4	q_2	$bb\epsilon$	aaa\$
5	q_3	$b\epsilon$	aa\$
6	q_3	ϵ	a\$

Die Eingabe ist zwar vollständig abgearbeitet, aber $q_3 \notin F$, also wird die Eingabe auf diesem Berechnungspfad nicht akzeptiert. Da es aber auch keinen anderen akzeptierenden Berechnungspfad für diese Eingabe gibt – was zu zeigen wäre – wird diese Eingabe nicht akzeptiert.

$aaabbb \in L(K_1)$ und $aaabb \notin L(K_1)$. Der Kellerautomat K_1 erkennt die Sprache $L(K_1) = \{a^n b^n \mid n \in \mathbb{N}_0\}$.

□

Ein Kellerautomat kann nun auch durch einen *Überführungsgraphen* dargestellt werden: Die Zustände werden als Knoten dargestellt. Ein Pfeil kennzeichnet den Startzustand. Ein zusätzlicher Kreis kennzeichnet die Endzustände. Die gerichteten Kanten definieren die Überführungen. Die Kantenmarkierung $x, a \to b$ von Zustand q nach q' hat dabei die Bedeutung $(q', b) \in \delta(q, x, a)$. In Abbildung 4.8 ist der Kellerautomat aus Beispiel 4.8 als Überführungsgraph dargestellt.

Beispiel 4.9
Sei $K_2 = (Q, \Sigma, \Gamma, \delta, q_1, F)$ ein KA mit

- $Q = \{q_1, q_2, q_3\}$,
- $\Sigma = \{(,)\}$,
- $\Gamma = \{x, \$\}$,
- $F = \{q_3\}$,
- Überführungsfunktion δ:

Q	Σ_ϵ	Γ_ϵ	$\mathcal{P}(Q \times \Gamma_\epsilon)$
q_1	ϵ	ϵ	$\{(q_2, \$)\}$
q_2	(	ϵ	$\{(q_2, x)\}$
q_2	)	x	$\{(q_2, \epsilon)\}$
q_2	ϵ	\$	$\{(q_3, \epsilon)\}$

Für alle in der Überführungstabelle nicht aufgeführten Tripel $(q, x, y) \in Q \times \Sigma_\epsilon \times \Gamma_\epsilon$ gilt $\delta(q, x, y) = \emptyset$, d. h. Überführungen sind für diese (q, x, y) nicht definiert.

K_2 speichert jedes gelesene (durch ein x im Keller und löscht bei jedem gelesenen) ein x im Keller. Ist nach Abarbeitung der Eingabe der Keller leer, dann wird die Eingabe akzeptiert. Abbildung 4.9 zeigt den Überführungsgraphen von K_2.

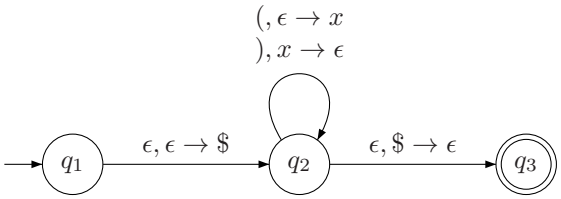


Abbildung 4.9: Kellerautomat K_2

Betrachte die folgenden Berechnungen von K_2:

I. Eingabewort (( ))( ):

i	Q	$y_{i+1} \dots y_m \in \Sigma_\epsilon^*$	$s_i \in \Gamma_\epsilon^*$
0	q_1	$\epsilon(( ))( )\epsilon$	ϵ
1	q_2	$(( ))( )\epsilon$	\$
2	q_2	$( ))( )\epsilon$	x\$
3	q_2	$))( )\epsilon$	xx\$
4	q_2	$)( )\epsilon$	x\$
5	q_2	$( )\epsilon$	\$
6	q_2	$)\epsilon$	x\$
7	q_2	ϵ	\$
8	q_3		

Die Eingabe ist vollständig abgearbeitet und $q_3 \in F$, also wird die Eingabe akzeptiert.

II. Eingabewort ( )):

i	Q	$y_{i+1} \dots y_m \in \Sigma_\epsilon^*$	$s_i \in \Gamma_\epsilon^*$
0	q_1	$\epsilon($ $)\epsilon)\epsilon$	ϵ
1	q_2	$($ $)\epsilon)\epsilon$	\$
2	q_2	$)\epsilon)\epsilon$	x\$
3	q_2	$\epsilon)\epsilon$	\$
4	q_3	$)\epsilon$	

Zwar ist $q_3 \in F$, aber die Eingabe ist nicht vollständig abgearbeitet. Also wird die Eingabe auf diesem Berechnungspfad nicht akzeptiert. Auch kein anderer Berechnungspfad führt zum Akzeptieren dieser Eingabe.

III. Eingabewort (( ):

i	Q	$y_{i+1} \dots y_m \in \Sigma_\epsilon^*$	$s_i \in \Gamma_\epsilon^*$
0	q_1	$\epsilon(($ $)\epsilon$	ϵ
1	q_2	$(($ $)\epsilon$	\$
2	q_2	$($ $)\epsilon$	x\$
3	q_2	$)\epsilon$	xx\$
4	q_2	ϵ	x\$

$\delta(q_2, \epsilon, x)$ ist nicht definiert. Die Eingabe ist zwar vollständig abgearbeitet, aber $q_2 \notin F$, also wird die Eingabe auf diesem Berechnungspfad nicht akzeptiert. Auch kein anderer Berechnungspfad führt zum Akzeptieren dieser Eingabe.

Der Kellerautomaten K_2 erkannt die Sprache
$L(K_2) = \{w \mid w \in \{(,)\}^*$ ist ein Wort einer richtig geschachtelten Klammerung$\}$.

□

Das Modell eines Kellerautomaten kann also bei der Erkennung bzw. der Analyse von Klammerausdrücken jeder Art eingesetzt werden, z. B. arithmetische, logische und relationale Ausdrücke wie sie in Programmiersprachen vorkommen. Darüber hinaus kann auch ein gesamtes Programm einer höheren Programmiersprache als eine geklammerte Struktur aufgefaßt werden (Abbildung 4.10), so dass die Syntaxanalyse von Programmen in einem Compiler mit Hilfe des Modells eines Kellerautomaten durchgeführt wird.

Neben der Erkennung von Klammerstrukturen wird ein Kellerautomat auch zur Erkennung von Spiegelstrukturen eingesetzt. Eine Spiegelstruktur kann als eine spezielle Klammerstruktur aufgefaßt werden, denn sie hat nur genau in der Wortmitte eine Spiegel- bzw. Symmetrieachse, wie z.B. die Sprache $L = \{wxw^R \mid w \in \{a, b\}^*\}$ über dem Alphabet $\{a, b, x\}$.

Eingangs war schon erwähnt worden, dass die Kellerautomaten genau die kontextfreien Sprachen erkennen können. Zu jedem Kellerautomaten muss also eine äquivalente kontextfreie Grammatik existieren und zu jeder kontextfreien Grammatik ein äquivalenter Kellerautomat. Der Beweis zu dem Satz 4.2, der genau diese Äquivalenz ausdrückt, erfolgt durch zwei solche Konstruktionen (Lemmas 4.3 und 4.4).

Satz 4.2 (Kontextfreie Sprachen und Kellerautomaten)
Eine Sprache ist genau dann kontextfrei, wenn es einen Kellerautomaten gibt, der sie erkennt.

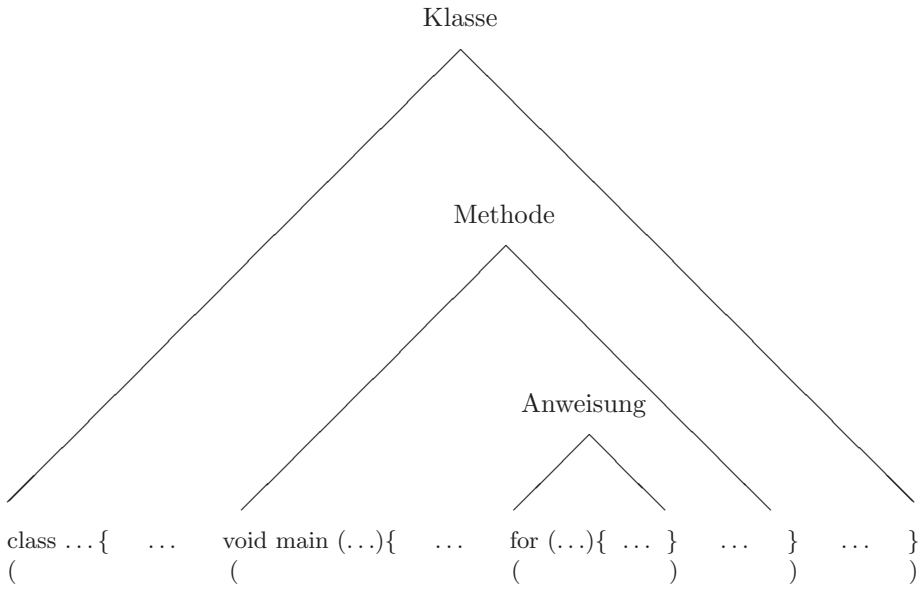


Abbildung 4.10: Struktur der Programmiersprache *Java*

Lemma 4.3
Eine Sprache wird von einem Kellerautomaten erkannt, wenn sie kontextfrei ist.

Beweis:
Sei eine kontextfreie Grammatik $G = (V, \Sigma, R, S)$ gegeben. Werde dazu ein KA $K = (Q, \Sigma, \Gamma, \delta, q_0, F)$ mit $L(G) = L(K)$ konstruiert.

Für diese Konstruktion wird eine Modifikation des Kellerautomaten benutzt: In einem Schritt kann der Kellerautomat mehr als ein Zeichen in den Kellerspeicher schreiben. Der Original-Kellerautomat kann dann den modifizierten Kellerautomaten durch eine Folge von Schritten dadurch simulieren, dass in jedem Schritt maximal nur ein Zeichen in den Kellerspeicher geschrieben wird und auf dem Eingabeband nicht weitergelesen wird. Für diese Simulation werden neue Zustände benötigt. Abbildung 4.11 zeigt ein Beispiel einer solchen Simulation.

Die Konstruktion von K ergibt sich aus der im Folgenden skizzierten Arbeitsweise:

1. Speichere das Zeichen \$ und die Startvariable S im Keller (\$ zuunterst).
2. Wiederhole die folgenden Schritte bis kein Schritt mehr ausgeführt werden kann:
   (a) Falls das oberste Kellerzeichen eine Variable A ist, wähle nichtdeterministisch eine der Regeln für A aus, z.B. $A \to u$, $u \in (V \cup \Sigma)^*$, und ersetze das oberste Kellerzeichen A durch u. Der Lesekopf wird auf dem Eingabeband nicht weiterbewegt.
   (b) Falls das oberste Kellerzeichen ein Terminalzeichen a ist, lese das nächste Eingabezeichen und vergleiche es mit a. Stimmen sie überein, dann wird das oberste Kellerzeichen gelöscht und der Lesekopf rückt auf dem Eingabeband zum nächsten Zeichen vor. Stimmen sie nicht überein, wird die Eingabe auf diesem Berechnungspfad zurückgewiesen.

(c) Falls das oberste Kellerzeichen das Zeichen \$ ist, gehe in den akzeptierenden Zustand. Ist die Eingabe vollständig abgearbeitet, dann wird die Eingabe akzeptiert. Falls nicht, wird die Eingabe auf diesem Berechnungspfad zurückgewiesen.

Der sich aus dieser Arbeitsweise ergebende Kellerautomat ist durch

$$K = (\{q_{start}, q_{loop}, q_{end}, \}, \ \Sigma, \ V \cup \Sigma \cup \{\$\}, \ \delta, \ q_{start}, \ \{q_{end}\})$$

definiert. Die Überführungsfunktion δ ergibt sich aus den Regeln R der gegebenen Grammatik G:

$$\begin{array}{lcl} \delta(q_{start}, \epsilon, \epsilon) & = & \{(q_{loop}, S\$)\} \\ \delta(q_{loop}, \epsilon, A) & = & \{(q_{loop}, w) \mid A \to w \in R\} \\ \delta(q_{loop}, a, a) & = & \{(q_{loop}, \epsilon) \mid a \in \Sigma\} \\ \delta(q_{loop}, \epsilon, \$) & = & \{(q_{end}, \epsilon)\} \end{array}$$

Sein Überführungsgraph ist in Abbildung 4.12 dargestellt.

Der so konstruierte Kellerautomat simuliert für ein gegebenes Eingabewort die Generierung dieses Wortes durch die gegebene Grammatik. Zu jedem Zeitpunkt entspricht die Konkatenation aus bereits abgearbeitetem Präfix der Eingabe und dem jeweiligen Kellerinhalt dem Wort nach dem jeweiligen Ableitungsschritt. □

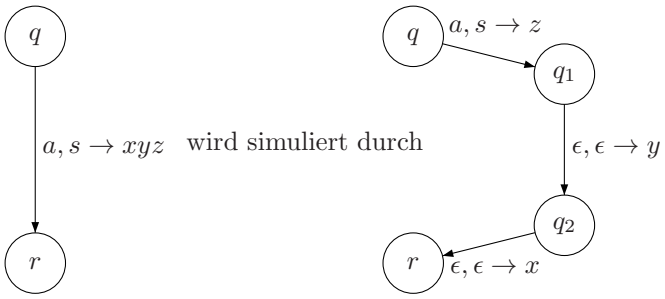


Abbildung 4.11: Simulation eines KA, der in einem Schritt mehr als ein Zeichen kellern kann. $(r, xyz) \in \delta(q, a, s)$, mit $x, y, z \in \Gamma$, ist also z.B. eine zulässige Überführung.

