Theoretische Informatik

Vorlesungsscriptum Sommersemester 2003

Dr. Bernd Reichel[1] und Dr. Ralf Stiebe[2]

Fakultät für Informatik

Otto-von-Guericke-Universität Magdeburg

[1]Tel.: +49'391'67'12851, e-mail: reichel@iws.cs.uni-magdeburg.de, URL: theo.cs.uni-magdeburg.de
[2]Tel.: +49'391'67'12457, e-mail: stiebe@iws.cs.uni-magdeburg.de, URL: theo.cs.uni-magdeburg.de

Inhaltsverzeichnis

Vorwort **3**

Literatur **4**

1 Mathematische Grundlagen **5**
1.1 Elementare Aussagenlogik . . . 5
1.2 Elementare Mengenlehre . . . 9
1.3 Relationen und Funktionen . . . 13
1.4 Über die Mächtigkeit von Mengen . . . 17
1.5 Algebraische Strukturen . . . 18
1.6 Alphabete, Wörter, Sprachen . . . 19

2 Berechenbarkeit **21**
2.1 Intuitiver Berechenbarkeitsbegriff . . . 21
2.2 Turing-Berechenbarkeit . . . 22
2.3 LOOP-, WHILE- und GOTO-Berechenbarkeit . . . 28
2.4 Die Churchsche These . . . 37
2.5 Halteproblem und Unentscheidbarkeit . . . 38

3 Komplexitätstheorie **44**
3.1 Komplexitätsklassen . . . 44
3.2 Das Domino-Problem . . . 45
3.3 Nichtdeterministische Turingmaschinen und die Klasse $\mathbb{NP}$ . . . 46
3.4 $\mathbb{NP}$-Vollständigkeit . . . 49

4 Formale Sprachen **54**
4.1 Einführung . . . 54
4.2 Grammatiken . . . 55
4.2.1 Chomsky-Hierarchie . . . 59
4.2.2 Wortproblem . . . 61
4.2.3 Syntaxbäume . . . 63
4.3 Reguläre Sprachen . . . 64
4.3.1 Endliche Automaten . . . 64
4.3.2 Nichtdeterministische endliche Automaten . . . 68
4.3.3 Reguläre Ausdrücke . . . 74
4.3.4 Das Pumping Lemma . . . 81
4.3.5 Abschlusseigenschaften . . . 83
4.3.6 Wortproblem und andere Entscheidbarkeitsprobleme . . . 83
4.4 Kontextfreie Sprachen . . . 85
4.4.1 Normalformen . . . 85
4.4.2 Das Pumping Lemma . . . 87
4.4.3 Abschlusseigenschaften . . . 88
4.4.4 Entscheidbarkeit und der Algorithmus von Cocke, Younger und Kasami . . . 89
4.4.5 Kellerautomaten . . . 91
4.5 Rekursiv aufzählbare und kontextabhängige Sprachen . . . 95
4.6 Tabellarischer Überblick . . . 96

Vorwort

Das vorliegende Scriptum soll dem Leser eine Hilfe zum Studium und beim Besuch der gleichnamigen Vorlesung geben. Es orientiert sich an [13], aus dem auch wesentliche Teile übernommen wurden.

Für weitergehende oder vertiefende Studien ist [8] sehr geeignet und zu empfehlen. Auch die Lehrbücher [7, 5, 9, 14, 16, 17, 18, 2] sind zu empfehlen.

Das Scriptum ist (wie jedes solches Werk) unfertig und wird ständig weiterbearbeitet. Deshalb nehmen die Autoren Anregungen und Meinungen nicht nur gern entgegen, sondern wünschen sich von allen Lesern solche ausdrücklich.

Bernd Reichel und Ralf Stiebe Magdeburg, im April 2003

Literatur

[1] J. Albert und Th. Ottmann. *Automaten, Sprachen und Maschinen für Anwender*. Bibliographisches Institut, Mannheim, 1983.

[2] A. Asteroth und Ch. Baier. *Theoretische Informatik*. Pearson Studium, München, 2002.

[3] W. Brauer. *Automatentheorie*. B. G. Teubner, Stuttgart, 1984.

[4] W. Brecht. *Theoretische Informatik – Grundlagen und praktische Anwendungen*. Vieweg, Braunschweig, 1995.

[5] W. Bucher und H. Maurer. *Theoretische Grundlagen der Programmiersprachen*. B.I.-Wissenschaftsverlag, Mannheim, 1984.

[6] M. R. Garey und D. S. Johnson. *Computers and Intractability*. Freeman, New York 1979.

[7] J. E. Hopcroft und J. D. Ullman. *Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie*. Addison Wesley, Reading, Bonn, 1990.

[8] J. E. Hopcroft, R. Motwani und J. D. Ullman. *Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie*. Pearson Studium, München, 2002.

[9] H. L. Lewis und C. H. Papadimitriou. *Elements of the Theory of Computation*. Second edition, Prentice-Hall, New Jersey, 1988.

[10] Ch. Posthoff und K. Schulz. *Grundkurs Theoretische Informatik*. B. G. Teubner, Stuttgart, 1992.

[11] A. Salomaa. *Formale Sprachen*. Springer-Verlag, Berlin, 1978.

[12] U. Schöning. *Logik für Informatiker*. Spektrum Akademischer Verlag, Heidelberg, 1995.

[13] U. Schöning. *Theoretische Informatik kurz gefaßt*. Spektrum Akademischer Verlag, Heidelberg, 1995.

[14] V. Sperschneider und B. Hammer. *Theoretische Informatik – Eine problemorientierte Einführung*. Springer-Verlag, Berlin, Heidelberg, New York, 1996.

[15] Th. A. Sudkamp. *Languages and Machines*. Addison-Wesley, Reading, 1988.

[16] K. Wagner. *Einführung in die Theoretische Informatik*. Springer-Verlag, Berlin, 1994.

[17] D. Wätjen. *Theoretische Informatik*. Oldenbourg, München, 1994.

[18] I. Wegener. *Theoretische Informatik*. B. G. Teubner, Stuttgart, 1993.

[19] D. Wood. *Theory of Computation*. John Wiley & Sons, New York, 1987.

1 Mathematische Grundlagen

Wir geben hier einige mathematische Grundbegriffe, die eigentlich aus der Mathematik bekannt sind. Deshalb können die Kapitel 1.1–1.5 problemlos übersprungen werden.

Im Kapitel 1.6 werden Begriffe bereitgestellt, die im Scriptum verwendet werden.

1.1 Elementare Aussagenlogik

Bevor wir zu einigen Grundaussagen der elementaren Aussagenlogik kommen, führen wir hier den Begriff der Menge ein.

Definition 1.1 *Jede Zusammenfassung von bestimmten, wohlunterschiedenen Objekten zu einem Ganzen wird Menge genannt. Die so zusammengefassten Objekte heißen Elemente der Menge.*

In der Regel bezeichen wir Mengen mit großen lateinischen Buchstaben und Elemente mit kleinen lateinischen Buchstaben. Es folgen einige Beispiele von Mengen.

Beispiel 1.2 (i) M_1 sei die Menge der Vokale im deutschen Alphabet.

(ii) M_2 ist die Menge der natürlichen Zahlen, die kleiner als 10 sind.

(iii) M_3 ist die Menge der ganzen Zahlen, die Lösung der Gleichung $2x^2 + 3x - 2 = 0$ sind.

Die Zugehörigkeit eines Objektes zu einer Menge wird durch das Element-Zeichen „$\in$“ beschrieben. Entsprechend besagt „$\notin$“, dass ein Objekt nicht Element der Menge ist. So schreibt man zum Beispiel:

$$\begin{array}{lll} a \in M_1 & 5 \in M_2 & -2 \in M_3 \\ s \notin M_1 & 10 \notin M_2 & \frac{1}{2} \notin M_3 \end{array}$$

Sprechweisen sind z. B.: „a ist Element von M_1“, „s ist nicht Element von M_1“ oder „a ist kein Element von M_1“.

Im Folgenden bezeichnen wir die Menge der natürlichen Zahlen, die Menge der ganzen Zahlen, die Menge der rationalen Zahlen und die Menge der reellen Zahlen mit $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$ bzw. $\mathbb{R}$.

Wir wollen jetzt einige Grundbegriffe der Aussagenlogik zusammenstellen, insbesondere auch eine Symbolik einführen, die es uns erleichtert, umgangssprachlich kompliziertere Wendungen sauber und präzise aufzuschreiben.

Definition 1.3 *Unter einer Aussage verstehen wir ein sprachliches Gebilde, das die Eigenschaft hat, eindeutig entweder wahr oder falsch zu sein oder – wie wir auch sagen – genau einen der Wahrheitswerte „wahr“ $(W, 1)$ oder „falsch“ $(F, 0)$ zu haben.*

Beispiel 1.4 Wir geben hier einige Beispiele für Aussagen:

(i) 15 ist eine Primzahl.

(ii) Zu keiner natürlichen Zahl n mit $n > 2$ lassen sich drei positive ganze Zahlen x, y, z angeben, dass $x^n + y^n = z^n$ ist.

(iii) $\int_0^\pi \sin x \, \mathrm{d}x = 2.$

(iv) Es gibt unendlich viele Primzahlzwillinge.

Die Aussagen 2 und 3 sind wahr, die Aussage 1 ist falsch, von der Aussage 4 wissen wir leider nicht, ob sie wahr oder falsch ist. Aus der Aussage 3 erkennen wir, dass wir natürlich bei der Formulierung von Aussagen eine mathematische Formelsprache oder andere eindeutige Vereinbarungen zulassen.

Beispiel 1.5 Wir wollen auch einige Sätze angeben, die leicht erkennbar keine Aussagen sind, da man ihnen nicht eindeutig genau einen Wahrheitswert zuordnen kann:

(i) x ist eine Primzahl.

(ii) Heute ist Dienstag.

Betrachtet man im Beispiel 1.5 den Satz 1, so erkennt man, dass es sich zwar um keine Aussage handelt, aber wenn man die Variable x durch konkrete Werte ersetzt, so erhält man eine Aussage. Damit kommen wir zur nächsten Definition.

Definition 1.6 *Eine Aussageform hat die Gestalt einer Aussage, in der eine oder mehrere Variable auftreten und besitzt die Eigenschaft, dass man jedesmal eine Aussage erhält, wenn man für diese Variablen beliebige Elemente eines Variablengrundbereiches einsetzt. Wir bezeichnen eine Aussageform mit den Variablen $x_1, x_2, \ldots, x_n$ mit $p(x_1, x_2, \ldots, x_n)$.*

Also handelt es sich im Beispiel 1.5 bei dem Satz 1 mit der Variablen x z. B. aus dem Grundbereich $\mathbb{N}$ um eine Aussageform. Sie wird wahr, wenn wir z. B. für x die Zahlen $2, 3, 31$ oder $2\,147\,483\,647$ einsetzen. Sie wird falsch, wenn wir z. B. für x die Zahlen $1, 4, 18$, oder $89\,687\,671\,441$ einsetzen.

Durch Aussageverbindungen können wir aus gegebenen Aussagen neue Aussagen konstruieren.

Definition 1.7 *Sei p eine Aussage, so ist $\neg p$ wiederum eine Aussage, deren Wahrheitswert genau der entgegengesetzte von p ist. Die einstellige Aussageverbindung $\neg p$ heißt Negation von p, gesprochen: „nicht p“ oder „non p“.*

Wir können die Definition durch folgende sogenannte *Wahrheitstabelle* veranschaulichen.

p	$\neg p$
1	0
0	1

Definition 1.8 *Seien p und q Aussagen, so führen wir folgende zweistellige Aussageverbindungen ein.*

(i) *$p \wedge q$ heißt Konjunktion von p und q und besitzt genau dann den Wahrheitswert 1, wenn sowohl p als auch q den Wahrheitswert 1 besitzen. Gesprochen: „p und q“.*

(ii) *$p \vee q$ heißt Alternative oder Disjunktion von p und q und besitzt genau dann den Wahrheitswert 0, wenn sowohl p als auch q den Wahrheitswert 0 besitzen. Gesprochen: „p oder q“.*

(iii) *$p \Rightarrow q$ heißt Implikation von p und q und besitzt genau dann den Wahrheitswert 0, wenn p den Wahrheitswert 1 und q den Wahrheitswert 0 besitzen. Gesprochen: „p impliziert q“ oder aus „p folgt q“ oder auch „Wenn p, so q“.*

(iv) *$p \Leftrightarrow q$ heißt Äquivalenz von p und q und besitzt genau dann den Wahrheitswert 1, wenn p und q den gleichen Wahrheitswert besitzen. Gesprochen: „p genau dann, wenn q“.*

Wir können die Definitionen wiederum durch folgende Wahrheitstabellen verdeutlichen.

p	q	$p \wedge q$	$p \vee q$	$p \Rightarrow q$	$p \Leftrightarrow q$
1	1	1	1	1	1
1	0	0	1	0	0
0	1	0	1	1	0
0	0	0	0	1	1

Entsprechend den Aussageverbindungen können wir auch von Aussageformen Verbindungen bilden, deren Variable durch Elemente ein und desselben Variablengrundbereichs ersetzt werden dürfen.

In der Menge der Aussageverbindungen definieren wir:

Definition 1.9 *Zwei Aussageverbindungen p und q heißen logisch äquivalent, wenn sie unabhängig von der Belegung der Einzelaussagen stets den gleichen Wahrheitswert liefern. In Zeichen: $p \equiv q$.*

Es folgen ohne Wertigkeit der Reihenfolge und ohne Anspruch auf Vollständigkeit einige logische Äquivalenzen von Aussageverbindungen.

Satz 1.10 *Seien p, q und r Aussagen, dann gelten folgende logische Äquivalenzen.*

$$\neg(\neg p) \equiv p \quad (\textit{Doppelte Negation}), \tag{1.1}$$
$$p \wedge q \equiv q \wedge p \quad (\textit{Kommutativität von } \wedge), \tag{1.2}$$
$$p \vee q \equiv q \vee p \quad (\textit{Kommutativität von } \vee), \tag{1.3}$$
$$(p \wedge q) \wedge r \equiv p \wedge (q \wedge r) \quad (\textit{Assoziativität von } \wedge), \tag{1.4}$$
$$(p \vee q) \vee r \equiv p \vee (q \vee r) \quad (\textit{Assoziativität von } \vee), \tag{1.5}$$
$$p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \wedge r) \quad (\textit{Distributivität von } \wedge \textit{ bezüglich } \vee), \tag{1.6}$$
$$p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r) \quad (\textit{Distributivität von } \vee \textit{ bezüglich } \wedge), \tag{1.7}$$
$$p \Rightarrow q \equiv \neg p \vee q \quad (\textit{Darstellung der Implikation durch eine Alternative}), \tag{1.8}$$
$$p \Rightarrow q \equiv \neg q \Rightarrow \neg p \quad (\textit{Kontraposition}), \tag{1.9}$$
$$p \Leftrightarrow q \equiv (p \Rightarrow q) \wedge (q \Rightarrow p), \tag{1.10}$$
$$\neg(p \wedge q) \equiv \neg p \vee \neg q \quad (\textit{De Morgansche Regel}), \tag{1.11}$$
$$\neg(p \vee q) \equiv \neg p \wedge \neg q \quad (\textit{De Morgansche Regel}), \tag{1.12}$$
$$p \wedge (q \vee \neg q) \equiv p, \tag{1.13}$$
$$p \vee (q \wedge \neg q) \equiv p. \tag{1.14}$$

Man kann logische Äquivalenzen (also auch die im Satz 1.10) über Wahrheitstabellen oder durch die Benutzung bereits bewiesener logischer Äquivalenzen beweisen. Wir geben je ein Beispiel:

Wir beweisen die logische Äquivalenz (1.8) durch eine Wahrheitstabelle. Wir gehen also einfach alle Fälle der Belegungen der Einzelaussagen durch und zeigen, in jedem Fall liefern die beiden Aussageverbindungen $p \Rightarrow q$ und $\neg p \vee q$ den gleichen Wahrheitswert:

p	q	$\boldsymbol{p \Rightarrow q}$	$\neg p$	$\boldsymbol{\neg p \vee q}$
1	1	**1**	0	**1**
1	0	**0**	0	**0**
0	1	**1**	1	**1**
0	0	**1**	1	**1**

Wir führen jetzt den Beweis für die Aussage (1.9) durch die Benutzung bekannter logischer Äquivalenzen, wobei wir annehmen, die logischen Äquivalenzen (1.1), (1.3) und (1.8). seien schon bewiesen:

$$\begin{aligned} p \Rightarrow q &\equiv \neg p \vee q && \text{nach (1.8)},\\ &\equiv q \vee \neg p && \text{nach (1.3)},\\ &\equiv \neg(\neg q) \vee \neg p && \text{nach (1.1)},\\ &\equiv \neg q \Rightarrow \neg p && \text{nach (1.8)}. \end{aligned}$$

Schließlich können wir aufgrund der Transitivität der logischen Äquivalenz $p \Rightarrow q \equiv \neg q \Rightarrow \neg p$ schließen, womit die Gleichung (1.9) bewiesen ist.

Definition 1.11 *Eine Aussageverbindung heißt Tautologie, wenn sie unabhängig von der Belegung der Einzelaussagen stets den Wahrheitswert* 1 *besitzt.*

Definition 1.12 *Eine Aussageverbindung heißt Kontradiktion, wenn sie unabhängig von der Belegung der Einzelaussagen stets den Wahrheitswert* 0 *besitzt.*

Satz 1.13 *Es seien p und q Aussagen. Folgende Aussageverbindungen sind Tautologien:*

$$p \vee \neg p \quad (\textit{Satz vom ausgeschlossenen Dritten}), \tag{1.15}$$
$$(p \Rightarrow \neg p) \Rightarrow \neg p, \tag{1.16}$$
$$(p \wedge (p \Rightarrow q)) \Rightarrow q, \tag{1.17}$$
$$(p \wedge q) \Rightarrow p, \tag{1.18}$$
$$p \Rightarrow (p \vee q), \tag{1.19}$$
$$(p \Rightarrow q) \Rightarrow ((q \Rightarrow r) \Rightarrow (p \Rightarrow r)), \tag{1.20}$$
$$(((p \Rightarrow q) \wedge (q \Rightarrow r)) \wedge p) \Rightarrow r. \tag{1.21}$$

Man kann Tautologien (also auch die im Satz 1.13) über Wahrheitstabellen oder aber durch Benutzung von logischen Äquivalenzen beweisen. Beispiele seien dem Leser überlassen.

Wir nennen hier noch einen Zusammnhang zwischen den Begriffen *Tautologie* und *logischer Äquivalenz*:

Satz 1.14 *Es seien $\alpha(p_1, p_2, \ldots, p_n)$ und $\beta(p_1, p_2, \ldots, p_n)$ Aussageverbindungen über den Einzelaussagen $p_1, p_2, \ldots, p_n$. Dann gilt*

$$\alpha(p_1, p_2, \ldots, p_n) \equiv \beta(p_1, p_2, \ldots, p_n)$$

genau dann, wenn

$$\alpha(p_1, p_2, \ldots, p_n) \Leftrightarrow \beta(p_1, p_2, \ldots, p_n)$$

eine Tautologie ist.

Wir haben oben eine Möglichkeit kennengelernt, aus Aussageformen Aussagen zu machen, nämlich durch das Belegen der Variablen mit Objekten aus dem Variablengrundbereich. Eine weitere Möglichkeit besteht in der *Quantifizierung* von Variablen. Wir führen den *Allquantor* $\forall$ („Für alle ...“) und den *Existenzquantor* $\exists$ („es gibt ein ...“) ein, die manchmal auch *Generalisator* bzw. *Partikularisator* genannt werden.

Definition 1.15 *Es sei $p(x)$ eine Aussageform mit der Variablen x. Dann ist $\forall x \in G\ (p(x))$ eine Aussage und ist wahr genau dann, wenn $p(x)$ mit jeder Belegung $x \in G$ zu einer wahren Aussage wird.*

Definition 1.16 *Es sei $p(x)$ eine Aussageform mit der Variablen x. Dann ist $\exists x \in G\ (p(x))$ eine Aussage und ist wahr genau dann, wenn es eine Belegung $x \in G$ gibt, die $p(x)$ zu einer wahren Aussage macht.*

Oft wird bei Quantoren der Variablengrundbereich G nicht mit genannt, falls er aus dem Kontext zweifelsfrei ersichtlich ist. Wir benutzen also $\forall x\ (p(x))$ bzw. $\exists x\ (p(x))$ statt $\forall x \in G\ (p(x))$ und $\exists x \in G\ (p(x))$.

Für Quantifizierungen mehrstelliger Aussageformen verzichten wir auf die Klammern um die jeweilige äußere Aussageform und schreiben:

$$\forall x\ \forall y\ (p(x,y)) \ \text{ statt } \ \forall x\ (\forall y\ (p(x,y))),$$
$$\exists x\ \exists y\ (p(x,y)) \ \text{ statt } \ \exists x\ (\exists y\ (p(x,y))),$$
$$\forall x\ \exists y\ (p(x,y)) \ \text{ statt } \ \forall x\ (\exists y\ (p(x,y))),$$
$$\exists x\ \forall y\ (p(x,y)) \ \text{ statt } \ \exists x\ (\forall y\ (p(x,y))).$$

Es gilt folgender Satz.

Satz 1.17 *Es seien $p(x)$ und $p(x,y)$ Aussageformen mit den Variablen x und y. Dann gilt*

$$\neg\forall x \in G\ (p(x)) \equiv \exists x \in G\ (\neg p(x)), \tag{1.22}$$
$$\neg\exists x \in G\ (p(x)) \equiv \forall x \in G\ (\neg p(x)), \tag{1.23}$$
$$\forall x \in G_1\ \forall y \in G_2\ (p(x,y)) \equiv \forall y \in G_2\ \forall x \in G_1\ (p(x,y)), \tag{1.24}$$
$$\exists x \in G_1\ \exists y \in G_2\ (p(x,y)) \equiv \exists y \in G_2\ \exists x \in G_1\ (p(x,y)), \tag{1.25}$$
$$\exists x \in G_1\ \forall y \in G_2\ (p(x,y)) \Rightarrow \forall y \in G_2\ \exists x \in G_1\ (p(x,y)). \tag{1.26}$$

1.2 Elementare Mengenlehre

Im Kapitel 1.1 wurden bereits die Begriffe *Menge* und *Element* einer Menge eingeführt. Wir wollen uns in diesem Kapitel mit einigen Aspekten der elementaren Mangenlehre beschäftigen, insbesondere mit Operationen auf Mengen.

Aber zuerst zu möglichen Beschreibungsarten von Mengen. Im Beispiel 1.2 haben wir bereits die verbale Beschreibung kennengelernt. Oft ist sie unzweckmäßig und unübersichtlich. Manchmal ist es günstiger, die Menge durch die Aufzählung aller ihrer Elemente anzugeben:

$$M_4 = \{a, e, i, o, u\},$$
$$M_5 = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}.$$

Die Aufzählung aller Elemente ist wiederum ungünstig, wenn es sehr viele sind und gar nicht möglich, wenn es unendlich viele Elemente sind (man nennt eine Menge mit unendlich vielen Elementen *unendlich*, sonst *endlich*). In diesem Fall ist die Beschreibung der Menge durch eine definierende Eigenschaft angebracht:

$$M_6 = \{x \in \mathbb{N} \mid x^2 < 400\},$$
$$M_7 = \{x \in \mathbb{R} \mid 0 \le x \le 1\}.$$

gelesen als: „M_6 ist die Menge aller natürlichen Zahlen x für die $x^2 < 400$ gilt“. Mathematisch gesehen handelt es sich um die allgemeine Form

$$M = \{x \in G \mid H(x)\},$$

wobei G ein gewisser Variablengrundbereich ist und die „definierende Eigenschaft“ $H(x)$ eine Aussageform mit der Variablen x.

Es kann vorkommen, dass die Aussageform $H(x)$ durch kein Element x aus dem Variablengrundbereich G zu einer wahren Aussage gemacht wird, so dass wir von der *leeren Menge* sprechen, also von der Menge, die kein Element enthält. Sie wird mit dem Symbol $\emptyset$ bezeichnet.

Eine Bemerkung: die Menge $\{\emptyset\}$ ist nicht die leere Menge, da sie ein Element enthält, nämlich die leere Menge.

Definition 1.18 *Mit $|M|$ bezeichnen wir die Kardinalzahl einer Menge M.*

Für endliche Mengen M ist die Kardinalzahl $|M|$ die Anzahl ihrer Elemente. Insbesondere gilt $|\emptyset| = 0$. Für unendliche Mengen weise ich darauf hin, dass $|\mathbb{N}| \neq |\mathbb{R}|$ gilt. Wir kommen später darauf zurück.

Jetzt kommen einige wichtige Definitionen für Beziehungen zwischen Mengen.

Definition 1.19 *Zwei Mengen M_1 und M_2 in einem Variablengrundbereich G heißen gleich genau dann (in Zeichen $M_1 = M_2$), wenn*

$$\forall x \in G \ (x \in M_1 \Leftrightarrow x \in M_2)$$

gilt.

Definition 1.20 *Eine Menge M_1 heißt Teilmenge einer Menge M_2 in einem Variablengrundbereich G (in Zeichen $M_1 \subseteq M_2$), falls*

$$\forall x \in G \ (x \in M_1 \Rightarrow x \in M_2)$$

gilt.

Definition 1.21 *Eine Menge M_1 heißt echte Teilmenge einer Menge M_2 in einem Variablengrundbereich G (in Zeichen $M_1 \subsetneq M_2$), falls*

$$M_1 \subseteq M_2 \quad \textit{und} \quad M_1 \neq M_2$$

gilt.

Definition 1.22 *Sei M eine Menge. Die Menge aller Teilmengen von M heißt Potenzmenge und wird mit 2^M oder auch mit $\mathcal{P}(M)$ bezeichnet.*

Beispiel 1.23 Sei $M = \{1, 2, 3\}$, so ist $2^M = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}$.

Die Bezeichnung 2^M leitet sich von folgender Tatsache ab.

Folgerung 1.24 *Sei M eine endliche Menge mit $|M| = n$, $n \in \mathbb{N}$, so gilt $|2^M| = 2^n$.*

Wir betrachten folgende Operationen auf Mengen, das heißt, wir konstruieren aus gegebenen Mengen neue Mengen.

Definition 1.25 *Es sind die Mengen M_1 und M_2 in einem Grundbereich G gegeben, dann gelte:*

$$\begin{aligned}
\overline{M_1} &:= \{x \in G \mid x \notin M_1\} && (\textit{Komplement von } M_1 \textit{ bez. } G),\\
M_1 \cup M_2 &:= \{x \in G \mid x \in M_1 \vee x \in M_2\} && (\textit{Vereinigung von } M_1 \textit{ und } M_2),\\
M_1 \cap M_2 &:= \{x \in G \mid x \in M_1 \wedge x \in M_2\} && (\textit{Durchschnitt von } M_1 \textit{ und } M_2),\\
M_1 \setminus M_2 &:= \{x \in G \mid x \in M_1 \wedge x \notin M_2\} && (\textit{Differenz von } M_1 \textit{ und } M_2).
\end{aligned}$$

Beispiel 1.26 Es seien der Grundbereich $G = \{x \in \mathbb{N} \mid x < 6\}$ und die Mengen $M_1 = \{1, 2, 3\}$ und $M_2 = \{2, 3, 4\}$ gegeben. Dann ist $\overline{M_1} = \{4, 5, 6\}$, $M_1 \cup M_2 = \{1, 2, 3, 4\}$, $M_1 \cap M_2 = \{2, 3\}$ sowie $M_1 \setminus M_2 = \{1\}$.

Die oben definierten Operationen auf Mengen kann man durch sogenannte VENN-Diagramme veranschaulichen (siehe Abbildung 1.1). Man möge aber beachten, dass solche graphischen Veranschaulichungen nicht geeignet sind, Allaussagen über Mengen zu beweisen. Allerdings kann man mit solchen Diagrammen Existenzaussagen beweisen, da die Darstellungen nämlich Punktmengen in der Ebene verkörpern.

Für die definierten Mengenoperationen können wir folgende Aussagen aufstellen (ohne Anspruch auf Vollständigkeit).

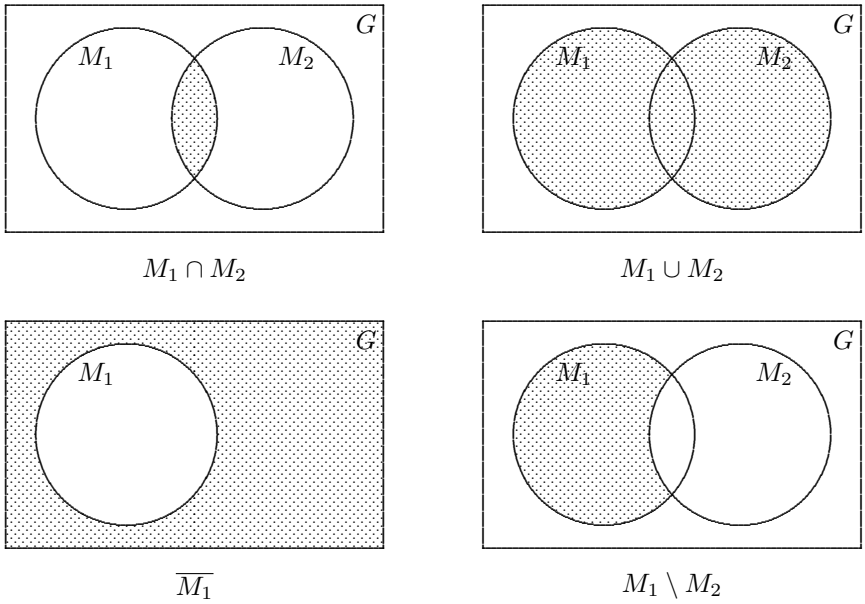


Abbildung 1.1: VENN-Diagramme zur graphischen Darstellung von Operationen auf Mengen

Satz 1.27 *Seien A, B und C Mengen in einem Grundbereich G, dann gelten folgende Aussagen.*

$$(A = B) \Leftrightarrow (A \subseteq B \wedge B \subseteq A), \tag{1.27}$$
$$\overline{\overline{A}} = A, \tag{1.28}$$
$$A \cup B = B \cup A \quad \textit{(Kommutativität bezüglich } \cup\textit{)}, \tag{1.29}$$
$$(A \cup B) \cup C = A \cup (B \cup C) \quad \textit{(Assoziativität bezüglich } \cup\textit{)}, \tag{1.30}$$
$$\emptyset \cup A = A, \tag{1.31}$$
$$A \cup A = A, \tag{1.32}$$
$$(A \subseteq B) \Longrightarrow (A \cup B = B), \tag{1.33}$$
$$A \cap B = B \cap A \quad \textit{(Kommutativität bezüglich } \cap\textit{)}, \tag{1.34}$$
$$(A \cap B) \cap C = A \cap (B \cap C) \quad \textit{(Assoziativität bezüglich } \cap\textit{)}, \tag{1.35}$$
$$\emptyset \cap A = \emptyset, \tag{1.36}$$
$$A \cap A = A, \tag{1.37}$$
$$(A \subseteq B) \Longrightarrow (A \cap B = A), \tag{1.38}$$
$$A \cup (B \cap C) = (A \cup B) \cap (A \cup C) \quad \textit{(Distributivität)}, \tag{1.39}$$
$$A \cap (B \cup C) = (A \cap B) \cup (A \cap C) \quad \textit{(Distributivität)}, \tag{1.40}$$
$$\overline{A \cup B} = \overline{A} \cap \overline{B} \quad \textit{(De Morgansche Regel)}, \tag{1.41}$$
$$\overline{A \cap B} = \overline{A} \cup \overline{B} \quad \textit{(De Morgansche Regel)}, \tag{1.42}$$
$$(A \cap B = \emptyset) \Leftrightarrow (A \subseteq \overline{B}), \tag{1.43}$$
$$(A \cup B = G) \Leftrightarrow (\overline{A} \subseteq B), \tag{1.44}$$
$$A \cap (B \setminus C) = (A \cap B) \setminus (A \cap C) \quad \textit{(Distributivität)}, \tag{1.45}$$
$$(A \cap B) \setminus C = (A \setminus C) \cap (B \setminus C) \quad \textit{(Distributivität)}, \tag{1.46}$$
$$(A \cup B) \setminus C = (A \setminus C) \cup (B \setminus C) \quad \textit{(Distributivität)}, \tag{1.47}$$
$$A \setminus (B \cup C) = (A \setminus B) \cap (A \setminus C) \quad \textit{(Distributivität)}, \tag{1.48}$$
$$A \setminus (B \cap C) = (A \setminus B) \cup (A \setminus C) \quad \textit{(Distributivität)}. \tag{1.49}$$

Bemerkung: $A \cup (B \setminus C) = (A \cup B) \setminus (A \cup C)$ gilt **nicht** allgemein.

Die Sachverhalte in obigem Satz kann man sich an sogenannten *Venn-Diagrammen* verdeutlichen, zum Beweis sind sie allerdings nicht geeignet. Die Beweise aller Aussagen aus obigem Satz werden geführt, indem auf die entsprechenden Gesetzmäßigkeiten der Aussagenlogik zurückgegriffen wird.

Wir wollen hier den Beweis der Aussage 1.49 aus obigem Satz angeben. Zu zeigen ist $A \setminus (B \cap C) = (A \setminus B) \cup (A \setminus C)$. Aufgrund der Definition der Gleichheit haben wir also folgende äquivalente Aussage für alle $x \in G$ zu beweisen:

$$x \in (A \setminus (B \cap C)) \Leftrightarrow x \in ((A \setminus B) \cup (A \setminus C)).$$

Das machen wir folgendermaßen:

$$\begin{aligned} x \in (A \setminus (B \cap C)) &\Leftrightarrow x \in A \wedge x \notin (B \cap C) \\ &\Leftrightarrow x \in A \wedge \neg(x \in (B \cap C)) \\ &\Leftrightarrow x \in A \wedge \neg(x \in B \wedge x \in C) \\ &\Leftrightarrow x \in A \wedge (\neg(x \in B) \vee \neg(x \in C)) \\ &\Leftrightarrow (x \in A \wedge \neg(x \in B)) \vee (x \in A \wedge \neg(x \in C)) \\ &\Leftrightarrow (x \in A \wedge x \notin B) \vee (x \in A \wedge x \notin C) \\ &\Leftrightarrow x \in (A \setminus B) \vee x \in (A \setminus C) \\ &\Leftrightarrow x \in ((A \setminus B) \cup (A \setminus C)) \end{aligned}$$

und schließlich folgt die Ausssage für alle $x \in G$ aus der Transitivität der aussagenlogischen Äquivalenz („$\Leftrightarrow$“).

Eine besondere Verknüpfung bei Mengen ist das sogenannte *Kreuzprodukt* von Mengen oder auch *Kartesisches Produkt* oder einfach *Produkt* von Mengen. Dazu benötigen wir aber noch den Begriff des n-Tupels:

Ein Element $(x_1, x_2, \ldots, x_n)$ heißt *n-Tupel*. Für $n = 2$ heißt es auch (*geordnetes*) *Paar*, für $n = 3$ *Tripel*, für $n = 4$ *Quadrupel*, für $n = 5$ *Quintupel* u. s. w.

Definition 1.28 *Sind M_1 und M_2 zwei Mengen, so ist ihr Kreuzprodukt $M_1 \times M_2$ definiert durch*

$$M_1 \times M_2 := \{(x, y) \mid (x \in M_1) \wedge (y \in M_2)\},$$

das heißt $M_1 \times M_2$ ist eine Menge geordneter Paare.

Das Ganze kann man auf n Mengen verallgemeinern:

Definition 1.29 *Sei $n > 0$ eine natürliche Zahl, $M_1, M_2, \ldots, M_n$ Mengen, so ist ihr Kreuzprodukt $M_1 \times M_2 \times \cdots \times M_n$ definiert durch*

$$M_1 \times M_2 \times \cdots \times M_n := \{(x_1, x_2, \ldots, x_n) \mid x_i \in M_i \text{ für } 1 \leq i \leq n\}.$$

Definition 1.30 *Gilt $M_1 = M_2 = \cdots = M_n = M$, so schreiben wir $M_1 \times M_2 \times \cdots \times M_n = M^n$.*

Ist $n = 0$, so ist $M^n = M^0 = \{\emptyset\}$.

Satz 1.31 *Seien A, B und C Mengen in einem Grundbereich G, dann gelten folgende Aussagen.*

$$A \times (B \times C) = (A \times B) \times C \quad \textit{(Assoziativität)}, \tag{1.50}$$

$$A \times (B \cup C) = (A \times B) \cup (A \times C) \quad \textit{(Distributivität)}, \tag{1.51}$$

$$A \times (B \cap C) = (A \times B) \cap (A \times C) \quad \textit{(Distributivität)}, \tag{1.52}$$

$$A \times \emptyset = \emptyset \times A = \emptyset. \tag{1.53}$$

Beachte, dass im Allgemeinen $M_1 \times M_2 \neq M_2 \times M_1$ gilt.

1.3 Relationen und Funktionen

Definition 1.32 *Eine Teilmenge $R \subseteq M_1 \times M_2 \times \cdots \times M_n$ nennt man eine n-stellige Relation zwischen den Mengen $M_1 \times M_2 \times \cdots \times M_n$.*

Gilt speziell $M_1 = M_2 = \cdots = M_n = M$, so nennt man $R \subseteq M^n$ eine n-stellige Relation in M.

Beispiel 1.33 Wir betrachten folgende Beispiele von Relationen. Dabei gilt $x \mid y$ genau dann, wenn es ein $k \in \mathbb{Z}$ mit $y = k \cdot x$ gibt.

$$
\begin{array}{ll}
R_0 \subseteq M_1 \times M_2 \times \cdots \times M_n, & R_0 = \emptyset, \\
R_1 \subseteq M_1 \times M_2 \times \cdots \times M_n, & R_1 = M_1 \times M_2 \times \cdots \times M_n, \\
R_2 \subseteq M^2, & R_2 = \{(x,y) \in M^2 \mid x = y\}, \\
R_3 \subseteq \{a,b,c\}^2, & R_3 = \{(a,a),(b,c),(c,c)\}, \\
R_4 \subseteq \{a,b,c\}^2, & R_4 = \{(a,a),(a,b),(b,a),(b,b),(b,c)\}, \\
R_5 \subseteq \mathbb{Z}^2, & R_5 = \{(x,y) \in \mathbb{Z}^2 \mid x \mid y\}, \\
R_6 \subseteq \mathbb{N}^2, & R_6 = \{(x,y) \in \mathbb{N}^2 \mid x \mid y\}, \\
R_7^{(m)} \subseteq \mathbb{Z}^2,\ m \in \mathbb{N},\ m > 0, & R_7^{(m)} = \{(x,y) \in \mathbb{Z}^2 \mid m \mid (x-y)\}.
\end{array}
$$

Die Relationen R_0 und R_1 im obigen Beispiel nennen wir *Nullrelation* bzw. *Allrelation*. Die Relation R_2 heißt *Diagonale* oder auch *Identität* in M und wird oft mit Δ_M oder auch id_M bezeichnet.

Definition 1.34 *Eine Relation R heißt binär oder zweistellig, wenn $R \subseteq M_1 \times M_2$ gilt. Für die Elemente der Relation $(x,y) \in R$ schreiben wir dann auch xRy (gelesen: „x steht in Relation R zu y“).*

In den obigen Beispielen handelt es sich außer bei den Relationen R_0 und R_1 um binäre Relationen. Das deutet schon darauf hin, dass wir es oft mit binären Relationen zu tun haben. Es gibt natürlich auch wichtige Relationen, die mehr als zweistellig sind. Ein Beispiel ist die sogenannte *Zwischenrelation*.

Endliche binäre Relationen kann man günstig graphisch veranschaulichen, indem man zwei Elemente, die zueinander in Relation stehen, durch einen Pfeil verbindet. In der Abbildung 1.2 ist die Relation R_4 aus Beispiel 1.33 dargestellt.

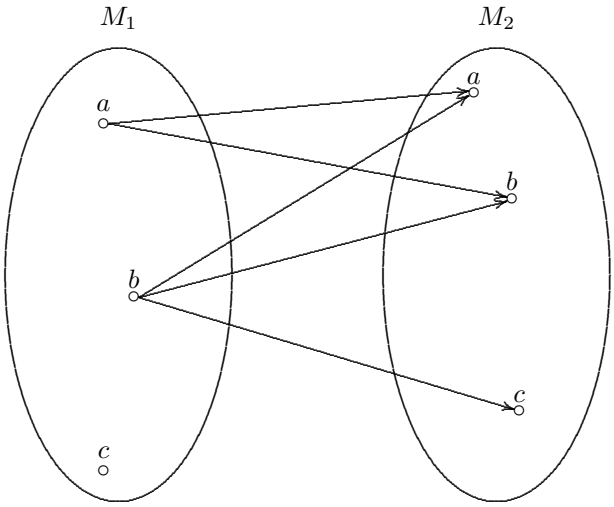


Abbildung 1.2: Graphische Darstellung einer Relation

Definition 1.35 *Sei $R \in M_1 \times M_2$ eine binäre Relationen, dann definieren wir:*

$$D(R) = \{x \in M_1 \mid \exists y \in M_2 \text{ mit } xRy\},$$
$$W(R) = \{y \in M_2 \mid \exists x \in M_1 \text{ mit } xRy\}.$$

Wir bezeichnen $D(R)$ als Definitionsbereich der Relation R und $W(R)$ als Wertebereich der Relation R.

Wir definieren im Folgenden einige Eigenschaften von Relationen, die für uns von Interesse sind, die aber auch in der Mathematik von herausragender Bedeutung sind. Es gibt außer den hier angeführten Eigenschaften natürlich auch noch weitere.

Definition 1.36 *Eine binäre Relation $R \in M_1 \times M_2$ heißt*

eindeutig genau dann, wenn $\forall x \in M_1,\ \forall y_1, y_2 \in M_2\ ((xRy_1 \wedge xRy_2) \Rightarrow (y_1 = y_2))$.

Eine binäre Relation $R \in M^2$ heißt

reflexiv	*genau dann, wenn* $\forall x \in M$	(xRx),
irreflexiv	*genau dann, wenn* $\forall x \in M$	$(\neg(xRx))$,
symmetrisch	*genau dann, wenn* $\forall x, y \in M$	$(xRy \Rightarrow yRx)$,
antisymmetrisch	*genau dann, wenn* $\forall x, y \in M$	$((xRy \wedge yRx) \Rightarrow (x = y))$,
transitiv	*genau dann, wenn* $\forall x, y, z \in M$	$((xRy \wedge yRz) \Rightarrow xRz)$.

Wir weisen noch mal ausdrücklich darauf hin, dass außer der Eigenschaft Eindeutigkeit alle anderen Eigenschaften nur für binäre Relationen **in** einer Menge definiert sind. Betrachten wir die binären Relationen aus Beispiel 1.33, so besitzen diese Relationen folgende Eigenschaften.

Eindeutigkeit:	R_2, R_3
Reflexivität:	R_2, R_5
Irreflexivität:	$--$
Symmetrie:	$R_2, R_5, R_6, R_7^{(m)}$
Antisymmetrie:	R_2, R_3, R_6
Transitivität:	$R_2, R_3, R_4, R_5, R_6, R_7^{(m)}$

Definition 1.37 *Eine Relation $R \in M^2$ heißt*

(i) *reflexive Halbordnung, falls sie reflexiv, antisymmetrisch und transitiv ist.*

(ii) *irreflexive Halbordnung, falls sie irreflexiv und transitiv ist.*

(iii) *Äquivalenzrelation, falls sie reflexiv, symmetrisch und transitiv ist.*

Definition 1.38 *Sei $R \in M^2$ eine Äquivalenzrelation, dann definieren wir*

$$[x]_R = \{y \in M \mid xRy\}$$

und bezeichnen $[x]_R$ als Äquivalenzklasse modulo R mit dem Repräsentanten x.

Die Menge aller Äquivalenzklassen von $R \in M^2$ heißt Faktormenge M/R und ist also definiert durch

$$M/R = \{[x]_R \mid x \in M\}$$

In einer Äquivalenzklasse einer Relation $R \in M^2$ befinden sich also alle Elemente der Menge M, die zueinander in Relation stehen. Wir können folgende Eigenschaften feststellen.

Satz 1.39 *Sei $R \in M^2$ eine Äquivalenzrelation, dann gilt*

(i) $\forall x \in M\ ([x]_R \neq \emptyset)$,

(ii) $\forall x, y \in M\ (([x]_R \neq [y]_R) \Rightarrow ([x]_R \cap [y]_R = \emptyset))$,

(iii) $\bigcup_{x \in M} [x]_R = M$.

Da Relationen Mengen sind, können wir natürlich die mengentheoretischen Operationen (Durchschnitt, Vereinigung, Differenz) auch auf Relationen anwenden. Wir führen jetzt zwei weitere Operationen in der Menge der binären Relationen ein.

Definition 1.40 *Sei $R \subseteq M_1 \times M_2$, dann ist die zu R inverse Relation R^{-1} definiert durch*

$$R^{-1} = \{(x, y) \in M_2 \times M_1 \mid yRx\}.$$

Die Invertierung bedeutet also im Prinzip eine „Umkehrung“. Es gilt also:

$$xRy \Leftrightarrow yR^{-1}x.$$

Daraus ergibt sich sofort:

Folgerung 1.41 *Sei $R \in M_1 \times M_2$ eine binäre Relation. Dann gilt*

$$D(R^{-1}) = W(R) \quad \textit{und} \quad W(R^{-1}) = D(R).$$

Beispiel 1.42 Die inversen Relationen zu den Relationen R_4 und R_6 aus Beispiel 1.33 sind folgende:

$$R_4^{-1} = \{(a, a), (a, b), (b, a), (b, b), (c, b)\},$$
$$R_6^{-1} = \{(x, y) \in \mathbb{N}^2 \mid y \,/\, x\}.$$

Die zweite Operation in der Menge der Relationen ist die sogenannte *Verkettung* oder *Verknüpfung* oder auch ganz einfach das *Produkt* zweier Relationen.

Definition 1.43 *Seien $R \subseteq M_1 \times M_2$ und $S \subseteq M_2 \times M_3$ zwei binäre Relationen, so ist ihre Verkettung $R \circ S$ definiert durch*

$$R \circ S = \{(x, z) \in M_1 \times M_3 \mid \exists y \in M_2 \textit{ mit } (x, y) \in R \wedge (y, z) \in S\}.$$

Beispiel 1.44 Gegeben sind $R \subseteq \{1, 2, 3, 4\} \times \{a, b, c, d\}$ sowie $S \subseteq \{a, b, c, d\} \times \{\alpha, \beta, \gamma, \delta\}$ durch $R = \{(1, a), (2, b), (2, d), (4, d), (4, e)\}$ und $S = \{(a, \alpha), (a, \beta), (c, \gamma), (d, \delta), (e, \delta)\}$. Dann ist $R \circ S = \{(1, \alpha), (1, \beta), (2, \delta), (4, \delta)\}$. Die graphischen Veranschaulichung ist in der Abbildung 1.3 dargestellt (R – gepunktete Pfeile, S – gestrichelte Pfeile, $R \circ S$ – durchgezogene Pfeile).

Wir kommen jetzt zu einem der wichtigsten Begriffe dieser Vorlesung, nämlich dem Begriff der *Funktion* oder *Abbildung*.

Definition 1.45 *Eine eindeutige Relation $R \subseteq M_1 \times M_2$ heißt Funktion oder auch Abbildung aus M_1 in M_2. Schreibweise: $R \colon M_1 \to M_2$.*

Für Funktionen benutzen wir in der Regel in Zukunft kleine lateinische Buchstaben. Falls $f \colon M_1 \to M_2$, so sagen wir, f ist eine Funktion „aus M_1 in M_2“.

Da bei einer Funktion $f : M_1 \to M_2$ jedem $x \in M_1$ höchstens ein $y \in M_2$ zugeordnet wird, schreiben wir

$$y = f(x) \ \text{ statt } \ \{y\} = f(x), \ \text{ falls } \ (x, y) \in f.$$

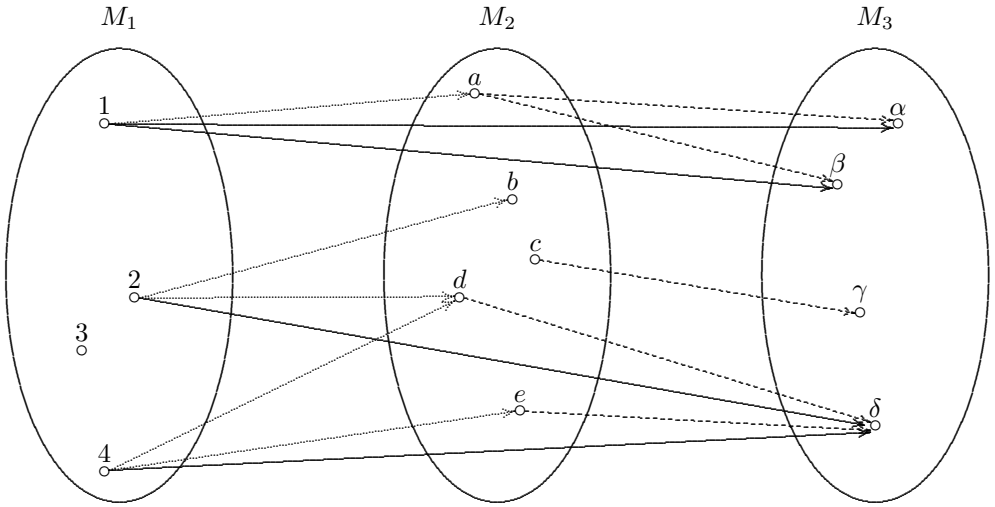


Abbildung 1.3: Graphische Darstellung einer Verküpfung von Relationen

Definition 1.46 *Sei $f\colon M_1 \to M_2$ eine Funktion. Dann heißt f*

(i) *partiell, falls $D(f) \subseteq M_1$ ist,*

(ii) *total, falls $D(f) = M_1$ ist,*

(iii) *surjektiv, falls $W(f) = M_2$ ist,*

(iv) *injektiv oder umkehrbar eindeutig, falls für alle $x_1, x_2 \in M_1$ gilt*

$$(f(x_1) = f(x_2)) \Rightarrow (x_1 = x_2),$$

(v) *bijektiv, falls f injektiv und surjektiv ist.*

Wir führen folgende Sprechweisen ein. $f\colon M_1 \to M_2$ ist eine Funktion

aus M_1 in M_2,	falls f partiell,
von M_1 in M_2,	falls f total,
aus M_1 auf M_2,	falls f partiell und surjektiv,
von M_1 auf M_2,	falls f total und surjektiv.

Wir möchten bemerken, dass man beim Begriff der Funktion in der Literatur oft schon die totale Funktion meint, insbesondere in der Analysis. Für uns ist allerdings die partielle Funktion von besonderem Interesse, so dass wir mit Funktion immer die partielle Funktion meinen.

Jede Funktion ist ja eine Relation, deshalb gelten unsere für Relationen gemachten Definitionen natürlich genauso für Funktionen, insbesondere können wir also Funktionen verknüpfen und ihre inverse Relation bilden. Die Verknüpfung von Funktionen und die Invertierung von Funktioenn liefern natürlich wieder Relationen. Es ergibt sich die Frage, ob es jedoch notwendig wieder Funktionen sind. Man erkennt relativ schnell folgende Sachverhalte:

Satz 1.47 *Seien $f\colon M_1 \to M_2$ und $g\colon M_2 \to M_3$ Funktionen. Dann ist ihre Verknüpfung $f \circ g\colon M_1 \to M_3$ eindeutig, also wieder eine Funktion.*

Satz 1.48 *Sei $f\colon M_1 \to M_2$ eine Funktion. Dann gilt: f^{-1} ist eine Funktion genau dann, wenn f injektiv ist.*

Im Allgemeinen ist also die Inverse einer Funktion nicht wieder eine Funktion. In der Literatur taucht oft der Begriff der „inversen Funktion“ oder „Umkehrfunktion“ auf, damit bezeichnet man

die Inverse, die wieder eine Funktion ist. Wir müssen also zwischen „Inversen einer Funktion“, die immer existiert, und „inverser Funktion“ (existiert nicht immer) wohl unterscheiden.

Wir möchten darauf hinweisen, dass die Verkettung von Funktionen nichts anderes ist als das bekannte *Einsetzen*.

Beispiel 1.49 Gegeben sind die Funktionen $f : \mathbb{R} \to \mathbb{R}$ vermöge $x \mapsto y = f(x) = \sqrt{x}$ und $g\colon \mathbb{R} \to \mathbb{R}$ vermöge $y \mapsto z = g(y) = \sin y$. Dann ist $f \circ g\colon \mathbb{R} \to \mathbb{R}$ vermöge $x \mapsto z = (f \circ g)(x) = g(f(x)) = \sin(\sqrt{x})$.

Es folgt noch eine Charakterisierung von Funktionen, die wir öfter in der Vorlesung benutzen werden.

Definition 1.50 *Sei $n \in \mathbb{N}$, dann heißt die Funktion $f\colon \mathbb{N}^n \to \mathbb{N}$ vermöge $(x_1, x_2, \ldots, x_n) \mapsto y = f(x_1, x_2, \ldots, x_n)$ n-stellige Funktion aus $\mathbb{N}^n$ in $\mathbb{N}$.*

Für $n = 0$ ist ja $\mathbb{N}^n = \mathbb{N}^0 = \{\emptyset\}$, so dass wir nur einen Funktionswert $f(\emptyset)$ haben. Durch solch eine 0-stellige Funktion wird also ein Wert $f(\emptyset)$ in $\mathbb{N}$ ausgezeichnet, also wird nichts anderes durch die Funktion realisiert, als eine Konstante zu vereinbaren.

1.4 Über die Mächtigkeit von Mengen

Im Unterkapitel 1.2 haben wir die Kardinalzahl $|M|$ einer Menge eingeführt und für endliche Mengen als die Anzahl der Elemente der Menge M vereinbart. Wir möchten in diesem Kapitel den Begriff der Kardinalzahl sauber definieren, so dass er auch für unendliche Mengen anwendbar ist.

Dazu benötigen wir zuerst den Begriff der Gleichmächtigkeit zweier Mengen.

Definition 1.51 *Zwei Mengen M_1 und M_2 heißen gleichmächtig, wenn es eine totale bijektive Funktion von M_1 auf M_2 gibt. Schreibweise: $M_1 \overset{\sim}{\longleftrightarrow} M_2$.*

Beispiel 1.52 Die endlichen Mengen $\{1, 2, 3, 4\}$ und $\{6, 7, 8, 9\}$ sind gleichmächtig, in Zeichen $\{1, 2, 3, 4\} \overset{\sim}{\longleftrightarrow} \{6, 7, 8, 9\}$, da die tolale bijektive Funktion

$$f\colon \{1, 2, 3, 4\} \to \{6, 7, 8, 9\} \quad \text{mit} \quad f = \{(1, 6), (2, 7), (3, 8), (4, 9)\}$$

existiert. Die unendlichen Mengen $\mathbb{N}$ und $\{n \in \mathbb{N} \mid \exists k \in \mathbb{N} \text{ mit } n = 2k\}$ sind gleichmächtig, d. h. $\mathbb{N} \overset{\sim}{\longleftrightarrow} \{n \in \mathbb{N} \mid \exists k \in \mathbb{N} \text{ mit } n = 2k\}$, da die tolale bijektive Funktion

$$g\colon \mathbb{N} \to \{n \in \mathbb{N} \mid \exists k \in \mathbb{N} \text{ mit } n = 2k\} \quad \text{vermöge} \quad x \mapsto g(x) = 2 \cdot x$$

existiert. Man beachte, dass zwei unendliche Mengen, von denen die eine Menge eine echte Teilmenge der anderen Menge ist, gleichmächtig sein können. Für endliche Mengen trifft das natürlich nicht zu.

Wie man sich leicht überlegen kann, gilt:

Folgerung 1.53 *Die Relation „$\overset{\sim}{\longleftrightarrow}$“ in der Menge aller Mengen ist eine Äquivalenzrelation.*

Folglich können wir die Äquivalenzklassen für diese Relation bilden. Diese Äquivalenzklassen $[M]_{\overset{\sim}{\longleftrightarrow}}$ sind genau die schon intuitiv eingeführten Kardinalzahlen von Mengen. Wir setzen also

$$|M| := [M]_{\overset{\sim}{\longleftrightarrow}}.$$

Man erkennt, dass für endliche Mengen der Kardinalzahlbegriff mit dem Anzahlbegriff übereinstimmt, dass es aber auch mindestens eine Kardinalzahl „Unendlich“ gibt. Das wollen wir näher spezifizieren.

Definition 1.54 *Eine Menge M heißt abzählbar unendlich, falls sie zur Menge der natürlichen Zahlen $\mathbb{N}$ gleichmächtig ist.*

Definition 1.55 *Eine Menge M heißt abzählbar, falls sie abzählbar unendlich oder endlich ist.*

Die in Beispiel 1.52 betrachteten Mengen sind somit abzählbar, inbesondere ist $\{n \in \mathbb{N} \mid \exists k \in \mathbb{N} \text{ mit } n = 2k\}$, die Menge der natürlichen geraden Zahlen, abzählbar unendlich. Weitere abzählbar unendliche Mengen sind zum Beispiel $\mathbb{Z}$, $\mathbb{Q}$, die Menge der gebrochenen Zahlen, die Menge der Primzahlen und viele andere. Die Beweise überlassen wir dem Leser.

Es gibt allerdings auch unendliche Mengen, die nicht abzählbar unendlich sind. Wir nennen sie *überabzählbar unendlich*. Dazu gehören zum Beispiel $\mathbb{R}$ sowie auch schon $\{x \in \mathbb{R} \mid 0 < x < 1\}$. Beweise dafür findet man in der mathematischen Literatur. Wir wollen hier von einer Menge zeigen, dass sie überabzählbar unendlich ist, die für uns später noch von Interesse sein wird.

Lemma 1.56 *Die Menge $F_{\mathbb{N}}$ aller einstelligen totalen Funktionen von $\mathbb{N}$ in $\mathbb{N}$ ist überabzählbar unendlich.*

Beweis. Wir führen den Beweis indirekt, d. h. wir machen die Annahme, die Menge aller einstelligen totalen Funktionen von $\mathbb{N}$ in $\mathbb{N}$ ist abzählbar unendlich. Nach Definition gibt es also eine totale bijektive Funktion g von der Menge $\mathbb{N}$ in die Menge $\{f : \mathbb{N} \to \mathbb{N} \mid D(f) = \mathbb{N}\}$. Für $n \in \mathbb{N}$ sei $g(n) = f_n$.

Wir konstruieren eine totale Funktion $f : \mathbb{N} \to \mathbb{N}$ durch $f(n) = f_n(n) + 1$. Da f total ist und eine Funktion von $\mathbb{N}$ in $\mathbb{N}$ ist, ist sie ein Element der Menge $F_{\mathbb{N}}$. Andererseits gilt für jedes $n \in \mathbb{N}$ $f \neq f_n$, da $f(n) = f_n(n) + 1 \neq f_n(n)$. Folglich haben wir einen Widerspruch konstruiert, somit ist unsere Annahme falsch und das Lemma bewiesen. □

Folgerung 1.57 *Die Menge aller Funktionen von $\mathbb{N}^n$ in $\mathbb{N}$, $n \in \mathbb{N}$, ist überabzählbar unendlich.*

Wir möchten an dieser Stelle bemerken, dass es in der Informatik den Begriff der *rekursiv aufzählbaren Menge* oder oft auch kurz den Begriff der *aufzählbaren Menge* gibt. Dieser ist mit dem Begriff der *Abzählbarkeit* nicht identisch. Allerdings gibt es Beziehungen. Wir werden später darauf zurückkommen.

1.5 Algebraische Strukturen

In diesem Kapitel wollen wir sehr kurz ganz wenige Begriffe aus der Algebra zur Verfügung stellen.

Zunächst wollen wir den Begriff der *Operation* einführen.

Definition 1.58 *Eine zweistellige totale Funktion $\circ : M^2 \to N$ heißt (binäre) Operation oder Verknüpfung.*

Gilt $N = M$, so nennen wir $\circ$ vollständig oder abgeschlossen und sprechen von einer Operation in M.

Für $\circ(x, y)$ schreibt man auch $x \circ y$ oder kurz xy, falls $\circ$ aus dem Kontext eindeutig hervorgeht.

Folgende Eigenschaften von Operationen sind oft von Interesse.

Definition 1.59 *Sei $\circ : M^2 \to M$ eine Operation.*

(i) *$\circ$ heißt assoziativ, wenn $(x \circ y) \circ z = x \circ (y \circ z)$ für alle $x, y, z \in M$ gilt.*

(ii) *$\circ$ heißt kommutativ, wenn $x \circ y = y \circ x$ für alle $x, y \in M$ gilt.*

Definition 1.60 *Sei M eine nichtleere Menge und $\circ : M^2 \to M$ eine Operation in M, so heißt $(M, \circ)$ (algebraische) Struktur mit der Trägermenge M.*

Algebraische Strukturen kommen in vielen Wissenschaftsdisziplinen vor und sind ein ausgezeichnetes Mittel der Abstraktion, das eine gemeine Sicht von Sachverhalten zulässt, die ursprünglich nichts miteinander zu tun haben.

Wir betrachten hier nur einen sehr, sehr kleinen Ausschnitt der algebraishen Strukturen. Folgende Strukturen sind insbesondere für uns von Interesse.

Definition 1.61 *Sei $(M, \circ)$ eine algebraische Struktur.*

(i) *$(M, \circ)$ heißt Halbgruppe, falls $\circ$ assoziativ ist.*

(ii) *$(M, \circ)$ heißt abelsch oder kommutativ, falls $\circ$ kommutativ ist.*

(iii) *Ist $(M, \circ)$ eine Halbgruppe, so heißt sie Monoid, falls*

$$\exists e \in M \ \forall x \in M \ (x \circ e = e \circ x = x).$$

e heißt neutrales Element der Struktur.

(iv) *Ist $(M, \circ)$ ein Monoid, so heißt es Gruppe, falls*

$$\forall x \in M \ \exists x^{-1} \in M \ (x \circ x^{-1} = x^{-1} \circ x = e)$$

gilt, wobei e das neutrale Element des Monoids ist.

Folgende Strukturen werden durch Zahlbereiche gebildet, wie man sich leicht überlegen kann.

Beispiel 1.62 (i) $(\mathbb{N}, +)$, $(\mathbb{N}, \cdot)$, $(\mathbb{Z}, \cdot)$ sowie $(\{x \in \mathbb{Q} \mid x \geq 0\}, +)$ sind abelsche Monoide, aber keine Gruppen.

(ii) $(\mathbb{Z}, +)$ und $(\{x \in \mathbb{Q} \mid x \geq 0\}, +)$ sind abelsche Gruppen.

(iii) Die Mengen $\mathbb{Q}$ sowie $\mathbb{R}$ bilden sowohl mit der Addition als auch der Multiplikation abelsche Gruppen.

1.6 Alphabete, Wörter, Sprachen

Wir wollen hier einige Grundbegriffe für das folgende Kapitel geben.

Unter einem *Alphabet* verstehen wir eine *endliche nichtleere Menge*. Zum Beispiel ist $\Sigma = \{a, b, c\}$ ein Alphabet. Die Elemente eines Alphabets heißen Buchstaben, Zeichen oder Symbole. Endliche Folgen von Buchstaben des Alphabets nennen wir Wörter über dem Alphabet; sie werden durch einfaches Hintereinanderschreiben der Buchstaben angegeben. Zum Beispiel sind *aba*, *abba* und *aaaa* Wörter über dem Alphabet Σ. ε bezeichnet das Leerwort oder leere Wort, das der leeren Folge entspricht, also aus keinem Buchstaben besteht. Die Menge aller Wörter über einem Alphabet Σ (einschließlich ε) bezeichnen wir mit Σ^*. Wir setzen $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$.

Folgerung 1.63 *Es sei Σ ein Alphabet. Dann enthält Σ^* abzählbar unendlich viele Elemente.*

Wir sind jetzt in der Lage, einen der beiden Begiffe aus dem Titel dieser Vorlesung zu definieren.

Definition 1.64 *Sei Σ ein Alphabet. Eine Teilmenge $L \subseteq \Sigma^*$ heißt formale Sprache über Σ.*

In Σ^* definieren wir das *Produkt* oder die *Konkatenation* $w_1 \cdot w_2$ (oder kurz $w_1 w_2$) der Wörter w_1 und w_1 durch einfaches Hintereinanderschreiben. Für $w_1 = abba$ und $w_2 = ba$ erhalten wir beispielsweise $w_1 \cdot w_2 = abbaba$.

Man sieht leicht folgenden Satz.

Satz 1.65 *Es sei Σ ein Alphabet. Für alle Wörter $w, w_1, w_2, w_3 \in \Sigma^*$ gelten dann folgende Beziehungen:*

$$w_1 \cdot (w_2 \cdot w_3) = (w_1 \cdot w_2) \cdot w_3 \quad (\textit{Assoziativgesetz}),$$
$$w\varepsilon = \varepsilon w = w \quad (\varepsilon \textit{ neutrales Element}).$$

Wie man sich aber leicht überzeugen kann, ist die Konkatenation nicht kommutativ. Für $w_1 = abba$ und $w_2 = ba$ gilt zum Beispiel $w_1 \cdot w_2 = abbaba$ und $w_2 \cdot w_1 = baabba$.

Die Konkatenation ist aber eine abgeschlossene Operation in Σ^*. Somit bildet $(\Sigma^*, \cdot)$ eine algebraische Struktur. Und wegen Satz 1.65 ist $(\Sigma^*, \cdot)$ sogar ein *Monoid*.

Wegen der Assoziativität der Konkatenation können wir folgende abkürzende Schreibweise einführen.

$$\underbrace{w \cdot w \cdot \cdots \cdot w}_{n\text{-mal}} = w^n$$

Es gilt $w^0 = \varepsilon$.

Wir erweitern die Konkatenation auf formale Sprachen.

Definition 1.66 *Sei Σ ein Alphabet, $L_1, L_2 \subseteq \Sigma^*$ Sprachen über Σ. Die Konkatenation $L_1 \cdot L_2$ ist definiert durch:*

$$L_1 \cdot L_2 = \{w_1 \cdot w_2 \mid w_1 \in L_1,\ w_2 \in L_2\}.$$

Unter der *Länge* $|w|$ eines Wortes w verstehen wir die Anzahl der in w vorkommenden Buchstaben, wobei jeder Buchstabe sooft gezählt wird wie er in w vorkommt. Es gilt zum Beispiel $|aba| = 3$, $|abba| = 4$, $|aaaa| = 4$ und $|\varepsilon| = 0$.

Aus der Definition der Länge ergibt sich sofort:

Folgerung 1.67 *Es sei Σ ein Alphabet. Für alle Wörter $w_1, w_2 \in \Sigma^*$ gilt dann*

$$|w_1 \cdot w_2| = |w_1| + |w_2|.$$

Zum Abschluss noch eine nützliche Definition.

Definition 1.68 *Sei $w \in \Sigma^*$ ein Wort mit $w = x \cdot y \cdot z$.*

(i) *y heißt Teilwort von w.*

(ii) *Falls $y \neq \varepsilon$ und $y \neq w$, heißt y echtes Teilwort von w.*

(iii) *y heißt Präfix von w, falls $x = \varepsilon$.*

(iv) *y heißt Suffix von w, falls $z = \varepsilon$.*

2 Berechenbarkeit

Dieses Kapitel entspricht im Wesentlichen dem Kapitel 2 (*Berechenbarkeitstheorie*) in [13].

Jeder, der programmieren kann, weiß, dass es so etwas wie einen *intuitiven Berechenbarkeitsbegriff* gibt. Man hat eine Vorstellung davon, welche Funktionen *berechenbar* sind.

Will man allerdings zeigen, dass gewisse Funktionen *nicht* berechenbar sind, reicht dieser intuitive Berechenbarkeitsbegriff nicht aus. Es ist deshalb notwendig, den intuitiven Begriff der Berechenbarkeit formal sauber, d. h. mathematisch korrekt zu definieren.

Es tritt jedoch das Problem auf, dass man zwar einen formalen Berechenbarkeitsbegriff hat, allerdings wiederum nicht zeigen kann, dass er genau dem intuitiven Begriff entspricht. Deshalb werden wir verschiedene Berechenbarkeitsbegriffe definieren (siehe Kapitel 2.2 und 2.3). Letztlich werden wir zeigen, dass alle diese verschiedenen Berechenbarkeitsbegriffe äquivalent sind und somit annnehmen, dass wir wahrscheinlich den intuitiven Begriff getroffen haben (siehe Kapitel 2.4). Beweisen können wir es aber nach wie vor *nicht*!

Wir beginnen mit der Diskussion des intuitiven Berechenbarkeitsbegriffes.

2.1 Intuitiver Berechenbarkeitsbegriff

Eine (evtl. partielle Funktion) $f : \mathbb{N}^k \to \mathbb{N}$ soll als *intuitiv berechenbar* angesehen werden, falls es ein Rechenverfahren, einen *Algorithmus*[3] gibt, das/der f *berechnet*, d. h. gestartet mit $(n_1, n_2, \ldots, n_k) \in \mathbb{N}^k$ als Eingabe soll der Algorithmus nach endlich vielen Schritten mit der Ausgabe $f(n_1, n_2, \ldots, n_k) \in \mathbb{N}$ stoppen.

Im Falle einer partiellen Funktion (also einer Funktion, die an manchen Stellen undefiniert sein kann) soll der Algorithmus bei der entsprechenden Eingabe nicht stoppen (z. B. durch eine unendliche Schleife).

Jedem Algorithmus wird also eine Funktion, die durch ihn berechnet wird, zugeordnet.

Beispiel 2.1 Der Algorithmus

```
INPUT(n);
REPEAT UNTIL FALSE
```

„berechnet“ die total undefinierte Funktion $\Omega\colon \mathbb{N} \to \mathbb{N}$ vermöge $n \mapsto f(n) = \textit{nicht definiert}$.

Beispiel 2.2 Die Funktion $f\colon \mathbb{N} \to \mathbb{N}$, definiert durch

$$f(n) = \begin{cases} 1 & \text{falls } n \text{ ein Anfangsabschnitt der Dezimalbruchentwicklung von } \pi \text{ ist,} \\ 0 & \text{sonst} \end{cases}$$

(zum Beispiel $f(314) = 1$, $f(314158) = 0$, $f(31415926535897932384626433832795028841) = 1$) ist berechenbar, denn es gibt Näherungsverfahren für die Zahl π, mit denen man π beliebig genau bestimmen kann (auch wenn es für sehr große n sehr lange dauern kann, bis man $f(n)$ kennt).