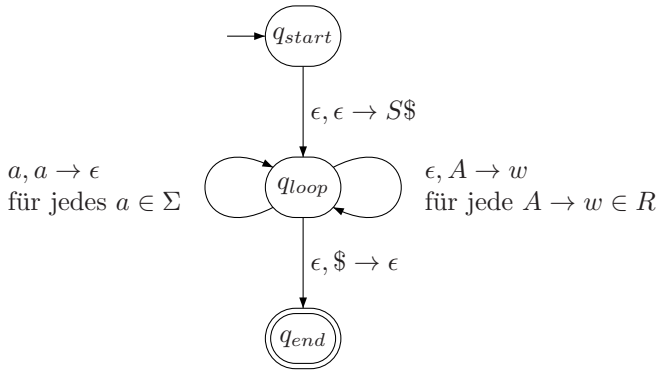


Abbildung 4.12: Überführungsgraph des KA K, Konstruktion entsprechend Beweis zu Lemma 4.3

Beispiel 4.10
Sei die kontextfreie Grammatik $G_3 = (\{S\}, \{(,)\}, \{S \to (S) \mid SS \mid \epsilon\}, S)$ gegeben (Beispiel 4.4). Es werde dazu entsprechend dem Beweis zu Lemma 4.3 ein äquivalenter Kellerautomat $K = (Q, \Sigma, \Gamma, \delta, q_0, F)$ konstruiert:

- $Q = \{q_{start}, q_{loop}, q_{end}\}$
- $\Sigma = \{(,)\}$
- $\Gamma = \{S, (,), \$\}$
- $q_0 = q_{start}$
- $F = \{q_{end}\}$
- δ ist in Abbildung 4.13 dargestellt

Die Auflösung von Überführungen, die je Schritt mehr als ein Zeichen kellern, zu einem Kellerautomaten in der originalen Definition ist in Abbildung 4.14 dargestellt.

Betrachte hier noch einmal das Wort (( ))( ), das von G_3 durch die Ableitung

$$S \Rightarrow SS \Rightarrow (S)S \Rightarrow ((S))S \Rightarrow ((\ ))S \Rightarrow ((\ ))(S) \Rightarrow ((\ ))(\ )$$

erzeugt wird. Der konstruierte Kellerautomat K (modifizierte Version) erkennt dieses Wort durch die folgende Berechnung:

i	Q	$y_{i+1} \dots y_m \in \Sigma_\epsilon^*$	$s_i \in \Gamma_\epsilon^*$
0	q_{start}	$\epsilon\epsilon\epsilon(\epsilon(\epsilon))\epsilon(\epsilon)\epsilon$	ϵ
1	q_{loop}	$\epsilon\epsilon(\epsilon(\epsilon))\epsilon(\epsilon)\epsilon$	$S\$$
2	q_{loop}	$\epsilon(\epsilon(\epsilon))\epsilon(\epsilon)\epsilon$	$SS\$$
3	q_{loop}	$(\epsilon(\epsilon))\epsilon(\epsilon)\epsilon$	$(S)S\$$
4	q_{loop}	$\epsilon(\epsilon))\epsilon(\epsilon)\epsilon$	$S)S\$$
5	q_{loop}	$(\epsilon))\epsilon(\epsilon)\epsilon$	$(S))S\$$
6	q_{loop}	$\epsilon))(\epsilon)\epsilon$	$S))S\$$
7	q_{loop}	$))\epsilon(\epsilon)\epsilon$	$))S\$$
8	q_{loop}	$)\epsilon(\epsilon)\epsilon$	$)S\$$
9	q_{loop}	$\epsilon(\epsilon)\epsilon$	$S\$$
10	q_{loop}	$(\epsilon)\epsilon$	$(S)\$$
11	q_{loop}	$\epsilon)\epsilon$	$S)\$$
12	q_{loop}	$)\epsilon$	$)\$$
13	q_{loop}	ϵ	$\$$
14	q_{end}		

Betrachte z.B. K zum Zeitpunkt $i = 6$: Bereits gelesen und verarbeitet ist ((und der Kellerinhalt ist $S))S\$$. Die Konkatenation aus ((und $S))S\$$ ohne dem speziellen Kellerzeichen \$ ergibt genau das diesem Ableitungsschritt entsprechende Wort mittels der gegebenen Grammatik, $S \Rightarrow^* ((S))S$.

Diese Beziehung gilt zu jedem Zeitpunkt. Der Kellerautomat simuliert also die Ableitung mittels der Grammatik für das gegebene Wort.

□

Lemma 4.4
Eine Sprache ist kontextfrei, wenn es einen Kellerautomaten gibt, der sie erkennt.

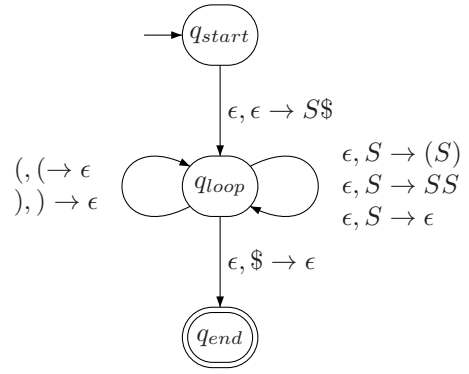


Abbildung 4.13: Überführungsgraph des KA K aus Beispiel 4.10

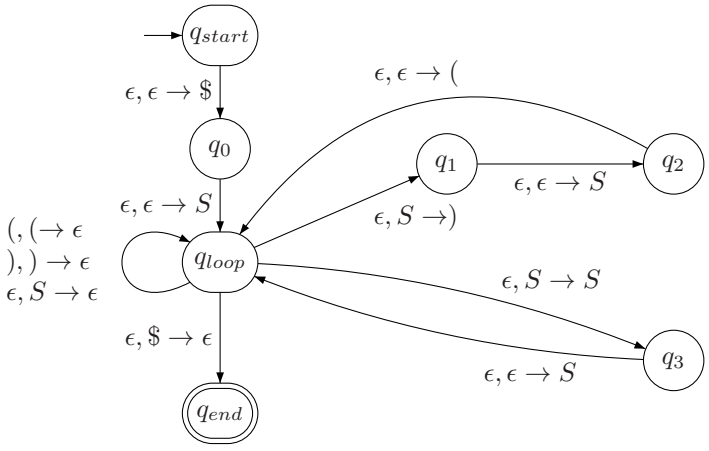


Abbildung 4.14: Überführungsgraph des KA K aus Beispiel 4.10 (in originaler Definition des KA)

Beweis:
Sei ein KA $K = (Q, \Sigma, \Gamma, \delta, q_0, F)$ gegeben. Konstruiert werde eine kontextfreie Grammatik $G = (V, \Sigma, R, S)$ mit $L(K) = L(G)$.

K muss die folgenden Merkmale erfüllen:

1. K hat nur einen akzeptierenden Zustand, q_{end}.
2. K leert den Keller vor dem Akzeptieren.
3. Jede Überführung speichert entweder ein Zeichen oben in dem Keller (*push*) oder löscht das oberste Kellerzeichen (*pop*)

Die Regeln von G werden wie folgt konstruiert: Für jedes Zustandspaar $p, q \in Q$ gibt es in der Grammatik eine Variable A_{pq}. Diese Variable generiert Wörter, die K von dem Zustand p bei leerem Keller zum Zustand q wieder bei leerem Keller führen. Diese Wörter führen K ebenso vom Zustand p mit einem beliebigen Kellerinhalt zum Zustand q mit genau diesem Kellerinhalt.

Für jedes dieser Wörter w ist der erste Überführungsschritt für K ein *push* und der letzte ein *pop*. Für die Berechnung von K gibt es damit zwei Möglichkeiten für den Fall $w \neq \epsilon$ und eine Möglichkeit für den Fall $w = \epsilon$:

1. Der Keller ist nur zu Beginn und zum Ende der Berechnung leer. Sei a das Eingabezeichen zu Beginn und b das Eingabezeichen am Ende, und sei r der auf p folgende Zustand und s der q vorangehende Zustand. Diese Berechnung wird dann durch die Regel $A_{pq} \to aA_{rs}b$ simuliert.

   Für jedes $p, q, r, s \in Q,\ t \in \Gamma,\ a, b \in \Sigma_\epsilon$, falls $(r, t) \in \delta(p, a, \epsilon)$ und $(q, \epsilon) \in \delta(s, b, t)$, füge die Regel $A_{pq} \to aA_{rs}b$ in G hinzu.

2. Der Keller wird vor dem Ende der Berechnung leer. Sei r der Zustand zu diesem Zeitpunkt. Diese Berechnung wird dann durch die Regel $A_{pq} \to A_{pr}A_{rq}$ simuliert.

   Für jedes $p, q, r \in Q$, füge die Regel $A_{pq} \to A_{pr}A_{rq}$ in G hinzu.

3. Für jedes $q \in Q$, füge die Regel $A_{pp} \to \epsilon$ in G hinzu.

Die Variablen von G sind $\{A_{pq} \mid p, q \in Q\}$ und die Startvariable ist $A_{q_0 q_{accept}}$. □

Beispiel 4.11
Sei der KA K_1 aus Beispiel 4.8 gegeben. Um die Voraussetzung zu erfüllen, nur einen Endzustand zu haben, nämlich q_4, wird die Überführung

$$(q_4, \epsilon) \in \delta(q_2, \epsilon, \$)$$

hinzugefügt. Die beiden anderen Voraussetzungen gelten ohnehin.

Konstruktion von G mit $L(K_1) = L(G)$:

1. Selektiere die Überführungen danach, ob sie ein Zeichen im Keller speichern (*push*) oder ein Zeichen löschen (*pop*), z.B.

$$\begin{array}{ll}
(q_2, \$) \in \delta(q_1, \epsilon, \epsilon) \wedge (q_4, \epsilon) \in \delta(q_3, \epsilon, \$): & A_{14} \to \epsilon A_{23} \epsilon = A_{23} \\
(q_2, a) \in \delta(q_2, a, \epsilon) \wedge (q_3, \epsilon) \in \delta(q_2, b, a): & A_{23} \to aA_{22}b \\
(q_2, a) \in \delta(q_2, a, \epsilon) \wedge (q_3, \epsilon) \in \delta(q_3, b, a): & A_{23} \to aA_{23}b \\
(q_2, \$) \in \delta(q_1, \epsilon, \epsilon) \wedge (q_4, \epsilon) \in \delta(q_2, \epsilon, \$): & A_{14} \to \epsilon A_{22} \epsilon = A_{22}
\end{array}$$

2. Über Regeln von diesem Typ werden in diesem Beipiel keine terminalen Wörter abgeleitet.

3. Z.B. $A_{22} \to \epsilon$.

Die Startvariable ist A_{14}. Insgesamt gibt es 16 Zustände, die aber nicht alle benötigt werden.

Das Wort $aaabbb \in L(K_1)$ wird in G wie folgt generiert:

$$A_{14} \Rightarrow A_{23} \Rightarrow aA_{23}b \Rightarrow aaA_{23}bb \Rightarrow aaaA_{22}bbb \Rightarrow aaabbb$$

□

Aus der Äquivalenz von kontextfreien Grammatiken und Kellerautomaten lässt sich weiterhin auch schließen, dass jede reguläre Sprache auch kontextfrei ist. Ein Kellerautomat simuliert einfach einen endlichen Automaten, indem er den Keller ignoriert.

Satz 4.5
Jede reguläre Sprache ist kontextfrei. Es gilt darüber hinaus $\mathcal{L}_{reg} \subset \mathcal{L}_{kf}$.

Beweis:

I. Sei $L \in \mathcal{L}_{reg}$. Dann gibt es einen deterministischen endlichen Automaten $E = (Q, \Sigma, \delta, q_0, F)$ mit $L = L(E)$. Mit $K = (Q, \Sigma, \emptyset, \delta', q_0, F)$ und

$$\delta'(q, x, \epsilon) = \{(q', \epsilon)\} \text{ genau dann, wenn } \delta(q, x) = q'$$

ist dann ein Kellerautomat gegeben, für den $L(E) = L(K)$ gilt, denn für alle $w \in \Sigma^*$ gibt es genau dann eine Berechnung in E, wenn es eine Berechnung in K gibt.

Damit gilt auch $L \in \mathcal{L}_{kf}$, also $\mathcal{L}_{reg} \subseteq \mathcal{L}_{kf}$.

II. Die echte Teilmengenbeziehung zwischen den Sprachfamilien ergibt sich unmittelbar aus der Existenz einer Sprache L mit $L \in \mathcal{L}_{kf}$ und $L \notin \mathcal{L}_{reg}$, z. B. $L = \{a^i b^i \mid i \in \mathbb{N}_0\}$. Also $\mathcal{L}_{reg} \subset \mathcal{L}_{kf}$.

□

Beispiel 4.12
Betrachte das Beispiel 2.3. Der KA $K = (\{0, 1, 2\}, \{a, b\}, \emptyset, \delta, 0, \{0\})$ mit

δ :

Q	Σ_ϵ	Γ_ϵ	$\mathcal{P}(Q \times \Gamma_\epsilon)$
0	a	ϵ	$\{1, \epsilon\}$
1	a	ϵ	$\{2, \epsilon\}$
2	a	ϵ	$\{0, \epsilon\}$
0	b	ϵ	$\{0, \epsilon\}$
1	b	ϵ	$\{1, \epsilon\}$
2	b	ϵ	$\{2, \epsilon\}$

erkennt die Sprache $L(K) = \{w \mid w \in \{a, b\}^*, |w|_a \bmod 3 = 0\}$.

□

4.3 Eigenschaften kontextfreier Sprachen

Es werden nun wieder einige Operationen auf kontextfreien Sprachen und ihre Abschlusseigenschaften betrachtet. Sie weichen in einigen Punkten von den Ergebnissen der regulären Sprachen ab. Darüber hinaus gibt es auch für kontextfreie Sprachen ein notwendiges Kriterium, auch *Pumping Lemma* genannt. Es wird zum Nachweis herangezogen, dass eine Sprache nicht kontextfrei ist. Es gibt also Sprachen, die außerhalb der bisher betrachteten Sprachfamilien liegen.