Beispiel 2.3 Die Funktion $g\colon \mathbb{N} \to \mathbb{N}$, definiert durch

$$g(n) = \begin{cases} 1 & \text{falls } n \text{ irgendwo in der Dezimalbruchentwicklung von } \pi \text{ vorkommt,} \\ 0 & \text{sonst} \end{cases}$$

ist möglicherweise nicht berechenbar. Wir können zwar zum Beispiel $g(141) = 1$ und $g(3589) = 1$ angeben. Falls aber ein n in der bis heute bekannten Dezimaldarstellung von π nicht vorkommt, können wir momentan keine Aussage treffen, da wir ja nicht wissen, wo in der Dezimaldarstellung n auftaucht.

Unser bisheriges Wissen über die Zahl π reicht nicht aus, um eine Entscheidung über Berechenbarkeit oder Nicht-Berechenbarkeit zu treffen.

[3] Benannt nach Mohamed Ibn Musa Alchwarizmi, geboren in Chiwa (Chorism), gestorben in Bagdad nach 846.

Berechenbarkeit ist also eng mit dem Begriff eines *Algorithmus* verbunden. Intuitiv stellen wir dabei folgende Forderungen an einen Algorithmus.

Ein Algorithmus

- überführt Eingabedaten in Ausgabedaten (wobei die Art der Daten vom Problem, das durch den Algorithmus gelöst werden soll, abhängig ist),
- besteht aus einer endlichen Folge von Anweisungen mit folgenden Eigenschaften:
  - es gibt eine eindeutig festgelegte Anweisung, die als erste auszuführen ist,
  - nach Abarbeitung einer Anweisung gibt es eine eindeutig festgelegte Anweisung, die als nächste abzuarbeiten ist, oder die Abarbeitung des Algorithmus ist beendet und hat eindeutig bestimmte Ausgabedaten geliefert,
  - die Abarbeitung einer Anweisung erfordert keine Intelligenz (ist also prinzipiell durch eine Maschine realisierbar).

Mit diesem intuitiven Konzept lässt sich leicht feststellen, ob ein Verfahren ein Algorithmus ist. Betrachten wir als Beispiel die schriftliche Addition. Als Eingabe fungieren die beiden gegebenen zu addierenden Zahlen; das Ergebnis der Addition liefert die Ausgabe. Der Algorithmus besteht im Wesentlichen aus der sukzessiven Addition der entsprechenden Ziffern unter Beachtung des jeweils entstehenden Übertrags, wobei mit den „letzten“ Ziffern angefangen wird. Zur Ausführung der Addition von Ziffern ist keine Intelligenz notwendig (obwohl wir in der Praxis dabei das scheinbar Intelligenz erfordernde Kopfrechnen benutzen), da wir eine Tafel benutzen können, in der alle möglichen Additionen von Ziffern enthalten sind (und wir davon ausgehen, dass das Ablesen eines Resultats aus einer Tafel oder Liste ohne Intelligenz möglich ist). In ähnlicher Weise kann man leicht überprüfen, dass z. B.

- der Gaussschе[4] Algorithmus zur Lösung von linearen Gleichungssystemen (über den rationalen Zahlen),
- Kochrezepte (mit Zutaten und Kochgeräten als Eingabe und dem fertigen Gericht als Ausgabe),
- Bedienungsanweisungen für Geräte,
- PASCAL-Programme

Algorithmen sind.

Jedoch ist andererseits klar, dass dieser Algorithmenbegriff nicht ausreicht, um zu klären, ob es für ein Problem einen Algorithmus zur Lösung gibt. Falls man einen Algorithmus zur Lösung hat, so sind nur obige Kriterien zu testen. Um jedoch zu zeigen, dass es keinen Algorithmus gibt, ist es erforderlich, eine Kenntnis aller möglichen Algorithmen zu haben; und dafür ist der obige intuitive Begriff zu unpräzise. Folglich wird es unsere Aufgabe sein, eine Präzisierung des Algorithmenbegriffs vorzunehmen, die es gestattet, in korrekter Weise Beweise führen zu können.

2.2 Turing-Berechenbarkeit

Wir beginnen mit Turings[5] Vorschlag zu einer formalen Definition des Berechenbarkeitsbegriffs, der auf der nach ihm benannten Turingmaschine basiert. Er ging hierbei von der Idee aus nachzuempfinden, wie ein Mensch eine systematische Berechnung, wie etwa eine schriftliche Multiplikation nach der Schulmethode, durchführt. Er verwendet hierzu ein Rechenblatt – in Felder eingeteilt – auf dem die Rechnung, samt aller Zwischenergebnisse, notiert wird. Zur Verfügung stehen ihm hierbei ein Schreibwerkzeug und eventuell ein Radierer, um Zeichen auf Felder zu notieren bzw. wieder zu löschen.

[4] Carl Friedrich Gauss, 1777–1855, deutscher Mathematiker.

[5] Alan Mathison Turing (1912–1954), britischer Mathematiker, Logiker, Kryptoanalytiker und Computerkonstrukteur.

Die jeweilige Aktion hängt nur von wenigen (endlich vielen) Symbolen ab, die sich im Umfeld der aktuellen Position des Schreibwerkzeuges befinden. Die Rechnung wird hierbei gesteuert von einem endlichen Programm.

Die formale Definition der Turingmaschine ist eine gewisse Vereinfachung obiger Ideen.

- Das zweidimensionale Rechenblatt wird reduziert zu einem eindimensionalen, beidseitig (potentiell) unendlichen Band, das in Felder eingeteilt ist. Jedes Feld kann ein einzelnes Zeichen des sogenannten Bandalphabets der Maschine enthalten. Zum Bandalphabet gehört auch das Leer- oder Blankzeichen $\square$, das in allen Feldern steht, in denen kein anderes Zeichen des Bandalphabets steht.
- Das Schreibwerkzeug und der Radierer verschmelzen zu einem einzigen Schreib-Lesekopf. Dieser kann sich auf dem Band bewegen. Nur solche Zeichen, auf denen sich dieser Kopf gerade befindet, können in einem momentanen Rechenschritt gelesen und verändert werden. Der Kopf kann in einem Rechenschritt dann um maximal eine Position nach links oder rechts bewegt werden.
- Die Steuerung übernimmt die endliche Kontrolle.

In Abbildung 2.1 ist diese Modell veranschaulicht.

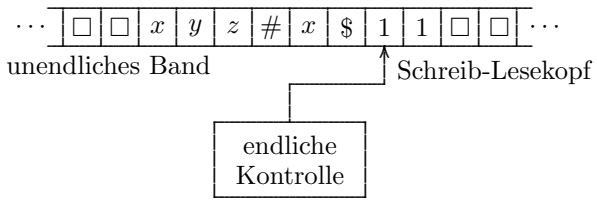


Abbildung 2.1: Veranschaulichung einer Turingmaschine

Formalisiert ergibt dieses Konzept folgende Definition.

Definition 2.4 *Eine (deterministische) Turingmaschine (kurz: TM) ist gegeben durch ein 7-Tupel*

$$M = (Z, \Sigma, \Gamma, \delta, z_0, \square, E).$$

Hierbei sind

- Z *eine endliche Menge (Zustandsmenge),*
- Σ *ein Alphabet (Eingabealphabet),*
- Γ *ein Alphabet (Bandalphabet) mit* $\Sigma \subseteq \Gamma$*,*
- $\delta \colon Z \times \Gamma \to Z \times \Gamma \times \{L, R, N\}$ *eine Funktion (Überführungsfunktion),*
- $z_0 \in Z$ *(Anfangszustand),*
- $\square \in \Gamma \setminus \Sigma$ *(Leerzeichen, Blank),*
- $E \subseteq Z$ *(Menge der Endzustände).*

Zur Interpretaion der Arbeitsweise der Turingmaschine: Falls

$$\delta(z, a) = (z', b, r)$$

gilt, bedeutet das: Wenn sich die TM M im Zustand z befindet und sie auf dem Band unter dem Schreib-Lesekopf das Zeichen a liest, so geht M in den Zustand z' über, schreibt auf das Band ein b (genau dorthin, wo der Kopf steht, sie also das a gelesen hat, d. h. a wird durch b überschrieben) und führt danach eine Kopfbewegung $r \in \{R, L, N\}$ aus. Hiebei bedeuten L: ein Schritt nach links, R: ein Schritt nach rechts, N: keine Kopfbewegung.

Die Turingmaschine beginnt ihre Arbeit, wenn das Eingabewort w auf dem Band steht, in allen anderen Zellen steht das Blankzeichen $\Box$, und sie sich im Anfangszustand z_0 befindet. Sie beendet ihre Arbeit, wenn sie einen Stopzustand, also einen Zustand aus E erreicht. Dabei wird vereinbart, dass auf dem Band dann das Ausgabewort steht, sonst nur Blankzeichen und der Kopf der Turingmaschine wiederum über dem ersten Symbol der Ausgabe steht.

Um die Arbeitsweise der Turingmaschine formalisiert darstellen zu können, benötigen wir eine Beschreibung der augenblicklichen Situation einer Turingmaschine, das geschieht mit der Konfiguration.

Definition 2.5 *Eine Konfiguration k einer Turingmaschine $M = (Z, \Sigma, \Gamma, \delta, z_0, \Box, E)$ ist ein Wort $k \in \Gamma^* Z \Gamma^*$.*

Dabei soll $k = \alpha z \beta$ folgendermaßen interpretiert werden:

- $\alpha\beta$ steht auf dem Eingabeband, des weiteren stehen nur noch Blankzeichen $\Box$ auf dem Band,
- die Turingmaschine befindet sich im Zustand z und
- der Kopf der Turingmaschine befindet sich über dem ersten Symbol von β.

Eine Startkonfiguration ist somit $k_0 = z_0 w$ mit $w \in \Sigma^*$. Eine Endkonfiguration $k_e = \Box q w'$ mit $q \in E$, $w' \in \Sigma^*$.

Jetzt können wir die Arbeitsweise formalisieren.

Definition 2.6 *Wir definieren in der Menge der Konfigurationen einer Turingmaschine $M = (Z, \Sigma, \Gamma, \delta, z_0, \Box, E)$ die binäre Relation „$\vdash$“ wie folgt. Es gilt*

$$a_1 \dots a_m z b_1 \dots b_n \vdash \begin{cases} a_1 \dots a_m z' c b_2 \dots b_n & \text{falls } \delta(z, b_1) = (z', c, N),\ m \geq 0,\ n \geq 1, \\ a_1 \dots a_m c z' b_2 \dots b_n & \text{falls } \delta(z, b_1) = (z', c, R),\ m \geq 0,\ n \geq 2, \\ a_1 \dots a_{m-1} z' a_m c b_2 \dots b_n & \text{falls } \delta(z, b_1) = (z', c, L),\ m \geq 1,\ n \geq 1, \end{cases}$$

$$a_1 \dots a_m z b_1 \vdash a_1 \dots a_m c z \Box \text{ falls } \delta(z, b_1) = (z', c, R),\ m \geq 0,$$

$$z b_1 \dots b_n \vdash z \Box c b_2 \dots b_n \text{ falls } \delta(z, b_1) = (z', c, L),\ n \geq 1.$$

Die Definition ist so gestaltet, dass die Konfigurationsbeschreibungen bei Bedarf verlängert werden, wenn die Maschine links oder rechts ein neues, bisher noch nicht besuchtes, Zeichen liest. Dieses kann natürlich nur das Blankzeichen $\Box$ sein.

Definition 2.7 *Mit $\vdash^*$ bezeichnen wir den reflexiven und transitiven Abschluss der binären Relation $\vdash$, also es gilt $k_0 \vdash^* k_e$ genau dann, wenn*

(i) *$k_0 = k_e$ ist, oder*

(ii) *eine Zahl $n \geq 1$ und Konfigurationen $k_1, k_2, \dots, k_n$ existieren, so dass*

$$k_0 \vdash k_1 \vdash k_2 \vdash \dots \vdash k_n \vdash k_e.$$

Betrachten wir ein Beispiel.

Beispiel 2.8 Gegeben sei die Turingmaschine

$$M = (\{z_0, z_1, z_2, z_e\}, \{0, 1\}, \{0, 1, \Box\}, \delta, z_0, \Box, \{z_e\}),$$

wobei δ wie folgt definiert ist. Wir geben dabei δ in einer Tabelle an, wobei im Kreuzungspunkt der Zeile mit der Bezeichnung a und der Spalte mit der Bezeichnung z der Funktionswert $\delta(z, a)$

steht.

δ	z_0	z_1	z_2
$\square$	$(z_1, \square, L)$	$(z_e, 1, N)$	$(z_e, \square, R)$
0	$(z_0, 0, R)$	$(z_2, 1, L)$	$(z_2, 0, L)$
1	$(z_0, 1, R)$	$(z_1, 0, L)$	$(z_2, 1, L)$

Wenn diese Maschine mit der Eingabe 101 gestartet wird, so stoppt sie schließlich mit 110 auf dem Band, wobei der Schreib-Lesekopf wieder auf dem ersten Zeichen der Ausgabe steht.

Präzisiert, haben wir im Einzelnen folgende Überführungsschritte:

$$z_0101 \vdash 1z_001 \vdash 10z_01 \vdash 101z_0\square \vdash 10z_11\square \vdash 1z_100\square \vdash z_2110\square \vdash z_2\square110\square \vdash \square z_e110\square$$

Nachdem wir die Turingmaschine und ihre Arbeitsweise eingeführt und definiert haben, sind wir jetzt in der Lage, den Begriff der *Turingberechenbaren Funktion* zu definieren. Dazu wird zunächst der Begriff für Funktionen $f : \Sigma^* \to \Sigma^*$ eingeführt und danach für Funktionen $f : \mathbb{N}^k \to \mathbb{N}$ für ein $k \in \mathbb{N}$ erweitert, wobei $\text{bin} : \mathbb{N} \to \{0,1\}^*$ eine Codierung der natürlichen Zahlen in der Menge $\{0,1\}^*$ darstellt, nämlich die übliche Binärdarstellung der natürlichen Zahlen, wobei wir allerdings keine führenden Nullen zulassen (die Zahl „0“ wird also durch das leere Wort dargestellt).

Definition 2.9 *Eine Funktion $f : \Sigma^* \to \Sigma^*$ heißt Turingberechenbar, falls es eine Turingmaschine $M = (Z, \Sigma, \Gamma, \delta, z_0, \square, E)$ gibt, so dass für alle $x, y \in \Sigma^*$ gilt:*

$$f(x) = y \quad \textit{genau dann, wenn} \quad z_0x \vdash^* \square \ldots \square z_e y \square \ldots \square \textit{ für } z_e \in E.$$

Definition 2.10 *Eine Funktion $f : \mathbb{N}^k \to \mathbb{N}$ für ein $k \in \mathbb{N}$ heißt Turingberechenbar, falls es eine Turingmaschine $M = (Z, \{0,1\}, \Gamma, \delta, z_0, \square, E)$ gibt, so dass für alle $n_1, n_2, \ldots, n_k, m \in \mathbb{N}$ gilt:*

$$f(n_1, n_2, \ldots, n_k) = m \quad \textit{genau dann, wenn}$$
$$z_0\text{bin}(n_1)\#\text{bin}(n_2)\#\ldots\#\text{bin}(n_k) \vdash^* \square \ldots \square z_e \text{bin}(m) \square \ldots \square \textit{ für } z_e \in E.$$

Man beachte, dass bei beiden Definitionen implizit ausgedrückt wird, dass im Falle $f(x) =$ *undefiniert* die Maschine M *keinen* Stopzustand erreicht, also zum Beispiel in eine unendliche Schleife gerät.

Beispiel 2.11 Die Nachfolgerfunktion $f : \mathbb{N} \to \mathbb{N}$ vermöge $n \mapsto f(n) = n + 1$ ist Turingberechenbar, da die Turingmaschine aus Beispiel 2.8 die Eingabe $\text{bin}(n)$ in die Ausgabe $\text{bin}(n + 1)$ transformiert.

Beispiel 2.12 Die für alle Wörter aus $\{a, b\}^*$ nicht-definierte Funktion $\Omega : \{a, b\}^* \to \{a, b\}^*$ vermöge $w \mapsto \Omega(w) =$ *nicht definiert* ist Turingberechenbar, da sie von der Turingmaschine $M = (\{z_0, z_e\}, \{a, b\}, \{a, b, \square\}, \delta, z_0, \square, \{z_e\})$ mit $\delta(z_0, x) = (z_0, x, N)$ für alle $x \in \{a, b, \square\}$ berechnet wird.

Eine *Mehrband-Turingmaschine* kann auf k Bändern, $k \geq 1$, unabhängig voneinander operieren, d. h. sie hat k Schreib-Leseköpfe, die in jedem Schritt lesen, schreiben und sich unabhängig voneinander bewegen können. Formal kann eine solche Maschine erfasst werden, indem wir δ als eine Funktion von $Z \times \Gamma^k$ in $Z \times \Gamma^k \times \{R, L, N\}^k$ ansetzen. Die Begriffe *Konfiguration*, *direkter Überführungsschritt* sowie *berechnete Funktion* können dementsprechend erweitert werden. Wir wollen es aber an dieser Stelle nicht tun, sondern es mit der informellen Darstellung belassen. In der Literatur, z. B. in [7] kann der interessierte Leser die Formalisierungen finden.

Wir wollen uns nun der Frage widmen, ob die Mehrband-Turingmaschine mehr *Berechnungskraft* besitzt als die einfache Turingmaschine, wobei wir den Beweis folgender Aussage nur ganz grob skizzieren und wieder auf die Literatur verweisen.

Satz 2.13 *Zu jeder Mehrband-Turingmaschine M gibt es eine (Einband-) Turingmaschine M', die dieselbe Funktion berechnet wie M.*

Beweis. Hier nur die Beweisidee:

Sei k die Anzahl der Bänder und Γ das Bandalphabet von M. Das Arbeitsband von M' soll dann $2k$ *Spuren* enthalten, k davon werden genutzt, um die Inhalte der k Bänder von M zu speichern und die restlichen k Bänder werden nur benutzt, um die jeweilige Kopfposition über dem simulierten Band zu markieren. M' simuliert dann einen Arbeitsschritt von M, indem sukzessive alle Inhalte der Zellen der Spuren gelesen werden, die durch die Kopfposition markiert sind, dann gemäß der Überführungsfunktion von M alle diese Inhalte wieder sukzessive neu beschrieben werden und anschließend gemäß der Überführungsfunktion wiederum nacheinander alle Marker für die Kopfpositionen verschoben werden. Genaueres lese der geneigte Leser bitte in der Literatur nach. □

Dieser Satz ermöglicht es uns, gewisse Funktionen durch Mehrband-Turingmaschinen berechnen zu lassen, was oftmals wesentlich „günstiger“ ist als die Nutzung von Einband-Turingmaschinen. Wir aber wissen, dass es dann natürlich auch Einband-Turingmaschinen gibt, die diese Funktionen realisieren.

Wir werden davon im folgenden Gebrauch machen und zur einfacheren Darstellung folgende Notationen einführen.

Notation 2.14 Wenn M eine 1-Band-Turingmaschine ist, so bezeichnet $M(i,k)$, $i \leq k$, diejenige k-Band-Turingmaschine, die wir aus M dadurch erhalten, dass die Aktionen von M auf dem Band i ablaufen und alle anderen Bänder unverändert bleiben. Falls die Gesamtzahl der Bänder k „genügend“ groß ist, also eigentlich keine Rolle spielt, so schreiben wir einfach $M(i)$ statt $M(i,k)$.

Notation 2.15 Die Turingmaschine aus Beispiel 2.8 (die zu einer Zahl 1 dazuaddiert) bezeichnen wir mit „Band := Band + 1“ und anstelle von „Band := Band + 1 (i)“ scheiben wir „Band i := Band $i+1$“. In ähnlicher Weise können wir auch Mehrband-Turingmaschinen erhalten, die die Operationen „Band i := Band $i \dot{-} 1$“, „Band $i := 0$“ sowie „Band i := Band j“ ausführen. Dabei bedeutet „ $\dot{-}$ “ die modifizierte Subtraktion $\dot{-} : \mathbb{N}^2 \to \mathbb{N}$ vermöge

$$(n_1, n_2) \mapsto n_1 \dot{-} n_2 = \begin{cases} n_1 - n_2 & \text{falls } n_1 \geq n_2, \\ 0 & \text{sonst.} \end{cases}$$

Als nächstes wollen wir Turingmaschinen „hintereinanderschalten“.

Notation 2.16 Seien M_1 und M_2 zwei Turingmaschinen, so wollen wir durch

$$\text{start} \longrightarrow M_1 \longrightarrow M_2 \longrightarrow \text{stop}$$

oder auch durch

$$M_1; M_2$$

diejenige Turingmaschine verstehen, die zuerst wie die Turingmaschine M_1 arbeitet und wenn M_1 einen Stopzustand erreichen würde in den Anfangszustand von M_2 übergeht und jetzt wie die Turingmaschine M_2 arbeitet. Sie stoppt dann, wenn M_2 einen Stopzustand erreichen würde.

Beispiel 2.17 Betrachten wir das Diagramm im Bild 2.2. So erkennen wir, dass dort das schematische Flussbild einer Turingmaschine steht, welche dreimal nacheinander zur Zahl auf dem Band 1 addiert, also es sich um die Turingmaschine „Band := Band + 3“ handelt.

Notation 2.18 Analog soll die Turingmaschine im Bild 2.3 nach Simulation der Turingmaschine M die Turingmaschine M_1 simulieren, falls sie bei der Simulation von M im Zustand z_{e1} stoppt. Analog soll sie die Turingmaschine M_2 abarbeiten, falls sie bei der Simulation von M im Zustand z_{e2} stoppt.

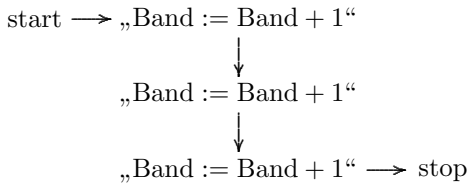


Abbildung 2.2: Die Turingmaschine „Band := Band + 3“

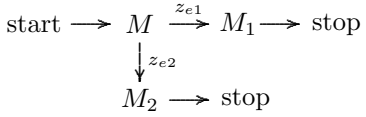


Abbildung 2.3: Eine sich verzweigende Turingmaschine

Betrachten wir in folgendem Beispiel noch eine spezielle Turingmaschine.

Beispiel 2.19 Es sei $M = (\{z_0, z_1, ja, nein\}, \Sigma, \Gamma, \delta, z_0, \square, \{ja, nein\})$ mit $0 \in \Sigma,\ 0, \square \in \Gamma$ sowie mit der Überführungsfunktion δ, gegeben durch

$$\begin{aligned}
\delta(z_0, a) &= (nein, a, N) \quad \text{für } a \neq 0,\\
\delta(z_0, 0) &= (z_1, 0, R),\\
\delta(z_1, a) &= (nein, a, L) \quad \text{für } a \neq \square,\\
\delta(z_1, \square) &= (ja, \square, L).
\end{aligned}$$

Diese Turingmaschine testet, ob die Eingabe genau das Wort 0 ist. Falls ja, stoppt sie im Zustand *ja*, falls nein, stoppt sie im Zustand *nein*. Wir wollen diese Turingmaschine mit „Band = 0 ?“ bezeichnen.

Anstatt von „Band = 0 ? (i)“ schreiben wir „Band $i = 0$?“

Beispiel 2.20 Betrachten wir noch ein weiteres Beispiel einer Turingmaschine. Sei M eine beliebige Turingmaschine, dann bezeichnen wir die Turingmaschine, die durch das Diagramm in der

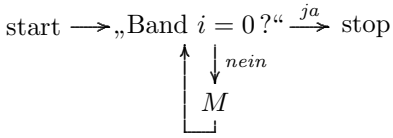


Abbildung 2.4: Die Turingmaschine „WHILE Band $i \neq 0$ DO M“

Abbildung 2.4 gegeben ist, mit „WHILE Band $i \neq 0$ DO M“. Die Arbeitsweise ist einfach zu erkennen.

Wie wir aus obigem Bespiel erkennen, können wir durch unsere eingeführten Bezeichnungen bereits verschiedene einfache Programmiersprachen-ähnliche Konzepte mit einer Mehrband-Turingmaschine simulieren. Dabei können die Bandinhalte als Variablenwerte angesehen werden. Es gibt einfache Wertzuweisungen, Hintereinanderreihung von Programmteilen ist möglich, und einfache Abfrage und WHILE-Schleifen können wir „programmieren“. Wir möchten daran erinnern, dass wir all dieses natürlich auch mit Einband-Turingmaschinen simulieren können.

2.3 LOOP-, WHILE- und GOTO-Berechenbarkeit

Nachdem wir im Kapitel 2.2 die Turingmaschine zur Annäherung an den Begriff *berechenbare Funktion* eingeführt haben, betrachten wir in diesem Kapitel einfache *Programmiersprachen.*

Zunächst führen wir LOOP-Programme in drei Stufen ein.

Zunächst werden die *syntaktischen Komponenten*, der *Zeichensatz* festgelegt, bevor die Syntax induktiv und schließlich die Semantik definiert wird.

Definition 2.21 LOOP-*Programme bestehen aus folgenden Zeichen* (*syntaktischen Komponenten*) :

- *Variablen:* $x_0 \;\; x_1 \;\; x_2 \;\; \ldots$
- *Konstanten:* $0 \;\; 1 \;\; 2 \;\; \ldots$
- *Operationssymbole:* $+ \;\; -$
- *Trennsymbole:* $; \;\; :=$
- *Schlüsselwörter:* LOOP DO END

Definition 2.22 *Die Syntax von* LOOP-*Programmen wird wie folgt induktiv definiert.*

(i) *Jede Wertzuweisung der Form*

$$x_i := x_j + c \qquad \textit{bzw.} \qquad x_i := x_j - c$$

ist ein LOOP-*Programm, wobei c eine Konstante ist.*

(ii) *Sind* P_1, P_2 LOOP-*Programme, dann sind auch*

$$P_1; P_2$$

sowie

$$\text{LOOP } x_i \text{ DO } P_1 \text{ END}$$

LOOP-*Programme.*

Definition 2.23 *Die Semantik von* LOOP-*Programmen ist wie folgt definiert.*

(i) *Jede Wertzuweisung der Form*

$$x_i := x_j + c$$

wird wie „üblich“ interpretiert: der neue Wert der Variablen x_i berechnet sich als Summe des Wertes der Variablen x_j und der Konstanten c, wobei der Wert in der Variablen x_j erhalten bleibt.
Die Wertzuweisung

$$x_i := x_j - c$$

wird analog interpretiert, wobei sich aber die Werte nach der sogenannten modifizierte Differenz „$\dot{-}$“, die wie folgt definiert ist

$$n_1 \dot{-} n_2 = \begin{cases} n_1 - n_2 \textit{ falls } n_1 \geq n_2, \\ 0 \qquad \textit{sonst,} \end{cases}$$

berechnen.

(ii) *Ein* LOOP*-Programm der Form* $P_1; P_2$ *soll die Hintereinanderausführung der Programme* P_1 *und* P_2 *bedeuten, also zuerst wird das Programm* P_1*, dann das Programm* P_2 *ausgeführt. Ein* LOOP*-Programm der Form* LOOP x_i DO P_1 END *bedeutet, dass das Programm* P_1 *sooft ausgeführt wird, wie der Wert der Variablen* x_i *zu Beginn angibt. Änderungen des Wertes der Variablen* x_i *haben also keinen Einfluss auf die Anzahl der Wiederholungen.*

Wir können jedem LOOP-Programm P eine Funktion zuordnen, nämlich die von diesem LOOP-Programm P *berechnete Funktion.* Präzisiert wird das mit dem Begriff der LOOP-berechenbaren Funktion.

Definition 2.24 *Eine Funktion* $f\colon \mathbb{N}^k \to \mathbb{N}$, $k \in \mathbb{N}$, *heißt* LOOP*-berechenbar, falls es ein* LOOP*-Programm* P *gibt, das* f *in dem Sinne berechnet, dass* P*, gestartet mit* $n_1, n_2, \ldots, n_k$ *in den Variablen* $x_1, x_2, \ldots, x_k$ *und* 0 *in den restlichen Variablen, mit dem Wert* $f(n_1, n_2, \ldots, n_k)$ *in der Variablen* x_0 *stoppt.*

Betrachten wir dazu einige Beispiele.

Beispiel 2.25 Gegeben sei das LOOP-Programm

$x_0 := x_1 + 0;$
LOOP x_2 DO $x_0 := x_0 + 1$ END

Man erkennt leicht, dass das Programm mit dem Wert der Summe der Anfangsbelegungen der Variablen x_1 und x_2 in der Variablen x_0 stoppt. Es berechnet also die Addition $+ : \mathbb{N}^2 \to \mathbb{N}$ vermöge $(x_1, x_2) \mapsto +(x_1, x_2) = x_1 + x_2$.

Also ist die Addition LOOP-berechenbar.

Bemerkung 2.26 Im obigen Beispiel 2.25 lautet die erste Programmzeile

$$x_0 := x_1 + 0$$

Normalerweise würde man dafür

$$x_0 := x_1$$

schreiben. Das wollen wir in Zukunft auch machen. D. h., wir werden die enge Definition der Wertzuweiseung in der Definition der Syntax von LOOP-Programmen etwas aufweiten, indem wir auch Wertzuweisungen der Form

$x_i := x_j$ und
$x_i := c$

zulassen wollen, da wir natürlich diese durch die Programme

$$x_i := x_j + 0$$

bzw.

$x_k = 0;$
$x_i := x_k + c$

simulieren können.

Beispiel 2.27 Gegeben sei das LOOP-Programm

LOOP x_2 DO
LOOP x_1 DO $x_0 := x_0 + 1$ END
END

Eine genaue Betrachtung des Programms zeigt, dass damit die Funktion $\cdot : \mathbb{N}^2 \to \mathbb{N}$ vermöge $(x_1, x_2) \mapsto \cdot(x_1, x_2) = x_1 \cdot x_2$, also die Multiplikation zweier Zahlen berechnet wird, die damit also LOOP-berechenbar ist.

Man beachte, dass die Anfangsbelegung der Variablen x_0 natürlich laut Definition 0 ist. Das wird hier gebraucht und verwendet.

Bemerkung 2.28 Wir haben es im Beispiel 2.27 mit zwei ineinander verschachtelten LOOP-Schleifen zu tun. Man erkennt, dass die innere Schleife eigentlich die Addition aus Beispiel 2.25 ist. Wir könnten also das Programm im Beispiel 2.27 im Prinzip auch als

LOOP x_2 DO $x_0 := x_0 + x_1$ END

schreiben, wobei das natürlich kein „LOOP-Programm“ im strengen Sinne der Definition ist. Da wir allerdings einmal nachgewiesen haben, dass die Addition „$x_0 := x_0 + x_1$“ durch ein LOOP-Programm berechnet werden kann, wollen wir im Folgenden solche Konstruktionen verwenden und sind uns dabei bewusst, dass wir eigentlich dafür die vollständigen LOOP-Programme benutzen müssten, wobei wir natürlich auf den korrekten Gebrauch der einzelnen Variablen achten müssen und uns immer bewusst sein müssen, dass es bei LOOP-Programmen ausschließlich „globale“ Variable gibt.

Die Syntax der LOOP-Programme ist sehr „kurz“, d. h., viele Kontrukte höherer Programmiersprachen wie z. B. IF-THEN-ELSE stehen nicht zur Verfügung. Ist das nun eine Einschränkung der LOOP-Programme?

Betrachten wir etwa die Anweisung

IF $x_1 = 0$ THEN A ELSE B END

wobei A und B LOOP-Programme sein sollen. Können wir dann diese Anweisung durch ein LOOP-Programm simulieren? Die Antwort gibt folgendes Beispiel.

Beispiel 2.29 Das Konstrukt

IF $x_1 = 0$ THEN A ELSE B END

wird durch das LOOP-Programm

$x_2 := 1;\ x_3 := 0;$
LOOP x_1 DO $x_2 := 0;\ x_3 := 1$ END;
LOOP x_2 DO A END;
LOOP x_3 DO B END

simuliert. Dabei sind die Variablen x_2 und x_3 natürlich nicht in den Programmen A und B enthalten.

Bemerkung 2.30 Wie wir eben gesehen haben, können wir IF-THEN-ELSE Anweisungen durch LOOP-Programme simulieren. Ist die Abfragebedingung komplizierter, so müssen wir natürlich die Simulation anpassen, aber man erkennt nach einiger Übung, dass man dieses entsprechend formulieren kann.

Deshalb werden wir auch bei der Angabe von speziellen LOOP-Programmen, die wir noch benutzen werden, solche IF-THEN-ELSE-Konstrukte verwenden, um Schreibarbeit zu sparen. Dabei haben wir natürlich das Wissen, dass wir daraus problemlos „reine“ LOOP-Programme konstruieren können.

Eine genaue Betrachtung der LOOP-Programme zeigt, dass bei der Abarbeitung jede LOOP-Schleife nur endlich oft durchlaufen werden kann. Insgesamt gibt es natürlich in einem LOOP-Programm auch nur endlich viele LOOP-Schleifen, also stoppt jedes LOOP-Programm. Das aber heißt, die von LOOP-Programmen berechneten Funktionen enthalten keine undefinierten Stellen. Also:

Lemma 2.31 *Jede von einem* LOOP*-Programm berechnete Funktion ist total.* □

Das aber wiederum hat natürlich folgende Konsequenz.

Folgerung 2.32 *Es gibt Funktionen, die nicht* LOOP*-berechenbar sind.* □

Etwas schwieriger ist folgendes Lemma zu zeigen, das wir hier deshalb auch nur nennen wollen.

Lemma 2.33 *Es gibt totale Funktionen, die nicht* LOOP*-berechenbar sind.* □

Dem interessierten Leser sei gesagt, dass zum Beispiel die sogenannte *Ackermannfunktion* eine solche ist. Einen Beweis dafür kann man zum Beispiel in [13] finden.

Wir wollen wegen oben genannter Unzulänglichkeiten der LOOP-Programme dieses Konzept erweitern, indem wir WHILE-Schleife einführen und so zu den sogenannten WHILE-Programmen kommen.

Wir geben hier noch einmal die vollständige Definition von WHILE-Programmen an, wobei es sich immer nur um kleinere Erweiterungen der LOOP-Programme handelt.

Definition 2.34 WHILE*-Programme bestehen aus folgenden Zeichen (syntaktischen Komponenten)* :

- *Variablen:* $x_0 \;\; x_1 \;\; x_2 \;\; \ldots$
- *Konstanten:* $0 \;\; 1 \;\; 2 \;\; \ldots$
- *Operationssymbole:* $+ \;\; -$
- *Trennsymbole:* $; \;\; := \;\; \neq$
- *Schlüsselwörter:* LOOP WHILE DO END

Definition 2.35 *Die Syntax von* WHILE*-Programmen wird wie folgt induktiv definiert.*

(i) *Jede Wertzuweisung der Form*

$$x_i := x_j + c \qquad \textit{bzw.} \qquad x_i := x_j - c$$

ist ein WHILE*-Programm, wobei c eine Konstante ist.*

(ii) *Sind* P_1, P_2 WHILE*-Programme, dann sind auch*

$$P_1; P_2$$

sowie

$$\text{LOOP } x_i \text{ DO } P_1 \text{ END}$$

und

$$\text{WHILE } x_i \neq 0 \text{ DO } P_1 \text{ END}$$

WHILE*-Programme.*

Definition 2.36 *Die Semantik von* WHILE*-Programmen ist wie folgt definiert.*

(i) *Jede Wertzuweisung der Form*

$$x_i := x_j + c$$

wird wie „üblich“ interpretiert: der neue Wert der Variablen x_i berechnet sich als Summe des Wertes der Variablen x_j und der Konstanten c, wobei der Wert in der Variablen x_j erhalten bleibt.