Satz 4.6 (Vereinigung, Konkatenation, Stern)
Es seien L_1 und L_2 kontextfreie Sprachen. Dann sind auch

- $L_1 \cup L_2$
- $L_1 L_2$
- L_1^*

kontextfreie Sprachen.

Beweis:
Da L_1 und L_2 kontextfreie Sprachen sind, gibt es sie erzeugende kontextfreie Grammatiken $G_1 = (V_1, \Sigma_1, R_1, S_1)$ und $G_2 = (V_2, \Sigma_2, R_2, S_2)$. Es seien $V_1 \cap V_2 = \emptyset$ und $S \notin V_1 \cup V_2$.

- Vereinigung:
  Die kontextfreie Grammatik $G_V = (V, \Sigma, R, S)$ mit

  $$\begin{aligned} V &= V_1 \cup V_2 \cup \{S\} \\ \Sigma &= \Sigma_1 \cup \Sigma_2 \\ R &= R_1 \cup R_2 \cup \{S \to S_1, S \to S_2\} \end{aligned}$$

  erzeugt dann die Sprache $L(G_V) = L_1 \cup L_2$. Also ist $L_1 \cup L_2$ kontextfrei.

- Konkatenation:
  Die kontextfreie Grammatik $G_K = (V, \Sigma, R, S)$ mit

  $$\begin{aligned} V &= V_1 \cup V_2 \cup \{S\} \\ \Sigma &= \Sigma_1 \cup \Sigma_2 \\ R &= R_1 \cup R_2 \cup \{S \to S_1 S_2\} \end{aligned}$$

  erzeugt dann die Sprache $L(G_K) = L_1 L_2$. Also ist $L_1 L_2$ kontextfrei.

- Stern (Kleeneabschluss):
  Die kontextfreie Grammatik $G_S = (V, \Sigma, R, S)$ mit

  $$\begin{aligned} V &= V_1 \cup \{S\} \\ \Sigma &= \Sigma_1 \\ R &= R_1 \cup \{S \to \epsilon, S \to S_1 S\} \end{aligned}$$

  erzeugt dann die Sprache $L(G_S) = L_1^*$. Also ist L_1^* kontextfrei.

Es ist offensichtlich, dass die so konstruierten Grammatiken die jeweils gewünschten Sprachen und nur diese erzeugen. □

Das Pumping Lemma (Satz 4.7) spezifiziert eine notwendige Eigenschaft von kontextfreien Sprachen. Ist diese Eigenschaft nicht vorhanden, dann ist die Sprache nicht kontextfrei. Dieser Satz wird primär – wie der entsprechende Satz für reguläre Sprachen – zum Nachweis angewendet, dass eine Sprache nicht kontextfrei ist.

Satz 4.7 (Pumping Lemma)
Es sei L eine kontextfreie Sprache. Dann gibt es eine Konstante $n \in \mathbb{N}$, so dass jedes Wort $w \in L$ mit $|w| \geq n$ in fünf Teilwörter $w = uvxyz$ mit

1. $vy \neq \epsilon$
2. $|vxy| \leq n$
3. $\forall\ k \in \mathbb{N}_0 : uv^k xy^k z \in L$

zerlegt werden kann.

Beweis:
Sei $G = (V, \Sigma, R, S)$ eine kontextfreie Grammatik in Chomsky Normalform, die die Sprache L erkennt. In jedem Ableitungsbaum dieser Grammatik kann ein Knoten dann höchstens 2 Kinder haben. Wenn also die Höhe des Ableitungsbaums höchstens h ist, dann ist die Länge eines erzeugten Wortes höchstens 2^{h-1}. Im jeweils letzten Schritt wird eine Variable nur noch durch genau ein Terminalzeichen ersetzt, es findet also keine Verdoppelung der Zeichen mehr statt. Umgekehrt, wenn ein erzeugtes Wort mindestens die Länge $2^{h-1} + 1$ hat, dann hat der zugehörige Ableitungsbaum mindestens die Höhe $h + 1$.

Setze $n = 2^{|V|}$. Falls $w \in L$ mit $|w| \geq n$, dann hat der zugehörige Ableitungsbaum mindestens eine Höhe von $|V| + 1$. Falls w mehrere Ableitungsbäume hat, wähle den mit der kleinsten Anzahl von Knoten.

Ein Pfad von der Wurzel bis zu einem Blatt hat nun mindestens eine Länge von $|V|+2$ Knoten, ein Terminalzeichen und $|V| + 1$ Variable. Wegen des Schubfachprinzips muss mindestens eine Variable mehrfach in diesem Pfad auftreten, z.B. R. Wähle R so, dass diese Variable diejenige ist, die unter den mehrfach auftretenden die ist, die dem Blatt am nächsten ist.

Teile w entsprechend Abbildung 4.15 in $uvxyz$. Das obere Vorkommen der Variablen R generiert das Teilwort vxy und das untere Vorkommen generiert x. Beide Teilbäume werden von der gleichen Variablen generiert,

$$\begin{aligned} R &\Rightarrow^* vxy \\ R &\Rightarrow^* x, \end{aligned}$$

also können sie auch gegeneinander ausgetauscht werden und bleiben dabei ein gültiger Ableitungsbaum, d.h.

$$R \Rightarrow^* v^i xy^i, \quad i \geq 0.$$

Eine Ableitung, die mit der Startvariablen beginnt, führt zu

$$S \Rightarrow^* uv^i xy^i z, \quad i \geq 0,$$

was Bedingung 3 beweist.

Für Bedingung 1 ist zu zeigen, dass v und y nicht gleichzeitig ϵ sind. Sei angenommen, sie wären es. Der Ableitungsbaum, den man erhalten würde, indem der untere

(kleinere) Teilbaum den oberen (größeren) ersetzen würde, würde weniger Knoten haben und dennoch w erzeugen. Das wäre aber ein Widerspruch dazu, dass der Ableitungsbaum eine minimale Anzahl von Knoten hätte.

Für Bedingung 2 ist zu zeigen, dass $|vxy| \leq n = 2^{|V|}$ gilt. In dem Ableitungsbaum für w generiert das obere Vorkommen von R das Wort vxy. R wurde so gewählt, dass beide möglichst weit am Pfadende der $|V|+1$ Variablen vorkommen, und dass der Teilbaum, in dem R vxy generiert, höchstens eine Höhe von $|V|+1$ aufweist. Ein Baum dieser Höhe aber kann ein Wort höchstens der Länge von $2^{|V|} = n$ erzeugen. □

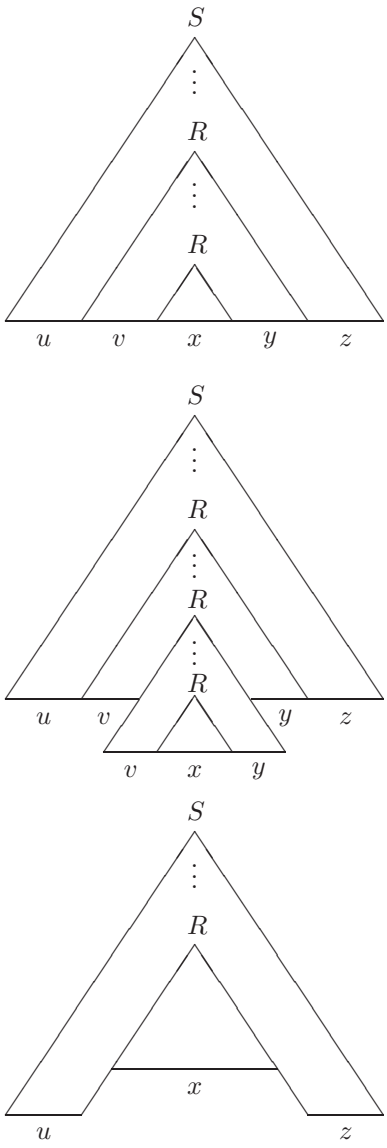


Abbildung 4.15: Ableitungsbäume für Wörter $w = uv^ixy^iz$, $i = 0, 1, 2$

Beispiel 4.13
Betrachte die kontextfreie Sprache $L = \{a^ib^i \mid i \in \mathbb{N}_0\}$.

Wähle $n = 2$ und betrachte die Wörter $a^m b^m$, $m \in \mathbb{N}$. Dann gilt $|a^m b^m| = 2m \geq 2$. $a^m b^m$ kann dann in die Teilwörter $u = a^{m-1}, v = a, x = \epsilon, y = b$ und $z = b^{m-1}$ zerlegt werden. Für $a^m b^m = uvxyz$ gilt dann

1. $vy = ab \neq \epsilon$,
2. $|vxy| = |ab| \leq 2$,
3. $\forall\ k \in \mathbb{N}_0 : uv^k xy^k z = a^{m-1} a^k b^k b^{m-1} = a^{m-1+k} b^{m-1+k} \in L$.

□

Beispiel 4.14
Betrachte die Sprache $L = \{a^i b^i c^i \mid i \in \mathbb{N}_0\}$.

Annahme: L sei kontextfrei.

Dann müssen die Eigenschaften des Pumping Lemmas auch für L erfüllt sein: Wähle $n \in \mathbb{N}$ und betrachte das Wort $w = a^n b^n c^n$. Es gilt $|w| \geq n$ und w kann in die Teilwörter $uvxyz$ zerlegt werden, so dass die Eigenschaften (1), (2) und (3) gelten.

Wegen $|vxy| \leq n$ (2) ist vxy entweder ein Teilwort von $a^n b^n$ oder von $b^n c^n$. a,b und c können nicht gemeinsam in vxy enthalten sein.

Da $v \neq \epsilon$ oder $y \neq \epsilon$ (1), können die Anzahlen der Vorkommen von a, b und c in uv^2xy^2z nicht gleich sein:

Sei z. B. vxy ein Teilwort von $a^n b^n$, d. h. vxy enthält kein c, ist also vollständig in z enthalten. In v oder y ist mindestens ein a oder b enthalten. In uv^2xy^2z wird damit mindestens ein a oder b verdoppelt, die Anzahl c bleibt auf jeden Fall unverändert. Also sind die Anzahlen nicht gleich. Analog kann mit dem anderen Teilwort argumentiert werden.

Also, $uv^2xy^2z \notin L$. Das ist ein Widerspruch zu (3).

Die Annahme, L sei kontextfrei, muss also falsch sein.

□

Satz 4.8 (Existenz nichtkontextfreier Sprachen)
Es gibt Sprachen, die nicht kontextfrei sind.

Beweis:
Die Sprache $\{a^i b^i c^i \mid i \in \mathbb{N}_0\}$ ist nicht kontextfrei (Beispiel 4.14). □

Mit diesem Ergebnis kann nun auch leicht gezeigt werden, dass einige der wichtigen Mengenoperationen für die Sprachfamilie der kontextfreien Sprachen nicht abgeschlossen sind.

Satz 4.9 (Durchschnitt, Komplement)
Die Klasse der kontextfreien Sprachen ist nicht abgeschlossen unter Durchschnitts- und Komplementbildung.

Beweis:
Betrachte die kontextfreien Sprachen

$$\begin{aligned} L_1 &= \{a^i b^i c^j \mid i, j \in \mathbb{N}_0\}, \\ L_2 &= \{a^j b^i c^i \mid i, j \in \mathbb{N}_0\}. \end{aligned}$$

Für diese Sprachen können leicht kontextfreie Grammatiken angegeben werden.

Die Durchschnittssprache

$$L_1 \cap L_2 = \{a^i b^i c^i \mid i \in \mathbb{N}_0\}$$

ist jedoch nicht kontextfrei (Beispiel 4.14).

Dann ist aber die Familie der kontextfreien Sprachen auch nicht unter Komplementbildung abgeschlossen. Denn wegen

$$L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$$

würde dann ja die Abgeschlossenheit unter Durchschnittsbildung folgen. □

Kapitel 5

Turingmaschinen und Berechenbarkeit

5.1 Deterministische Turingmaschinen

Die *Turingmaschine* erweitert nun die bisher betrachteten Automatenmodelle. Das Eingabeband wird zu einem sequentiellen Speicherband, auf dem beliebig viele Zeichen gespeichert und auch ohne Einschränkungen wieder gelesen werden können. Dafür entfällt der Kellerspeicher, den Kellerautomaten zur Verfügung haben. Die Turingmaschine wurde 1936 von *Alan Turing* als Berechnungsmodell zur Formalisierung des Algorithmusbegriffs eingeführt, der bis dahin nur auf einer informellen Ebene existierte.

Die Turingmaschine wird heute als mathematisch-formales Modell eines universellen Computers benutzt. Sie kann alle die Berechnungen durchführen, die auch ein Computer erledigen kann. Allerdings gibt es gewisse Probleme, die von einer Turingmaschine nicht gelöst werden können. Diese können aber auch nicht von realen Computern gelöst werden. Mittels Turingmaschinen werden somit auch die Grenzen algorithmisch-lösbarer Probleme aufgezeigt.

Eine Turingmaschine besteht aus einer Steuereinheit und einem Speicherband. Die Steuereinheit enthält das Turingmaschinen-Programm, sie kann endlich viele Zustände annehmen und über einen Lese- und Schreibkopf auf die einzelnen Felder des Speicherbandes sequentiell zugreifen. Jedes Feld kann ein Zeichen aus einem endlichen Zeichenvorrat aufnehmen. Auf dem Speicherband stehen unbegrenzt viele Felder zur Verfügung. In Abbildung 5.1 ist eine Turingmaschine (TM) schematisch dargestellt.