Die Wertzuweisung

$$x_i := x_j - c$$

wird analog interpretiert, wobei sich aber die Werte nach der sogenannten modifizierte Differenz „ $\dot{-}$ “, die wie folgt definiert ist

$$n_1 \dot{-} n_2 = \begin{cases} n_1 - n_2 & \textit{falls } n_1 \geq n_2, \\ 0 & \textit{sonst,} \end{cases}$$

berechnen.

(ii) *Ein* WHILE-*Programm der Form* $P_1; P_2$ *soll die Hintereinanderausführung der Programme* P_1 *und* P_2 *bedeuten, also zuerst wird das Programm* P_1*, dann das Programm* P_2 *ausgeführt.*
Ein WHILE-*Programm der Form* LOOP x_i DO P_1 END *bedeutet, dass das Programm* P_1 *sooft ausgeführt wird, wie der Wert der Variablen* x_i *zu Beginn angibt. Änderungen des Wertes der Variablen* x_i *haben also keinen Einfluss auf die Anzahl der Wiederholungen.*
Ein WHILE-*Programm der Form* WHILE $x_i \neq 0$ DO P_1 END *bedeutet, dass das Programm* P_1 *solange ausgeführt wird, wie der Wert der Variablen* x_i *ungleich Null ist. Es findet also vor jedem erneuten Durchlauf des Programms* P_1 *eine Abfrage der Variablen* x_1 *statt.*

Wir können wiederum jedem WHILE-Programm P eine Funktion zuordnen, nämlich die von diesem WHILE-Programm P *berechnete Funktion*. Präzisiert wird das mit dem Begriff der WHILE-berechenbaren Funktion.

Definition 2.37 *Eine Funktion* $f : \mathbb{N}^k \to \mathbb{N}$, $k \in \mathbb{N}$, *heißt* WHILE-*berechenbar, falls es ein* WHILE-*Programm* P *gibt, das* f *in dem Sinne berechnet, dass* P*, gestartet mit* $n_1, n_2, \ldots, n_k$ *in den Variablen* $x_1, x_2, \ldots, x_k$ *und* 0 *in den restlichen Variablen, mit dem Wert* $f(n_1, n_2, \ldots, n_k)$ *in der Variablen* x_0 *stoppt. Ist* $f(n_1, n_2, \ldots, n_k)$ *dagegen nicht definiert, so stoppt* P *nicht.*

Da die Definition der WHILE-Programme die LOOP-Programme auch enthalten, haben wir

Folgerung 2.38 *Jede* LOOP-*berechenbare Funktion ist* WHILE-*berechenbar.* □

Betrachten wir ein Beispiel.

Beispiel 2.39 Das WHILE-Programm

$x_3 := x_1 - 5$;
WHILE $x_3 \neq 0$ DO $x_1 := x_1 + 1$ END;
LOOP x_1 DO $x_0 := x_0 + 1$ END;
LOOP x_2 DO $x_0 := x_0 + 1$ END

berechnet die Funktion $f\colon \mathbb{N}^2 \to \mathbb{N}$ vermöge

$$f(x_1, x_2) = \begin{cases} x_1 + x_2 & \text{falls } x_1 \leq 5, \\ \textit{nicht definiert} & \text{sonst.} \end{cases}$$

Aus dem Beispiel, das zeigt, dass WHILE-Programme auch echt partielle Funktionen berechnen können haben wir sofort:

Folgerung 2.40 *Es gibt* WHILE-*berechenbare Funktionen, die nicht* LOOP-*berechenbar sind.* □

Bemerkung 2.41 Sei LOOP x_i DO P END ein LOOP-Programm. Sei weiterhin y eine Variable, die in dem Programm P nicht vorkommt. Dann wird offensichtlich diese LOOP-Schleife durch das WHILE-Programm

$y := x_i$;
WHILE $y \neq 0$ DO P; $y := y - 1$ END

simuliert. Also benötigen wir in der Programmiersprache WHILE die LOOP-Schleife eigentlich nicht mehr.

Im Kapitel 2.2 wurde angedeutet, dass Wertzuweisungen, Hintereinanderschalten von Turingmaschinen und WHILE-Schleifen auf einer Mehrband-Turingmaschine simulierbar sind. Hierbeir entspricht dem i-ten Band der Turingmaschine gerade die Variable x_1 des WHILE-Programms, wobei der Wert einer Variablen auf dem Band als Binärzahl dargestellt wird.

Schließlich kann noch jede Mehrband-Turingmaschine durch eine „normale“ Turingmaschine (also 1-Band-Turingmaschine) simuliert werden.

Damit erhalten wir:

Satz 2.42 *Jede* WHILE*-berechenbare Funktion ist Turingberechenbar.* □

Wir wollen jetzt versuchen, die Umkehrung zu zeigen, dabei ist es günstig, wenn wir noch einen Zwischenschritt einfügen, nämlich die GOTO-*Berechenbarkeit.*

Definition 2.43 GOTO*-Programme bestehen aus folgenden Zeichen (syntaktischen Komponenten)* :

- *Variablen:* x_0 x_1 x_2 ...
- *Konstanten:* 0 1 2 ...
- *Operationssymbole:* + −
- *Trennsymbole:* : ; := =
- *Marken:* M_1 M_2 M_3 ...
- *Schlüsselwörter:* GOTO IF THEN HALT

Definition 2.44 *Die Syntax von* GOTO*-Programmen wird wie folgt definiert. Ein* GOTO*-Programm besteht aus einer endlichen Folge von Anweisungen A_i, die jeweils durch eine Marke M_i eingeleitet werden:*

$M_1 : A_1$;
$M_2 : A_2$;
$\vdots$
$M_k : A_k$

Dabei ist jede Anweisung A_i, $1 \leq i \leq k$ von folgender Form:

- Wertzuweisung: $x_i := x_j + c$ *oder* $x_i := x_j - c$*, wobei c eine Konstante ist,*
- unbedingter Sprung: GOTO M_i*,*
- bedingter Sprung: IF $x_i = c$ THEN GOTO M_i *oder*
- Stopanweisung: HALT.

Definition 2.45 *Die Semantik von* GOTO*-Programmen ist wie „üblich“ definiert. Lautet eine Programmzeile*

- M_i: $x_j := x_k + c$, *so wird die Wertanweisung* $x_j := x_k + c$ *gemäß der Semantik der* LOOP- *oder* WHILE-*Programme ausgeführt. Danach wird die nächste Programmzeile abgearbeitet.*
- M_i: GOTO M_j, *wird als nächste Programmzeile die mit der Marke* M_j *ausgeführt.*
- M_i: IF $x_j = c$ THEN GOTO M_k, *wird als nächste Programmzeile die mit der Marke* M_k *ausgeführt, falls der Wert der Variablen* x_j *gleich* c *ist, sonst wird die folgende Programmzeile abgearbeitet.*
- M_i: HALT, *stoppt das Programm.*

Die von GOTO-Programmen berechnete Funktionen werden analog den WHILE-Programmen definiert, dabei ist klar, dass GOTO-Programme auch in eine unendliche Schleife (M_i: GOTO M_i) geraten können.

Definition 2.46 *Eine Funktion* $f : \mathbb{N}^k \to \mathbb{N}$, $k \in \mathbb{N}$, *heißt* GOTO-*berechenbar, falls es ein* GOTO-*Programm* P *gibt, das* f *in dem Sinne berechnet, dass* P, *gestartet mit* $n_1, n_2, \ldots, n_k$ *in den Variablen* $x_1, x_2, \ldots, x_k$ *und* 0 *in den restlichen Variablen, mit dem Wert* $f(n_1, n_2, \ldots, n_k)$ *in der Variablen* x_0 *stoppt. Ist* $f(n_1, n_2, \ldots, n_k)$ *dagegen nicht definiert, so stoppt* P *nicht.*

Beispiel 2.47 Eine WHILE-Schleife

WHILE $x_i \neq 0$ DO P END

kann durch folgendes GOTO-Programm simuliert werden.

M_1: IF $x_i = 0$ THEN GOTO M_4;
M_2: P;
M_3: GOTO M_1;
M_4: ...

Bemerkung 2.48 Oft werden in GOTO-Programmen nur die Marken aufgeschrieben, die wirklich gebraucht werden, nämlich durch einen Sprung. Das GOTO-Programm im obigen Beispiel würde damit folgende Gestalt haben:

M_1: IF $x_i = 0$ THEN GOTO M_4;
P;
GOTO M_1;
M_4: ...

Mit dem Beispiel 2.47 erhalten wir sofort:

Satz 2.49 *Jede* WHILE-*berechenbare Funktion ist auch* GOTO-*berechenbar.* □

Wir wollen jetzt die Umkehrung zeigen.

Beispiel 2.50 Gegeben sei das GOTO-Programm

M_1: A_1;
M_2: A_2;
⋮
M_k: A_k

Wir simulieren dies durch ein WHILE-Programm mit *nur einer* WHILE-Schleife wie folgt:

$count := 1;$
WHILE $count \neq 0$ DO
IF $count = 1$ THEN A'_1 END;
IF $count = 2$ THEN A'_2 END;
$\vdots$
IF $count = k$ THEN A'_k END
END

wobei die Anweisungen A'_i folgendermaßen definiert werden:

$$A'_i = \begin{cases} x_j := x_\ell + c;\ count := count + 1 & \text{falls } A_i = x_j := x_\ell + c \\ x_j := x_\ell - c;\ count := count + 1 & \text{falls } A_i = x_j := x_\ell - c \\ count := n & \text{falls } A_i = \text{GOTO } M_n \\ \text{IF } x_j = c \text{ THEN } count := n & \\ \qquad \text{ELSE } count := count + 1 \text{ END} & \text{falls } A_i = \text{IF } x_j = c \text{ THEN GOTO } M_n \\ count := 0 & \text{falls } A_i = \text{HALT} \end{cases}$$

Aus Beispiel 2.50 erhalten wir sofort folgenden Satz.

Satz 2.51 *Jede* GOTO-*berechenbare Funktion ist auch* WHILE-*berechenbar.* □

Wir heben den Aspekt, dass die Simulation mit nur einer WHILE-Schleife auskommt, im folgenden Satz besonders hervor.

Satz 2.52 (Kleene Normalform für WHILE-Programme) *Jede* WHILE-*berechenbare Funktion kann durch ein* WHILE-*Programm mit nur einer* WHILE-*Schleife berechnet werden.*

Beweis. Sei P ein beliebiges WHILE-Programm zur Berechnung der Funktion f. Wir simulieren P zunächst durch ein GOTO-Programm P' gemäß Beispiel 2.47. Dann simulieren wir das GOTO-Programm P' durch ein WHILE-Programm P'' gemäß Beispiel 2.50. Offensichtlich berechnet das WHILE-Programm P'' dann die Funktion f und besitzt nur eine WHILE-Schleife. □

Jetzt zeigen wir noch, dass Turingmaschinen durch GOTO-Programme simuliert werden können.

Satz 2.53 *Jede Turingberechenbare Funktion ist auch* GOTO-*berechenbar.*

Beweis. Sei $M = (Z, \Sigma, \Gamma, \delta, z_1, \square, E)$ eine Turingmaschine zur Berechnung einer Funktion f. Wir simulieren M durch ein GOTO-Programm, das folgendermaßen aufgebaut ist:

$M_1 : P_1;$
$M_2 : P_2;$
$M_3 : P_3$

Hierbei transformiert P_1 die eingegebenen Anfangswerte der Variablen in Binärdarstellung und erzeugt eine Darstellung der Startkonfiguration von M, die sich in den Variablenwerten dreier Variablen x, y, z widerspiegelt. Wir geben diese Codierung von Turingmaschinen-Konfigurationen in drei natürliche Zahlen gleich im Detail an.

P_2 führt eine Schritt-für-Schritt-Simulation der Rechnung von M durch – durch entsprechendes Verändern der Variablenwerte von x, y und z.

P_3 schließlich erzeugt aus der codierten Form der Endkonfiguration in x, y, z die eigentliche Ausgabe in der Ausgabevariablen x_0

Man beachte, dass P_1 und P_3 gar nicht von M abhängen, sondern nur P_2.

Betrachten wir jetzt die schon erwähnte Codierung einer Konfiguration einer Turingmaschine M. Seien die Mengen

$$Z = \{z_1, z_2, \ldots, z_k\} \text{ und}$$
$$\Gamma = \{a_1, a_2, \ldots, a_m\}$$

durchnumeriert. Sei außerdem b eine Zahl mit $b > |\Gamma|$. Dann repräsentieren wir eine Turingmaschinen-Konfiguration

$$a_{i_1} a_{i_2} \ldots a_{i_p} z_\ell a_{j_1} a_{j_2} \ldots a_{j_q}$$

dadurch, dass die drei Programmvariablen x, y, z die Werte

$$x = (i_1 i_2 \ldots i_p)_b$$
$$y = (j_q j_{q-1} \ldots j_1)_b$$
$$z = \ell$$

annehmen, dabei bedeutet $(i_1 i_2 \ldots i_p)_b$ die Zahl $i_1 i_2 \ldots i_p$ in b-närer Darstellung, also

$$x = \sum_\mu = 1^p i_\mu \cdot b^{p-\mu}.$$

Analoges gilt für y (die Ziffern stehen hier jedoch in umgekehrter Reihenfolge).

Das GOTO-Programmstück $M_2 : P_2$ hat nun folgende Form:

M_2: $a := y \bmod b$;
IF $(z = 1)$ AND $(a = 1)$ THEN GOTO $M_{1,1}$;
IF $(z = 1)$ AND $(a = 2)$ THEN GOTO $M_{1,2}$;
$\vdots$
IF $(z = k)$ AND $(a = m)$ THEN GOTO $M_{k,m}$;
$M_{1,1}$: $\circledast$;
GOTO M_2;
$M_{1,2}$: $\circledast$;
GOTO M_2;
$\vdots$
$M_{k,m}$: $\circledast$;
GOTO M_2

Wir beschreiben nun, was an den mit $\circledast$ bezeichneten Stellen passiert. Greifen wir repräsentativ das Programmstück, das mit der Marke M_{ij} beginnt, heraus. Nehmen wir an, dass die entsprechende δ-Anweisung

$$\delta(z_i, a_j) = (z_{i'}, a_{j'}, L)$$

lautet. Dies kann durch folgende Anweisungen simuliert werden:

$z := i'$;
$y := y \text{ div } b$;
$y := b * y + j'$;
$y := b * y + (x \bmod b)$;
$x := x \text{ div } b$

Entsprechend kann man sich die anderen Fälle vorstellen.

Falls z_i Endzustand ist, so setzen wir für $\circledast$ einfach

GOTO M_3

Die Konstruktionen von P_1 und P_3 sind nun entsprechend auszuführen und überlassen wir dem geneigten Leser. □

2.4 Die Churchsche These

Wir stellen die Ergebnisse aus dem vorangegangenen Kapitel noch mal im Folgenden zusammenfassenden Satz dar.

Satz 2.54 *Sei $f : \mathbb{N}^k \to \mathbb{N}$ eine Funktion, dann sind folgende Aussagen äquivalent:*

(i) *f ist* WHILE-*berechenbar.*

(ii) *f ist Turing-berechenbar.*

(iii) *f ist* GOTO-*berechenbar.* □

Außerdem erinnern wir, dass jede LOOP-berechenbare Funktion zwar WHILE-berechenbar ist, aber die Umkehrung nicht gilt.

Es gibt neben den von uns hier betrachteten Berechenbarkeitsbegriffen noch viele andere Begriffe, die in den letzten Jahrzehnten betrachtet wurden. Dazu zählen z. B. Berechenbarkeitsbegriffe, die auf *Flussdiagramme*, *Registermaschinen*, *partiell-rekursive Funktionen* zurückgeführt werden. Erstaunlicherweise musste man erkennen, dass all diese Begriffe nicht über den Begriff der Turingberechenbarkeit hinausführen. Das führt zu der Tatsache, dass man heute davon ausgeht, dass man mit dem Begriff der Turingberechenbarkeit genau den eingangs besprochenen Begriff der intuitiven Berechbarkeit getroffen hat.

Diese Überzeugung fasst man unter dem Namen *Churchche*[6] These zusammen:

Churchsche These *Jede intuitiv berechenbare Funktion ist Turing-berechenbar.*

Diese These ist *nicht* beweisbar, da der intuitive Berechenbarkeitsbegriff eben nicht formal mathematisch erfasst werden kann. Allerdings könnte man sie sofort widerlegen, indem man einen Berechenbarkeitsbegriff angibt, der eben nicht durch Turingmaschinen simuliert werden kann – das aber wurde in den vergangenen Jahrzehnten (die historisch ersten Definitionen von Turing und Church gehen auf 1936 zurück) immer wieder versucht, allerdings bis heute vergeblich, so dass man, wie oben schon ausgeführt, heute allgemein von der Richtigkeit der These ausgeht.

[6] A. Church, Amerikanischer Mathematiker

2.5 Halteproblem und Unentscheidbarkeit

Der Berechenbarkeitsbegriff ist auf Funktionen zugeschnitten. Wir wollen nun einen entsprechenden Begriff für Mengen einführen.

Definition 2.55 *Eine Menge $A \subseteq \Sigma^*$ heißt entscheidbar, falls die charakteristische Funktion von A, nämlich $\chi_A \colon \Sigma^* \to \{0,1\}$, berechenbar ist. Hierbei ist für alle $w \in \Sigma^*$:*

$$\chi_A(w) = \begin{cases} 1 & \textit{falls } w \in A, \\ 0 & \textit{falls } w \notin A, \end{cases}$$

Definition 2.56 *Eine Menge $A \subseteq \Sigma^*$ heißt semi-entscheidbar, falls die „halbe“ charakteristische Funktion von A, nämlich $\chi'_A \colon \Sigma^* \to \{0,1\}$, berechenbar ist. Hierbei ist für alle $w \in \Sigma^*$:*

$$\chi'_A(w) = \begin{cases} 1 & \textit{falls } w \in A, \\ \textit{nicht definiert} & \textit{falls } w \notin A, \end{cases}$$

Beide Definitionen lassen sich natürlich auch für Teilmengen $A \subseteq \mathbb{N}$ erweitern.

Bemerkung 2.57 Im Zusammenhang mit der Frage der Entscheidbarkeit werden Mengen oft auch als *Entscheidungs-Probleme* dargestellt (in der Form: gegeben – gefragt).

Der Menge

$$\{x \in \mathbb{N} \mid x \text{ gerade}\}$$

würde dann das Problem

Gegeben: $x \in \mathbb{N}$
Frage: Ist x gerade?

entsprechen.

Bildhaft gesprochen besagen die beiden Definitionen, dass im ersten Fall ein immer stoppender Algorithmus zur Verfügung steht, der das Entscheidungsproblem für A löst (siehe Abbildung 2.5).

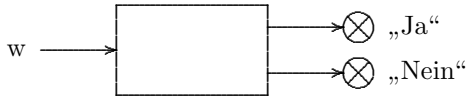


Abbildung 2.5: Veranschaulichung von entscheidbaren Mengen

Im Fall der Semi-Entscheidbarkeit sieht das Bild so aus, dass der Algorithmus *nur einen* definitiven Ausgang hat. Falls der Algorithmus also für lange Zeit nicht gestoppt hat, so ist es nicht klar, ob der „Nein“-Fall ($w \notin A$) vorliegt, oder ob er doch noch mit der Ausgabe „Ja“ stoppt (siehe Abbildung 2.6).

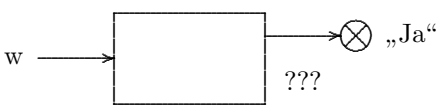


Abbildung 2.6: Veranschaulichung von semi-entscheidbaren Mengen

In diesem Fall ist die Situation also etwas unbefriedigend, aber bei vielen algorithmischen Problemen ist es das Beste, was erreichbar ist (z. B. bei Entscheidungsverfahren für die Prädikatenlogik, „Theorembeweisern“).

Mit Hilfe von Turingmaschinen lassen sich die Begriffe Entscheidbarkeit bzw. Semi-Entscheidbarkeit mit Hilfe der folgenden Sätze charakterisieren.

Satz 2.58 *Eine Menge $A \subseteq \Sigma^*$ ist genau dann entscheidbar, wenn es eine Turingmaschine M mit der Menge der Endzustände $\{ja, nein\}$ gibt, die für jedes Wort $w \in \Sigma^*$ einen Endzustand erreicht und genau dann den Endzustand ja erreicht, wenn $w \in A$ gilt.*
Sprechweise: M entscheidet *A.*

Satz 2.59 *Eine Menge $A \subseteq \Sigma^*$ ist genau dann semi-entscheidbar, wenn es eine Turingmaschine M gibt, die für jedes Wort $w \in \Sigma^*$ genau dann einen Endzustand erreicht, wenn $w \in A$ gilt.*
Sprechweise: M akzeptiert *A.*

Zwischen Entscheidbarkeit und Semi-Entscheidbarkeit existiert der folgende offensichtliche Zusammenhang.

Satz 2.60 *Eine Menge A ist entscheidbar genau dann, wenn sowohl die Menge A als auch ihr Komplement $\overline{A}$ semi-entscheidbar sind.* □

Im folgenden vergleichen wir den Begriff der Semi-Entscheidbarkeit mit der *rekursiven Aufzählbarkeit.*

Definition 2.61 *Eine Menge $A \subseteq \Sigma^*$ heißt rekursiv aufzählbar, falls $A = \emptyset$ oder falls eine totale und berechenbare Funktion $f : \mathbb{N} \to \Sigma^*$ gibt, so dass*

$$A = \{f(0), f(1), f(2), \dots\}$$

gilt. Sprechweise: f zählt A auf. Man beachte, dass $f(i) = f(j)$ zulässig ist.

Wir bringen den folgenden Satz wiederum ohne Beweis, der geneigte Leser möge ihn in der Literatur nachlesen.

Satz 2.62 *Eine Menge ist rekursiv aufzählbar genau dann, wenn sie semi-entscheidbar ist.* □

Zusammenfassend können wir folgendes konstatieren.

Folgerung 2.63 *Sei $A \subseteq \Sigma^*$ eine Menge. Dann sind folgende Aussagen äquivalent.*

(i) *A ist rekursiv aufzählbar.*

(ii) *A ist semi-entscheidbar.*

(iii) *χ'_A ist berechenbar.*

(iv) *A ist Definitionsbereich einer berechenbaren Funktion.*

(v) *A ist Wertebereich einer berechenbaren Funktion.* □

Der Begriff der *Abzählbarkeit* von Mengen kann (siehe Definitionen 1.54 und 1.55) auch so definiert werden, dass er dem der rekursiven Aufzählbarkeit – bis auf einen kleinen, aber wichtigen Unterschied – ähnlich sieht:

Definition 2.64 *Eine Menge A heißt abzählbar, falls $A = \emptyset$ oder falls es eine Funktion f gibt, so dass*

$$A = \{f(0), f(1), f(2), \dots\}$$

gilt.

Der Unterschied ist der, dass hier nicht die *Berechenbarkeit* der Funktion f verlangt wird. Er wird bei folgendem Beispiel klar: Jede Teilmenge A' einer abzählbaren Menge

$$A = \{f(0), f(1), f(2), \dots\}$$

ist wieder abzählbar. Sei etwa $a \in A' \neq \emptyset$ ein festgehaltenes Element. Wenn wir

$$g(n) = \begin{cases} f(n) & \text{falls } f(n) \in A', \\ a & \text{sonst.} \end{cases}$$

setzen, so ist g sicher eine wohl-definierte (aber nicht notwendigerweise berechenbare) Funktion. Es gilt nun

$$A' = \{g(0), g(1), g(2), \dots\}$$

Nicht jede Teilmenge einer rekursiv aufzählbaren Menge muss dagegen wieder rekursiv aufzählbar sein.

Wir wollen nun ein paar nicht-entscheidbare Probleme kennenlernen. Bei den ersten dieser Probleme sollen Turingmaschinen selbst (in geeignet codierter Form) als Eingaben vorkommen. Wir müssen uns also kurz darum kümmern, wie man Turingmaschinen als Wort über $\{0, 1\}$ schreiben kann.

Zunächst nehmen wir an, dass die Elemente von Γ und Z durchnummeriert sind, also

$$\Gamma = \{a_0, a_1, \dots, a_k\} \qquad \text{und}$$
$$Z = \{z_0, z_1, \dots, z_\ell\},$$

wobei festgelegt sein soll, welche Nummern die Symbole $\square, 0, 1, \#$ und die Start- und Endzustände erhalten. Jeder δ-Regel der Form

$$\delta(z_i, a_j) = (z_{i'}, a_{j'}, y)$$

ordnen wir das Wort

$$w_{i,j,i',j',y} = \#\#\text{bin}(i)\#\text{bin}(j)\#\text{bin}(i')\#\text{bin}(j')\#\text{bin}(m)$$

zu, wobei

$$m = \begin{cases} 0 & \text{falls } y = L, \\ 1 & \text{falls } y = R, \\ 2 & \text{falls } y = N. \end{cases}$$

Alle diese zu δ gehörenden Wörter schreiben wir nun in beliebiger Reihenfolge hintereinander und erhalten – als Zwischenschritt – einen Code der zugrundeliegenden Turingmaschine über dem Alphabet $\{0, 1, \#\}$.

Jedem solchen Wert können wir nun noch ein Wort über $\{0, 1\}$ zuordnen, indem wir noch folgende Codierung vornehmen

$$0 \mapsto 00,$$
$$1 \mapsto 01,$$
$$\# \mapsto 11.$$

Es ist klar, dass auf diese Weise nicht jedes Wort in $\{0, 1\}^*$ ein sinnvoller Code einer Turingmaschine ist. Sei aber M_0 irgendeine beliebige feste Turingmaschine, dann können wir *für jedes* $w \in \{0, 1\}^*$ festlegen, dass M_w eine bestimmte Turingmaschine bezeichnet, nämlich

$$M_w = \begin{cases} M & \text{falls } w \text{ Codewort von } M \text{ ist}, \\ M_0 & \text{sonst.} \end{cases}$$

Definition 2.65 *Unter dem speziellen Halteproblem für Turingmaschinen verstehen wir die Menge*

$$K = \{w \in \{0,1\}^* \mid M_w \text{ angesetzt auf } w \text{ hält}\}.$$

Satz 2.66 *Das spezielle Halteproblem für Turingmaschinen ist nicht entscheidbar.*

Beweis. Angenommen, K ist entscheidbar. Dann ist χ_K berechenbar mittels einer Turingmaschine M. Diese Maschine M könnte nun leicht zu einer Turingmaschine M' umgebaut werden, die durch Abbildung 2.7 definiert ist.

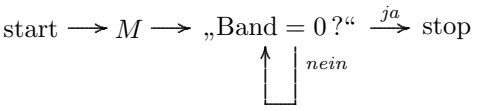


Abbildung 2.7: Die Turingmaschine M' aus dem Beweis zu Satz 2.66

Das heißt, M' stoppt genau dann, wenn M den Wert 0 ausgeben würde. Falls M den Wert 1 ausgibt, gerät M' in eine Endlosschleife. Sei w' ein Codewort der Maschine M'. Nun gilt

M' angesetzt auf w' hält
- $\Leftrightarrow$ M angesetzt auf w' gibt 0 aus (wegen Definition von M'),
- $\Leftrightarrow$ $\chi_K(w') = 0$ (da M die Menge K entscheidet),
- $\Leftrightarrow$ $w' \notin K$ (wegen Definition von χ_K),
- $\Leftrightarrow$ $M_{w'}$ angesetzt auf w' hält nicht (wegen Definition von K),
- $\Leftrightarrow$ M' angesetzt auf w' hält nicht (da w' Code von M' ist).

Dieser Widerspruch beweist, dass die Eingangsannahme falsch war, also ist K nicht entscheidbar. □

Dieses spezielle Halteproblem für Turingmaschinen kann zum (allgemeinen) *Halteproblem für Turingmaschinen* verallgemeinert werden in dem Sinne, dass wir als Eingabe eine beliebige Turingmaschine und ein beliebiges Wort zulassen und fragen, ob die Turingmaschine bei dieser Eingabe in einen Stopzustand kommt, also

Definition 2.67 *Das Halteproblem für Turingmaschinen ist die Menge*

$$H = \{w\#x \mid M_w \text{ angesetzt auf } x \text{ hält}\}.$$

Intuitiv ist klar, dass mit dem speziellen Halteproblem auch das allgemeine Halteproblem unentscheidbar ist. Es gilt nämlich $w \in K$ genau dann, wenn $w\#w \in H$. Könnte man nun H entscheiden, so könnte man das Problem K entscheiden, indem man die Eingabe w für K auf die Eingabe $w\#w$ für H transformiert und den Entscheidungsalgorithmus für H anwendet. Diese Vorgehensweise der Überführung auf ein anderes Problem wird jetzt formalisiert.

Definition 2.68 *Eine Sprache $A \subseteq \Sigma^*$ heißt* reduzierbar *auf die Sprache $B \subseteq \Gamma^*$, Notation $A \leq B$, wenn es eine totale und berechenbare Funktion $f : \Sigma^* \to \Gamma^*$ gibt, so dass*

für alle $w \in \Sigma^*$ gilt: $w \in A$ genau dann, wenn $f(w) \in B$.

In diesem Fall nennt man f eine Reduktion von A auf B.

Satz 2.69 *Es seien A und B Sprachen mit $A \leq B$.*
Ist B entscheidbar, so ist auch A entscheidbar.
Ist B semi-entscheidbar, so ist auch A semi-entscheidbar.

Beweis. Sei $f : \Sigma^* \to \Gamma^*$ eine Reduktion von A auf B. Es gilt: $\chi_A(w) = \chi_B(f(w))$ sowie $\chi'_A(w) = \chi'_B(f(w))$; aus der Berechenbarkeit von χ_B bzw. χ'_B sowie der Berechenbarkeit von f folgt die Berechenbarkeit von χ_A bzw. χ'_A. □

Die Reduktion findet häufig Anwendung in Beweisen:
Ist die Entscheidbarkeit von B bekannt, so genügt für den Beweis der Entscheidbarkeit von A die Angabe einer Reduktion von A auf B.
Ist die Unentscheidbarkeit von A bekannt, so genügt für den Beweis der Unentscheidbarkeit von B die Angabe einer Reduktion von A auf B.

Als Beispiel zeigen wir die Unentscheidbarkeit des allgemeinen Halteproblems H durch Reduktion des speziellen Halteproblems K auf H.

Satz 2.70 *Das Halteproblem für Turingmaschinen* (H) *ist nicht entscheidbar.*

Beweis. Die Funktion $f : \{0,1\}^* \to \{0,1,\#\}^*$ mit $f(w) = w\#w$ ist berechenbar. Es gilt für alle $w \in \{0,1\}^*$: $w \in K$ genau dann, wenn $f(w) \in H$, d.h. $K \leq H$. Wegen der Unentscheibarkeit von K ist auch H unentscheidbar. □

Dieses Halteproblem wiederum kann natürlich auch auf für andere Berechenbarkeitsmodelle formuliert werden, also z. B. für WHILE-Programme und GOTO-Programme. Wegen der auch dort geltenden Unentscheidbarkeit bedeutet das, verallgemeinert auf eine beliebige höhere Programmiersprache: Es ist nicht berechenbar, ob ein Computerprogramm bei einer gegebenen Eingabe hält oder nicht!

Die Unentscheidbarkeit des speziellen Halteproblems wurde bewiesen, indem Turingmaschinen durch Wörter codiert als Eingaben verwendet wurden. In diesem Zusammenhang ist auch das folgende Resultat über die Existenz *universeller Turingmaschinen* interessant.

Satz 2.71 *Es gibt eine Turingmaschine U mit dem Eingabealphabet $\{0,1,\#\}$, die für jede Eingabe $u\#v$ mit $u, v \in \{0,1\}^*$ die Ausgabe $f_{M_u}(v)$ liefert bzw. nicht stoppt, falls M_u angesetzt auf v nicht stoppt oder die Eingabe nicht die Form $u\#v$ mit $u, v \in \{0,1\}^*$ besitzt.*

Die universelle Turingmaschine U kann man als das Modell eines universellen Computers ansehen, der zu einem Programm u und einer Eingabe v des Programmes die Ausgabe berechnet. Anzumerken ist, dass U effektiv konstruiert werden kann.

Satz 2.72 *Das Halteproblem für Turingmaschinen* (H) *ist semi-entscheidbar.*

Beweis. Die universelle Turingmaschine U stoppt für eine Eingabe $u\#v$ mit $u, v \in \{0,1\}^*$ genau dann, wenn M_u angesetzt auf v stoppt. Das heißt, H wird durch U akzeptiert.
(Intuitiv setzt man einfach M_u auf v an.) □

Ein weiteres unentscheidbares Problem ist das sogenannte X. Hilbertsche[7] Problem:

Definition 2.73 *Das X. Hilbertsche Problem ist definiert durch:*

Gegeben: $n \in \mathbb{N}$, *ein Polynom* $p(x_1, \ldots, x_n)$ *in* n *Unbekannten,*
Frage: *Besitzt* p *ganzzahlige Lösungen?*

Satz 2.74 *Das X. Hilbertsche Problem ist nicht entscheidbar.* □

Kommen wir nun zu einem Problem, das als *Postsches*[8] *Korrespondenzproblem* (PKP) bezeichnet wird.

[7] David Hilbert (1862–1943), deutscher Mathematiker, formulierte dieses Problem neben anderen auf dem Mathematikerkongress 1900 in Paris.
[8] Emil Post, Mathematiker.

Definition 2.75 *Das Postsche Korrespondenzproblem ist definert durch:*

Gegeben: *Alphabet* A, $k \in \mathbb{N}$ *sowie die Folge von Wortpaaren* $(x_1, y_1), (x_2, y_2), \ldots, (x_k, y_k)$ *mit* $x_i, y_i \in A^+$ *für* $1 \leq i \leq k$.

Frage: *Gibt es eine Folge von Indizes* $i_1, i_2, \ldots, i_n$ *mit* $i_j \in \{1, 2, \ldots, k\}$ *für* $1 \leq j \leq n$, $n \in \mathbb{N}$, *so dass* $x_{i_1} x_{i_2} \ldots x_{i_n} = y_{i_1} y_{i_2} \ldots y_{i_n}$ *gilt?*

Beispiel 2.76 Das Korrespondenzproblem

$$K = ((1, 101), (10, 00), 011, 11)),$$

also

$$\begin{array}{lll} x_1 = 1 & x_2 = 10 & x_3 = 011 \\ y_1 = 101 & y_2 = 00 & y_3 = 11 \end{array}$$

besitzt die Lösung $(1, 3, 2, 3)$, denn es gilt

$$x_1 x_3 x_2 x_3 = 101110011 = y_1 y_3 y_2 y_3.$$

Das Postsche Korrespondenzproblem besitzt ein hohes Maß an „Komplexität“, wie das folgende harmlos aussehende Beispiel zeigt.