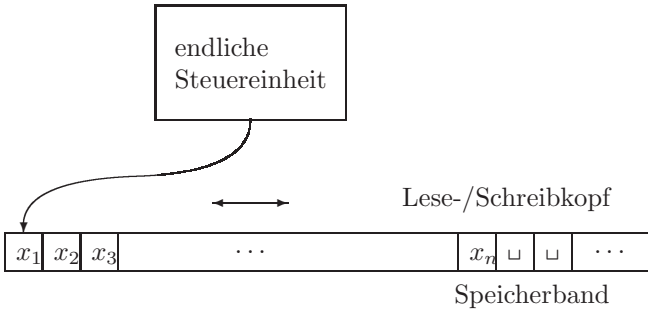


Abbildung 5.1: Turingmaschine (TM)

Initial enthält das Speicherband linksbündig das Eingabewort und der Lese-/Schreibkopf befindet sich auf dem Feld ganz links. Rechts von der Eingabe stehen auf dem Speicherband unendlich viele weitere Felder zur Verfügung. Diese Felder enthalten ein besonderes Zeichen, das so genannte *Blank*, geschrieben ⊔. Die Turingmaschine befindet sich initial im Startzustand. Sie verarbeitet die Eingabe getaktet entsprechend ihres Programms. Je Takt werden in Abhängigkeit des momentanen Zustands und dem unter dem Lesekopf befindlichen Zeichens die folgenden drei Aktionen ausgeführt:

1. Die Steuereinheit geht in einen neuen Zustand über, dem Folgezustand. Dies kann natürlich auch der bisherige sein.
2. Das Feld unter dem Lese-/Schreibkopf wird mit einem neuen Zeichen beschrieben. Dies kann natürlich auch das bisherige sein.
3. Der Lese-/Schreibkopf wird um ein Feld nach rechts oder um ein Feld nach links weiterbewegt.

Die Turingmaschine beendet ihre Arbeit und hält, wenn sie in einen der zwei besonderen Zustände gelangt, in den *akzeptierenden* Zustand oder in den *verwerfenden*

Zustand. Im ersten Fall wird die Eingabe akzeptiert, im zweiten wird sie verworfen (nicht akzeptiert). Gelangt die Turingmaschine nicht in einen dieser Zustände, dann hält sie nicht an und läuft unbegrenzt weiter.

Formal wird eine Turingmaschine durch ein 7-Tupel analog zu den bisherigen Automatenmodellen beschrieben. Es handelt sich dabei um ein deterministisches Automatenmodell. Auf die nichtdeterministische Variante wird hier nicht eingegangen.

Definition 5.1 (Turingmaschine)
Ein 7-Tupel $T = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ heißt Turingmaschine (TM), wobei:

- *Q ist eine nicht-leere, endliche Zustandsmenge.*
- *Σ ist das Eingabealphabet, das nicht das Blank $\sqcup$ enthält, $\sqcup \notin \Sigma$.*
- *Γ ist das Bandalphabet mit $\Sigma \subseteq \Gamma$ und $\sqcup \in \Gamma$.*
- *$\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$*
  mit $\delta(q, x) = (q', x', d)$ ist die Überführungsfunktion.

  Dabei ist q der aktuelle Zustand, x das gelesene Zeichen, q' der Folgezustand, x' das zu schreibende Zeichen und d die Bewegungsrichtung des Lese- und Schreibkopfes. $d = L$ steht für ein Feld nach links und $d = R$ für ein Feld nach rechts.
- *$q_0 \in Q$ ist der Startzustand.*
- *$q_{accept} \in Q$ ist der akzeptierende Zustand.*
- *$q_{reject} \in Q$ ist der verwerfende Zustand, mit $q_{accept} \neq q_{reject}$.*

Die Überführungsfunktion δ definiert das Turingmaschinen-Programm. Es ist üblich die Überführungsfunktion in einer so genannten *Überführungstabelle* zu notieren. Tabelle 5.1 spezifiziert dabei die Bedeutung der einzelnen Spalten.

δ :	Q	Γ	Q	Γ	$\{L, R\}$
	aktueller Zustand	gelesenes Zeichen	Folge-zustand	zu schreibendes Zeichen	Bewegungs-richtung

Tabelle 5.1: Überführungstabelle einer TM

Eine solche Überführungstabelle besteht aus endlich vielen Zeilen. Für jedes Argumentpaar $(q, x) \in Q \times \Gamma$ von δ gibt es genau eine Zeile in der Tabelle.

Zur formalen Beschreibung der Arbeitsweise einer Turingmaschine werden *Konfigurationen* und *Berechnungen* eingeführt. Eine Konfiguration beschreibt einen momentanen Gesamtzustand einer Turingmaschine, bestehend aus Zustand, Position des Lese-/Schreibkopfes und Bandinhalt. Bei der Spezifikation des Bandinhalts genügt es, sich auf einen endlichen Ausschnitt des unendlichen Speicherbandes zu beschränken. Die unendlich vielen *Blanks* rechts auf dem Speicherband werden meist nicht geschrieben. Initial wird der Bandinhalt durch die aus endlich vielen Zeichen bestehende Eingabe spezifiziert. Es befinden sich keine *Blanks* in der Eingabe. Die Eingabe kann dann während der Verarbeitung mit anderen Zeichen des Bandalphabets überschrieben werden, durchaus auch mit *Blanks*. Dabei bleibt die Anzahl der belegten Felder aber gleich und damit auch die Länge des relevanten Bandinhalts. In endlich vielen Takten können über die von der Eingabe initial beschriebenen Felder hinaus maximal nur endlich viele Felder neu beschrieben werden. Der relevante Bandausschnitt bleibt somit in jedem Fall endlich. Ein Bandinhalt kann also

immer als eine endliche Zeichenfolge spezifiziert werden, innere *Blanks* sind dabei mit aufzuführen, *Blanks* am rechten bzw. linken Rand nur soweit wie die jeweiligen Felder gerade von dem Lese-/Schreibkopf gelesen werden(Abbildung 5.2, Definition 5.2).

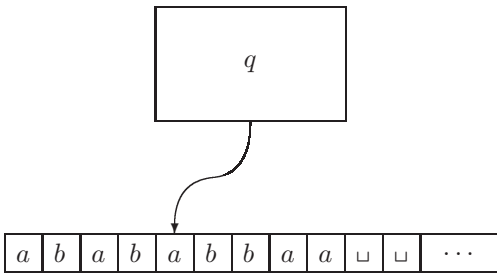


Abbildung 5.2: Turingmaschine in der Konfiguration *ababqabbaa*

Definition 5.2 (Konfiguration)
Es sei $T = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ *eine Turingmaschine,* $Q \cap \Gamma = \emptyset$*. Ein Tripel aus* $\Gamma^* \times Q \times \Gamma^*$*, geschrieben als Zeichenfolge* uqv *mit* $u, v \in \Gamma^*, q \in Q$*, heißt Konfiguration, falls gilt:*

- $q \in Q$ *ist der momentane Zustand von* T*.*
- $uv \in \Gamma^*$ *ist der momentane Bandinhalt von* T*.*
- *Der Lese-/Schreibkopf steht momentan auf dem ersten Zeichen von* v*, falls* $v \neq \epsilon$*. Ist* $v = \epsilon$*, so steht der Lese-/Schreibkopf auf einem Blank.*

Um die Darstellung des Bandinhalts auf eine endliche Zeichenfolge beschränken zu können, gilt für die Konfiguration am rechten Rand

$$u\,q\,\sqcup = uq\ .$$

Die Disjunktheit von Q und Γ dient nur dazu, die Zeichen von dem Zustand in einer Konfiguration unterscheiden zu können. Ist eine Turingmaschine gegeben, bei der das nicht zutrifft, so sind die Zustände entsprechend umzubenennen. Das Verhalten der Turingmaschine ändert sich dadurch nicht.

Für die Konfiguration am rechten Rand des Bandes gilt $u\,q\,\sqcup = uq$. Hier folgen unendlich viele Blanks, die nicht geschrieben werden müssen. Am linken Rand gilt das nicht entsprechend, denn dann würde die Information über die Anzahl der dort befindlichen endlich vielen Blanks bis zum Bandanfang verloren gehen.

Definition 5.3 (Startkonfiguration)
Es seien $T = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ *eine Turingmaschine,* $Q \cap \Gamma = \emptyset$*, und* $w \in \Sigma^*$ *das Eingabewort, bzw. die Eingabe für* T*. Die Konfiguration* $q_0 w$ *heißt Startkonfiguration von* T*.*

Eine Berechnung einer Turingmaschine setzt sich nun aus einer Folge von Berechnungsschritten zusammen. Sie beginnt mit der Startkonfiguration und durchläuft eine Folge von Konfigurationen bis sie ggf. hält. Je nach Zustand akzeptiert oder

verwirft sie die Eingabe. Eine Turingmaschine muss nicht nach endlich vielen Berechnungsschritten halten.

Definition 5.4 (Berechnungsschritt)
Es sei $T = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ eine Turingmaschine, $Q \cap \Gamma = \emptyset$. Seien $a, b, c \in \Gamma$, $u, v \in \Gamma^*$, $q \in Q \setminus \{q_{accept}, q_{reject}\}$ und $q' \in Q$.

Eine Konfiguration C_1 wird in eine Konfiguration C_2 überführt, wenn T in einem Berechnungsschritt (Schritt) von C_1 nach C_2 gelangt, d.h.

- *$C_1 = uqav$ wird in $ubq'v = C_2$ überführt, falls $\delta(q, a) = (q', b, R)$ und*
- *$C_1 = ucqav$ wird in $uq'cbv = C_2$ überführt, falls $\delta(q, a) = (q', b, L)$.*

Für die Spezialfälle gelten

- *$C_1 = uq$ wird in $ubq' = C_2$ überführt, falls $\delta(q, \sqcup) = (q', b, R)$ und*
- *$C_1 = qav$ wird in $q'bv = C_2$ überführt, falls $\delta(q, a) = (q', b, L)$.*

C_2 wird auch die Folgekonfiguration von C_1 genannt.

Der erste Spezialfall, die Überführung am rechten Rand, ist eigentlich bereits in der allgemeinen Definition mit $a = \sqcup$ und $v = \epsilon$ enthalten und könnte entfallen. Der zweite Spezialfall, die Überführung am linken Rand, dagegen ist notwendig. Er legt fest, dass der Lese-/Schreibkopf auf dem Feld ganz links bleibt, auch wenn die Überführungsfunktion eine Linksbewegung vorschreibt. Über den linken Rand kann eben nicht hinausgegangen werden. Achtung: Mit der Überführung $\delta(q, a) = (q, a, L)$ gerät eine Turingmaschine am linken Rand in eine nicht haltende Schleife.

Definition 5.5 (Akzeptierende und verwerfende Konfiguration)
Es seien $T = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ eine Turingmaschine, $Q \cap \Gamma = \emptyset$, und $u, v \in \Gamma^*$.

- *Eine Konfiguration $uq_{accept}v$ heißt akzeptierende Konfiguration.*
- *Eine Konfiguration $uq_{reject}v$ heißt verwerfende Konfiguration.*

Beide Konfigurationen heißen auch haltende Konfiguration. T hält an, sobald sie eine haltende Konfiguration erreicht. Ansonsten hält T nicht an.

Die Funktionswerte der Überführungsfunktion für den akzeptierenden Zustand und den verwerfenden Zustand sind also stets irrelevant, da die Turingmaschine bei Erreichen eines dieser Zustände immer anhält. Ließe die Definition einer Turingmaschine eine unvollständig definierte Überführungsfunktion zu, so könnten diese Funktionswerte undefiniert bleiben.

Eine Turingmaschine wird nun wie die bisher betrachteten Automatenmodelle zur Erkennung von Sprachen eingesetzt. Die dabei von der Startkonfiguration bis zur akzeptierenden Konfiguration durchlaufene Folge von Konfigurationen wird auch Berechnung genannt.

Definition 5.6 (Berechnung einer Turingmaschine)
Seien $T = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ eine Turingmaschine und $w \in \Sigma^*$ ein Wort. T akzeptiert dann w, wenn eine Folge von Konfigurationen $C_1, C_2, \ldots, C_k$ mit den folgenden Eigenschaften existiert:

1. *$C_1 = q_0 w$ ist die Startkonfiguration,*
2. *C_i wird in C_{i+1} für $1 \leq i \leq k-1$ überführt und*
3. *C_k ist eine akzeptierende Konfiguration.*

Man sagt dann, dass T die Sprache L erkennt, falls $L = \{w \mid T \text{ akzeptiert } w\}$, geschrieben $L(T) = L$.

Ein Wort wird also von einer Turingmaschine akzeptiert, wenn die Konfigurationenfolge ausgehend von der Startkonfiguration endlich ist und die Turingmaschine in dem akzeptierenden Zustand anhält. Der verbleibende Bandinhalt ist für die Spracherkennung ohne Bedeutung. Für die Nicht-Akzeptierung eines Wortes gibt es zwei Ursachen: Die Konfigurationenfolge ist zwar endlich, aber die Turingmaschine hält in dem verwerfenden Zustand oder die Konfigurationenfolge ist unendlich, die Turingmaschine hält also nicht an.

Definition 5.7 (Turing-erkennbar)
Eine Sprache L heißt Turing-erkennbar, erkennbar oder rekursiv aufzählbar, falls es eine Turingmaschine gibt, die L erkennt. Die Sprachfamilie oder Klasse der Turing-erkennbaren Sprachen wird mit $\mathcal{L}_{erk}$ bezeichnet.

Es ist häufig von großem Vorteil, insbesondere bei sehr langen Berechnungen, wenn man weiß, dass eine Turingmaschine für alle möglichen Eingaben anhält. Sie gelangt also immer entweder in eine akzeptierende oder eine verwerfende Konfiguration. Die Eingabe wird dann entweder akzeptiert oder verworfen. Die Turingmaschine trifft also auf jeden Fall eine Entscheidung über die Zugehörigkeit eines Wortes zu der Sprache, sie *entscheidet* die Sprache.

Definition 5.8 (Turing-entscheidbar)
Eine Sprache L heißt Turing-entscheidbar, entscheidbar oder rekursiv, falls es eine Turingmaschine gibt, die immer hält und L erkennt. Die Sprachfamilie oder Klasse der entscheidbaren Sprachen wird mit $\mathcal{L}_{ent}$ bezeichnet.

Beispiel 5.1
Konstruktion einer Turingmaschine, die die Sprache $L = \{a^i b^i c^i \mid i \in \mathbb{N}_0\}$ entscheidet.

Informelle Beschreibung der Turingmaschine:

1. Erstes a durch ein Blank überschreiben, weitere a überlesen.

2. Bereits überschriebene b überlesen, erstes b durch x überschreiben, weitere b überlesen. Falls kein b mehr vorhanden, halte verwerfend.

3. c überlesen, letztes c durch ein Blank überschreiben. Falls kein c mehr vorhanden, halte verwerfend.

4. Zurück an den Wortanfang.

5. Falls noch a vorhanden, gehe nach 1, sonst überprüfen, ob alles überschrieben.

6. Falls kein a mehr vorhanden und alles überschrieben, halte akzeptierend.

Formale Beschreibung:
$T = (\{0,1,2,3,4,5,6,7,8,9\}, \{a,b,c\}, \{a,b,c,x,\sqcup\}, \delta, 0, 8, 9)$ mit

δ:	Q	Γ	Q	Γ	$\{L,R\}$	
	0	a	1	$\sqcup$	R	// Schritt 1/5
	0	$\sqcup$	8	$\sqcup$	R	
	1	a	1	a	R	
	1	x	2	x	R	// Schritt 2
	2	x	2	x	R	
	2	b	3	x	R	
	1	b	3	x	R	
	3	b	3	b	R	
	3	c	4	c	R	// Schritt 3
	4	c	4	c	R	
	4	$\sqcup$	5	$\sqcup$	L	
	5	c	6	$\sqcup$	L	
	6	c	6	c	L	// Schritt 4
	6	b	6	b	L	
	6	x	6	x	L	
	6	a	6	a	L	
	6	$\sqcup$	0	$\sqcup$	R	
	0	x	7	x	R	// Schritt 5/6
	7	x	7	x	R	
	7	$\sqcup$	8	$\sqcup$	R	

Für alle hier nicht aufgeführten Paare $(q,y) \in Q \times \Gamma$ wird $\delta(q,y) = (9,y,R)$ definiert.
Nachfolgend ist für das Wort *aabbcc* die Berechnung von T aufgeführt:

0	a		a		b		b		c		c	
	$\sqcup$	1	a		b		b		c		c	
	$\sqcup$		a	1	b		b		c		c	
	$\sqcup$		a		x	3	b		c		c	
	$\sqcup$		a		x		b	3	c		c	
	$\sqcup$		a		x		b		c	4	c	
	$\sqcup$		a		x		b		c		c	4 $\sqcup$
	$\sqcup$		a		x		b		c	5	c	
	$\sqcup$		a		x		b	6	c		$\sqcup$	
	$\sqcup$		a		x	6	b		c		$\sqcup$	
	$\sqcup$		a	6	x		b		c		$\sqcup$	
	$\sqcup$	6	a		x		b		c		$\sqcup$	
6	$\sqcup$		a		x		b		c		$\sqcup$	
	$\sqcup$	0	a		x		b		c		$\sqcup$	
	$\sqcup$		$\sqcup$	1	x		b		c		$\sqcup$	
	$\sqcup$		$\sqcup$		x	2	b		c		$\sqcup$	
	$\sqcup$		$\sqcup$		x		x	3	c		$\sqcup$	
	$\sqcup$		$\sqcup$		x		x		c	4	$\sqcup$	
	$\sqcup$		$\sqcup$		x		x	5	c		$\sqcup$	
	$\sqcup$		$\sqcup$		x	6	x		$\sqcup$		$\sqcup$	
	$\sqcup$		$\sqcup$	6	x		x		$\sqcup$		$\sqcup$	
	$\sqcup$	6	$\sqcup$		x		x		$\sqcup$		$\sqcup$	
	$\sqcup$		$\sqcup$	0	x		x		$\sqcup$		$\sqcup$	
	$\sqcup$		$\sqcup$		x	7	x		$\sqcup$		$\sqcup$	
	$\sqcup$		$\sqcup$		x		x	7	$\sqcup$		$\sqcup$	
	$\sqcup$		$\sqcup$		x		x		$\sqcup$	8	$\sqcup$	

$aabbcc \in L(T)$, da T hält und 8 der akzeptierende Zustand ist.

Es gilt $L(T) = L$. Da T für jede Eingabe hält, ist L entscheidbar.

□

Eine Turingmaschine $T = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ kann in der folgenden Weise als *Überführungsgraph* dargestellt werden:

- Die Zustände $q \in Q$ werden als Knoten dargestellt und mit dem Zustandsnamen benannt.
- Der Startzustand wird mit einem Pfeil spezifiziert, der Endzustand mit einem zusätzlichen Kreis.
- Die Überführung vom Zustand q zum Zustand q' durch eine gerichtete Kante vom Knoten q zum Knoten q'. Die Überführungsfunktion $\delta(q, a) = (q', b, d)$ definiert dabei die Markierung der Kante mit $a \to b, d$.

In Abbildung 5.3 ist die Turingmaschine aus Beispiel 5.1 als Überführungsgraph dargestellt. Der verwerfende Zustand 9 und alle Überführungen dorthin sind der Übersichtlichkeit halber nicht dargestellt.

Abschließend sei noch auf die sich unmittelbar ergebende Beziehung zwischen Turing-erkennbaren und Turing-entscheidbaren Sprachen hingewiesen.

Satz 5.1
Jede Turing-entscheidbare Sprache ist auch Turing-erkennbar.

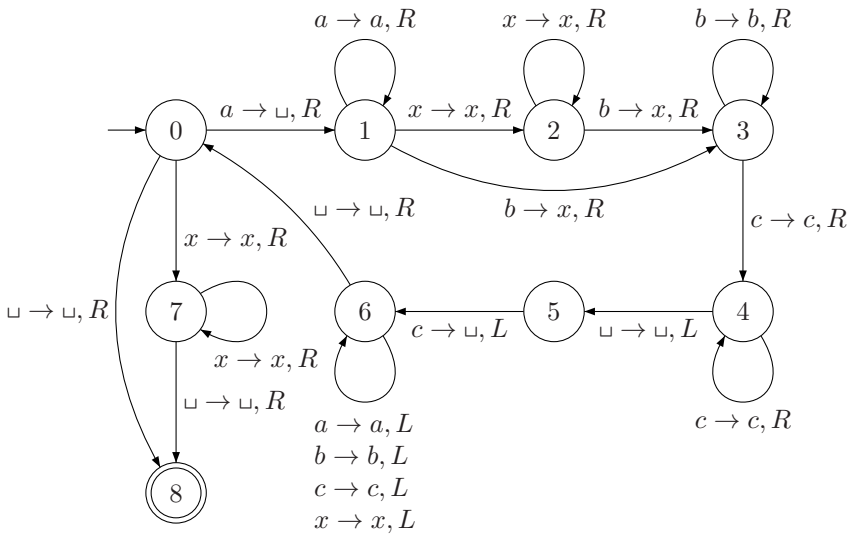


Abbildung 5.3: Überführungsgraph der TM T

Beweis:
Sei L eine Turing-entscheidbare Sprache. Dann gibt es eine TM T mit $L = L(T)$, die immer hält. Also ist L auch Turing-erkennbar. □

5.2 Intuitiver Algorithmusbegriff

Der Begriff *Algorithmus* ist heute untrennbar mit der Erstellung von Computerprogrammen verbunden. Aber schon lange bevor es Computer überhaupt gab, gab es Algorithmen, auch wenn dieser Begriff noch nicht verwendet wurde. So z. B. die im *Papyrus Rhind* zusammengestellten Rechenaufgaben aus dem 16. Jh. v. Chr. oder das von Euklid um 300 v. Chr. aufgeschriebene Verfahren zur Bestimmung des größten gemeinsamen Teilers zweier Zahlen, das heute *Euklidischer Algorithmus* genannt wird. Informell bzw. intuitiv läßt sich die Bedeutung des Begriffs Algorithmus folgendermaßen beschreiben:

> *Ein Algorithmus ist ein allgemeines, eindeutiges Verfahren zur Lösung einer Klasse gleichartiger Probleme, gegeben durch einen aus elementaren Anweisungen bestehenden Text.*

Hierunter fallen natürlich insbesondere mathematische Verfahren und – wie eingangs schon festgestellt – Computerprogramme. Aber auch Bedienungsanleitungen von technischen Anlagen, Zusammenbauanleitungen von Bausätzen oder auch Kochrezepte können in einem erweiterten Sinn als Algorithmen bezeichnet werden.

Die Bedeutung des Begriffs *Algorithmus* wird nach Donald E. Knuth (*1938) durch Angabe weiterer Kriterien präzisiert (Begriffsbestimmung 5.9).

Begriffsbestimmung 5.9 (Intuitiver Algorithmus)
Ein intuitiver Algorithmus ist ein allgemeines, eindeutiges Verfahren zur Lösung einer Klasse gleichartiger Probleme, gegeben durch einen aus elementaren Anweisungen bestehenden Text, der die nachfolgend aufgeführten Kriterien erfüllt:

- ***Endlichkeit:*** *Ein Algorithmus besteht aus endlich vielen elementaren Anweisungen (Schritten).*
- ***Bestimmtheit:*** *Jeder Schritt des Algorithmus muss eindeutig definiert sein (determiniert). Der jeweils nächste auszuführende Schritt muss eindeutig bestimmt sein (deterministisch). Dazu gehört auch die Bestimmung des ersten Schrittes und des Algorithmusendes.*
- ***Eingabe:*** *Ein Algorithmus hat keine oder endlich viele Eingabewerte aus einer Eingabemenge. Diese Werte werden initial an den Algorithmus gegeben bevor er mit der Ausführung des ersten Schritts beginnt.*
- ***Ausgabe:*** *Ein Algorithmus hat mindestens einen Ausgabewert aus einer Ausgabemenge. Der Algorithmus ordnet der Eingabe eine Ausgabe zu, er definiert also eine Funktion.*
- ***Effektivität:*** *Die einzelnen Anweisungen müssen überhaupt ausführbar sein.*

Die obige Beschreibung eines Algorithmus bleibt aber trotz der Präzisierungen sehr vage. Begriffe wie „Verfahren“, „Anweisung“, „Schritt“ oder „Wert“ sind zwar intuitiv verständlich, aber nicht im mathematischen Sinn exakt definiert. Insbesondere erfordert die Interpretation des Begriffs „elementar“ eine vage Vorstellung von der Zielgruppe oder der Zielmaschine, die den Algorithmus ausführen soll. Deshalb wird dieser Algorithmusbegriff auch *intuitiv* genannt, und der Begriff wird deshalb auch nur erläutert und nicht definiert.

Die Effektivität der einzelnen Anweisungen ist ein ganz wichtiges Kriterium. Wenn z. B. eine Bedingung in einer Anweisung auf der Lösung eines nicht lösbaren oder bisher noch ungelösten Problems beruht, dann ist diese Anweisung nicht ausführbar, also nicht effektiv. Des Weiteren fordert diese Eigenschaft, aber auch die der Elementarität, dass eine Anweisung in endlicher Zeit zum Abschluss kommen muss. Eine unendlich viel Zeit benötigende Anweisung ist weder effektiv noch elementar.

Knuth führt als ein weiteres Kriterium eines intuitiven Algorithmus auch die Endlichkeit der Schrittausführungen auf und nennt Verfahren, in denen dies für zumindest einige Eingaben nicht erfüllt ist, *Berechnungsmethoden*. Andere Autoren dagegen verzichten auf die Forderung dieses Kriteriums, so dass auch Verfahren, die für gewisse Eingaben möglicherweise unendlich viele Schrittausführungen aufweisen, auch als Algorithmen bezeichnet werden können. Es wird hier also von Algorithmen, die immer terminieren, und von Algorithmen, die nicht immer terminieren, gesprochen.

Die hier verwendete Begriffsbestimmung eines Algorithmus schließt dann auch z. B. Betriebssysteme und sonstige Programme mit ein, die so lange nicht terminieren, bis sie explizit durch eine externe Eingabe oder durch Abbruch des Geräts beendet werden.

Darüber hinaus gibt es die so genannten *probabilistischen Algorithmen*, die das Kriterium der Bestimmtheit nicht erfüllen und dennoch als Algorithmen bezeichnet werden. Allerdings wird eine derartige Erweiterung des Algorithmusbegriffs hier nicht vorgenommen, und es werden auch solche Algorithmen nicht betrachtet. Ein weiteres Konzept, das gegen das Kriterium der Bestimmtheit verstößt, ist der *Nichtdeterminismus*. Insofern werden in diesem Kontext auch nur deterministische Verfahren betrachtet.

Mittels Algorithmen wird nun der *Berechenbarkeitsbegriff* eingeführt. Ein Algorithmus berechnet in Abhängigkeit von seiner Eingabe eine Ausgabe, durch ihn wird also eine Funktion bestimmt. Eine *partielle* Funktion ist dabei eine Funktion, bei der nicht für alle Argumente Funktionswerte definiert sein müssen.

Begriffsbestimmung 5.10 (Intuitiv berechenbar)
Eine partielle Funktion f heißt intuitv berechenbar, falls es einen intuitiven Algorithmus gibt, der f berechnet.

In der Diskussion des Berechenbarkeitsbegriffs nimmt die Frage nach der Terminierung eines Algorithmus für gewisse Eingaben eine zentrale Position ein. Eine wie hier vorgenommene Verallgemeinerung des Algorithmusbegriffs erscheint auch deshalb angemessen, da sich ansonsten die Frage nach der Terminierung eines Algorithmus – zumindest in der strengen Formulierung – nicht stellen würde, denn diese terminierten ja immer.