Beispiel 2.77 Gegeben ist folgende Belegung des PKP:

$$\begin{array}{llll} x_1 = 001 & x_2 = 01 & x_3 = 01 & y_4 = 10 \\ y_1 = 0 & y_2 = 011 & y_3 = 101 & y_4 = 001. \end{array}$$

Dieses Problem besitzt eine Lösung, aber die kürzeste Lösung besteht aus 66 Indizes, nämlich

$$2, 4, 3, 4, 4, 2, 1, 2, 4, 3, 4, 3, 4, 4, 3, 4, 4, 2, 1, 4, 4, 2, 1, 3, 4, 1, 1, 3, 4, 4, 4, 2, 1,$$
$$2, 1, 1, 1, 3, 4, 3, 4, 1, 1, 1, 4, 4, 2, 1, 4, 1, 1, 3, 4, 1, 1, 3, 1, 1, 3, 1, 2, 1, 4, 1, 1, 3$$

Der naive Algorithmus, der bei gegebener Eingabe (x_1, y_1), (x_2, y_2), $\ldots$, (x_k, y_k) systematisch alle immer länger werdende Indexfolgen $i_1, i_2, \ldots, i_n$ daraufhin untersucht, ob sie eine Lösung darstellen und im positiven Fall stoppt, demonstriert, dass das PKP semi-entscheidbar ist. Bei Eingaben, die keine Lösung besitzen stoppt das Verfahren allerdings nicht. Man kann beweisen, dass es kein Verfahren gibt, das das PKP entscheidet, also

Satz 2.78 *Das Postsche Korrespondenzproblem ist nicht entscheidbar.* □

3 Komplexitätstheorie

3.1 Komplexitätsklassen

Im Kapitel 2 haben wir verschiedene Berechenbarkeitsbegriffe für Funktionen definiert und ihre Äquivalenz festgestellt. Leider haben wir auch gelernt, dass es Funktionen gibt, die *nicht* berechenbar sind.

In diesem Kapitel nun beschäftigen wir uns nur mit den *berechenbaren* Funktionen und interessieren uns nun dafür, wie *schwierig* es ist, sie zu berechnen, also welche *Ressourcen* zu ihrer Berechnung benötigt werden. Die wichtigsten Ressourcen sind sicherlich *Rechenzeit und Speicherplatz*. Das findet seinen Niederschlag in der Betrachtung von *Zeit- und Raumkomplexität* von Algorithmen und Problemen.

Bevor wir uns jedoch mit diesen Komplexitäten beschäftigen, möchten wir hier die LANDAUschen Symbole angeben, um Funktionen vergleichen zu können:

Definition 3.1 *Seien f und g zwei Funktionen $f, g\colon \mathbb{N} \to \mathbb{R}$.*

(i) *Wir sagen $f = O(g)$ genau dann, wenn $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$ mit einem $c \geq 0$ gilt.*

(ii) *Wir sagen $f = \Theta(g)$ genau dann, wenn $f = O(g)$ und $g = O(f)$ gelten.*

Eine dazu äquivalente Definition, wie man zeigen kann, ist

Definition 3.2 *Seien f und g zwei Funktionen $f, g\colon \mathbb{N} \to \mathbb{R}$.*

(i) *Wir sagen $f = O(g)$ genau dann, wenn es positive Konstanten c_1 und c_2 gibt, so dass*

$$f(n) \leq c_1 \cdot g(n) + c_2$$

(ii) *Wir sagen $f = \Theta(g)$ genau dann, wenn $f = O(g)$ und $g = O(f)$ gelten.*

Gilt also $f = O(g)$, so wächst f asymptotisch nicht schneller als g; gilt $f = \Theta(g)$, so haben f und g asymptotisch gleiches Wachstum.

Welchen Einfluss die Anzahl der Rechenschritte in Abhängigkeit von der Länge der Eingabe auf die Laufzeit eines Programms hat, wird in der folgenden Tabelle verdeutlicht. Annahme: ein Programm wird berechnet, wobei die Funktion f die Anzahl der auszuführenden Operationen angibt. In Tabelle 3.1 ist die Zeit für die Ausführung angegeben, falls ein Computer eine Million Operationen pro Sekunde ausführt. (Wenn wir eine andere Geschwindigkeit des Computers annehmen, z.B. 10^9 Operationen je Sekunde, so ändern sich die Tabellenwerte nur um einen konstanten Faktor. Wir merken aber an, daß aus physikalischen Gründen eine Schranke für die Geschwindigkeit existiert.)

$f \setminus n$	5	10	50	100	200
n^2	0,000 025 s	0,0001 s	0,0025 s	0,01 s	0,04 s
n^5	0,003 125 s	0,1 s	312,5 s	3 h	89 h
2^n	0,000 032 s	0,001 024 s	36 a	10^{17} a	10^{47} a
n^n	0,003 125 s	3 h	10^{71} a	10^{185} a	10^{447} a

Tabelle 3.1: Rechenzeiten eines Computers

Definition 3.3 *Es sei M eine Mehrband-Turingmaschine mit dem Eingabealphabet Σ. Dann ist $time_M$ die Funktion $time_M : \Sigma^* \to \mathbb{N}$, wobei $time_M(x)$ die Anzahl der Rechenschritte von M bei der Abarbeitung der Eingabe x ist.*

Beispiel 3.4 Ist M die Turingmaschine zur Berechnung der Nachfolgerfunktion aus Beispiel 2.8, so gilt $|x| + 2 \leq time_M(x) \leq 2|x| + 2$ für alle $x \in \{0,1\}^+$.

Definition 3.5 *Es seien $\varphi : \Sigma^* \to \Sigma^*$ und $f : \mathbb{N} \to \mathbb{N}$ totale Funktionen. Wir sagen, dass φ mit einem Aufwand von f berechnet werden kann, wenn eine Mehrband-Turingmaschine M existiert, die φ berechnet und die $time_M(x) \leq f(|x|)$ für alle $x \in \Sigma^*$ erfüllt.*

Beispiel 3.6 Die Nachfolgerfunktion kann mit einem Aufwand von $f(n) = 2n + 2$ berechnet werden. Dabei ist n die Länge der Binärdarstellung des Eingabewertes x, es gilt also $n \approx \log_2 x$.

Bei der Untersuchung von Komplexitäten wollen wir uns wieder auf Entscheidungsprobleme, d.h. auf charakteristische Funktionen von Mengen, konzentrieren.

Definition 3.7 *Sei $f : \mathbb{N} \to \mathbb{N}$ eine Funktion. Die Klasse $TIME(f(n))$ besteht aus allen Mengen, die von einer Mehrband-Turingmaschine M entschieden werden können mit $time_M(x) \leq f(|x|)$ für alle $x \in \Sigma$.*

Definition 3.8 *Ein Polynom ist eine Funktion $p : \mathbb{N} \to \mathbb{N}$ der Form*

$$p(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_2 n^2 + a_1 n + a_0$$

wobei $k \in \mathbb{N}$ und $a_i \in \mathbb{N}$ für $i = 0, 1, 2, \ldots, k$ gilt.

Definition 3.9 *Die Komplexitätsklasse $\mathbb{P}$ ist wie folgt definiert:*

$$\mathbb{P} = \bigcup_{p \text{ Polynom}} TIME(p(n))$$

Wir bemerken, dass bei der Simulation von Mehrband-TM durch (Einband-)TM eine Berechnung einer Mehrband-Turingmaschine mit t Schritten durch eine Berechnung der Einband-Turingmaschine mit $O(t^2)$ Schritten simuliert wird. Eine Menge A liegt also genau dann in $\mathbb{P}$, wenn A durch eine (Einband-)TM in Polynomialzeit entschieden wird.

Die Churchsche These kann wie folgt erweitert werden: Ein Problem kann in polynomial beschränkter Zeit genau dann gelöst werden, wenn es in $\mathbb{P}$ liegt. Um zu zeigen, dass eine Menge in $\mathbb{P}$ liegt, reicht es zu zeigen dass es eine TM (einen Algorithmus) gibt, die (der) das Problem in einer Zeitkomplexität $O(n^k)$ für ein $k \in \mathbb{N}$ entscheidet.

In $\mathbb{P}$ liegen zum Beispiel

- das Palindromproblem,
- das Problem, ob zwei Zahlen teilerfremd sind,
- das Problem, ob ein gegebener Graph planar ist,
- das Problem, ob ein gegebener Graph einen Eulerkreis enthält,

usw.

3.2 Das Domino-Problem

Wir haben eine endliche Anzahl von Dominostein-Typen gegeben, wobei es von jedem Typ beliebig viele Exemplare gibt. Jeder Dominostein ist in vier Dreiecke aufgeteilt und in jedem dieser Dreiecke steht ein Symbol, das aus einer endlichen Menge Σ kommt. Ein Dominostein hat also die folgende Form:

Außerdem ist ein Rahmen vorgegeben, dessen Rand in Zellen aufgeteilt ist. Jede Zelle ist genau so groß wie ein Dominostein und ist mit einem Symbol versehen.

Die Frage ist, ob dieses Domino-Spiel eine Lösung hat, d.h. ob wir den Rahmen mit den Dominosteinen ausfüllen können, so daß

- für jedes Paar von benachbarten Dominosteinen s und s', die zwei Dreiecke, die die gemeinsame Kante von s und s' enthalten, das gleiche Symbol haben, und
- jedes Dreieck, das den Rand des Rahmens berührt, das gleiche Symbol hat, wie die Zelle des Randes, die berührt wird.

Bei dem Ausfüllen des Rahmens dürfen die Dominosteine nicht gedreht werden.

Bei der Beschäftigung mit dem Domino-Problem fällt uns folgendes auf:

1. Man kann ein gegebenes Problem durch systematisches Probieren lösen. Eine bessere Lösung ist im allgemeinen Fall nicht zu sehen. Es ist auch tatsächlich unbekannt, ob das Domino-Problem in Polynomialzeit gelöst werden kann.
2. Hat man dagegen einen mit Domino-Steinen ausgefüllten Rahmen, so kann man in polynomieller Zeit bezüglich der Rahmengröße entscheiden, ob die Ausfüllung korrekt ist.

Diese beiden Eigenschaften (keine bessere Lösung als systematisches Probieren in Sicht, Verifizierung einer möglichen Lösung in Polynomialzeit) besitzen auch viele andere Probleme. Im folgenden Abschnitt soll diese Klasse von Problemen formal definiert werden.

3.3 Nichtdeterministische Turingmaschinen und die Klasse $\mathbb{NP}$

Definition 3.10 *Eine nichtdeterministische Turingmaschine ist gegeben durch ein 7-Tupel $M = (Z, \Sigma, \Gamma, \delta, z_0, \square, E)$. Hierbei sind*

- Z *eine endliche Menge (Zustandsmenge),*
- Σ *ein Alphabet (Eingabealphabet),*
- Γ *ein Alphabet (Bandalphabet) mit* $\Sigma \subseteq \Gamma$*,*
- $\delta\colon Z \times \Gamma \to 2^{Z \times \Gamma \times \{L,R,N\}}$ *eine Funktion (Überführungsfunktion),*
- $z_0 \in Z$ *(Anfangszustand),*
- $\square \in \Gamma \setminus \Sigma$ *(Leerzeichen, Blank),*
- $E \subseteq Z$ *(Menge der Endzustände).*

Die NTM unterscheidet sich von der TM lediglich durch die Überführungsfunktion, die für einen gegebenen Zustand und ein gegebenes Bandsymbol mehrere Aktionen erlaubt. Konfigurationen sind für NTM genauso definiert wie für TM. Formal definieren wir die Arbeitsweise einer NTM, d.h. die Nachfolgerelation $\vdash$, wie folgt:

Definition 3.11 *Wir definieren in der Menge der Konfigurationen einer nichtdeterministischen Turingmaschine $M = (Z, \Sigma, \Gamma, \delta, z_0, \square, E)$ die binäre Relation „$\vdash$“ wie folgt. Es gilt*

$$a_1 \dots a_m z b_1 \dots b_n \vdash \begin{cases} a_1 \dots a_m z' c b_2 \dots b_n & \text{falls } (z', c, N) \in \delta(z, b_1),\ m \geq 0,\ n \geq 1, \\ a_1 \dots a_m c z' b_2 \dots b_n & \text{falls } (z', c, R) \in \delta(z, b_1),\ m \geq 0,\ n \geq 2, \\ a_1 \dots a_{m-1} z' a_m c b_2 \dots b_n & \text{falls } (z', c, L) \in \delta(z, b_1),\ m \geq 1,\ n \geq 1, \end{cases}$$

$$a_1 \dots a_m z b_1 \vdash a_1 \dots a_m c z \square \text{ falls } (z', c, R) \in \delta(z, b_1),\ m \geq 0,$$

$$z b_1 \dots b_n \vdash z \square c b_2 \dots b_n \text{ falls } (z', c, L) \in \delta(z, b_1),\ n \geq 1.$$

Beispiel 3.12 Es sei $\delta(z, a) = \{(z_1, a, R), (z_2, b, L), (z_3, c, N)\}$. Dann gelten für die Konfiguration $aczabb$ folgende Nachfolgebeziehungen:

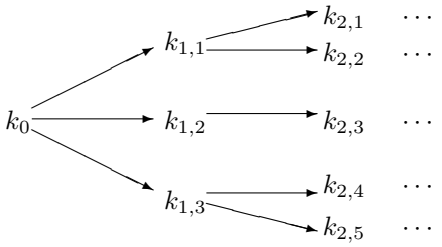
$$aczabb \vdash acaz_1bb, \qquad aczabb \vdash az_2cbbb, \qquad aczabb \vdash acz_3cbb.$$

Die Arbeitsweisen von TM und NTM lassen sich wie folgt veranschaulichen:

TM: Es gibt für jede Eingabe *genau einen* Berechnungspfad:

$$k_0 \vdash k_1 \vdash k_2 \vdash \dots$$

NTM: Es gibt für jede Eingabe einen *Baum* der möglichen Berechnungspfade:



Da es zu einer Eingabe einer NTM in der Regel mehrere mögliche Berechnungen gibt, definieren wir für die NTM keine berechnete Funktion, sondern die *akzeptierte Menge*.

Definition 3.13 *Sei $M = (Z, \Sigma, \Gamma, \delta, z_0, \Box, E)$ eine nichtdeterministische Turingmaschine. Die von M* akzeptierte Menge *$T(M)$ ist definiert durch*

$$T(M) = \{x \in \Sigma^* \mid z_0 x \vdash^* wzy \text{ für } z \in E,\ w, y \in \Gamma^*\}.$$

Eine Eingabe x wird also genau dann von der NTM akzeptiert, wenn *eine der möglichen* Berechnungen zu einem Endzustand führt, d.h. wenn es im zugehörigen Berechnungsbaum eine Endkonfiguration gibt. Die bisher betrachteten (deterministsichen) Turingmaschinen sind ein Spezialfall der nichtdeterministischen Turingmaschinen. Bezüglich der akzeptierten Sprachen sind nichtdeterministische und deterministische Turingmaschinen äquivalent.

Satz 3.14 *Für jede NTM M existiert eine TM M' mit $T(M') = T(M)$.*

Beweis. Die Beweisidee (*Dovetailing*) ist wie folgt:

- Bestimme für wachsendes n alle Konfigurationen, die in n Schritten erreichbar sind.
- Wird eine Endkonfiguration erreicht, so akzeptiere.

Etwas genauer erfolgt die Simulation der NTM M durch eine 2-Band-TM mit folgender Arbeitsweise:

- Band 1 enthält alle nach n Schritten von M erreichbaren Konfigurationen (zuerst also die Startkonfiguartion).
- Ist eine der Konfigurationen auf Band 1 eine Endkonfiguration von M, so akzeptiere.
- Anderenfalls schreibe für jede Konfiguration auf Band 1 alle Nachfolgekonfigurationen auf Band 2. Lösche dabei Band 1.
- Verschiebe den Inhalt von Band 2 nach Band 1. Band 1 enthält nun alle nach $n+1$ Schritten erreichbaren Konfigurationen. Beginne nun wieder von vorn.

□

Folgerung 3.15 *Eine Menge ist genau dann semi-entscheidbar, wenn sie durch eine NTM akzeptiert wird.*

Beweis. Man kann aus einer TM, die eine Menge A akzeptiert, sehr leicht eine TM konstruieren, die die "halbe" charakteristische Funktion χ'_A berechnet, und umgekehrt. Die Behauptung folgt dann aus der Äquivalenz von TM und NTM. □

Als nächstes sollen nichtdeterministische Komplexitätsklassen definiert werden.

Definition 3.16 *Sei $M = (Z, \Sigma, \Gamma, \delta, z_0, \square, E)$ eine nichtdeterministische Turingmaschine; die Funktion $time_M : \Sigma^* \to \mathbb{N}$ ist definiert durch*

$$time_M(x) = \begin{cases} \text{Minimum der Längen aller akzeptierenden Rechnungen von } M \text{ auf } x & \text{falls } x \in T(M), \\ 0 & \text{falls } x \notin T(M). \end{cases}$$

Für eine von M akzeptierte Eingabe x ist $time_M(x)$ also die *kleinste Tiefe* einer Endkonfiguration im zugehörigen Berechnungsbaum.

Definition 3.17 *Sei $f : \mathbb{N} \to \mathbb{N}$ eine totale Funktion. Die Klasse $NTIME(f(n))$ besteht aus allen Mengen A, die von einer nichtdeterministischen Mehrband-Turingmaschine M [9] akzeptiert werden können mit $time_M(x) \leq f(|x|)$ für alle Eingaben x.*

Definition 3.18 *Die Komplexitätsklasse $\mathbb{NP}$ ist wie folgt definiert:*

$$\mathbb{NP} = \bigcup_{p \text{ Polynom}} NTIME(p(n))$$

Erneut gehört eine Menge A genau dann zu $\mathbb{NP}$, wenn A durch eine nichtdeterministische (Einband-)TM in Polynomialzeit akzeptiert wird.

Abbildung 3.1 gibt einen Überblick über die mengentheoretischen Beziehungen der bisher definierten Klassen. Dabei heißt eine Menge "LOOP-berechenbar", wenn ihre charakteristische Funktion LOOP-berechenbar ist. Es ist bisher unbekannt, ob die Inklusion $\mathbb{P} \subseteq \mathbb{NP}$ echt ist. Diese als "$\mathbb{P} = \mathbb{NP}$ - Problematik" bekannte Frage ist eines der wichtigsten offenen Probleme der heutigen Mathematik. Alle anderen Inklusionen sind echt.

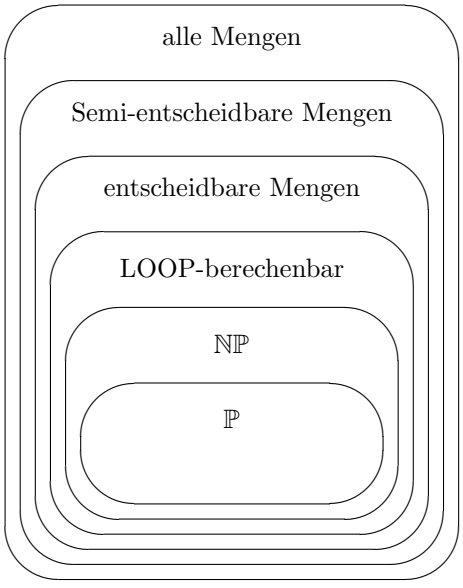


Abbildung 3.1: Komplexitätsklassen

[9] Nichtdeterministische Mehrband-Turingmaschinen lassen sich analog zu deterministischen Mehrband-Turingmaschinen definieren. Wir verzichten auf eine formale Definition, da dieser Begriff nur in dieser Definition benötigt wird.

3.4 $\mathbb{NP}$-Vollständigkeit

Ob es ein Problem aus $\mathbb{NP}$ gibt, das nicht in Polynomialzeit lösbar ist, weiß man also zur Zeit nicht. Wir wollen in diesem Unterabschnitt die Klasse der $\mathbb{NP}$-vollständigen Probleme kennen lernen. Diese können im folgenden Sinne als die "schwierigsten Probleme" dieser Klasse $\mathbb{NP}$ angesehen werden: Kann ein $\mathbb{NP}$-vollständiges Problem in Polynomialzeit lösen, so kann man alle Probleme aus $\mathbb{NP}$ in Polynomialzeit lösen. Wir werden insbesondere sehen, dass das Domino-Problem $\mathbb{NP}$-vollständig ist.

Definition 3.19 *Seien $A, B \subseteq \Sigma^*$ Mengen. Dann heißt A auf B polynomial reduzierbar (in Zeichen $A \leq_p B$) falls es eine totale und in polynomialer Zeit berechenbare Funktion $f : \Sigma^* \to \Sigma^*$ gibt, so dass für alle $x \in \Sigma^*$ gilt:*

$$x \in A \implies f(x) \in B.$$

Wir wollen hier nur eine einfache Reduktion vorführen. Wir definieren die folgenden Mengen:

$$\begin{aligned}\mathbf{SOS} := \{(a_1, \ldots, a_m, b) : \quad & m, a_1, \ldots, a_m, b \in \mathbb{N} \text{ und es gibt} \\ & c_1, \ldots, c_m \in \{0,1\}, \text{ so daß } \textstyle\sum_{i=1}^m c_i a_i = b\}\end{aligned}$$

und

$$\begin{aligned}\mathbf{RS} := \{(g_1, \ldots, g_m, k_1, \ldots, k_m, G, K) : \\ m, g_1, \ldots, g_m, k_1, \ldots, k_m, G, K \in \mathbb{N} \\ \text{und es gibt } c_1, \ldots, c_m \in \{0,1\}, \\ \text{so daß } \textstyle\sum_{i=1}^m c_i g_i \leq G \text{ und } \sum_{i=1}^m c_i k_i \geq K\}.\end{aligned}$$

SOS ist als Teilmengensummenproblem (*Sum of Subsets*) bekannt; **RS** steht für *Rucksackproblem*: Wir haben m Päckchen mit Lebensmitteln. Das i-te Päckchen hat Gewicht g_i und enthält k_i Kalorien. Wir wollen entscheiden, ob wir einen Rucksack mit einer Teilmenge der Päckchen packen können, so daß das Gesamtgewicht höchstens G und die Kalorienzahl mindestens K ist.

Satz 3.20 $\mathbf{SOS} \leq_p \mathbf{RS}$.

Beweis. Nach Definition suchen wir eine in Polynomialzeit berechenbare Funktion f, die Eingaben für **SOS** auf Eingaben für **RS** abbildet, so daß

$$(a_1, \ldots, a_m, b) \in \mathbf{SOS} \iff f(a_1, \ldots, a_m, b) \in \mathbf{RS}.$$

Damit $f(a_1, \ldots, a_m, b)$ eine Eingabe für **RS** ist, muß dieser Funktionswert also die Form

$$f(a_1, \ldots, a_m, b) = (g_1, \ldots, g_m, k_1, \ldots, k_m, G, K)$$

haben. Wir definieren

$$f(a_1, \ldots, a_m, b) := (a_1, \ldots, a_m, a_1, \ldots, a_m, b, b).$$

Es ist klar, daß f in polynomieller Zeit berechenbar ist. Außerdem gilt

$$\begin{aligned}&(a_1, \ldots, a_m, b) \in \mathbf{SOS} \\ &\iff \text{es gibt } c_1, \ldots, c_m \in \{0,1\} \text{ mit } \textstyle\sum_{i=1}^m c_i a_i = b \\ &\iff \text{es gibt } c_1, \ldots, c_m \in \{0,1\} \text{ mit } \textstyle\sum_{i=1}^m c_i a_i \leq b \text{ und } \sum_{i=1}^m c_i a_i \geq b \\ &\iff (a_1, \ldots, a_m, a_1, \ldots, a_m, b, b) \in \mathbf{RS} \\ &\iff f(a_1, \ldots, a_m, b) \in \mathbf{RS}.\end{aligned}$$

□

Lemma 3.21 (i) *Gilt $A \leq_p B$ und $B \in \mathbb{NP}$, so ist auch $A \in \mathbb{NP}$.*

(ii) *Gilt* $A \leq_p B$ *und* $B \in \mathbb{P}$*, so ist auch* $A \in \mathbb{P}$.

Beweis. Wir zeigen nur die erste Behauptung; der Beweis der zweiten Behauptung erfolgt analog. Es gelte $A, B \subseteq \Sigma^*$, $A \leq_p B$ und $f : \Sigma^* \to \Sigma^*$ sei die nach Definition verlangte Funktion mit $x \in A \iff f(x) \in B$. Es sei M_1 eine TM, die f berechnet und $time_{M_1}(x) \leq p(|x|)$ für ein Polynom p und alle $x \in \Sigma^*$ erfüllt. Da $B \in \mathbb{NP}$ gilt, existiert eine NTM M_2, die B akzeptiert und $time_{M_2}(y) \leq q(|y|)$ für ein Polynom q und alle $y \in \Sigma^*$ erfüllt.

Wir konstruieren eine NTM M mit $T(M) = A$, indem auf eine Eingabe x zunächst die TM M_1 angesetzt wird und anschließend die NTM M_2 mit dem Ergebnis $f(x)$ weiter arbeitet. Da M_1 zur Bearbeitung von x höchstens $p(|x|)$ Schritte ausführt, gilt $|f(x)| \leq p(|x|)$, wobei wir o.B.d.A. $p(n) \geq n$ für alle $n \in \mathbb{N}$ voraussetzen. Wir erhalten $time_M(|x|) \leq p(|x|) + q(p(|x|))$, d.h. $time_M$ ist polynomiell beschränkt. □

Definition 3.22 *Eine Menge* A *heißt* $\mathbb{NP}$*-vollständig genau dann, wenn*

(i) $A \in \mathbb{NP}$ *und*

(ii) *für alle Mengen* $A' \in \mathbb{NP}$ $A' \leq_p A$ *gilt.*

Mengen, due Bedingung (ii) erfüllen, werden auch $\mathbb{NP}$-hart *genannt.*

Dann gilt offensichentlich folgender Satz.

Satz 3.23 *Sei* A $\mathbb{NP}$*-vollständig, dann gilt*

$$A \in \mathbb{P} \Leftrightarrow \mathbb{P} = \mathbb{NP}.$$

Wir wollen als nächstes zeigen, dass das Domino-Problem $\mathbb{NP}$-vollständig ist. Dazu müssen wir nach Definition beweisen, dass man jede Menge $A \in \mathbb{NP}$ in Polynomialzeit auf das Domino-Problem reduzieren kann.

Satz 3.24 Domino *ist* **NP***-vollständig.*

Beweis. Es ist klar, daß **Domino** $\in$ **NP**: Eine Lösung besteht aus einer Auslegung des Rahmens mit Dominosteinen. Die Anzahl der Bits, die wir brauchen um so eine Auslegung darzustellen ist polynomiell in der Länge der Eingabe. Außerdem können wir in Polynomialzeit überprüfen, ob eine vorgegebene Lösung korrekt ist.

Wir müssen also noch beweisen, daß

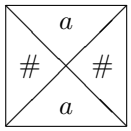
$A \leq_p$ **Domino** für alle Mengen $A \in$ **NP**.

Sei A eine Menge aus **NP**. Dann gibt es ein Polynom p und eine nichtdeterministische Turing-Maschine $M = (Z, \Sigma, \Gamma, \delta, z_0, \Box, E)$, die die Menge A in der Zeit p akzeptiert. Wir dürfen o.B.d.A. annehmen, dass am Ende einer erfolgreichen Berechnung von M das Band leer ist.

Wir konstruieren jetzt für M ein Dominospiel Π_M. Ein Eingabewort w wird dann derart auf einen Rahmen R_w transformiert, dass eine akzeptierende Berechnung von M einer Auslegung des Rahmens R_w mit Steinen aus Π_M entspricht.

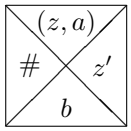
Das Dominospiel hat die folgenden Typen von Dominosteinen:

1. Für jedes Bandsymbol a der Turing-Maschine M:



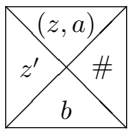
   Bedeutung: Vor und nach dem Befehl steht der Kopf nicht in diesem Feld.

2. Für jeden Befehl $(z, a) \to (z', b, R)$ der Turing-Maschine M:



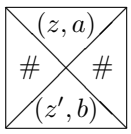
Bedeutung: Vor dem Befehl steht der Kopf in diesem Feld; er macht einen Schritt nach rechts.

3. Für jeden Befehl $(z, a) \to (z', b, L)$ der Turing-Maschine M:



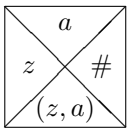
Bedeutung: Vor dem Befehl steht der Kopf in diesem Feld; er macht einen Schritt nach links.

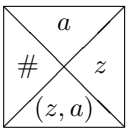
4. Für jeden Befehl $(z, a) \to (z', b, N)$ der Turing-Maschine M:



Bedeutung: Vor und nach dem Befehl steht der Kopf in diesem Feld.

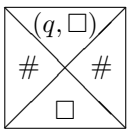
5. Für jeden Zustand z und jedes Alphabet-Symbol a der Turing-Maschine M gibt es zwei Dominostein-Typen:





Bedeutung: Der linke Dominostein-Typ gibt an, daß der Kopf von links in dieses Feld kommt; der rechte Dominostein-Typ gibt an, daß der Kopf von rechts in dieses Feld kommt.

6. Für jeden Endzustand q der Turing-Maschine M gibt es den Dominostein-Typ:



Bedeutung: Nach einer akzeptierenden Rechnung werden die Information über den Endzustand und die Position des Lese/Schreibkopfes gelöscht.

Damit liegen die Dominostein-Typen fest. Der Rahmen R_w für ein Eingabewort $w = a_1 a_2 \cdots a_n$ ist in Abbildung 3.2 dargestellt. Der obere Rand dieses Rahmens entspricht dem Anfang einer Berechnung, der untere Rand entspricht dem Ende einer akzeptierenden Berechnung. Der linke und rechte Rand entsprechen Grenzen für den Kopf. Offenbar kann R_w in Polynomialzeit bezüglich $|w|$ bestimmt werden.

Für eine Lösung des Dominospiels gelten folgende Feststellungen:

- Der obere Teil der ersten Zeile enthält die Startkonfiguration von M für die Eingabe w. Der untere Teil der letzten Zeile enthält nur Blanks.
- Enthält der obere Teil einer Zeile eine Konfiguration k der NTM M, so enthält der untere Teil dieser Zeile eine Nachfolgekonfiguration von k, falls k keine Endkonfiguration ist, bzw. nur Blanks, falls k eine Endkonfiguration ist.
- Der obere Teil einer Zeile hat den gleichen Inhalt wie der untere Teil der Vorgängerzeile.

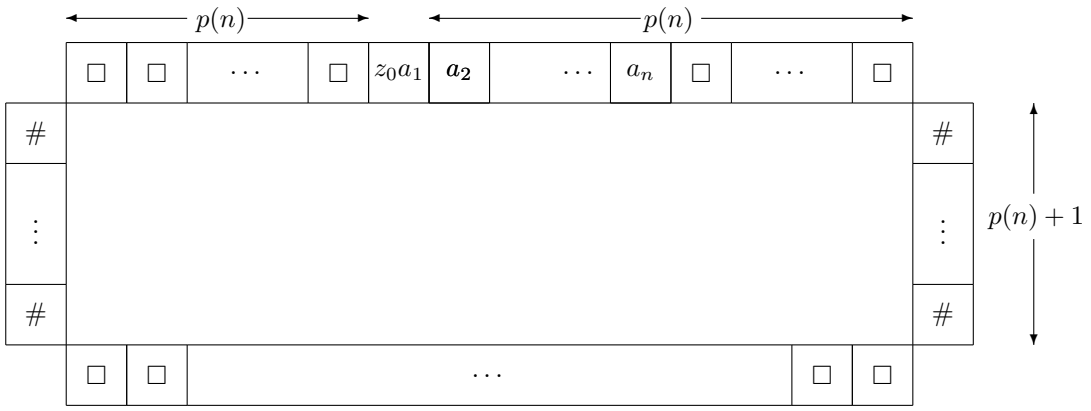


Abbildung 3.2: *Rahmen des Dominospiels.*

Somit entspricht eine Lösung des Domino-Problems für den Rahmen R_w einer akzeptierenden Rechnung von M für die Eingabe w mit höchstens $p(|w|)$ Schritten. Umgekehrt kann man aus jeder akzeptierenden Rechnung mit höchstens $p(|w|)$ Schritten eine Lösung des Domino-Problems konstruieren. Es gilt also:

$$\begin{aligned} w \in A \iff & \text{es gibt mindestens eine akzeptierende Berechnung von } M, \\ & \text{die höchstens } p(n) \text{ Schritte macht} \\ \iff & \text{das Dominoproblem } (\Pi_M, R_w) \text{ ist lösbar.} \end{aligned}$$

Damit haben wir nachgewiesen, dass $A \leq_p$ **Domino** für jede beliebige Menge $A \in \mathbb{NP}$ gilt. Also ist die Menge **Domino NP**-vollständig. □

Satz 3.25 *Sei A $\mathbb{NP}$-vollständig, $B \in \mathbb{NP}$ und gelte $A \leq_p B$. Dann ist auch B $\mathbb{NP}$-vollständig.*

Der letzte Satz hat große Bedeutung bei der Untersuchung von Problemen. Will man nämlich zeigen, dass ein Problem B $\mathbb{NP}$-vollständig ist, so braucht man nicht auf die Definition zurück zu gehen, sondern es reicht die Reduktion eines $\mathbb{NP}$-vollständigen Problems A.

Ohne Beweis führen wir hier an, dass folgende Probleme $\mathbb{NP}$-vollständig sind (für die genauen Definitionen siehe Literatur).

- **SAT**, das Erfüllbarkeitsproblem der Aussagenlogik:
  Eingabe: Boolesche Formel φ in konjunktiver Normalform
  Frage: Ist die Formel φ erfüllbar?
  Für dieses Problem wird die $\mathbb{NP}$-Vollständigkeit in den meisten Lehrbüchern direkt gezeigt. Man kann aber auch relativ einfach zeigen, dass **Domino** $\leq_p$ **SAT** gilt.
- **3SAT** (wie **SAT**, jedoch höchstens 3 Literale pro Alternative).
- **Clique**, das Cliquenproblem:
  Eingabe: ungerichteter Graph G, $k \in \mathbb{N}$
  Frage: Enthält G einen vollständigen Teilgraphen mit k Knoten?
- **Set Partition**, das Partitionierungsproblem:
  Eingabe: Menge U, Gewichtsfunktion $g : U \to \mathbb{N}$
  Frage: Gibt es eine Zerlegung $U = V \cup W$, $V \cap W = \emptyset$, so dass $\sum_{v \in V} g(v) = \sum_{w \in W} g(w)$?
- **Bin Packing**:

Eingabe: m Gegenstände mit Volumen $a_1, a_2, \ldots a_m$,
k Behälter mit Volumen jeweils ℓ.
Frage: Kann man die Gegenstände auf die k Behälter verteilen?

- Hamiltonkreis-Problem:
  Eingabe: gerichteter Graph G
  Frage: Gibt es in G einen Kreis, der jeden Knoten genau einmal passiert?
- **TSP**, das Rundreiseproblem:
  Eingabe: Schranke $M \in \mathbb{N}$, n Städte, $n \times n$-Kostenmatrix C mit
  $C_{i,j}$ = Kosten einer Fahrt von Stadt i nach Stadt j
  Frage: Gibt es eine Rundreise durch alle Städte
  mit Kosten von höchstens M?

Eine umfangreiche Sammlung $\mathbb{NP}$-vollständiger Probleme findet man im Buch von Garey und Johnson [6].

4 Formale Sprachen

4.1 Einführung

Wir erinnern an die Definitionen im Unterkapitel 1.6, die im weiteren Verlauf der Vorlesung von Bedeutung sein werden. Insbesondere werden wir uns mit *formalen Sprachen* über gegebenen *Alphabeten* Σ befassen.

Betrachten wir dazu ein Beispiel.

Beispiel 4.1 Sei $\Sigma = \{(,), +, -, *, /, a\}$, so betrachten wir die Menge *EXPR* der korrekt geklammerten arithmetischen Ausdrücke über diesem Alphabet, wobei a als Platzhalter für beliebige Konstanten und Variablen dienen soll. Zum Beispiel soll gelten:

$$\begin{aligned} (a-a)*a+a\,/\,(a+a)-a &\in \mathit{EXPR} \\ (((a))) &\in \mathit{EXPR} \\ ((a+)-a(&\notin \mathit{EXPR} \end{aligned}$$

Wie das Beispiel zeigt, sind formale Sprachen im Allgemeinen unendliche Mengen von Wörtern. In diesem Kapitel wird es darum gehen, diese unendlichen Mengen durch endliche Konstrukte zu charakterisieren, zu beschreiben. Dazu gehören zum Beispiel *Grammatiken* und *Automaten*.