Beispiele 5.2
Betrachte die folgenden Funktionen:

1. Addition zweier nicht-negativer ganzer Zahlen:

   $f_{add} : \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0$ mit $f_{add}(a, b) = a + b$.

   Zur Berechnung von f_{add} kann leicht ein Algorithmus angegeben werden.

2. Übersetzung von Programmen:

   Ein Compiler übersetzt ein in einer anwendungsorientierten Programmiersprache geschriebenes Programm $w \in X^+$ in ein Programm $w' \in Y^+$ einer

maschinenorientierten Programmiersprache, das auf einem Zielrechner ausgeführt werden kann. Ein Compiler, selbst ein auf einem Rechner ausführbares Programm, stellt die Implementierung eines Algorithmus dar. Ein Compiler definiert also eine Funktion

$f_{comp} : X^+ \to Y^*$ mit

$$f_{comp}(w) = \begin{cases} w' \neq \epsilon & : \quad w \text{ ist korrekt} \\ \epsilon & : \quad w \text{ ist nicht korrekt.} \end{cases}$$

f_{comp} wird durch den Compiler berechnet.

3. Terminierung von Programmen:

   Ein Programm $P \in Y^+$ wird auf einem Rechner mit der Eingabe $w \in X^*$ ausgeführt. Einige Programme benötigen etwas mehr Ausführungszeit, und es stellt sich machmal die Frage, ob das Programm noch läuft, weil es mit der Berechnung noch nicht fertig ist, oder ob es noch läuft, weil es für diese Eingabe nicht terminiert. Die Funktion

   $f_{term} : Y^+ \times X^* \to \{0, 1\}$ mit

   $$f_{term}(P, w) = \begin{cases} 1 & : \quad P \text{ terminiert mit } w \\ 0 & : \quad P \text{ terminiert nicht mit } w \end{cases}$$

   beschreibt die gewünschte Ein-/Ausgabezuordnung eines Algorithmus, der diese Frage beantwortet.

Die drei Funktionen sind wohldefiniert und dennoch gibt einen grundlegenden Unterschied. Während für f_{add} und f_{comp} Algorithmen zur Berechnung der Funktionen angegeben werden können, so ist für f_{term} kein Algorithmus bekannt, der die Funktion berechnet. Man kann sogar zeigen, daß es keinen gibt.

1. f_{add} ist intuitiv berechenbar.
2. f_{comp} ist intuitiv berechenbar.
3. f_{term} ist nicht intuitiv berechenbar.

□

Auf der Basis dieser intuitiven Begriffe können nun aber keine formalen Beweise und Schlussfolgerungen durchgeführt werden. Es ist deshalb notwendig diesen Begriffen einen formalen Algorithmus- und einen formalen Berechenbarkeitsbegriff beiseite zu stellen. Z.B. die Tatsache, dass es bisher nicht gelungen ist, einen intuitiven Algorithmus zur Berechnung der Funktion f_{term} zu finden, ist kein Beweis, dass es keinen gibt. Gerade ein solcher Beweis erfordert eine Formalisierung.

5.3 Turing-Berechenbarkeit

Die Formalisierung des Algorithmusbegriffs wurde 1936 aus unterschiedlichen Ansätzen entwickelt. Ein funktionenorientierter Ansatz kam von Alonzo Church, der *λ-Kalkül*, ein über ein Maschinenmodell entwickelter Ansatz stammt von Alan Turing, die heute nach ihm benannte *Turingmaschine*. Daneben gibt es weitere Algorithmus-Formalisierungen. Im Abschnitt 5.1 sind Turingmaschinen, die Sprachen erkennen bzw. entscheiden, bereits eingeführt worden. Auf das λ-Kalkül und andere Formalisierungen wird hier nicht eingegangen.

Die Turingmaschine als Sprachakzeptor berechnet nun eine Funktion mit einem Wort über dem Eingabealphabet als Argument und zwei möglichen Funktionswerten, „akzeptieren“ oder „verwerfen“. Entscheidet die Turingmaschien eine Sprache, so ist diese Funktion für alle Wörter über dem Eingabealphabet definiert, man spricht dann auch von einer *totalen* Funktion. Erkennt die Turingmaschine aber eine Sprache nur, d.h. sie hält für einige Wörter ggf. nicht an, so muss diese Funktion nicht für alle Wörter über dem Eingabealphabet definiert sein, man spricht dann von einer *partiellen* Funktion.

Zur Berechnung von Funktionen mit mehr als zwei Funktionswerten, wie z.B. der Addition zweier Zahlen (Funktion f_{add} aus Beispiel 5.2), muss das Modell der Turingmaschine um die Ausgabemöglichkeit von Funktionswerten erweitert werden. Die Ausgabe einer Turingmaschine werde hier wie folgt definiert: Hält die Turingmaschine akzeptierend, dann wird der Bandinhalt rechts vom Lese-/Schreibkopf, einschließlich des Feldes unter dem Lese-/Schreibkopf, als Funktionswert interpretiert.

Definition 5.11 (Turing-berechenbar)

- *Die Turingmaschine $T = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ berechnet die Funktion $f_T : \Sigma^* \to \Gamma^*$, wenn T mit der Eingabe $w \in \Sigma^*$ in eine akzeptierende Konfiguration $uq_{accept}v$ gelangt. Die Zeichenfolge v ist der Funktionswert $f_T(w) = v$, $f_T(w)$ ist definiert. Andernfalls berechnet T keinen Funktionswert und $f_T(w)$ ist undefiniert.*
- *Eine Funktion f heißt Turing-berechenbar, falls es eine Turingmaschine T gibt, die f berechnet, also mit $f = f_T$.*
- *Ist f_T eine totale Funktion, dann berechnet T für jedes $w \in \Sigma^*$ einen Funktionswert.*
- *Ist f_T eine partielle Funktion, dann kann es $w \in \Sigma^*$ geben, für die T keinen Funktionswert berechnet.*

Beispiel 5.3
Konstruktion einer Turingmaschine, die die Inkrementfunktion $f : \mathbb{N}_0 \to \mathbb{N}_0$ mit $f(n) = n+1$ berechnet, wobei $n \in \mathbb{N}$ als Binärzahl ohne führende Nullen dargestellt wird und $n = 0$ als 0.

Informelle Beschreibung der Turingmaschine:

1. Eingabewort (Binärzahl) um ein Feld nach rechts schieben. Das zusätzliche Feld links dient zur Erkennung des linken Rands und es kann ggf. bei einem Übertrag bis zur höchsten Stelle mit 1 überschrieben werden.
2. Auf die letzte Ziffer der Zahl gehen.

3. Falls diese Ziffer gleich 1, auf 0 setzen, ein Feld nach links gehen und diese Anweisung wiederholen.
4. Falls diese Ziffer gleich 0 oder $\sqcup$, auf 1 setzen.
5. Auf die erste Ziffer der Zahl gehen und akzeptierend terminieren.

Formale Beschreibung:

$T_{ink} = (\{S, T, E, N, B, C, L, A, R\}, \{0, 1\}, \{0, 1, \sqcup\}, \delta, S, A, R)$ mit

δ:

Q	Γ	Q	Γ	$\{L, R\}$	
S	0	T	$\sqcup$	R	// $n = 0$
T	$\sqcup$	B	0	R	
S	1	E	$\sqcup$	R	// $n \neq 0$,
E	1	E	1	R	// Zahl ein Feld
E	0	N	1	R	// nach rechts schieben
E	$\sqcup$	B	1	R	
N	0	N	0	R	
N	1	E	0	R	
N	$\sqcup$	B	0	R	
B	$\sqcup$	C	$\sqcup$	L	// auf letzte Ziffer gehen
C	1	C	0	L	// $n \leftarrow n + 1$
C	0	L	1	L	
C	$\sqcup$	A	1	L	
L	0	L	0	L	// auf erste
L	1	L	1	L	// Ziffer gehen
L	$\sqcup$	A	$\sqcup$	R	// akzeptieren

Alle in der Tabelle nicht aufgeführten Paare $(q, y) \in Q \times \Gamma$ sind wie folgt definiert: $\delta(q, y) = (R, y, R)$.

Nachfolgend ist für die Binärzahl $n = 10011$ die Berechnung von T_{ink} aufgeführt:

S	1		0		0		1		1		
	$\sqcup$	E	0		0		1		1		
	$\sqcup$		1	N	0		1		1		
	$\sqcup$		1		0	N	1		1		
	$\sqcup$		1		0		0	E	1		
	$\sqcup$		1		0		0		1	E	$\sqcup$
	$\sqcup$		1		0		0		1		1 B
	$\sqcup$		1		0		0		1	C	1
	$\sqcup$		1		0		0	C	1		0
	$\sqcup$		1		0	C	0		0		0
	$\sqcup$		1	L	0		1		0		0
	$\sqcup$	L	1		0		1		0		0
L	$\sqcup$		1		0		1		0		0
	$\sqcup$	A	1		0		1		0		0

$f_{T_{ink}}(10011) = 10100$, da T_{ink} akzeptierend hält.

Es gilt $f_{T_{ink}} = f$, was eigentlich zu beweisen wäre. f ist damit Turing-berechenbar.

Zu beachten ist aber:
Genau genommen berechnet T die partielle Funktion $f_{T_{ink}} : \{0, 1\}^* \to \{0, 1, \sqcup\}^*$ mit

$$f_{T_{ink}}(w) = \begin{cases} f(n) & : \; w = n \text{ ist Binärzahl ohne führende Nullen} \\ \text{undefiniert} & : \; w \text{ enthält führende Nullen oder } w = \epsilon \end{cases}$$

und nicht $f : \mathbb{N}_0 \to \mathbb{N}_0$. Dieser Unterschied werde hier aber nicht weiter beachtet.

In Abbildung 5.4 ist die Turingmaschine T_{ink} als Überführungsgraph dargestellt. Der verwerfende Zustand R und alle dahingehenden Überführungen sind weggelassen.

□

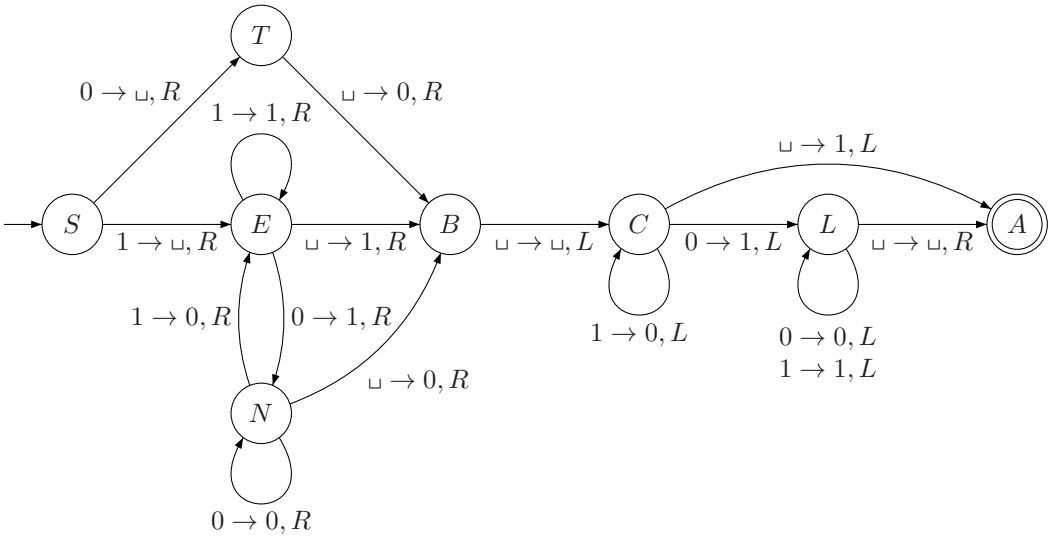


Abbildung 5.4: Überführungsgraph der TM T_{ink}

Die Definition und Arbeitsweise einer Turingmaschine erfüllt sicherlich ausnahmslos die Anforderungen des intuitiven Algorithmusbegriffs nach der Begriffsbestimmung 5.9. Es kann deshalb die nachfolgende Folgerung geschlossen werden.

Folgerung 5.2 (Turing-berechenbar und intuitiv-berechenbar)
Jede Turing-berechenbare Funktion ist auch intuitv-berechenbar.

Wie aber steht es mit der Umkehrung dieser Aussage? Es gibt neben der Turing-Berechenbarkeit diverse weitere Berechenbarkeits- und damit auch weitere formale Algorithmusbegriffe, *allgemein-rekursiv berechenbar*, *λ-berechenbar*, *Markov-berechenbar*, *RAM-berechenbar* oder auch *PL-berechenbar*. Es kann gezeigt werden, daß alle diese Berechenbarkeitsbegriffe äquivalent sind. Für alle Formalisierungen des Berechenberkeitsbegriffs gilt damit natürlich auch die intuitive Berechenbarkeit. Da andererseits bisher jeder intuitive Algorithmus in einen formalen Algorithmus nach irgendeinem formalen Algorithmusbegriff umgeformt werden konnte, ist man davon überzeugt, dass auch die Umkehrung gilt. Diese Aussage läßt sich natürlich nur unter Plausibilitätsbetrachtungen herleiten und kann nicht allgemein bewiesen werden. Sie wird als *Churchsche These* oder auch als *Church-Turing These* bezeichnet.

Hypothese 5.3 (Church-Turing These)
Jede intuitiv-berechenbare Funktion ist auch Turing-berechenbar.

Natürlich gilt diese Hypothese auch für alle anderen formalen Berechenbarkeitsbegriffe. Insbesondere sei hier auf die RAM- und die PL-Berechenbarkeit hingewiesen. Eine Random-Access-Maschine (RAM) ist ein formales Modell, das sich in

Aufbau und Funktionsweise sehr eng an den heutigen grundlegenden Rechnerarchitekturen und ihren maschinenorientierten Programmiersprachen orientiert, dem *von-Neumann Rechner*. PL ist eine algorithmische Sprache, die in ihren grundlegenden Konzepten heutigen anwendungsorientierten Programmiersprachen entspricht. Man kann sich vor diesem Hintergrund sehr leicht plausibel machen, daß jeder in intuitiver Notation vorliegende Algorithmus zur Lösung eines Problems mittels realer Programmiersprachen auf realen Rechnern ausgeführt und damit gelöst werden kann.