Zunächst werden wir uns mit Grammatiken beschäftigen, die für die Theorie der Informatik eine ausgezeichnete Rolle spielen, insbesondere im Compilerbau. Allerdings sind die historischen Wurzeln in der Linguistik zu suchen, deshalb an dieser Stelle ein (stark vereinfachtes) Beispiel aus der Linguistik. (Dass die Linguistik nicht so einfach zu formalisieren ist, erkennt man an den Schwierigkeiten, die automatische Übersetzungssysteme (noch?) machen. Noch kann kein solches System einen Dolmetscher ersetzen.)

Beispiel 4.2 Wir betrachten eine Grammatik mit folgenden Regeln, wobei sogenannte *Variable* oder *Phrasen* durch spitze Klammern gekennzeichnet sind. Das heißt, sie gehören eigentlich nicht zum Alphabet, über dem die Sprache definiert werden soll.

$$\begin{aligned}
\text{<Satz>} &\to \text{<Subjekt><Prädikat><Objekt>} \\
\text{<Subjekt>} &\to \text{<Artikel><Attribut><Substantiv>} \\
\text{<Artikel>} &\to \varepsilon \\
\text{<Artikel>} &\to \text{<der>} \\
\text{<Artikel>} &\to \text{<die>} \\
\text{<Artikel>} &\to \text{<das>} \\
\text{<Attribut>} &\to \varepsilon \\
\text{<Attribut>} &\to \text{<Adjektiv>} \\
\text{<Attribut>} &\to \text{<Adjektiv><Attribut>} \\
\text{<Adjektiv>} &\to \text{<kleine>} \\
\text{<Adjektiv>} &\to \text{<bissige>} \\
\text{<Adjektiv>} &\to \text{<große>} \\
\text{<Substantiv>} &\to \text{<Hund>} \\
\text{<Substantiv>} &\to \text{<Katze>} \\
\text{<Prädikat>} &\to \text{<jagt>} \\
\text{<Objekt>} &\to \text{<Artikel><Attribut><Substantiv>}
\end{aligned}$$

Durch die obige Grammatik können wir zum Beispiel folgenden Satz bilden:

der kleine bissige Hund jagt die große Katze

Wie wir diesen Satz mittels der Regeln abgeleitet haben, kann man sehr gut an einem sogenannten *Syntaxbaum* veranschaulichen. Dieser ist in der Abbildung 4.1 dargestellt. Hierbei werden die linke und rechte Seite einer angewendeten Regel durch einen Vater- bzw. Sohnknoten dargestellt.

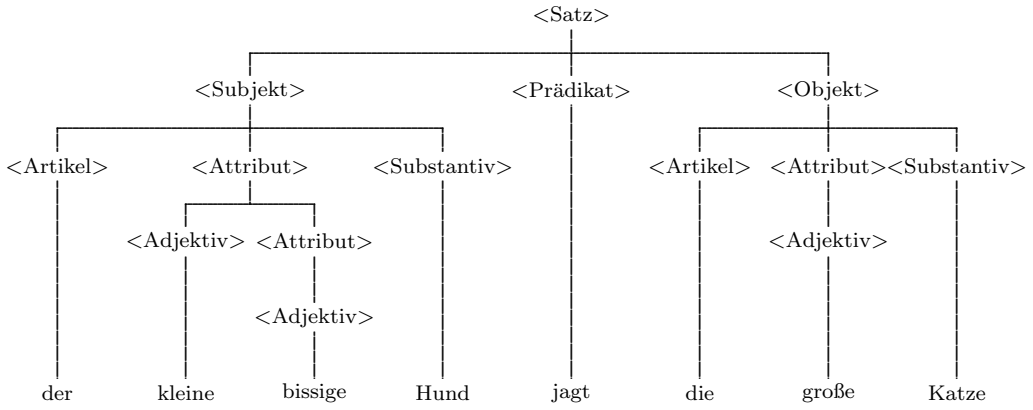


Abbildung 4.1: Syntaxbaum für den Satz „*der kleine bissige Hund jagt die große Katze*“

Wir können natürlich auch noch andere Sätze mittels dieser Grammatik bilden, zum Beispiel

der kleine bissige Hund jagt die große große große Katze

und

die kleine Katze jagt die große große Hund

Ersterer Satz macht deutlich, dass wir mit obiger (endlicher) Grammatik bereits unendlich viele Sätze bilden können. Der letze Satz zeigt die Schwächen und auch Grenzen von Grammatiken auf:

1. Die Grammatik lässt keine Fälle zu und ist somit unzureichend. Diese Schwäche allerdings könnten wir durch eine verbesserte Grammatik (dann aber wesentlich umfangreicher) beheben.
2. Es zeigt sich aber auch eine andere Schwäche, die nicht so einfach zu beheben ist: Ein *syntaktisch* einwandfreier Satz ist im Allgemeinen nicht automatisch *semantisch* korrekt.

4.2 Grammatiken

Wir bemerken, dass im obigen Beispiel zwei Arten von Symbolen auftauchen: Erstens sogenannte *Terminalsymbole*, das sind Symbole, aus denen die Wörter eigentlich bestehen, und sogenannte *Nichtterminalsymbole* oder *Variablen*. Das sind Symbole, die zwar während des Ableitungsprozesses benutzt werden (zum Beispiel <Attribut>), aber im eigentlichen Wort oder Satz nicht mehr auftauchen. In unserem Beispiel besteht jede Regel aus einer linken Seite und einer rechten Seite, wobei die linke Seite hier immer aus genau einer Variablen besteht. Im Allgemeinen kann eine linke Seite auch aus einem Wort mit mehreren Symbolen bestehen. Ein besonderes Symbol in unserer Beispielgrammatik war <Satz>, damit beginnt eine Ableitung. Eine Ableitung wiederum ist eine mehrfache Anwendung der Regeln, wobei eine Anwendung einer Regel bedeutet, dass in einem Wort ein Teilwort (die linke Seite einer Regel) durch die rechte Seite derselben Regel ersetzt wird.

Wir wollen jetzt den Begriff der Grammatik formalisieren.

Definition 4.3 *Eine Grammatik ist ein 4-Tupel $G = (V, \Sigma, P, S)$, wobei*

- *V ein Alphabet ist (Nichtterminalalphabet oder Alphabet der Variablen),*
- *Σ ein Alphabet ist (Terminalalphabet),*
- *$V \cap \Sigma = \emptyset$ gilt,*

- *P eine endliche Teilmenge von $((V \cup \Sigma)^* \setminus \Sigma^*) \times (V \cup \Sigma)^*$ ist (Menge der Regeln),*
- *$S \in V$ ist (die Startvariable oder Axiom).*

Die Elemente von P, also die *Regeln* oder auch *Produktionen* sind eigentlich geordnete Paare. Zur besseren Lesbarkeit werden wir aber $u \to v \in P$ für $(u, v) \in P$ schreiben.

Den Begriff der Ableitung haben wir schon informal eingeführt. Wir wollen jetzt definieren, was wir unter der *direkten Ableitung* exakt verstehen wollen.

Definition 4.4 *Sei $G = (V, \Sigma, P, S)$ eine Grammatik und $u, v \in (V \cup \Sigma)^*$ Wörter. Dann gilt $u \Longrightarrow_G v$ (in Worten: u erzeugt bezüglich G direkt v) genau dann, wenn*

(i) *$u = \gamma_1 \alpha \gamma_2$ mit $\gamma_1, \gamma_2 \in (V \cup \Sigma)^*$,*

(ii) *$v = \gamma_1 \beta \gamma_2$ und*

(iii) *$\alpha \to \beta \in P$ ist.*

Falls aus dem Kontext eindeutig hervorgeht, welche Grammatik G gemeint ist, schreiben wir $u \Longrightarrow v$ statt $u \Longrightarrow_G v$.

Wir können $\Longrightarrow$ als Relation in der Menge $(V \cup \Sigma)^*$ ansehen. Dann wollen wir mit $\overset{*}{\Longrightarrow}$ den reflexiven und transitiven Abschluss der Relation $\Longrightarrow$ bezeichnen. Wir können ihn auch elementweise definieren:

Definition 4.5 *Sei $G = (V, \Sigma, P, S)$ eine Grammatik und $u, v \in (V \cup \Sigma)^*$ Wörter. Dann gilt $u \overset{*}{\Longrightarrow}_G v$ genau dann, wenn $u = v$ gilt oder es ein $n \in \mathbb{N}$ und Wörter $w_0, w_1, \ldots, w_n$ gibt, so dass*

$$u = w_0 \Longrightarrow_G w_1 \Longrightarrow_G w_2 \Longrightarrow_G \cdots \Longrightarrow_G w_n = v$$

gilt.

Wiederum schreiben wir $u \overset{*}{\Longrightarrow} v$ statt $u \overset{*}{\Longrightarrow}_G v$, falls es kein Missverständnis geben kann.

Nun sind wir in der Lage, die *erzeugte Sprache* $L(G)$ einer Grammatik G als die Menge aller Wörter zu definieren, die von dieser Grammatik erzeugt wird.

Definition 4.6 *Sei $G = (V, \Sigma, P, S)$ eine Grammatik. Die von G erzeugte Sprache $L(G)$ wird definiert als*

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

Betrachten wir ein Beispiel.

Beispiel 4.7 Es sei die Grammatik

$$G = (\{E, T, F\}, \{(,), a, +, *\}, P, E)$$

mit

$$P = \{E \to T,\ E \to E + T,\ T \to F,\ T \to T * F,\ F \to a,\ F \to (E)\}$$

gegeben. Diese Grammatik beschreibt eine Teilmenge der Menge *EXPR*, der Menge der exakt geklammerten arithmetischen Ausdrücke aus Beispiel 4.1, nämlich die Teilmenge ohne die Operationen *Division* / und *Subtraktion* $-$. Es gilt zum Beispiel

$$a * a * (a + a) + a \in L(G),$$

denn das Wort $a * a * (a + a) + a$ wird durch die Ableitung

$$\begin{aligned} E &\Longrightarrow E + T \Longrightarrow T + T \Longrightarrow T * F + T \Longrightarrow T * F * F + T \Longrightarrow F * F * F + T \\ &\Longrightarrow a * F * F + T \Longrightarrow a * a * F + T \Longrightarrow a * a * (E) + T \Longrightarrow a * a * (E + T) + T \\ &\Longrightarrow a * a * (T + T) + T \Longrightarrow a * a * (F + T) + T \Longrightarrow a * a * (a + T) + T \\ &\Longrightarrow a * a * (a + F) + T \Longrightarrow a * a * (a + a) + T \Longrightarrow a * a * (a + a) + F \\ &\Longrightarrow a * a * (a + a) + a \end{aligned}$$

aus dem Startwort E erzeugt. Hierbei wurde in jedem Ableitungsschritt immer die am weitesten links stehende Variable ersetzt. Wir können auch eine andere Ableitung für dieses Wort konstruieren, zum Beispiel:

$$\begin{aligned} E &\Longrightarrow E + T \Longrightarrow T + T \Longrightarrow T * F + T \Longrightarrow T * F * F + T \Longrightarrow F * F * F + T \\ &\Longrightarrow a * F * F + T \Longrightarrow a * a * F + T \Longrightarrow a * a * (E) + T \Longrightarrow a * a * (E + T) + T \\ &\Longrightarrow a * a * (T + T) + T \Longrightarrow a * a * (F + T) + T \Longrightarrow a * a * (F + F) + T \\ &\Longrightarrow a * a * (F + F) + F \Longrightarrow a * a * (a + F) + F \Longrightarrow a * a * (a + a) + F \\ &\Longrightarrow a * a * (a + a) + a. \end{aligned}$$

Beiden Ableitungen wird in diesem Beispiel ein und derselbe Syntaxbaum zugeordnet (siehe Abbildung 4.2).

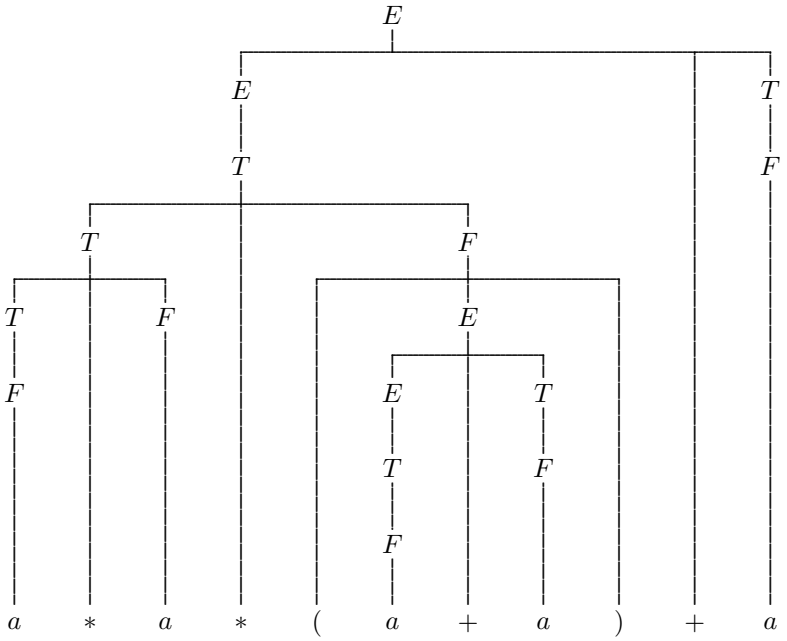


Abbildung 4.2: Ein Syntaxbaum für das Wort $a * a * (a + a) + a$

Betrachten wir ein weiteres, etwas komplexeres Beispiel.

Beispiel 4.8 Es sei die Grammatik

$$G = (\{S, B, C\}, \{a, b, c\}, P, S)$$

mit

$$P = \{S \to aSBC,\ S \to aBC,\ CB \to BC,\ aB \to ab,\ bB \to bb,\ bC \to bc,\ cC \to cc\}$$

gegeben. Wir können zum Beispiel die Ableitung

$$\begin{aligned} S &\Longrightarrow aSBC \Longrightarrow aaSBCBC \Longrightarrow aaaBCBCBC \\ &\Longrightarrow aaaBBCCBC \Longrightarrow aaaBBCBCC \Longrightarrow aaaBBBCCC \\ &\Longrightarrow aaabBBCCC \Longrightarrow aaabbBCCC \Longrightarrow aaabbbCCC \\ &\Longrightarrow aaabbbcCC \Longrightarrow aaabbbccC \Longrightarrow aaabbbccc \end{aligned}$$

aufstellen, also gehört das Wort $aaabbbccc = a^3b^3c^3$ zur erzeugten Sprache $L(G)$, es gilt also $a^3b^3c^3 \in L(G)$.

Nun fragen wir uns natürlich, welche Menge $L(G)$ genau darstellt. Wenn man sich die Ableitung genauer anschaut, vermutet man leicht

$$L(G) = \{a^n b^n c^n \mid n \geq 1\}. \tag{4.1}$$

Der Nachweis dafür muss eigentlich exakt mathematisch geführt werden. Man macht es in zwei Schritten:

(i) Zunächst wird $L(G) \supseteq \{a^n b^n c^n \mid n \geq 1\}$ gezeigt,

(ii) dann $L(G) \subseteq \{a^n b^n c^n \mid n \geq 1\}$.

Uns genügt es allerdings, den Nachweis nur zu skizzieren:

(i) Um $L(G) \supseteq \{a^n b^n c^n \mid n \geq 1\}$ zu zeigen, müssen wir also nachweisen, dass jedes Wort $a^n b^n c^n$ für ein $n \geq 1$ vom Startwort S abgeleitet werden kann. Dazu schauen wir uns obige Ableitung für $a^3b^3c^3$ näher an und sehen, dass sie einfach verallgemeinert werden kann. Zunächst wird $(n-1)$-mal die Regel $S \to aSBC$ angewendet und einmal die Regel $S \to aBC$. Dann erhält man das Wort $a^n(BC)^n$ (siehe Zeile 1 in obiger Ableitung). Dann werden durch mehrmalige Anwendung der Regel $CB \to BC$ alle B's vor die C's getauscht (Zeile 2). Dann werden durch die Regeln $aB \to ab$ und $bB \to bb$ alle B's in b's umgewandelt (Zeile 3) und schließlich durch die Regeln $bC \to bc$ und $cC \to cc$ alle C's in c's (Zeile 4).

(ii) Schwieriger zu zeigen ist die Behauptung $L(G) \subseteq \{a^n b^n c^n \mid n \geq 1\}$. Das heißt, es ist zu zeigen, dass *nur* Wörter der Form $a^n b^n c^n$ für ein $n \geq 1$ abgeleitet werden können. Dies geschieht eigentlich streng mathematisch (wie eigentlich auch schon die Behauptung (i)) über das Beweisverfahren der vollständigen Induktion. Für uns reicht eine Diskussion in der Hinsicht, dass eine genaue Analyse der Regeln Folgendes zeigt:

Erstens werden nur Worte mit gleicher Anzahl von a's und b's und c's erzeugt. (Sieht man daran, dass durch die erste und zweite Regel jeweils für jedes a auch genau ein B und ein C erzeugt wird. Und die anderen Regeln wandeln höchstens Großbuchstaben in Kleinbuchstaben um, verändern aber nicht die Anzahl!)

Zweitens werden nur Wörter erzeugt, in denen alle a's vor allen b's stehen, und alle b's vor allen c's. (Zeigt eine genaue Diskussion aller Regeln.)

Beide Behauptungen (i) und (ii) zusammen bringen dann unsere gewünschte Aussage 4.1.

Wir werden jetzt noch zwei weitere Beispiele für Grammatiken bringen, wobei wir den Nachweis für die erzeugte Sprache nicht bringen werden. Wie im obigen Beispiel führt oft eine genaue Analyse der Regeln und deren Zusammenspiel zur gewünschten Aussage.

Beispiel 4.9 Es sei

$$G = (\{S\}, \{a, b\}, \{S \to aSb,\ S \to ab\}, S)$$

eine Grammatik, dann gilt

$$L(G) = \{a^n b^n \mid n \geq 1\}.$$

Eine Ableitung für das Wort a^4b^4 sieht dann so aus:

$$S \Longrightarrow aSb \Longrightarrow aaSbb \Longrightarrow aaaSbbb \Longrightarrow aaaabbbb.$$

Beispiel 4.10 Es sei

$$G = (\{S\}, \{a\}, \{S \to aS,\ S \to a\}, S)$$

eine Grammatik, dann gilt

$$L(G) = \{a^n \mid n \geq 1\}.$$

Eine Ableitung für das Wort a^5 sieht dann so aus:

$$S \Longrightarrow aS \Longrightarrow aaS \Longrightarrow aaaS \Longrightarrow aaaaS \Longrightarrow aaaaa.$$

Beispiel 4.11 Es sei

$$G = (\{S\}, \{a, b\}, \{S \to aS,\ S \to bS,\ S \to a,\ S \to b\}, S)$$

eine Grammatik, dann gilt

$$L(G) = \{a, b\}^+ = \{w \in \{a, b\}^* \mid w \neq \varepsilon\}.$$

Eine Ableitung für das Wort *aaba* sieht dann so aus:

$$S \Longrightarrow aS \Longrightarrow aaS \Longrightarrow aabS \Longrightarrow aaba.$$

4.2.1 Chomsky-Hierarchie

Wir wollen in diesem Kapitel eine Klassifikation der Grammatiken in sogenannte *Typ*-0- bis *Typ*-3-*Grammatiken* angeben. Sie stammt von Noam Chomsky aus dem Jahre 1958, einem Linguisten aus der Frühzeit der Theorie formaler Sprachen, trotzdem hat sie nichts an Aktualität verloren, im Gegenteil.

Definition 4.12 *Eine Grammatik $G = (V, \Sigma, P, S)$ heißt vom*

- *Typ* 0 *oder* Phrasenstrukturgrammatik*, wenn sie keinen Beschränkungen unterliegt,*
- *Typ* 1 *oder* kontextabhängig*, falls für alle Regeln $\alpha \to \beta$ in P gilt: $|\alpha| \leq |\beta|$, mit der Ausnahme $S \to \varepsilon$, falls S nicht auf der rechten Seite einer Regel vorkommt.*
- *Typ* 2 *oder* kontextfrei*, wenn jede Regel von der Form $A \to \beta$ mit $A \in V$ und $\beta \in (V \cup \Sigma)^*$ ist.*
- *Typ* 3 *oder* regulär*, wenn jede Regel von der Form $A \to wB$ oder $A \to w$ mit $A, B \in V$ und $w \in \Sigma^*$ ist.*

Bevor wir die Begriffe auf Sprachen erweitern, eine Bemerkung zu den Bezeichnungen *kontextfrei* und *kontextabhängig* (auch manchmal *kontextsensitiv* genannt):

Bei einer *kontextfreien* Regel $A \to \alpha$ kann in einem Wort der Buchstabe A unabhängig vom *Kontext* des Buchstaben A (d. h. des Textes links und rechts von A) durch α ersetzt werden.

Bei *kontextabhängigen* Grammatiken kann man zeigen, dass man sich auf Regeln der Form $\gamma_1 A \gamma_2 \to \gamma_1 \alpha \gamma_2$ mit $\gamma_1, \gamma_2 \in (V \cup \Sigma)^*$ und $\alpha \in (V \cup \Sigma)^+$beschränken kann (mit Ausnahme $S \to \varepsilon$), d. h. wiederum wird letztendlich die Variable A durch ein Wort α ersetzt, allerdings können wir diese Ersetzung nur dann vornehmen, wenn A in einem gewissen *Kontext* steht (hier γ_1 und γ_2), d. h. die Ersetzung ist vom *Kontext abhängig.*

Definition 4.13 *Eine Sprache $L \subseteq \Sigma^*$ heißt vom Typ* 0 *oder rekursiv aufzählbar (Typ* 1 *oder kontextabhängig, Typ* 2 *oder kontextfrei, Typ* 3 *oder regulär), falls es eine Grammatik $G = (V, \Sigma, P, S)$ vom Typ* 0 *(Typ* 1*, Typ* 2*, Typ* 3*) gibt, so dass $L = L(G)$ gilt.*

Betrachten wir unsere Beispielgrammatiken, so gilt:

- Die Grammatik und somit auch die erzeugte Sprache aus Beispiel 4.8 ist vom Typ 1.
- Die Grammatiken und somit auch die erzeugten Sprachen aus den Beispielen 4.7 sowie 4.9 sind vom Typ 2.
- Die Grammatiken und somit auch die erzeugten Sprachen aus den Beispielen 4.10 sowie 4.11 sind vom Typ 3.

Aus den Definitionen der Chomsky-Grammatiken folgt sofort:

Folgerung 4.14 (i) *Jede Typ-*1*-Grammatik ist vom Typ* 0*.*

(ii) *Jede Typ-*2*-Grammatik ist vom Typ* 0*.*

(iii) *Jede Typ-*3*-Grammatik ist vom Typ* 2*.* □

Wegen der kanonischen Definition der Sprachen folgt aus der Folgerung 4.14 sofort:

Lemma 4.15 (i) *Jede Typ-*1*-Sprache ist vom Typ* 0*.*

(ii) *Jede Typ-*2*-Sprache ist vom Typ* 0*.*

(iii) *Jede Typ-*3*-Sprache ist vom Typ* 2*.* □

Wir bemerken, dass nicht jede Typ-2-Grammatik vom Typ 1 ist, da Typ-2-Grammatiken sogenannte ε-Regeln enthalten dürfen (auch für Variablen, die nicht das Startwort sind). Wir werden aber später zeigen, dass man jede kontextfreie Grammatik „ε-Regel-frei“ machen kann, so dass man auch zeigen kann, dass jede Typ-2-sprache auch vom Typ 1 ist.

Führen wir die Bezeichnungen **Typ 0**, **Typ 1**, **Typ 2** und **Typ 3** für die Menge aller Typ-0-Sprachen, Typ-1-Sprachen, Typ-2-Sprachen bzw. Typ-3-Sprachen ein, so werden wir letztendlich folgenden Satz beweisen, der hier an dieser Stelle schon mal wegen der Vollständigkeit genannt wird.

Satz 4.16 *Es gilt:*

$$\textbf{Typ 3} \subsetneq \textbf{Typ 2} \subsetneq \textbf{Typ 1} \subsetneq \textbf{Typ 0}.$$

Das heißt, alle Inklusionen in Lemma 4.15 sind echt. Satz 4.16 stellt eine der wichtigsten Aussagen nicht nur in dieser Vorlesung, sondern in der gesamten Theorie der Informatik dar und wird als sogenannte *Chomsky-Hierarchie* bezeichnet.

Beispiele für Sprachen, die die *Echtheit der Inklusionen* in der Chomsky-Hierarchie zeigen, sind folgende (hier ohne Beweis):

Satz 4.17 (i) *Die Sprache* $L = \{a^n b^n \mid n \geq 1\}$ *ist vom Typ* 2*, aber nicht vom Typ* 3*.*

(ii) *Die Sprache* $L' = \{a^n b^n c^n \mid n \geq 1\}$ *ist vom Typ* 1*, aber nicht vom Typ* 2*.*

(iii) *Die Sprache* $L'' = L_H$ *ist vom Typ* 0*, aber nicht vom Typ* 1*, dabei ist* L_H *das „Halteproblem“ aus dem ersten Teil der Vorlesung* (*siehe Definition* 2.67)*.*

Folgender Satz gibt die Beziehung der Chomsky-Hierarchie zu weiteren Sprachklassen.

Satz 4.18 (i) *Die Menge der Typ-*0*-Sprachen und die Menge der semi-entscheidbaren Sprachen* (*siehe Definition* 2.56) *sind identisch.*

(ii) *Es gibt Sprachen, die sind nicht vom Typ* 0*.* □

Zusammengefasst stellen wir die Aussagen aus den Sätzen 4.16, 4.18 und 2.62 in der Abbildung 4.3 dar.

Von Interesse für die Informatik sind insbesondere die kontextfreien und die regulären Sprachen. Deshalb werden sie auch in unseren weiteren Überlegungen die Hauptrolle spielen. Regulären

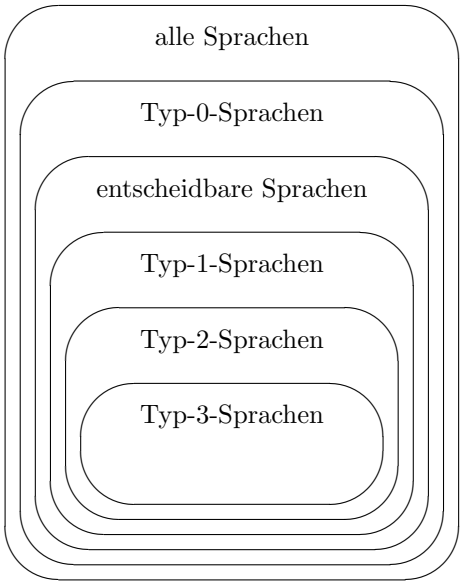


Abbildung 4.3: Die Chomsky-Hierarchie mit weiteren Sprachklassen

Sprachen spielen unter Anderem eine große Rolle bei der lexikalischen Analyse im Compilerbau, beim Suchen und Ersetzen in Editoren, bei Netzwerkprotokollen etc. Die Theorie kontextfreier Sprachen ist eng mit dem Compilerbau, insbesondere mit der Syntaxanalyse verbunden. Eine weitere Sprachklasse, die hier von besonderem Interesse ist, ist die Klasse der *deterministisch kontextfreien* Sprachen, die in der Hierarchie unterhalb der kontextfreien aber oberhalb der regulären Sprachen liegen. In diesem Zusammenhang wurden auch die $LL(k)$- und $LR(k)$-Sprachen untersucht. Obige Gründe führten zu einer weitgehenden Theorie der regulären und kontextfreien Sprachen, insbesondere auch deshalb, da diese Sprachklassen sich theoretisch „leicht“ erschließen ließen. Allerdings gilt: „Die Welt ist nicht kontextfrei“. Schon die Menge aller korrekten Programme in einer gängigen Programmiersprache (PASCAL, C++, PROLOG, EIFFEL, JAVA, etc.) ist leider nicht kontextfrei. Allerdings wurde für die Beschreibung dieser Sprachen trotzdem eine kontextfreie Syntax benutzt (zu den Gründen später). Das hat zur Folge, dass ein syntaktisch korrektes Programm noch lange nicht korrekt sein muss, sondern dass noch weitere Überprüfungen notwendig sind.

4.2.2 Wortproblem

Ein gegebenes Programm ist syntaktisch korrekt, falls es der Syntax *entspricht*, d. h. falls es aus den syntaktischen Regeln abgeleitet werden kann. Für die Syntaktische Überprüfung eines Prgrammes muss man also untersuchen, ob es aus den syntaktische Regeln aufgebaut werden kann. Wenn man bedenkt, dass die Syntax nichts Anderes ist als eine Menge von Regeln, stellt also ein Programm nichts Anderes dar als ein Wort. Syntaktisch korrekt ist das Programm (Wort), wenn es der Syntax (Regeln) entspricht. Mit anderen Worten, interessiert die Frage, ob ein gegebenes Wort von einer gegebenen Grammatik erzeugt werden kann, das aber ist genau das *Wortproblem* für Grammatiken. Genauer formuliert:

Definition 4.19 (Wortproblem) *Sei $i \in \{0,1,2,3\}$. Unter dem Wortproblem für Typ-i-Grammatiken versteht man folgendes Problem:*

Gegeben: *Grammatik $G = (V, \Sigma, P, S)$ vom Typ i, $i \in \{0,1,2,3\}$, und Wort $w \in \Sigma^*$,*

Frage: *Gilt $w \in L(G)$?*

Da das Halteproblem für Turingmaschinen unentscheidbar ist und es nach Satz 4.17 in der Menge der Typ-0-Sprachen liegt, gilt:

Folgerung 4.20 *Das Wortproblem für Typ-0-Sprachen ist unentscheidbar.* □

Das ist natürlich hinsichtlich der eingangs gemachten Bemerkungen zur Syntaxüberprüfung eine katastrophale Aussage. Glücklicherweise kann man schon für Typ-1-Grammatiken die Entscheidbarkeit retten. Es gibt also einen Algorithmus, der bei Eingabe einer Typ-1-Grammatik $G = (V, \Sigma, P, S)$ und einem Wort $w \in \Sigma^*$ in endlicher Zeit entscheidet, ob $w \in L(G)$ gilt oder nicht. Der folgende Satz hält die Aussage exakt fest. Ursache ist die Monotonie der Ableitungen, d. h. die Bedingung $|\alpha| \leq |\beta|$ für alle Regeln $\alpha \to \beta$ in P. Deshalb brauchen nämlich nur endlich viele Ableitungen untersucht werden, dem geneigten Leser ist der korrekte Beweis angefügt.

Satz 4.21 *Das Wortproblem für Typ-1-Grammatiken ist entscheidbar.*

Beweis. Sei $G = (V, \Sigma, P, S)$ die gegebene Grammatik vom Typ 1 und $w \in \Sigma^*$ das gegebene Wort. Wir definieren Mengen T_m^n für alle $m, n \in \mathbb{N}$ wie folgt.

$$T_m^n = \{w \in (V \cup \Sigma)^* \mid |w| \leq n \text{ und } w \text{ lässt sich aus } S \text{ in höchstens } m \text{ Schritten ableiten}\}.$$

Diese Mengen T_m^n, $n \geq 1$ lassen sich induktiv über m wie folgt definieren:

$$\begin{aligned} T_0^n &= \{S\}, \\ T_{m+1}^n &= T_m^n \cup \{w \in (V \cup \Sigma)^* \mid |w| \leq n \text{ und } w' \Longrightarrow w \text{ für ein } w' \in T_m^n\}. \end{aligned}$$

Diese Darstellung ist natürlich nur für Typ-1-Grammatiken anwendbar.

Da es nur endlich viele Wörter in $(V \cup \Sigma)^*$ gibt, die höchstens die Länge n haben, ist $\bigcup_{m \geq 0} T_m^n$ für jedes $n \in \mathbb{N}$ eine endliche Menge. Folglich gibt es ein m mit

$$T_m^n = T_{m+1}^n = T_{m+2}^n = \cdots$$

Falls nun w, mit $|w| = n$, in $L(G)$ liegt, so muss w in $\bigcup_{m \geq 0} T_m^n$ und damit in T_m^n für ein m liegen. Das ist aber in endlicher Zeit überprüfbar. □

Der aus dem Beweis des Satzes 4.21 resultierende Algorithmus zur Entscheidung des Wortproblems ist leider exponentiell. Bis heute ist auch kein „besserer“ Algorithmus bekannt. Es ist auch nicht zu vermuten, dass es bald einen besseren Algorithmus gibt, da das Wortproblem für Typ-1-Grammatiken $\mathbb{NP}$-*hart* (siehe Kapitel 3) ist.

Für eine praktikable Syntaxüberprüfung ist also eine Syntax, die als allgemeine Typ-1-Grammatik konstruiert wurde, nicht zu gebrauchen, denn über die katastrophalen Auswirkungen von Algorithmen mit exponentiellem Laufzeitverhalten wurde bereits auch in Kapitel 3 informiert.

Glücklicherweise kann man zeigen, dass die Wortprobleme von Teilklassen der Typ-1-Sprachen auch eine kleinere Komplexität haben. Das Wortproblem für Typ-2-Grammatiken ist von kubischer Zeitkomplexität, das Wortproblem für $LL(k)$- und $LR(k)$-Grammatiken ist von linearer Zeitkomplexität. Glücklicherweise kann man die Syntax von gängigen Programmiersprachen schon durch $LL(k)$- und $LR(k)$-Grammatiken realisieren. Das wird bei modernen Programmiersprachen verwendet, so dass also ein *heutiger* Compiler die Syntaxüberprüfung in Linearzeit erledigt.

Betrachten wir zum besseren Verständnis ein Beispiel zur Anwendung des Algorithmus, basierend auf dem Beweis zum Satz 4.21.

Beispiel 4.22 Gegeben sei die Grammatik aus Beispiel 4.8. Sei $n = 4$. Dann erhalten wir:

$$\begin{aligned} T_0^4 &= \{S\}, \\ T_1^4 &= \{S, aSBC, aBC\}, \\ T_2^4 &= \{S, aSBC, aBC, abC\}, \\ T_3^4 &= \{S, aSBC, aBC, abC, abc\}, \\ T_4^4 &= \{S, aSBC, aBC, abC, abc\} = T_3^4. \end{aligned}$$

Das heißt, das einzige terminale Wort der Sprache $L(G)$ der Länge ≤ 4 ist abc.

4.2.3 Syntaxbäume

In der Einleitung zu diesem Kapitel haben wir Syntaxbäume schon informell kennengelernt. Wir wollen auch hier nicht allzuviel hinzufügen, sondern nur einige wichtige Eigenschaften zusammenfassend ohne nähere Betrachtung nennen. Den interessierten Leser verweisen wir auf die reichhaltige Literatur.

1. Jeder Ableitung eines Wortes w in einer Typ-2- oder Typ-3-Grammatik kann ein Syntaxbaum zugeordnet werden.
2. Sei $w \in L(G)$ und $S = w_1 \Longrightarrow w_2 \Longrightarrow \cdots \Longrightarrow w_n = w$ eine Ableitung für w. Dann wird der Syntaxbaum folgendermaßen definiert: Die Wurzel wird S; falls bei der Ableitung eine Regel $A \to \alpha$ angewendet wird, heißt das, dem Vaterknoten A werden $|\alpha|$ viele Söhne zugeordnet, nämlich alle Symbole von α. Die Blätter des Syntaxbaumes sind dann genau (von links nach rechts gelesen) die Buchstaben von w.
3. Falls die Grammatik regulär ist, ist jeder Syntaxbaum „entartet“ (Kette).
4. Verschiedene Ableitungen für ein und dasselbe Wort können den gleichen Syntaxbaum haben, oder auch nicht.
5. Mehrdeutig heißt eine Grammatik, wenn es ein Wort gibt, für das verschiedene Syntaxbäume existieren.
6. Inhärent mehrdeutig heißt eine Sprache, wenn jede Grammatik G mit $L(G) = L$ mehrdeutig ist.
7. Die Sprache $L = \{a^i b^j c^k \mid i = j \text{ oder } j = k\}$ ist ein Beispiel für eine inhärent mehrdeutige, kontextfreie Sprache.
8. Es ist nicht entscheidbar, ob zu einer gegebenen kontextfreien Grammatik eine äquivalente kontextfreie Grammatik existiert, die nicht mehrdeutig ist.

4.3 Reguläre Sprachen

In diesem Kapitel beschäftigen wir uns etwas näher mit den regulären Sprachen, insbesondere mit der Möglichkeit verschiedener Charakterisierungen und den Eigenschaften.

4.3.1 Endliche Automaten

Zur Definition der regulären Sprachen benutzten wir reguläre Grammatiken. Grammatiken können Wörter über einem gewissen Alphabet *erzeugen* und so eine Menge von Wörtern, eine *Sprache*, beschreiben. Jetzt wollen wir einen anderen Mechanismus zur Beschreibung benutzen: die *endlichen Automaten*. Der Automat erhält ein Wort als Eingabe, „arbeitet“ über diesem Wort und *erkennt* (oder *akzeptiert*) es oder auch nicht. Alle Wörter, die ein Automat akzeptiert, bilden die von ihm beschriebene *Sprache*.

Definition 4.23 (Deterministischer endlicher Automat) *Ein deterministischer endlicher Automat (Wir wollen ihn kurz mit DEA bezeichnen) A ist ein 5-Tupel $A = (Z, \Sigma, \delta, z_0, E)$, wobei Z das Zustandsalphabet und Σ das Eingabealphabet mit $Z \cap \Sigma = \emptyset$ sind. $z_0 \in Z$ ist der Anfangszustand, $E \subseteq Z$ die Menge der Endzustände oder akzeptierenden Zustände. Mit δ bezeichnen wir die Zustandsüberführungsfunktion $\delta\colon Z \times \Sigma \to Z$.*

Wir interpretieren einen DEA als endliche Kontrolle, die sich in einem Zustand $z \in Z$ befindet und eine Folge von Symbolen aus Σ, die auf einem Band geschrieben stehen, liest (siehe Abbildung 4.4). In einem Schritt bewegt sich der Lesekopf des Automaten, der augenblicklich über der Zelle des

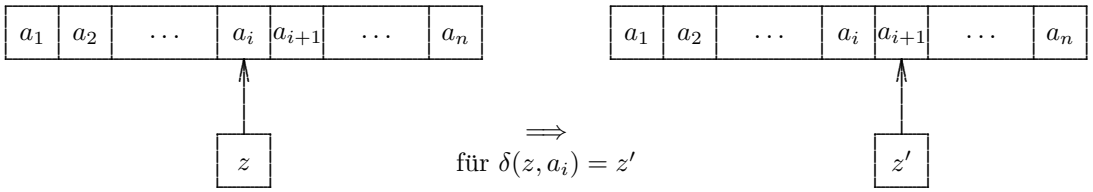


Abbildung 4.4: Interpretation der Arbeitsweise eines endlichen Automaten

Bandes mit dem Inhalt $a_i \in \Sigma$ steht und sich im Zustand $z \in Z$ befindet, einen Schritt nach rechts und geht in den Zustand $\delta(z, a_i) = z'$ über. Ist $\delta(z, a_i) = z' \in E$, d. h. ein akzeptierender Zustand, so hat der DEA das ganze Wort, das er, beginnend in Startzustand z_0, gelesen hat, akzeptiert.

Um die von einem DEA akzeptierte Sprache, d. h. die Menge aller von ihm akzeptierten Wörter, formal beschreiben zu können, erweitern wir die Zustandsfunktion δ zur Funktion $\hat{\delta}\colon Z \times \Sigma^* \to Z$ rekursiv durch folgende Definition.

Definition 4.24 (Erweiterte Zustandsüberführungsfunktion eines DEA) *Sei A ein deterministischer endlicher Automat $A = (Z, \Sigma, \delta, z_0, E)$, die erweiterte Zustandsüberführungsfunktion $\hat{\delta}\colon Z \times \Sigma^* \to Z$ wird definiert durch*

(i) $\hat{\delta}(z, \varepsilon) = z$ *für alle $z \in Z$,*

(ii) $\hat{\delta}(z, wa) = \delta(\hat{\delta}(z, w), a)$ *für alle $z \in Z$, $a \in \Sigma$, $w \in \Sigma^*$.*

Wir erwähnen, dass in der Literatur $\hat{\delta}$ oft auch nur als δ bezeichnet wird. Das ist in der Tatsache begründet, dass für alle $z \in Z$ und für alle $a \in \Sigma$ die Gleichheit $\hat{\delta}(z, a) = \delta(z, a)$ gilt, d. h. δ ist eine Einschränkung der Funktion $\hat{\delta}$ auf $Z \times \Sigma$.

Jetzt können wir die von dem Automaten akzeptierte Sprache definieren:

Definition 4.25 (Akzeptierte Sprache eines DEA) *Für einen DEA $A = (Z, \Sigma, \delta, z_0, E)$ sei die von ihm akzeptierte Sprache $T(A)$ definiert durch $T(A) = \{w \in \Sigma^* \mid \hat{\delta}(z_0, w) \in E\}$.*

Nun ist es Zeit für ein Beispiel.

Beispiel 4.26 Wir betrachten den deterministischen endlichen Automaten

$$A = (\{z_0, z_1, z_2, z_3\}, \{a, b\}, \delta, z_0, \{z_3\}),$$

wobei δ durch die Tabelle

δ	z_0	z_1	z_2	z_3
a	z_1	z_1	z_3	z_3
b	z_0	z_2	z_0	z_3

gegeben ist. Um zu entscheiden, ob zum Beispiel die Wörter ab, $aaba$ oder $bababb$ von dem Automaten akzeptiert werden, müssen wir $\hat{\delta}(z_0, ab)$, $\hat{\delta}(z_0, aaba)$ und $\hat{\delta}(z_0, bababb)$ bestimmen. Dazu benutzen wir die Definition von $\hat{\delta}$ und natürlich die Definition des Automaten, insbesondere der Überführungsfunktion δ.

$$\begin{aligned}
\hat{\delta}(z_0, ab) &= \delta(\hat{\delta}(z_0, a), b) \\
&= \delta(\delta(\hat{\delta}(z_0, \varepsilon), a), b) \\
&= \delta(\delta(z_0, a), b) \\
&= \delta(z_1, b) \\
&= z_2
\end{aligned}$$

Also gilt $\hat{\delta}(z_0, ab) = z_2$ und somit $ab \notin T(A)$, da $z_2 \notin E$ gilt, z_2 also kein akzeptierender Zustand ist.

$$\begin{aligned}
\hat{\delta}(z_0, aaba) &= \delta(\hat{\delta}(z_0, aab), a) \\
&= \delta(\delta(\hat{\delta}(z_0, aa), b), a) \\
&= \delta(\delta(\delta(\hat{\delta}(z_0, a), a), b), a) \\
&= \delta(\delta(\delta(\delta(\hat{\delta}(z_0, \varepsilon), a), a), b), a) \\
&= \delta(\delta(\delta(\delta(z_0, a), a), b), a) \\
&= \delta(\delta(\delta(z_1, a), b), a) \\
&= \delta(\delta(z_1, b), a) \\
&= \delta(z_2, a) \\
&= z_3
\end{aligned}$$

Wegen $z_3 \in E$ gilt demnach: $aaba$ wird vom Automaten akzeptiert.

$$\begin{aligned}
\hat{\delta}(z_0, bababb) &= \delta(\hat{\delta}(z_0, aab), a) \\
&= \delta(\delta(\hat{\delta}(z_0, baba), b), b) \\
&= \delta(\delta(\delta(\hat{\delta}(z_0, bab), a), b), b) \\
&= \delta(\delta(\delta(\delta(\hat{\delta}(z_0, ba), b), a), b), b) \\
&= \delta(\delta(\delta(\delta(\delta(\hat{\delta}(z_0, b), a), b), a), b), b) \\
&= \delta(\delta(\delta(\delta(\delta(\delta(\hat{\delta}(z_0, \varepsilon), b), a), b), a), b), b) \\
&= \delta(\delta(\delta(\delta(\delta(\delta(z_0, b), a), b), a), b), b) \\
&= \delta(\delta(\delta(\delta(\delta(z_0, a), b), a), b), b) \\
&= \delta(\delta(\delta(\delta(z_1, b), a), b), b) \\
&= \delta(\delta(\delta(z_2, a), b), b) \\
&= \delta(\delta(z_3, b), b) \\
&= \delta(z_3, b) \\
&= z_3
\end{aligned}$$

Wegen $z_3 \in E$ gilt auch $bababb \in T(A)$.

Wir haben also $ab \notin T(A)$ und $aaba$, $bababb \in T(A)$. Nun gilt es zu bestimmen, welche Sprache $T(A)$ von diesem deterministischen endlichen Automaten A akzeptiert wird. Eine genaue Analyse der Überführungsfunktion δ würde ergeben (den exakten Beweis führen wir nicht, wird auch vom Hörer dieser Vorlesung nicht verlangt)

$$T(A) = \{w \in \{a,b\}^* \mid w = uabav \text{ für Wörter } u, v \in \{a,b\}^*\},$$

also akzeptiert der Automat alle Wörter über dem Alphabet $\{a,b\}$, die das Teilwort aba besitzen.

Im obigen Beispiel ist es nicht so leicht, die akzeptierte Sprache des Automaten zu bestimmen. Oft wird es etwas leichter, falls wir die Darstellung eines gerichteten Graphen, des sogenannten *Überführungsgraphen* oder auch *Transitionsgraphen* des deterministischen endlichen Automaten benutzen. Diese Darstellung sieht folgendermaßen aus.

Die Menge der Zustände Z wird die Knotenmenge des Graphen, $\delta(z_1, a) = z_2$ wird durch eine gerichtete Kante von z_1 zu z_2, die mit a markiert ist, dargestellt. Der Anfangszustand wird durch einen zum Knoten gehenden Pfeil dargestellt, Endzustände durch zwei konzentrische Kreise (siehe Abbildung 4.5). Wollen wir jetzt wissen, ob ein Wort vom Automaten akzeptiert wird, dann

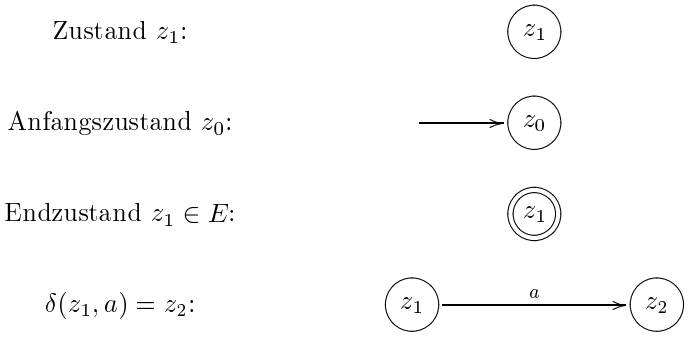


Abbildung 4.5: Konstruktion des Überführungsgraphen eines DEA

müssen wir, beginnend im Zustand z_0 die Kanten des Graphen entsprechend des Wortes entlang wandern.

In Abbildung 4.6 ist der Überführungsgraph zum DEA aus Beispiel 4.26 dargestellt. Wenn man

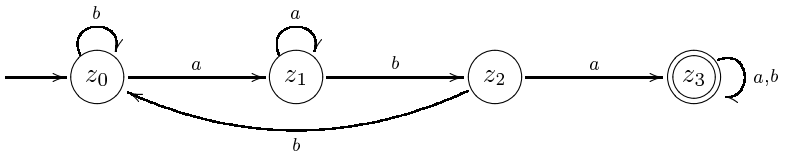


Abbildung 4.6: Überführungsgraph des DEA aus Beispiel 4.26

sich diesen Graphen genauer ansieht, erkennt man eher als aus der Tabelle, dass die akzeptierte Sprache genau die Menge aller Wörter mit dem Teilwort aba ist. Die Zustände sind assoziert mit den Teilen des Wortes aba, die bereits eingelesen wurden: z_0: noch nichts von aba, z_1: schon das a von aba, z_2: schon das ab von aba, z_3: das gesamte Teilwort aba. Wenn aba gelesen wurde (Zustand z_3), bleibt der Automat immer in diesem akzeptierenden Zustand.

Betrachten wir ein weiteres Beispiel.

Beispiel 4.27 Es sei A der deterministische endliche Automat

$$A = (\{z_0, z_1, z_2, z_3\}, \{0,1\}, \delta, z_0, \{z_2, z_3\}),$$

mit der Überführungsfunktion δ, gegeben durch folgende Tabelle.

δ	z_0	z_1	z_2	z_3
0	z_0	z_3	z_3	z_0
1	z_1	z_2	z_2	z_1

Der dazugehörige Graph ist in Abbildung 4.7 dargestellt. Auch hier nutzen wir wiederum die

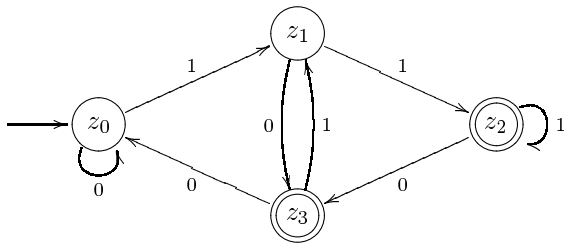


Abbildung 4.7: Überführungsgraph des DEA aus Beispiel 4.27

Zustände, um Informationen über den Teil des Wortes zu speichern, der bereits eingelesen wurde. Man erkennt, dass der Zustand die beiden zuletzt gelesenen Buchstaben repräsentiert. Der Zustand z_0 bedeutet, die letzten beiden Buchstaben waren 00, bei z_1: 01, bei z_2: 11 und bei z_3: 10. Akzeptiert wird im Zustand z_2 und z_3, also genau dann, wenn das vorletzte Zeichen eine 1 war. Also gilt

$$T(A) = \{w \in \{0,1\}^* \mid w = u1x \text{ mit } u \in \{0,1\}^*,\ x \in \{0,1\}\}$$

für die akzeptierte Sprache $T(A)$, also ist $T(A)$ die Menge aller Wörter über $\{0,1\}$, deren vorletztes Symbol eine 1 ist.

Es stellt sich natürlich die Frage, welche Sprachklasse durch deterministische endliche Automaten beschrieben wird. Die akzeptierten Sprachen in den Beispielen 4.26 und 4.27 sind regulär. Gilt das für alle von DEA akzeptierten Sprachen? Die Antwort gibt folgender Satz.

Satz 4.28 *Sei A ein deterministischer endlicher Automat. Dann ist die von A akzeptierte Sprache $T(A)$ regulär (vom Typ 3).*

Beweis. Sei $A = (Z, \Sigma, \delta, z_0, E)$ ein DEA. Wir konstruieren eine Grammatik $G = (V, \Sigma, P, S)$ folgendendermaßen: wir setzen $V = Z$ und $S = z_0$. Weiterhin sei

$$P = \{z_1 \to az_2 \mid \delta(z_1, a) = z_2\} \cup \{z_1 \to \varepsilon \mid z_1 \in E\}.$$

Offensichtlich ist die so konstruierte Grammatik G vom Typ 3, also regulär, und erzeugt die Sprache $T(A)$, was noch zu beweisen wäre. Wir verweisen aber an dieser Stelle für den interessierten Leser auf die Literatur. □

Wir wollen die Konstruktion aus obigem Beweis an einem Beispiel demonstrieren.

Beispiel 4.29 Sei $A = (\{z_0, z_1, z_2, z_3\}, \{a, b\}, \delta, z_0, \{z_3\})$ der DEA aus Beispiel 4.26 mit der Überführungsfunktion δ:

δ	z_0	z_1	z_2	z_3
a	z_1	z_1	z_3	z_3
b	z_0	z_2	z_0	z_3

Wir konstruieren jetzt die äquivalente Typ-3-Grammatik $G = (V, \Sigma, P, S)$:

$$
\begin{aligned}
V &= \{z_0, z_1, z_2, z_3\},\\
\Sigma &= \{a, b\},\\
S &= z_0,\\
P &= \{z_0 \to az_1,\ z_0 \to bz_0,\ z_1 \to az_1,\ z_1 \to bz_2,\ z_2 \to az_3,\ z_2 \to bz_0,\ z_3 \to az_3,\ z_3 \to bz_3\}\\
&\quad \cup \{z_3 \to \varepsilon\}.
\end{aligned}
$$

Bei der Konstruktion von P handelt sich in der ersten Zeile um die Regeln, die ausgehend von δ im ersten Schritt konstruiert werden, also zum Beispiel $z_0 \to az_1$ wegen $\delta(z_0, a) = z_1$ oder $z_0 \to bz_0$ wegen $\delta(z_0, b) = z_0$. In der zweiten Zeile kommt die terminierende Regel für den Endzustand $\{z_3\}$ hinzu.

4.3.2 Nichtdeterministische endliche Automaten

Nachdem wir wissen, dass jede von einem deterministischen endlichen Automaten akzeptierte Sprache vom Typ 3, also regulär ist, interessiert natürlich die Umkehrung, also ob jede Typ-3-Sprache auch von einem deterministischen endlichen Automaten akzeptiert werden kann. Eine naheliegende Idee wäre, die Konstruktion aus dem obigen Beweis einfach entsprechend zu invertieren.

Das führt allerdings zu zwei Schwierigkeiten.

1. Wenn wir zum Beispiel eine reguläre Grammatik mit den Regeln $A \to aB$ und $A \to aC$ hätten, dann müssten gleichzeitig $\delta(A, a) = B$ und $\delta(A, a) = C$ gelten, was nicht möglich ist.
2. Wenn wir zum Beispiel eine reguläre Grammatik mit einer Regel $A \to aabB$ hätten, müssten wir eine Überführung vom Zustand A zum Zustand B mit aab erzeugen, was nicht möglich ist.

Um die Idee der Invertierung des Beweises von Satz 4.28 jedoch nicht fallen zu lassen und zu zeigen, dass reguläre Sprachen von DEA akzeptiert werden können, führen wir einfach neue Modelle ein, um die Schwierigkeiten 1 und 2 zu überwinden.

Um die Schwierigkeit 2 zu beseitigen, betrachten wir eine Normalform von regulären Grammatiken:

Satz 4.30 *Zu jeder regulären (Typ-3) Grammatik $G = (V, \Sigma, P, S)$ gibt es eine äquivalente Grammatik $G' = (V', \Sigma, P', S')$, die nur Regeln der Form $A \to aB$ oder $A \to a$ mit $A, B \in V'$ und $a \in \Sigma$ hat, mit der Ausnahme $S' \to \varepsilon$, falls S' nicht auf der rechten Seite einer Regel vorkommt.*

Beweis. Der Beweis würde in zwei Schritten ablaufen: zuerst müssen wir die Regeln $A \to \varepsilon$ für $A \neq S$ der Grammatik G beseitigen. Das wollen wir hier nicht ausführen, sondern auf den entsprechenden Satz für kontextfreie Sprachen verweisen, dessen Beweis hier vollständig übernommen werden kann.

Zweitens müssen wir dann die Regeln $A \to wB$ oder $A \to w$ mit $|w| \geq 2$ ersetzen. Sei also $A \to a_1a_2 \dots a_nB$ eine solche Regel mit $A, B \in V$ und $n \geq 2$, dann nehmen wir neue Nichtterminale $A_1, A_2, \dots, A_{n-1}$ in die Menge V' auf, die noch nicht verwendet wurden und ersetzen die Regel $A \to a_1a_2 \dots a_nB$ durch die Regeln

$$A \to a_1A_1,\ A_1 \to a_2A_2\ , \dots,\ A_{n-1} \to a_nB$$

in P'. Entsprechend ersetzt man eine Regel $A \to a_1a_2 \dots a_n$ mit $A \in V$ und $n \geq 2$ durch die Regeln

$$A \to a_1B_1,\ B_1 \to a_2B_2\ , \dots,\ B_{n-1} \to a_n$$

mit den neuen Nichtterminalen $B_1, B_2, \dots, B_{n-1}$ in V'.

Man kann dann leicht zeigen, dass dann die Regeln $A \to a_1 a_2 \dots a_n B$ und $A \to a_1 a_2 \dots a_n$ durch die Ableitungen

$$A \Longrightarrow a_1 A_1 \Longrightarrow a_1 a_2 A_2 \Longrightarrow \cdots \Longrightarrow a_1 a_2 a_3 \dots a_{n-1} A_{n-1} \Longrightarrow a_1 a_2 a_3 \dots a_{n-1} a_n B$$

bzw.

$$A \Longrightarrow a_1 B_1 \Longrightarrow a_1 a_2 B_2 \Longrightarrow \cdots \Longrightarrow a_1 a_2 a_3 \dots a_{n-1} B_{n-1} \Longrightarrow a_1 a_2 a_3 \dots a_{n-1} a_n$$

simuliert werden und andererseits aber keine neuen Wörter durch die Grammatik G' erzeugt werden können, also $L(G) = L(G')$ gilt. □

Zur Umgehung der oben genannten Schwierigkeit 1 führen wir ein neues Automatenmodell ein, indem wie den Begriff des DEA erweitern und solche Überführungen $\delta(A, a) = B$ und $\delta(A, a) = C$ gleichzeitig zulassen, indem wir *Nichtdeterminismus* benutzen.

Definition 4.31 (Nichtdeterministischer endlicher Automat) *Ein nichtdeterministischer endlicher Automat (kurz mit NEA bezeichnet) A ist ein 5-Tupel $A = (Z, \Sigma, \delta, z_0, E)$, wobei Z das Zustandsalphabet und Σ das Eingabealphabet mit $Z \cap \Sigma = \emptyset$ sind. $z_0 \in Z$ ist der Anfangszustand, $E \subseteq Z$ die Menge der Endzustände. Mit δ bezeichnen wir die Zustandsüberführungsfunktion $\delta \colon Z \times \Sigma \to 2^Z$.*

Wie man an der Definition erkennt, ist der NEA gar nicht so weit vom DEA entfernt. Der einzige Unterschied liegt in der Definition der Überführungsfunktion δ. Funktionswerte von δ sind nicht einzelne Zustände (wie beim DEA) sondern *Mengen von Zuständen*, d. h. $\delta(z, a) = \{z_1, z_2, \dots, z_r\}$ mit $\{z_1, z_2, \dots, z_r\} \subseteq Z$. Wir bemerken, dass $\delta(z, a)$ auch die leere Menge sein kann.

Auch den NEA können wir wiederum in derselben Art und Weise wie beim DEA als Graph darstellen, wobei von einem Zustand für ein und dasselbe Symbol mehrere Pfeile ausgehen können (oder auch keiner), siehe Abbildung 4.8.

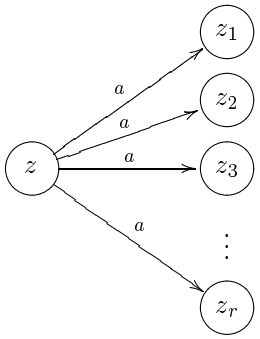


Abbildung 4.8: Nichtdeterminismus beim endlichen Automaten

Die Interpretation der Arbeitsweise des NEA ist analog der des DEA, wobei wir $\delta(z, a) = \{z_1, z_2, \dots, z_r\}$ so interpretieren, dass der Automat, wenn er sich im Zustand z befindet und ein a einliest, in einen der Zustände $z_1, z_2, \dots, z_r$ übergehen kann. Das heisst, der NEA kann ein und dasselbe Wort über *verschiedene* Wege einlesen, wobei alle Wege gleichwertig sein sollen. Das hat natürlich zur Folge, dass der NEA beim Einlesen ein und desselben Wortes einmal einen akzeptierenden Zustand erreichen kann und einmal nicht. Wir werden sagen, dass der NEA genau dann *ein Wort akzeptiert*, wenn er beim Einlesen des Wortes einen akzeptierenden Zustand erreichen *kann*, also *wenn es für das Wort im Graphen einen Pfad vom Anfangszustand zu einem Endzustand gibt.* Man kann diesen Nichtdeterminismus in gewisser Weise als Parallelverarbeitung auffassen.

Um die von einem NEA akzeptierte Sprache formal definieren zu können, benötigen wir wieder die erweiterte Zustandsfunktion $\hat{\delta} \colon Z \times \Sigma^* \to 2^Z$. Im Prinzip wird sie wieder wie beim NEA definiert, allerdings ist ja jetzt $\hat{\delta}(z, w)$ eine Menge von Zuständen und kann nicht direkt als Argument verwendet werden.

Definition 4.32 (Erweiterte Zustandsüberführungsfunktion eines NEA) *Sei A ein NEA mit $A = (Z, \Sigma, \delta, z_0, E)$, die erweiterte Zustandsfunktion $\hat{\delta}\colon Z \times \Sigma^* \to 2^Z$ wird definiert durch*

(i) $\hat{\delta}(z, \varepsilon) = \{z\}$ *für alle $z \in Z$,*

(ii) $\hat{\delta}(z, wa) = \bigcup_{z' \in \hat{\delta}(z,w)} \delta(z', a)$ *das heißt*
$= \{z'' \in Z \mid \exists z' \in Z$ *mit* $z' \in \hat{\delta}(z, w)$ *und* $z'' \in \delta(z', a)\}$ *für $z \in Z$, $a \in \Sigma$, $w \in \Sigma^*$.*

Definition 4.33 (Akzeptierte Sprache eines NEA) *Für einen NEA $A = (Z, \Sigma, \delta, z_0, E)$ sei die von ihm akzeptierte Sprache $T(A)$ definiert durch $T(A) = \{w \in \Sigma^* \mid \hat{\delta}(z_0, w) \cap E \neq \emptyset\}$.*

Bemerkung 4.34 Bitte beachten Sie, dass die Definitionen des NEA sowie der erweiterten Zustandsfunktion von den Definitionen bei Schöning in [13] etwas abweichen. Man kann aber sehr leicht zeigen, dass die Klasse der akzeptierbaren Mengen gleich sind.

Sehen wir uns ein Beispiel an, um den Mechanismus des Nichtdeterminismus besser zu verstehen.

Beispiel 4.35 Sei $A = (\{z_0, z_1, z_2\}, \{0, 1\}, \delta, z_0, \{z_2\})$, wobei δ durch die Tabelle

δ	z_0	z_1	z_2
0	$\{z_0\}$	$\{z_2\}$	$\emptyset$
1	$\{z_0, z_1\}$	$\{z_2\}$	$\emptyset$

gegeben ist, ein nichtdeterministischer endlicher Automat. Der dazugehörige Graph ist in der Abbildung 4.9 dargestellt. Betrachten wir nun die Eingabe 111 und fragen nach $\hat{\delta}(z_0, 111)$, also

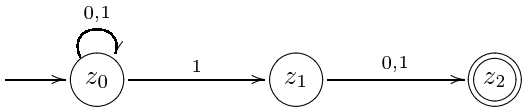


Abbildung 4.9: Überführungsgraph des NEA aus Beispiel 4.35

nach den Zuständen, die bei der Eingabe von 111 durch den NEA erreicht werden können. Beim Einlesen des ersten Symbols (eine 1) kann der Automat wählen zwischen dem Folgezustand z_0 oder z_1. Im letzteren Fall liest er dann die zweite 1 ein und landet (keine Wahlmöglichkeit) im Zustand z_2, von dem aus er die dritte 1 nicht mehr einlesen kann (es gibt keine Überführung mehr), also erreicht er über diesen Weg *keinen* Zustand. Im ersteren Fall jedoch kann er bei der Eingabe der zweiten 1 wiederum wählen zwischen den Folgezuständen z_0 und z_1: Wählt er z_1, landet er mit der letzten 1 in z_2, wählt er jedoch z_0, so kann er bei der letzten 1 wiederum zwischen den Folgezuständen z_0 und z_1 wählen. Summa summarum kann der Automat beim Einlesen von 111 die Zustände z_0, z_1, z_2 erreichen, also

$$\hat{\delta}(z_0, 111) = \{z_0, z_1, z_2\}.$$

Nun gilt

$$\hat{\delta}(z_0, 111) \cap E = \{z_0, z_1, z_2\} \cap \{z_2\} = \{z_2\} \neq \emptyset,$$

das heißt, für die Eingabe 111 gibt es einen Weg vom Anfangszustand in einen akzeptierenden Zustand, also gilt $111 \in T(A)$, d. h. die Eingabe 111 wird akzeptiert, gehört also zur akzeptierten Sprache.

Für weitere Eingaben gilt:

$$\begin{aligned}\hat{\delta}(z_0, 0) &= \{z_0\},\\ \hat{\delta}(z_0, 1) &= \{z_0, z_1\},\\ \hat{\delta}(z_0, 01) &= \{z_1\},\\ \hat{\delta}(z_0, 11) &= \{z_0, z_1, z_2\},\\ \hat{\delta}(z_0, 001) &= \{z_0, z_1\},\\ \hat{\delta}(z_0, 011) &= \{z_0, z_1, z_2\},\end{aligned}$$

also werden 11 sowie 011 akzeptiert und 0, 1, 01 sowie 001 nicht akzeptiert. Für die akzeptierte Sprache $T(A)$ gilt:

$$T(A) = \{w \in \{0,1\}^* \mid w = u1x \text{ mit } u \in \{0,1\}^*,\ x \in \{0,1\}\},$$

also ist $T(A)$ die Menge aller Wörter über $\{0,1\}$, deren vorletztes Symbol eine 1 ist.

Man kann jeden DEA natürlich als NEA auffassen, nämlich für den dann $\delta(z,a)$ immer eine Einermenge ist. Also:

Folgerung 4.36 *Jede von einem deterministischen endlichen Automaten akzeptierbare Sprache ist auch von einem nichtdeterministischen endlichen Automaten akzeptierbar.* □

Die Menge, die vom NEA im Beispiel 4.35 akzeptiert wurde, kann auch von einem DEA akzeptiert werden (siehe Beispiel 4.27). Natürlich ergibt sich sofort die Frage, ob jede von einem NEA akzeptierte Sprache auch von einem DEA akzeptiert werden kann. Die Antwort liefert der folgende Satz.

Satz 4.37 *Jede von einem nichtdeterministischen endlichen Automaten akzeptierbare Sprache ist auch von einem deterministischen endlichen Automaten akzeptierbar.*

Beweis. Sei $A = (Z, \Sigma, \delta, z_0, E)$ ein NEA. Wir konstruieren einen DEA $A' = (Z', \Sigma, \delta', z_0', E')$ durch:

$$\begin{aligned}Z' &= 2^Z,\\ z_0' &= \{z_0\},\\ E' &= \{z' \in Z' \mid z' \cap E \neq \emptyset\},\end{aligned}$$

sowie

$$\delta'(z', a) = \bigcup_{z \in z'} \delta(z, a)$$

für alle $z' \in Z'$ und $a \in \Sigma$. Dann gilt

$$T(A) = T(A'),$$

was wir an dieser Stelle nicht beweisen werden. □

Im obigen Beweis ist die Zustandsmenge des konstruierten DEA genau die Potenzmenge der Zustandsmenge des gegebenen NEA. Falls man zu einem konkret gegebenen DEA den äquivalenten NEA konstruiert, stellt man fest, dass oft nicht alle Teilmengen von Z auch wirklich erreicht werden können. Folglich reicht es aus, wenn wir, beginnend mit der Menge $\{z_0\}$ jeweils für alle $z' \in Z'$ die Teilmengen $\delta'(z', x)$ für alle $x \in \Sigma$ berechnen und die neu erzeugten Teilmengen in die Menge der Zustände Z' aufnehmen, falls sie noch nicht enthalten sind. Kommt kein neuer Zustand mehr hinzu, wären wir fertig mit der Konstruktion von δ'. Wir wollen dieses Vorgehen an einem Beispiel demonstrieren.

Beispiel 4.38 Wir nehmen den NEA

$$A = (\{z_0, z_1, z_2\}, \{0, 1\}, \delta, z_0, \{z_2\})$$

aus Beispiel 4.35 mit der in folgender Tabelle gegebenen Überführungsfunktion δ.

δ	z_0	z_1	z_2
0	$\{z_0\}$	$\{z_2\}$	$\emptyset$
1	$\{z_0, z_1\}$	$\{z_2\}$	$\emptyset$

Wir konstruieren jetzt den äquivalenten DEA

$$A' = (Z', \{0, 1\}, \delta', z', E')$$

gemäß Beweis des Satzes 4.37. Zuerst gilt $z'_0 = \{z_0\}$. Die weiteren Zustände sowie die Überführungsfunktion berechnen wir entsprechend der Definition in folgender Tabelle (spaltenweise).

δ'	$\{z_0\}$	$\{z_0, z_1\}$	$\{z_0, z_2\}$	$\{z_0, z_1, z_2\}$
0	$\{z_0\}$	$\{z_0, z_2\}$	$\{z_0\}$	$\{z_0, z_2\}$
1	$\{z_0, z_1\}$	$\{z_0, z_1, z_2\}$	$\{z_0, z_1\}$	$\{z_0, z_1, z_2\}$

Also gilt

$$Z' = \{\{z_0\}, \{z_0, z_1\}, \{z_0, z_2\}, \{z_0, z_1, z_2\}\}$$

und da nur die Zustände $\{z_0, z_2\}$ und $\{z_0, z_1, z_2\}$ einen Zustand aus E enthalten, sind sie die einzigen neuen Endzustände, also

$$E' = \{\{z_0, z_2\}, \{z_0, z_1, z_2\}\}.$$

Damit wäre der äquivalente DEA A' zum NEA A konstruiert. Zur besseren Lesbarkeit bezeichnen wir die Zustände um: $\{z_0\} =: q_0$, $\{z_0, z_1\} =: q_1$, $\{z_0, z_2\} =: q_2$ und $\{z_0, z_1, z_2\} =: q_3$. Dann gilt

$$A' = (\{q_0, q_1, q_2, q_3\}, \{0, 1\}, \delta', q_0, \{q_2, q_3\})$$

mit

δ'	q_0	q_1	q_2	q_3
0	q_0	q_2	q_0	q_2
1	q_1	q_3	q_1	q_3

In Abbildung 4.10 finden Sie den Graphen zum Automaten A'. Man erkennt, dass die Automaten in

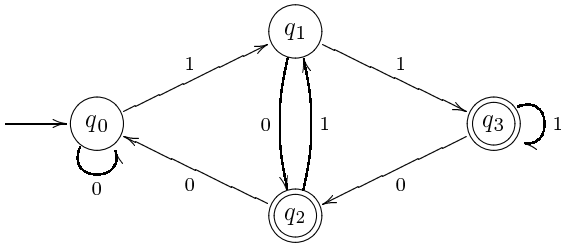


Abbildung 4.10: Überführungsgraph des DEA A' aus Beispiel 4.38

den Abbildungen 4.7 und 4.10 bis auf Bezeichnungen der Zustände identisch sind. Also akzeptieren sie auch die gleiche Sprache.