Die bewiesene bzw. plausible Äquivalenz unter den diversen Berechenbarkeits- und Algorithmusbegriffen führt zu der folgenden vereinheitlichenden Definition.

Definition 5.12 (berechenbar)
Eine Funktion heißt berechenbar, wenn sie intuitiv-berechenbar, Turing-berechenbar oder nach irgendeiner anderen Formalisierung berechenbar ist.

Es waren in den Beispielen 5.2 und 5.3 bereits berechenbare Funktionen und eine nicht berechenbare Funktion angegeben. Auf die nicht berechenbare Funktion des Beispiels 5.2 wird später im Abschnitt 6.2 eingegangen. Ausgehend von der Inkrementfunktion (Beispiel 5.3) könnte nun z.B. auch leicht gezeigt werden, dass die Addition, die Multiplikation, deren Umkehrfunktionen und alle darauf aufbauenden arithmetischen Funktionen berechenbar sind.

Turingmaschinen sind bisher auf zwei Arten beschrieben worden. Die exakte *formale Beschreibung* definiert eine Turingmaschine mit allen ihren Zuständen, Alphabeten und Überführungen. Die *informelle Beschreibung* einer Turingmaschine beschreibt ihre Kopfbewegungen auf dem Band und wie und welche Daten auf dem Band gespeichert und gelesen werden, aber Zustände und Überführungen werden nicht mehr spezifiziert. Beide Beschreibungsformen sind in den vorangegangen Beispielen verwendet worden. In einem weiteren Abstraktionsschritt werden nun Algorithmen vollkommen unabhängig von irgendwelchen formalen Berechnungsmodellen beschrieben, wie z.B. einer Turingmaschine, denn mit der Church-Turing These und der Äquivalenz aller formalen Berechenbarkeitsbegriffe wird es zu einem solchen *intuitiven Algorithmus* immer auch einen formalen Algorithmus geben, insbesondere auch eine Turingmaschine.

Kapitel 6

Entscheidbarkeit

6.1 Entscheidbare Probleme

Während im vorangehenden Abschnitt Algorithmen zur Berechnung beliebiger Funktionen und ihre Formalisierung mittels Turingmaschinen eingeführt und diskutiert wurden, geht es in diesem Abschnitt wieder nur um Algorithmen zur Lösung von *Entscheidungsproblemen*, Problemen, die mit *Ja* oder *Nein* beantwortet werden können, also die Berechnung von Funktionen mit nur zwei Funktionswerten. Es sollen die grundsätzlichen Fähigkeiten und Begrenzungen algorithmischer Problemlösungen diskutiert werden, und dazu genügt es, sich auf solche Probleme zu beschränken. Die Ergebnisse lassen sich dann leicht auf komplexere Probleme übertragen.

Ziel dieses Kapitels ist es zu zeigen, dass es unentscheidbare Probleme gibt. Dies wird am Beispiel des *Halteproblems* aufgezeigt. Vorab aber werden einige entscheidbare Probleme besprochen. Und auch hier wieder findet eine Beschränkung auf Sprachen statt. Probleme werden als so genanntes *Wortproblem* dargestellt. Gegeben sind eine Sprache über einem Alphabet und ein Wort über diesem Alphabet, und es ist die Frage zu beantworten, ob das Wort zur Sprache gehört oder nicht.

Definition 6.1 (Wortproblem)
Die Frage, ob ein Wort zu einer Sprache gehört oder nicht, heißt Wortproblem.

Für die Klasse der regulären Sprachen und die Klasse der kontextfreien Sprachen wird die Entscheidbarkeit des Wortproblems gezeigt. Diese Ergebnisse liefern die Grundlage dafür, dass ein Compiler von einem Programm, dessen Syntax mittels einer kontextfreien Grammatik definiert ist, entscheiden kann, ob es syntaktisch korrekt ist oder nicht, also ob das Wort zur Sprache gehört oder nicht. Das Wortproblem selbst kann wie folgt als eine Sprache dargestellt werden:

$$L_K := \{\langle L, w\rangle \mid L \text{ ist Element einer Klasse von Sprachen } K \text{ und das Wort } w \in L\}$$

Das Problem, ob w Wort der Sprache L ist, ist äquivalent zu dem Problem, ob $\langle L, w\rangle$ Wort der Sprache L_K ist. Kann nun gezeigt werden, dass L_K entscheidbar ist, dann ist damit gezeigt, dass das Wortproblem für jede Sprache dieser Klasse entscheidbar ist. Es werde nun zunächst die Klasse der regulären Sprachen betrachtet.

Definition 6.2
$L_{RA} := \{\langle R, w\rangle \mid R$ *ist ein regulärer Ausdruck und* $w \in L(R)\}$.

Satz 6.1
L_{RA} *ist eine entscheidbare Sprache.*

Beweis:
Zum Beweis wird eine Turingmaschine T spezifiziert, die L_{RA} entscheidet.

Informelle Beschreibung von T:

1. Analysiere die Eingabe $\langle R, w\rangle$ dahingehend, ob R ein regulärer Ausdruck ist und w ein Wort. Ist das nicht der Fall, so halte verwerfend.
2. Überführe den regulären Ausdruck R nach Lemma 3.6 in einen äquivalenten NEA N.

3. Überführe N nach Satz 2.2 in einen äquivalenten DEA E.
4. Simuliere E mit der Eingabe w.
5. Endet die Simulation in einem akzeptierenden Zustand von E, so halte akzeptierend, endet die Simulation in einem nichtakzeptierenden Zustand von E, so halte verwerfend.

T hält mit jeder Eingabe nach endlich vielen Schritten an, insbesondere weil auch ein DEA immer nach endlich vielen Schritten hält. Also entscheidet T die Sprache L_{RA}. □

Aus dem vorangehenden Satz ergibt sich natürlich auch unmittelbar, dass die entsprechende Fragestellung mit einem gegebenen deterministischen oder nichtdeterministischen endlichen Automaten auch entschieden werden kann. Die Eingabeanalyse ist zu modifizieren und die jeweiligen Überführungen in die äquivalenten Modelle können entfallen.

Satz 6.2
Jede reguläre Sprache ist entscheidbar.

Beweis:
Sei L eine reguläre Sprache. Dann gibt es einen regulären Ausdruck, der L repräsentiert. Seien R der reguläre Ausdruck mit $L = L(R)$ und w ein Wort.

Informelle Beschreibung von T:

1. Führe die TM aus dem Beweis von Satz 6.1 mit der Eingabe $\langle R, w\rangle$ aus.
2. Falls diese TM akzeptiert, dann halte akzeptierend, andernfalls halte verwerfend.

□

Als nächstes werde das Wortproblem für kontextfreie Sprachen betrachtet.

Definition 6.3
$L_{KF} := \{\langle G, w\rangle \mid G$ *ist eine kontextfreie Grammatik und* $w \in L(G)\}$.

Satz 6.3
L_{KF} *ist eine entscheidbare Sprache.*

Beweis:
Zum Beweis wird eine Turingmaschine T spezifiziert, die L_{KF} entscheidet.

Informelle Beschreibung von T:

1. Analysiere die Eingabe $\langle G, w\rangle$ dahingehend, ob G eine kontextfreie Grammatik ist und w ein Wort. Ist das nicht der Fall, so halte verwerfend.
2. Überführe die kontextfreie Grammatik G nach Satz 4.1 in eine äquivalente Grammatik in Chomsky Normalform.

3. Liste alle Ableitungen der Länge $2n-1$ auf. Dabei ist $n = |w|$.
   Ausnahme: Ist $n = 0$, dann liste alle Ableitungen der Länge 1 auf.

4. Ist unter diesen Ableitungen die Ableitung für w, dann halte akzeptierend, andernfalls halte verwerfend.

T hält mit jeder Eingabe nach endlich vielen Schritten an, insbesondere weil es nur endlich viele Ableitungen der Länge $2n-1$ gibt, und Wörter der Länge n haben in Grammatiken von Chomsky Normalform ausschließlich Ableitungen dieser Länge. Eine Ausnahme bildet das leere Wort, dies wird über eine Ableitung der Länge 1 erzeugt. Also entscheidet T die Sprache L_{KF}. □

Satz 6.4
Jede kontextfreie Sprache ist entscheidbar.

Beweis:
Sei L eine kontextfreie Sprache. Dann gibt es eine kontextfreie Grammatik, die L erzeugt. Seien G die kontextfreie Grammatik mit $L = L(G)$ und w ein Wort.

Informelle Beschreibung von T:

1. Führe die TM aus dem Beweis von Satz 6.3 mit der Eingabe $\langle G, w\rangle$ aus.

2. Falls diese TM akzeptiert, dann halte akzeptierend, andernfalls halte verwerfend.

□

Natürlich gibt es viele weitere entscheidbare Probleme im Zusammenhang mit Sprachen und auch unabhängig von Sprachen. Darauf soll hier aber nicht eingegangen werden. Bleibt für diesen Abschnitt zusammenfassend auf die Hierarchie unter den vier betrachteten Sprachfamilien hinzuweisen (Satz 6.5, Abbildung 6.1).

Satz 6.5
$\mathcal{L}_{reg} \subset \mathcal{L}_{kf} \subset \mathcal{L}_{ent} \subset \mathcal{L}_{erk}$

Beweis:

I. $\mathcal{L}_{reg} \subset \mathcal{L}_{kf}$: Satz 4.5.

II. $\mathcal{L}_{kf} \subset \mathcal{L}_{ent}$: Die Teilmengenbeziehung ergibt sich aus Satz 6.4 und die echte Teilmengenbeziehung aus den Beispielen 4.14 und 5.1.

III. $\mathcal{L}_{ent} \subset \mathcal{L}_{erk}$: Die Teilmengenbeziehung ergibt sich aus Satz 5.1. Die echte Teilmengenbeziehung wird im nächsten Abschnitt mit der Spezifikation einer Sprache gezeigt, die Turing-erkennbar aber nicht entscheidbar ist.

□

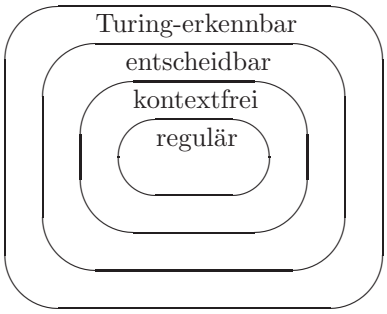


Abbildung 6.1: Hierarchie der Sprachfamilien

6.2 Das Halteproblem

Es sei ein Algorithmus durch ein Programm implementiert, und dieses Programm werde auf einem Computer mit einer Eingabe ausgeführt. Viele Programme verarbeiten die Eingabe, terminieren nach einer bestimmten Zeit und geben eine Ausgabe aus. Es gibt aber auch einige Programme, die haben eine längere Laufzeit, und man wartet auf die Ausgabe und die Terminierung und wartet und wartet und ... .

Terminiert das Programm nun irgendwann oder terminiert es nicht? Gerade bei Programmen, die ohnehin eine längere und unbekannte Laufzeit haben, wäre es hilfreich, ein anderes Programm zur Verfügung zu haben, das Programme und ihre Eingabe auf Terminierung überprüft. Das führt auf die Fragestellung des *Halteproblems* und die Frage nach der Existenz eines Algorithmus zur Lösung des Halteproblems. Das Halteproblem selbst lautet folgendermaßen: Hält ein Algorithmus mit einer bestimmten Eingabe? In Definition 6.4 ist das Halteproblem mittels Turingmaschinen formalisiert.

Definition 6.4 (Halteproblem)
Die Frage, ob eine Turingmaschine angesetzt auf eine Eingabe anhält oder nicht, heißt Halteproblem.

Die Frage nach der Existenz eines Algorithmus zur Lösung des Halteproblems muss leider mit *Nein* beantwortet werden (Satz 6.6). Die Nicht-Existenz eines solchen Algorithmus wird nach dem Prinzip des *Zweiten Cantorschen Diagonalverfahrens* gezeigt. Man spricht auch von einem Beweis durch *Diagonalisierung*.

Satz 6.6 (Unentscheidbarkeit des Halteproblems)
Es gibt keine Turingmaschine, die das Halteproblem entscheidet.

Dem Beweis dieses Satzes seien noch ein paar Bemerkungen vorangestellt. Sie sollen den Zugang zu dieser Beweistechnik erleichtern.

Sind zwei endliche Mengen bzgl. ihrer Größe miteinander zu verglichen, so zählt man einfach ihre Elemente. Die Mengen haben die gleiche Größe, die gleiche *Kardinalität* oder die gleiche *Mächtigkeit*, wenn ihre Elementanzahlen gleich sind, ansonsten ist die eine größer als die andere. Eine solche einfache Abzählung versagt zum Vergleich von unendlichen Mengen, denn sie haben unendlich viele Elemente. Hierzu wird der Begriff der *Abzählbarkeit* mittels Funktionen eingeführt (Definitionen 6.5 und 6.6).

Definition 6.5
Zwei Mengen A und B haben die gleiche Größe, wenn es eine bijektive Abbildung $f : A \to B$ gibt.

Zwei Mengen sind also gleich groß, wenn jedem Element von A genau ein Element von B zugeordnet werden kann und wenn dabei auch jedes Element von B in einer solchen Zuordnung vorkommt.

Definition 6.6 (Abzählbarkeit)
Eine Menge heißt abzählbar, wenn sie endlich ist oder die gleiche Größe wie $\mathbb{N}$ hat. Ansonsten heißt sie nicht abzählbar oder überabzählbar.

Beispiel 6.1
Die Menge der natürlichen Zahlen $\mathbb{N}$ ist abzählbar. Sie hat die gleiche Größe wie sie selbst, denn die Abbildung $f : \mathbb{N} \to \mathbb{N}$ mit $f(n) = n$ ist bijektiv.

Auch die Menge der geraden natürlichen Zahlen $\mathbb{N}_{gerade}$ ist abzählbar. Sie hat die gleiche Größe wie $\mathbb{N}$, denn die Abbildung $f : \mathbb{N} \to \mathbb{N}_{gerade}$ mit $f(n) = 2n$ ist bijektiv.

Dies scheint jetzt etwas merkwürdig zu sein, denn $\mathbb{N}_{gerade} \subset \mathbb{N}$. Eine bijektive Abbildung gibt es hier natürlich nur, weil es sich um unendliche Mengen handelt.

□

Das erste Cantorsche Diagonalverfahren zeigt nun, dass auch die Menge der rationalen Zahlen $\mathbb{Q}$ abzählbar ist und damit die gleiche Größe wie $\mathbb{N}$ hat. Mittels des zweiten Cantorschen Diagonalverfahrens dagegen wird die Überabzählbarkeit der reellen Zahlen $\mathbb{R}$ nachgewiesen. Beide Verfahren können in den entsprechenden Lehrbüchern zur Mathematik oder auch in [4] nachgelesen werden.

Mittels solcher Größenvergleiche kann nun bereits gezeigt werden, dass es auch Sprachen geben muss, die nicht Turing-erkennbar sind, die also außerhalb der in Abbildung 6.1 dargestellten Sprachhierarchie liegen.

Satz 6.7
Es gibt Sprachen, die nicht Turing-erkennbar sind.

Beweis:
Ein Existenzbeweis für Sprachen, die nicht Turing-erkennbar sind, wird über die Mächtigkeiten der Menge der Turingmaschinen und der Menge der Sprachen geführt.

Sei Σ das zugrunde liegende Alphabet. Es ist endlich. Die Menge aller Wörter Σ^* ist dann abzählbar unendlich. Eine Sprache L ist eine Teilmenge von Σ^* oder auch ein Element von $\mathcal{P}(\Sigma^*)$. Die Potenzmenge einer abzählbar unendlichen Menge ergibt eine überabzählbare Menge, d. h. es gibt überabzählbar viele Sprachen über Σ.

Betrachte nun die Menge aller Turingmaschinen mit dem Eingabealphabet Σ. Da Turingmaschinen über einem endlichen Alphabet in endlich vielen Zeichen beschrieben werden, lassen sie sich in Wortordnung anordnen, und somit hat die Menge aller Turingmaschinen eine abzählbar unendliche Mächtigkeit. Damit gibt es auch nur abzählbar unendlich viele Sprachen, die Turing-erkennbar sind.

Eine überabzählbare Menge ist echt mächtiger als eine abzählbar unendliche Menge. Also muss es Sprachen geben, die nicht Turing-erkennbar sind. □

Nun zurück zum Halteproblem und dem Nachweis seiner Unentscheidbarkeit. Mit Definition 6.7 wird das Halteproblem analog zu 6.2 und 6.3 in ein Wortproblem überführt.

Definition 6.7
$L_{TM} := \{\langle T, w\rangle \mid T \text{ ist eine Turingmaschine und } w \in L(T)\}$.

Wäre das Halteproblem entscheidbar, dann wäre auch die Sprache L_{TM} entscheidbar: Hält T angesetzt auf w akzeptierend oder verwerfend, dann wird $\langle T, w\rangle$ akzeptiert bzw. verworfen. Hält aber T angesetzt auf w nicht, dann könnte das mit dem Algorithmus erkannt werden, der das Halteproblem für T und w entscheidet, und $\langle T, w\rangle$ wird verworfen.

Kann nun aber gezeigt werden, dass L_{TM} nicht entscheidbar ist, dann folgt durch

Kontraposition der vorangehenden Aussage, dass auch das Halteproblem nicht entscheidbar ist. Vorab wird aber noch Turing-Erkennbarkeit von L_{TM} gezeigt.

Satz 6.8
L_{TM} ist eine Turing-erkennbare Sprache.

Beweis:
Zum Beweis wird eine Turingmaschine U spezifiziert, die L_{TM} erkennt.

Informelle Beschreibung von U:

1. Analysiere die Eingabe $\langle T, w\rangle$ dahingehend, ob T eine Turingmaschine ist und w ein Wort. Ist das nicht der Fall, so halte verwerfend.
2. Simuliere T mit der Eingabe w.
3. Falls T in den akzeptierenden oder verwerfenden Zustand gelangt, dann halte akzeptierend bzw. verwerfend.

Es ist zu bemerken, dass U nicht hält, wenn T angesetzt auf w nicht hält. U hält aber akzeptierend, wenn T angesetzt auf w akzeptierend hält. Also erkennt U die Sprache L_{TM}. □

Die hier informell beschriebene Turingmaschine U wird auch *universelle Turingmaschine* genannt. Sie kann jede andere Turingmaschine simulieren und kann als eine Formalisierung eines programm-gesteuerten Computers angesehen werden.

Satz 6.9
L_{TM} ist eine unentscheidbare Sprache.

Beweis:
Der Beweis wird als ein Widerspruchsbeweis geführt. Sei also angenommen L_{TM} sei entscheidbar und H ein Entscheider für L_{TM}. H ist somit eine Turingmaschine, die die folgende Funktion berechnet:

$$H\left(\langle T, w\rangle\right) = \begin{cases} accept & : \ T \text{ akzeptiert } w \\ reject & : \ T \text{ akzeptiert } w \text{ nicht.} \end{cases}$$

Es werde nun eine weitere Turingmaschine D konstruiert, die H quasi als Unterprogramm aufruft. H soll dann entscheiden, ob eine Turingmaschine T angesetzt auf ihre eigene Beschreibung als Eingabe hält.

Informelle Beschreibung von D:

1. Analysiere die Eingabe $\langle T\rangle$ dahingehend, ob T eine Turingmaschine ist. Ist das nicht der Fall, so halte verwerfend.
2. Simuliere H mit der Eingabe $\langle T, \langle T\rangle\rangle$.
3. Falls H akzeptiert, halte verwerfend, falls H verwirft, halte akzeptierend.

D liefert also immer die gegenteilige Ausgabe von H:

$$D\left(\langle T\rangle\right) = \begin{cases} accept & : \ T \text{ akzeptiert } \langle T\rangle \text{ nicht} \\ reject & : \ T \text{ akzeptiert } \langle T\rangle. \end{cases}$$

D kann nun beliebige Turingmaschinen als Eingabe haben, also auch sich selbst. Was liefert dann aber D als Ausgabe?

$$D(\langle D\rangle) = \begin{cases} accept & : \quad D \text{ akzeptiert } \langle D\rangle \text{ nicht} \\ reject & : \quad D \text{ akzeptiert } \langle D\rangle. \end{cases}$$

D gibt immer das Gegenteil von dem aus, was sie eigentlich ausgeben müßte. Das ergibt einen Widerspruch.

Für den Entscheider H mit der Eingabe $\langle D, \langle D\rangle\rangle$ ergeben sich somit die folgenden zwei Fälle:

- H entscheidet $\langle D, \langle D\rangle\rangle \in L_{TM}$, falls D die Eingabe $\langle D\rangle$ nicht akzeptiert, d.h. aber $\langle D, \langle D\rangle\rangle \notin L_{TM}$. Widerspruch.
- H entscheidet $\langle D, \langle D\rangle\rangle \notin L_{TM}$, falls D die Eingabe $\langle D\rangle$ akzeptiert, d.h. aber $\langle D, \langle D\rangle\rangle \in L_{TM}$. Widerspruch.

Weitere Möglichkeiten gibt es nicht. Damit kann also nur die eingangs gemachte Annahme, es existiere ein Entscheider für L_{TM}, falsch sein. Also ist L_{TM} unentscheidbar. □

Warum wird diese Beweistechnik nun Diagonalisierung genannt? Im Beweis von Satz 6.7 war die Abzählbarkeit von Turingmaschinen ausgeführt. In einer zweidimensionalen unendlichen Tabelle lassen sich somit alle Paare $\langle T_i, \langle T_j\rangle\rangle$ von Turingmaschinen (vertikal) und ihren Beschreibungen als Eingabe (horizontal) auflisten (Tabelle 6.1). Der Eintrag in Zeile i und Spalte j spezifiziert, ob $\langle T_i, \langle T_j\rangle\rangle \in L_{TM}$, d.h. die Tabelle liefert die Entscheidung von H. In der Diagonalen treten dabei genau die Entscheidungen für den Fall auf, daß eine Turingmaschine auf ihre eigene Beschreibung als Eingabe angesetzt wird.

	$\langle T_1\rangle$	$\langle T_2\rangle$	$\langle T_3\rangle$	$\langle T_4\rangle$	$\langle T_5\rangle$	$\cdots$
T_1	***reject***	*accept*	*accept*	*reject*	*reject*	
T_2	*accept*	***reject***	*accept*	*reject*	*accept*	
T_3	*accept*	*accept*	***accept***	*accept*	*accept*	
T_4	*accept*	*accept*	*reject*	***reject***	*reject*	
T_5	*reject*	*accept*	*reject*	*reject*	***accept***	
$\vdots$						

Tabelle 6.1: Akzeptierungstabelle mit beispielhaften Einträgen

D ist eine Turingmaschine, also muss sie auch in der Tabelle in irgendeiner Zeile auftreten. Sei das Zeile k, d.h. $D = T_k$. Nun liefert aber D genau das gegenteilige Ergebnis von Eintrag (k, k) der Tabelle. Dieser Widerspruch kann nur dadurch aufgelöst werden, dass es die Turingmaschine D und damit die Turingmaschine H nicht geben kann. D wurde genau so konstruiert, dass der Widerspruch in der Diagonalen auftritt.

Mit Satz 6.7 war die Existenz nicht Turing-erkennbarer Sprachen gezeigt worden. Abschließend für dieses Kapitel soll nun explizit eine solche Sprache angegeben werden. Dazu wird das Komplement einer Turing-erkennbaren Sprache benötigt.

Definition 6.8
Eine Sprache heißt co-Turing-erkennbar, wenn ihr Komplement Turing-erkennbar ist.

Satz 6.10
Eine Sprache ist genau dann entscheidbar, wenn sie Turing-erkennbar und co-Turing-erkennbar ist.

Beweis:

I. Sei L eine entscheidbare Sprache:

Jede entscheidbare Sprache ist auch Turing-erkennbar (Satz 5.1).

Mit L ist auch $\overline{L}$ entscheidbar: Der Entscheider für $\overline{L}$ ergibt sich aus dem Entscheider für L durch die Vertauschung von akzeptierendem und verwerfendem Zustand. Damit ist auch $\overline{L}$ Turing-erkennbar.

II. Sei L eine Turing-erkennbare und co-Turing-erkennbare Sprache:

Dann gibt es einen Erkenner T_1 für L und einen Erkenner T_2 für $\overline{L}$. Die Turingmaschine T entscheidet dann L.

Informelle Beschreibung von T:

1. Analysiere die Eingabe w dahingehend, ob w ein Wort über dem L zugrunde liegenden Alphabet ist. Ist das nicht der Fall, so halte verwerfend.
2. Simuliere abwechselnd T_1 und T_2 jeweils mit der Eingabe w, einen Berechnungsschritt T_1, dann einen Berechnungsschritt T_2.
3. Falls T_1 akzeptiert, halte akzeptierend, falls T_2 akzeptiert, halte verwerfend.

T läuft nun solange bis entweder T_1 oder T_2 akzeptiert. Einer von beiden Turingmaschinen muss akzeptieren, weil w entweder in L ist oder in $\overline{L}$. T entscheidet also die Sprache L.

□

Satz 6.11
$\overline{L_{TM}}$ ist eine nicht Turing-erkennbare Sprache.

Beweis:
L_{TM} ist Turing-erkennbar (Satz 6.8). Wäre auch $\overline{L_{TM}}$ Turing-erkennbar, dann wäre L_{TM} auch entscheidbar (Satz 6.10). L_{TM} ist aber eine unentscheidbare Sprache (Satz 6.9). Also ist $\overline{L_{TM}}$ nicht Turing-erkennbar. □

Damit ist auch die in Satz 6.5 angesprochene Hierarchie von Sprachfamilien vollständig gezeigt. Mit $L_{TM} \notin \mathcal{L}_{ent}$ aber $L_{TM} \in \mathcal{L}_{erk}$ ergibt sich die echte Teilmengen-Beziehung $\mathcal{L}_{ent} \subset \mathcal{L}_{erk}$. Darüber hinaus ist mit $\overline{L_{TM}} \notin \mathcal{L}_{erk}$ eine Sprache spezifiziert, die außerhalb dieser Hierarchie liegt.

Literaturverzeichnis

[1] Asteroth, A., Baier, Ch.: Theoretische Informatik.
Pearson Studium, München 2002. ISBN 3-8273-7033-7

[2] Hopcroft, J. E., Motwani, R., Ullman, J. D.:
Introduction to Automata Theory, Languages, and Computation.
Addison-Wesley, Boston 2001. ISBN 0-201-44124-1

[3] Hromkovič, J.: Theoretische Informatik.
Teubner Verlag, Wiesbaden 2004. ISBN 3-519-10332-X

[4] Sipser, M.: Introduction to the Theory of Computation. Third Edition.
Cengage Learning, Boston 2013. ISBN 978-1-133-18781-3

[5] Socher, R.: Grundkurs Theoretische Informatik.
Fachbuchverlag, Leipzig 2002. ISBN 3-446-22177-8

[6] Vossen, G., Witt, K.-U.: Grundkurs Theoretische Informatik.
Vieweg Verlag, Wiesbaden 2006. ISBN 3-8348-0153-4

Index