Bemerkung 4.39 1. Beim Konstruieren des äquivalenten DEA zum gegebenen NEA im obigen Beispiel haben wir den gleichen Automaten (bis auf Bezeichnungen) erhalten, den wir auch schon vorher betrachtet hatten. Das muss natürlich nicht immer sein. Insbesondere erhält man im Allgemeinen bei dieser Konstruktion Automaten, die nicht *minimal* in dem Sinne sind, dass man äquivalente DEA finden kann, die eventuell weniger Zustände haben.

2. Im obigen Beispiel hatte der NEA drei Zustände, der DEA vier. Führt man den gleichen Übergang vom NEA zum DEA für die (von der Struktur gleiche) Sprache

$$T(A) = \{w \in \{0,1\}^* \mid w = u1v \text{ mit } u \in \{0,1\}^*,\ v \in \{0,1\}^9\}$$

durch, also für die Menge aller Wörter über dem Alphabet $\{0,1\}$, deren zehntletztes Symbol eine 1 ist, so benötigt der NEA 11 Zustände, der DEA aber 2^{10} Zustände. Es lässt sich zeigen, dass es keinen DEA für diese Sprache mit weniger Zustände gibt. Dieses Resultat kann man für beliebiges n verallgemeinern. Die Anzahl der Zustände beim Übergang vom NEA zum DEA kann also *exponentiell wachsen*.

Wir haben das Modell des nichtdeterministischen endlichen Automaten eingeführt, weil wir letztendlich beweisen wollten, dass die Klasse der Typ-3-Sprachen und die Klasse der Sprachen, die von deterministischen endlichen Automaten akzeptiert werden, identisch sind. Mit dem folgenden Satz kommen wir diesem Beweis sehr nahe.

Satz 4.40 *Sei G eine reguläre Grammatik, also vom Typ 3, dann existiert ein nichtdeterministischer endlicher Automat A mit $T(A) = L(G)$.*

Beweis. Wie bereits angekündigt, benutzt der Beweis eigentlich die gleiche Idee wie beim Übergang vom DEA zur regulären Grammatik. Sei $G = (V, \Sigma, P, S)$ die reguläre Grammatik. Wir konstruieren einen NEA $A = (Z, \Sigma, \delta, z_0, E)$ wie folgt:

$$Z = V \cup \{X\},$$
$$z_0 = S,$$
$$E = \begin{cases} \{S, X\} & \text{für } S \to \varepsilon \in P, \\ \{X\} & \text{für } S \to \varepsilon \notin P, \end{cases}$$

Des weiteren definieren wir die Überführungsfunktion δ durch

$$\delta(A, a) = \begin{cases} \{B \mid A \to aB \in P\} \cup \{X\} & \text{für } A \to a \in P, \\ \{B \mid A \to aB \in P\} & \text{für } A \to a \notin P, \end{cases}$$

für $A \in V$ und $a \in \Sigma$.

Jetzt könnten und müssten wir beweisen, dass $T(A) = L(G)$ gilt, worauf wir aber an dieser Stelle wiederum verzichten wollen. □

Die Abbildung 4.11 fasst die Ergebnisse der Sätze 4.28, 4.40 sowie 4.37 zusammen. Damit gilt:

Folgerung 4.41 *Die Klasse der regulären Sprachen* (**Typ 3**) *ist gleich der Klasse der von nichtdeterministischen endlichen Automaten akzeptierten Sprachen* ($\mathcal{L}$(NEA)) *und der Klasse der von deterministischen endlichen Automaten akzeptierten Sprachen* ($\mathcal{L}$(DEA))*, also*

$$\textbf{Typ 3} = \mathcal{L}(\text{NEA}) = \mathcal{L}(\text{DEA}).$$

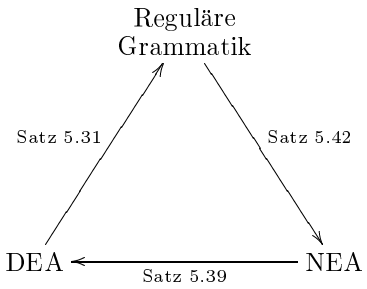


Abbildung 4.11: Beweisschema für Mechanismen zur Beschreibung regulärer Sprachen

4.3.3 Reguläre Ausdrücke

Nachdem wir in den vorhergehenden Kapiteln die Menge der regulären Sprachen (oder Typ-3-Sprachen) durch Grammatiken und Automaten beschrieben haben, wollen wir in diesem Kapitel eine Beschreibungsart betrachten, die algebraischer Natur ist, nämlich die regulären Ausdrücke. Obwohl die Beschreibung auf algebraische Operationen zurückgeht, also eigentlich recht mathematischer Natur ist, werden reguläre Ausdrücke in vielen Gebieten der Informatik angewendet, zum Beispiel bei der Suche in Editoren.

Die Menge der regulären Ausdrücke über einem Alphabet Σ werden wir induktiv definieren.

Definition 4.42 (Reguläre Ausdrücke) *Sei Σ ein Alphabet, dann gilt:*

(i) *$\emptyset$ ist ein regulärer Ausdruck über Σ.*

(ii) *ε ist ein regulärer Ausdruck über Σ.*

(iii) *Für jedes $a \in \Sigma$ ist a ein regulärer Ausdruck über Σ.*

(iv) *Wenn α und β reguläre Ausdrücke über Σ sind, so auch $\alpha\beta$, $(\alpha \mid \beta)$ und $(\alpha)^*$.*

Reguläre Ausdrücke über einem Alphabet Σ sind also erst einmal nur Wörter spezieller Art über diesem Alphabet. Nun ordnen wir solch einem Wort eine Sprache zu. Die Definition dieser Zuordnung erfolgt wiederum rekursiv.

Definition 4.43 (Sprache eines regulären Ausdrucks) *Sei Σ ein Alphabet und γ ein regulärer Ausdruck über Σ, dann wird die von γ beschriebene Sprache $L(\gamma) \subseteq \Sigma^*$ wie folgt definiert.*

(i) *Für $\gamma = \emptyset$ gilt $L(\gamma) = \emptyset$.*

(ii) *Für $\gamma = \varepsilon$ gilt $L(\gamma) = \{\varepsilon\}$.*

(iii) *Für $\gamma = a$ mit $a \in \Sigma$ gilt $L(\gamma) = \{a\}$.*

(iv) *Für $\gamma = \alpha\beta$ gilt $L(\gamma) = L(\alpha) \cdot L(\beta)$.*

(v) *Für $\gamma = (\alpha \mid \beta)$ gilt $L(\gamma) = L(\alpha) \cup L(\beta)$.*

(vi) *Für $\gamma = (\alpha)^*$ gilt $L(\gamma) = (L(\alpha))^*$.*

Beispiel 4.44 Wir betrachten den regulären Ausdruck

$$(0 \mid (0 \mid 1)^*00).$$

Dann können wir die zugeordnete Sprache wie folgt gemäß der Definition bilden (hier für ein erstes

Beispiel sehr ausführlich aufgeschrieben):

$$\begin{aligned}
L((0 \mid (0 \mid 1)^*00)) &= L(0) \cup L((0 \mid 1)^*00)) \\
&= L(0) \cup (L((0 \mid 1)^*0)) \cdot L(0)) \\
&= L(0) \cup ((L((0 \mid 1)^*)) \cdot L(0)) \cdot L(0)) \\
&= L(0) \cup (((L((0 \mid 1)))^* \cdot L(0)) \cdot L(0)) \\
&= L(0) \cup (((L(0) \cup L(1))^* \cdot L(0)) \cdot L(0)) \\
&= \{0\} \cup ((((\{0\} \cup \{1\})^* \cdot \{0\}) \cdot \{0\}) \\
&= \{0\} \cup ((\{0,1\}^* \cdot \{0\}) \cdot \{0\}) \\
&= \{0\} \cup (\{0,1\}^* \cdot \{00\}),
\end{aligned}$$

das heißt, die vom regulären Ausdruck $(0 \mid (0 \mid 1)^*00)$ beschriebene Sprache ist die Menge aller Wörter über dem Alphabet $\{0,1\}$, die gleich 0 sind oder auf 00 enden.

Bemerkung 4.45 1. Wir vereinbaren, dass wir Klammern, die nicht notwendigerweise gebraucht werden, weglassen können. Zum Beispiel können wir statt $(\alpha \mid (\beta \mid \gamma))$ auch $(\alpha \mid \beta \mid \gamma)$ schreiben. Wir schreiben auch $L(\alpha \mid \beta)$ statt $L((\alpha \mid \beta))$ sowie a^* statt $(a)^*$.

2. Wir benutzen die abkürzende Schreibweise α^n für $\underbrace{\alpha\alpha\ldots\alpha}_{n\text{-mal}}$.

3. Wir benutzen die abkürzende Schreibweise α^+ für $\alpha^*\alpha$.

4. In der Literatur findet man oft auch abweichende Definitionen der regulären Ausdrücke. Zum Beispiel findet man für $(\alpha \mid \beta)$ auch $(\alpha + \beta)$ oder auch $(\alpha \cup \beta)$. Auch wird natürlich oft $\alpha \cdot \beta$ für $\alpha\beta$ zugelassen.

5. Oft wird in der Literatur zwischen regulärem Ausdruck und beschriebener Sprache nicht unterschieden, das heißt, man identifiziert einen regulären Ausdruck mit der beschriebenen Sprache.

Wir geben noch ein paar weitere Beispiele an:

Beispiel 4.46 Weitere Beispiele für reguläre Ausdrücke über $\Sigma = \{a, b\}$ und deren zugeordneten Sprachen sind:

$(a \mid b)^*$ beschreibt die Menge aller Wörter über dem Alphabet $\{a, b\}$.

$(a \mid b)^+$ beschreibt die Menge aller Wörter über dem Alphabet $\{a, b\}$, die nicht dem leeren Wort entsprechen.

$(a \mid b)^*aba(a \mid b)^*$ beschreibt die Menge aller Wörter über dem Alphabet $\{a, b\}$, die das Teilwort aba haben.

$(a \mid b)^*a(a \mid b)^2$ beschreibt die Menge aller Wörter über dem Alphabet $\{a, b\}$, deren drittletztes Symbol ein a ist.

$((a \mid b)(a \mid b))^*$ beschreibt die Menge aller Wörter über dem Alphabet $\{a, b\}$, deren Länge gerade ist.

$(b \mid \varepsilon)(ab)^*(a \mid \varepsilon)$ beschreibt die Menge aller Wörter über dem Alphabet $\{a, b\}$, die nicht das Teilwort aa und nicht das Teilwort bb enthalten.

Wenn man sich die regulären Ausdrücke und die beschriebenen Sprachen im obigen Beispiel anschaut, erkennt man, dass alle Sprachen vom Typ 3 sind, also regulär. Man stellt sich natürlich die Frage, ob alle durch reguläre Ausdrücke beschreibbaren Sprachen regulär sind und ob umgekehrt für jede reguläre Sprache ein regulärer Ausdruck existiert, der sie beschreibt. Die Bezeichnung *reguläre* Ausdrücke suggeriert natürlich die Antwort.

Satz 4.47 (Kleene) *Die Menge der durch reguläre Ausdrücke beschreibbaren Sprachen ist genau die Menge der regulären Sprachen.*

Beweis. Der Beweis muss in zwei Richtungen geführt werden.

Teil 1: Einerseits muss gezeigt werden, dass jeder reguläre Ausdruck eine reguläre Sprache beschreibt. Diesen Teil wollen wir an dieser Stelle skizzieren. Wir werden zeigen, dass zu jedem regulären Ausdruck ein NEA existiert, der genau die vom regulären Ausdruck beschriebene Sprache akzeptiert, womit wegen Folgerung 4.41 diese Sprache regulär oder vom Typ 3 ist.

Sei Σ ein Alphabet und γ ein regulärer Ausdruck über Σ. Wir betrachten zuerst die Fälle, in denen γ die Form $\gamma = \emptyset$, $\gamma = \varepsilon$ oder $\gamma = a$ mit $a \in \Sigma$ hat. In Abbildung 4.12 sind NEA's

Abbildung 4.12: DEA's für $L(\emptyset)$, $L(\varepsilon)$ sowie $L(a)$ (von links nach rechts)

angegeben, die jeweils die Mengen $L(\emptyset) = \emptyset$, $L(\varepsilon) = \{\varepsilon\}$ sowie $L(a) = \{a\}$ akzeptieren.

Wir nehmen jetzt an, dass γ ein regulärer Ausdruck ist, der aus regulären Ausdrücken zusammengesetzt ist. Dann gibt es für γ die drei Fälle $\gamma = \alpha\beta$, $\gamma = (\alpha \mid \beta)$ und $\gamma = (\alpha)^*$.

Sei zunächst γ ein regulärer Ausdruck der Form $\gamma = \alpha\beta$. Dabei können wir annehmen, dass es bereits NEA's für $L(\alpha)$ und $L(\beta)$ gibt, also dass NEA's A_α und A_β existieren mit $T(A_\alpha) = L(\alpha)$ und $T(A_\beta) = L(\beta)$. Nun konstruieren wir den Automaten A, indem wir die Automaten A_α und

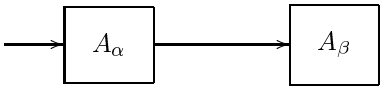


Abbildung 4.13: Schema des NEA's für $L(\alpha\beta)$

A_β im Prinzip hintereinander schalten („in Reihe") (siehe Schema in Abbildung 4.13).

Die genaue Konstruktion für den NEA A ist folgende: Die Automaten $A_\alpha = (Z_\alpha, \Sigma, \delta_\alpha, z_{\alpha 0}, E_\alpha)$ und $A_\beta = (Z_\beta, \Sigma, \delta_\beta, z_{\beta 0}, E_\beta)$ seien die Automaten mit $T(A_\alpha) = L(\alpha)$ und $T(A_\beta) = L(\beta)$. Wir konstruieren $A = (Z, \Sigma, \delta, z_0, E)$ wie folgt.

- Jeder Zustand von A_α und A_β ist auch Zustand von A, also $Z = Z_\alpha \cup Z_\beta$.
- Der Anfangszustand von A_α ist auch Anfangszustand für A, also $z_0 = z_{\alpha 0}$.
- Die Menge der Endzustände von A_β wird die Menge der Endzustände von A, also $E = E_\beta$.
- Alle Überführungen von A_α und A_β gelten auch für A. Von allen Zuständen von A_α, für die es Überführungen zu Endzuständen von A_α gibt, gibt es zusätzlich Überführungen für das gleiche Symbol zum Anfangszustand von A_β. Formal gilt für alle $z \in Z$ und $a \in \Sigma$:

$$\delta(z,a) = \begin{cases} \delta_\beta(z,a) & \text{für } z \in Z_\beta, \\ \delta_\alpha(z,a) & \text{für } z \in Z_\alpha \text{ und } \delta_\alpha(z,a) \cap E_\alpha = \emptyset, \\ \delta_\alpha(z,a) \cup \{z_{\beta 0}\} & \text{für } z \in Z_\alpha \text{ und } \delta_\alpha(z,a) \cap E_\alpha \neq \emptyset. \end{cases}$$

Für den Fall $\varepsilon \in L(\alpha)$ muss man noch Sonderregelungen treffen, darauf verzichten wir hier.

Wie man nachweisen kann, gilt dann $T(A) = L(\alpha\beta)$, das heißt A akzeptiert ein Wort genau dann, wenn ein erster Teil des Wortes von A_α und der Rest des Wortes von A_β akzeptiert wird.

Habe γ nun die Form $\gamma = (\alpha \mid \beta)$. Wir setzen wieder voraus, dass die Automaten $A_\alpha = (Z_\alpha, \Sigma, \delta_\alpha, z_{\alpha 0}, E_\alpha)$ und $A_\beta = (Z_\beta, \Sigma, \delta_\beta, z_{\beta 0}, E_\beta)$ die Automaten mit $T(A_\alpha) = L(\alpha)$ und $T(A_\beta) = L(\beta)$ seien. Wir konstruieren dann den Automaten A in folgender Art und Weise (wir schalten die Automaten A_α und A_β im Prinzip „parallel", siehe Abbildung 4.14):

$$A = (Z_\alpha \cup Z_\beta \cup \{z_0\}, \Sigma, \delta, z_0, E_\alpha \cup E_\beta),$$

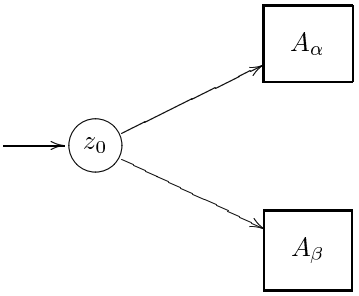


Abbildung 4.14: Schema des NEA's für $L((\alpha \mid \beta))$

die Funktion δ ist dabei wie folgt definiert.

$$\delta(z,a) = \begin{cases} \delta_\alpha(z,a) & \text{für } z \in Z_\alpha, \\ \delta_\beta(z,a) & \text{für } z \in Z_\beta, \\ \delta_\alpha(z_{\alpha 0},a) \cup \delta_\beta(z_{\beta 0},a) & \text{für } z = z_0. \end{cases}$$

Der Automat akzeptiert dann ein Wort genau dann, wenn es von A_α *oder* A_β akzeptiert wird. Also gilt $T(A) = A_\alpha \cup A_\beta$, das heißt $T(A) = L((\alpha \mid \beta))$.

Habe γ nun die Form $\gamma = (\alpha)^*$. Wir setzen wiederum voraus, dass der Automat $A_\alpha = (Z_\alpha, \Sigma, \delta_\alpha, z_{\alpha 0}, E_\alpha)$ der Automat mit $T(A_\alpha) = L(\alpha)$ ist. Wir konstruieren A, indem wir im Prinzip

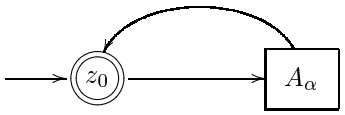


Abbildung 4.15: Schema des NEA's für $L((\alpha)^*)$

eine Schleife erzeugen (siehe Abbildung 4.15). Exakt wird dann $A = (Z, \Sigma, \delta, z_0, E)$ folgendermaßen konstruiert.

$$Z = Z_\alpha \cup \{z_0\}, \qquad E = \{z_0\}$$

und für die Überführungsfunktion δ gilt für alle $z \in Z$ und $a \in \Sigma$:

$$\delta(z,a) = \begin{cases} \delta_\alpha(z_{\alpha 0},a) & \text{für } z = z_0, \\ \delta_\alpha(z,a) & \text{für } z \in Z_\alpha \text{ und } \delta_\alpha(z,a) \cap E_\alpha = \emptyset, \\ \delta_\alpha(z,a) \cup \{z_0\} & \text{für } z \in Z_\alpha \text{ und } \delta_\alpha(z,a) \cap E_\alpha \neq \emptyset. \end{cases}$$

Zur Interpretation: Anfangszustand ist also ein neuer Zustand, der auch gleichzeitig der einzige Endzustand ist. Vom Anfangszustand gibt es dann die gleichen Überführungen wie vom Anfangszustand des Automaten A_α. Von allen Zuständen von A_α, für die es Überführungen zu Endzuständen von A_α gibt, gibt es zusätzlich Überführungen für das gleiche Symbol zum Anfangszustand z_0.

Damit wird das leere Wort akzeptiert (wegen $z_0 \in E$) und auch alle Wörter, die auch A_α akzeptiert. Weiter können wir natürlich die Schleife mehrmals durchlaufen, das heißt für die akzeptierte Sprache von A gilt

$$\begin{aligned} T(A) &= \{\varepsilon\} \cup T(A_\alpha) \cup (T(A_\alpha))^2 \cup (T(A_\alpha))^3 \cup \cdots \\ &= (T(A_\alpha))^0 \cup T(A_\alpha)^1 \cup (T(A_\alpha))^2 \cup (T(A_\alpha))^3 \cup \cdots \\ &= (T(A_\alpha))^*, \end{aligned}$$

also $T(A) = (L(\alpha))^*$.

Damit hätten wir alle Fälle behandelt und haben den ersten Teil des Beweises vollendet (abgesehen von den Lücken, die wir hier nicht vollständig bewiesen haben). Das heißt, wir haben gezeigt, zu jedem regulären Ausdruck gibt es einen äquivalenten nichtdeterministischen endlichen Automaten, also ist jede von einem regulären Ausdruck beschriebene Sprache regulär.

Noch eine allgemeine Bemerkung zu den obigen Konstruktionen, natürlich müssen wir $Z_\alpha \cap Z_\beta = \emptyset$ fordern, und falls wir einen Anfangszustand z_0 neu hinzunehmen, darf er natürlich in den gegebenen Automaten nicht vorkommen. Das ist aber keine Einschränkung, da wir durch einfache Umbenennungen der Zustände diese Bedingungen immer absichern können.

Teil 2: Im zweiten Teil des Beweises zeigen wir, dass jede reguläre Sprache von einem regulären Ausdruck beschrieben wird.

Sei L eine reguläre Sprache. Dann wird L von einem deterministischen endlichen Automaten $A = (Z, \Sigma, \delta, z_0, E)$ akzeptiert, wobei $\Sigma = \{a_1, a_2, \ldots, a_k\}$ gelte. Für jeden Zustand $z \in Z$ definieren wir die Wortmenge $L_z = \{w \in \Sigma^* \mid \hat{\delta}(z, w) \in E\}$. Für alle diese L_z werden wir zeigen, dass sie von regulären Ausdrücken beschrieben werden können. Wegen $L = T(A) = L_{z_0}$ würde dann die Behauptung folgen.

Die Mengen L_z stehen in folgenden Beziehungen zueinander.

$$\begin{aligned} L_z &= \bigcup_{i=1}^{k} \{a_i\} \cdot L_{\delta(z,a_i)} && \text{für } z \notin E \text{ und} \\ L_z &= \bigcup_{i=1}^{k} \{a_i\} \cdot L_{\delta(z,a_i)} \cup \{\varepsilon\} && \text{für } z \in E. \end{aligned} \tag{4.2}$$

Wir werden jetzt dieses System von Gleichungen, in denen die L_z als Unbekannte auftreten, schrittweise auflösen, wobei wir nur die Operationen Vereinigung, Konkatenation und Iteration verwenden, so dass letztendlich nur L_{z_0} stehen bleibt und somit von einem regulären Ausdruck beschrieben werden kann.

Die Tatsache, dass Unbekannte auf beiden Seiten ein und derselben Gleichung auftreten können, führt dazu, dass hier ein „normales“ Eliminieren nicht möglich ist.

Hier hilft folgendes Lemma,

Lemma 4.48 *Für $B, C \in \Sigma^*$ gilt: Ist $\varepsilon \notin B$, so ist $L = B^* \cdot C$ die einzige Lösung der Gleichung $L = B \cdot L \cup C$.*

Beweis. Der Beweis erfolgt durch das Zeigen der Inklusionen $B^* \cdot C \subseteq L$ sowie $L \subseteq B^* \cdot C$.

Teil 1: $B^* \cdot C \subseteq L$. Wir zeigen durch Induktion über k, dass jedes $B^k \cdot C$ in jeder Lösung L enthalten ist.

Induktionsanfang. Für $k = 0$ gilt:

$$B^0 \cdot C = \{\varepsilon\} \cdot C = C \subseteq B \cdot L \cup C = L.$$

Induktionsschritt. Mit der Induktionsvoraussetzung $B^k \cdot C \subseteq L$ schließt man

$$B^{k+1} \cdot C = B \cdot (B^k \cdot C) \subseteq B \cdot L \subseteq B \cdot L \cup C = L.$$

Teil 2: $L \subseteq B^* \cdot C$. Wir zeigen durch Induktion über die Länge des Wortes w, dass aus $w \in L$ stets $w \in B^* \cdot C$ folgt.

Induktionsanfang. Ist $|w| = 0$, so ist $w = \varepsilon$. Gilt $\varepsilon \in L$, so folgt aus $L = B \cdot L \cup C$ und $\varepsilon \notin B$ sofort $\varepsilon \in C$. Dann ist aber auch $\varepsilon \in B^* \cdot C$.

Induktionsschritt. Es sei $|w| > 0$, und nach Induktionsvoraussetzung folge für alle v mit $|v| < |w|$ aus $v \in L$ stets $v \in B^* \cdot C$.

Es sei nun $w \in L = B \cdot L \cup C$. Im Falle $w \in C$ folgt sofort $w \in B^* \cdot C$. Im Falle $w \in B \cdot L$ gibt es ein $u \in B$ und ein $v \in L$ mit $w = u \cdot v$. Wegen $\varepsilon \notin B$ gilt $|u| > 0$ und folglich $|v| < |w|$. Nach Induktionsvoraussetzung haben wir damit $v \in B^* \cdot C$, und wir schließen $w = u \cdot v \in B \cdot (B^* \cdot C) \subseteq B^* \cdot C$. □

Mit diesem Lemma können wir jetzt L_{z_0} bestimmen, indem wir schrittweise alle anderen Mengenvariablen L_z wie folgt eliminieren. Kommt in der Gleichung, bei der links L_z steht, auch rechts L_z vor, so wird das Lemma angewendet. In jedem Fall hat man dann eine Gleichung $L_z = R$, wobei in R die Variable L_z nicht mehr vorkommt. Ersetzt man jetzt in allen anderen Gleichungen L_z durch R, so ist L_z eliminiert.

Wenn wir alle Variablen bis auf L_{z_0} eliminiert haben, ist L_{z_0} dargestellt als endlich oftmalige Anwendung der Operationen Konkatenation, Vereinigung und Iteration über Symbolen aus Σ. Somit kann $L_{z_0} = T(A)$ durch einen regulären Ausdruck beschrieben werden.

Somit ist der zweite Teil des Satzes von Kleene und damit auch der Satz bewiesen. □

Wir betrachten ein Beispiel für die Konstruktion eines äquivalenten regulären Ausdrucks für einen gegebenen deterministischen endlichen Automaten.

Beispiel 4.49 In der Abbildung 4.16 ist ein DEA über dem Alphabet $\Sigma = \{a, b\}$ durch einen

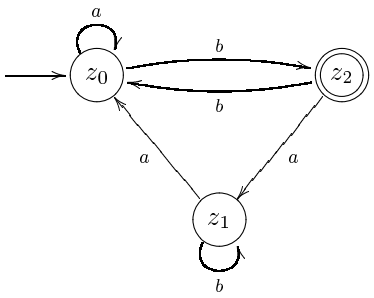


Abbildung 4.16: Ein deterministischer endlicher Automat

Überführungsgraphen gegeben. Das dazugehörige Gleichungssystem für die Wortmengen L_z besteht somit aus den drei Gleichungen

$$\left.\begin{aligned} L_{z_0} &= \{a\} \cdot L_{z_0} \cup \{b\} \cdot L_{z_2}, \\ L_{z_1} &= \{a\} \cdot L_{z_0} \cup \{b\} \cdot L_{z_1}, \\ L_{z_2} &= \{a\} \cdot L_{z_1} \cup \{b\} \cdot L_{z_0} \cup \{\varepsilon\}. \end{aligned}\right\} \tag{4.3}$$

Die Variable (Wortmenge) L_{z_2} in der ersten Gleichung können wir ohne Probleme eliminieren, indem wir sie durch die rechte Seite der dritten Gleichung ersetzen. Somit erhalten wir das zu (4.3) äquivalente Gleichungssystem

$$\begin{aligned} L_{z_0} &= \{a\} \cdot L_{z_0} \cup \{b\} \cdot (\{a\} \cdot L_{z_1} \cup \{b\} \cdot L_{z_0} \cup \{\varepsilon\}), \\ L_{z_1} &= \{a\} \cdot L_{z_0} \cup \{b\} \cdot L_{z_1} \end{aligned}$$

oder, indem wir die rechte Seite der ersten Gleichung zusammenfassen,

$$\begin{aligned} L_{z_0} &= (\{a\} \cup \{bb\}) \cdot L_{z_0} \cup \{ba\} \cdot L_{z_1} \cup \{b\}, \\ L_{z_1} &= \{a\} \cdot L_{z_0} \cup \{b\} \cdot L_{z_1}. \end{aligned}$$

Nun wenden wir auf die zweite Gleichung das Lemma 4.48 an und erhalten $L_{z_1} = \{b\}^*\{a\} \cdot L_{z_0}$. Damit können wir jetzt L_{z_1} in der ersten Gleichung des Gleichungssystems ersetzen.

$$\begin{aligned} L_{z_0} &= (\{a\} \cup \{bb\}) \cdot L_{z_0} \cup \{ba\} \cdot L_{z_1} \cup \{b\} \\ &= (\{a\} \cup \{bb\}) \cdot L_{z_0} \cup \{ba\} \cdot (\{b\}^*\{a\} \cdot L_{z_0}) \cup \{b\} \\ &= (\{a\} \cup \{bb\} \cup \{ba\} \cdot \{b\}^*\{a\}) \cdot L_{z_0} \cup \{b\}. \end{aligned}$$

Nun wenden wir abermals unser Lemma an und erhalten

$$L_{z_0} = (\{a\} \cup \{bb\} \cup \{ba\} \cdot \{b\}^*\{a\})^*\{b\}.$$

Somit können wir L_{z_0}, also die vom gegebenen Automaten akzeptierte Sprache, durch den regulären Ausdruck

$$(a \mid bb \mid bab^*a)^*b$$

beschreiben.

Betrachten wir ein weiteres Beispiel.

Beispiel 4.50 Es ist ein regulärer Ausdruck zu bestimmen, der die Menge aller Wörter über dem Alphabet $\{a, b\}$, die nicht das Teilwort *bab* enthalten, beschreibt.

Wir konstruieren zuerst den DEA, der die Menge aller Wörter über dem Alphabet $\{a, b\}$, die das Teilwort *bab* enthalten, beschreibt (siehe Abbildung 4.17). Jetzt können wir daraus sehr einfach

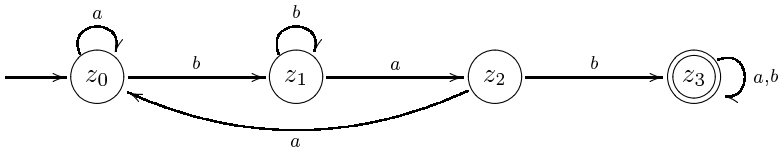


Abbildung 4.17: Ein DEA für die Menge aller Wörter über $\{a, b\}$, die das Teilwort *bab* enthalten

einen DEA konstruieren, der die Komplementärmenge akzeptiert, nämlich durch Vertauschen der akzeptierenden und nichtakzeptierenden Zustände (siehe Abbildung 4.18).

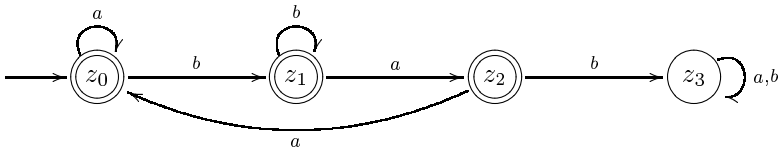


Abbildung 4.18: Ein DEA für die Menge aller Wörter über $\{a, b\}$, die nicht das Teilwort *bab* enthalten

Nun können wir für den DEA aus Abbildung 4.18 das Gleichungssystem der Wortmengen für die einzelnen Zustände aufstellen.

$$\begin{aligned}
L_{z_0} &= \{a\} \cdot L_{z_0} \cup \{b\} \cdot L_{z_1} \cup \{\varepsilon\},\\
L_{z_1} &= \{a\} \cdot L_{z_2} \cup \{b\} \cdot L_{z_1} \cup \{\varepsilon\},\\
L_{z_2} &= \{a\} \cdot L_{z_0} \cup \{b\} \cdot L_{z_3} \cup \{\varepsilon\},\\
L_{z_3} &= \{a\} \cdot L_{z_3} \cup \{b\} \cdot L_{z_3}.
\end{aligned}$$

Aus der vierten Gleichung erhalten wir $L_{z_3} = \{a, b\} \cdot L_{z_3}$ und durch Anwendung des Lemmas 4.48 schließlich $L_{z_3} = \{a, b\}^* \cdot \emptyset = \emptyset$. Setzen wir dies in die dritte Gleichung ein, erhalten wir folgendes neue Gleichungssystem.

$$\begin{aligned}
L_{z_0} &= \{a\} \cdot L_{z_0} \cup \{b\} \cdot L_{z_1} \cup \{\varepsilon\},\\
L_{z_1} &= \{a\} \cdot L_{z_2} \cup \{b\} \cdot L_{z_1} \cup \{\varepsilon\},\\
L_{z_2} &= \{a\} \cdot L_{z_0} \cup \{\varepsilon\}.
\end{aligned}$$

Mittels der dritten Gleichung können wir L_{z_2} in der zweiten Gleichung ersetzen und erhalten

$$\begin{aligned}
L_{z_1} &= \{a\} \cdot L_{z_2} \cup \{b\} \cdot L_{z_1} \cup \{\varepsilon\}\\
&= \{a\} \cdot (\{a\} \cdot L_{z_0} \cup \{\varepsilon\}) \cup \{b\} \cdot L_{z_1} \cup \{\varepsilon\}\\
&= \{aa\} \cdot L_{z_0} \cup \{b\} \cdot L_{z_1} \cup \{\varepsilon, a\}\\
&= \{b\} \cdot L_{z_1} \cup \{aa\} \cdot L_{z_0} \cup \{\varepsilon, a\}.
\end{aligned}$$

Wenden wir nun auf diese Gleichung unser Lemma an, erhalten wir

$$\begin{aligned} L_{z_1} &= \{b\}^* \cdot (\{aa\} \cdot L_{z_0} \cup \{\varepsilon, a\}) \\ &= \{b\}^* \cdot \{aa\} \cdot L_{z_0} \cup \{b\}^* \cdot \{\varepsilon, a\}. \end{aligned}$$

Setzen wir nun dieses Ergebnis in die Gleichung für L_{z_0} ein, so haben wir

$$\begin{aligned} L_{z_0} &= \{a\} \cdot L_{z_0} \cup \{b\} \cdot L_{z_1} \cup \{\varepsilon\} \\ &= \{a\} \cdot L_{z_0} \cup \{b\} \cdot (\{b\}^* \cdot \{aa\} \cdot L_{z_0} \cup \{b\}^* \cdot \{\varepsilon, a\}) \cup \{\varepsilon\} \\ &= (\{a\} \cup \{b\} \cdot \{b\}^* \cdot \{aa\}) \cdot L_{z_0} \cup \{b\} \cdot \{b\}^* \cdot \{\varepsilon, a\} \cup \{\varepsilon\}. \end{aligned}$$

Daraus erhalten wir nach Anwendung unseres Lemmas

$$L_{z_0} = (\{a\} \cup \{b\} \cdot \{b\}^* \cdot \{aa\})^* \cdot (\{b\} \cdot \{b\}^* \cdot \{\varepsilon, a\} \cup \{\varepsilon\}),$$

und somit beschreibt der reguläre Ausdruck

$$(a \mid bb^*aa)^*(bb^*(\varepsilon \mid a) \mid \varepsilon)$$

unsere gegebene Menge der Wörter über dem Alphabet $\{a, b\}$, die nicht das Teilwort *bab* enthalten.

4.3.4 Das Pumping Lemma

In diesem Kapitel behandeln wir einen Satz, mit dem man zeigen kann, dass eine Sprache *nicht* regulär ist.

Satz 4.51 (Pumping Lemma) *Sei L eine reguläre Sprache. Dann gibt es eine Konstante $k \in \mathbb{N}$, so dass für alle Wörter $z \in L$ mit $|z| \geq k$ Wörter u, v und w existieren, so dass gilt:*

(i) $z = uvw$,

(ii) $|uv| \leq k$ *and* $|v| \geq 1$,

(iii) *für alle* $i \in \mathbb{N}$ *gilt* $uv^iw \in L$.

Beweis. Sei L eine reguläre Sprache, dann gibt es einen DEA A, der L akzeptiert. Sei nun k die Anzahl der Zustände von A.

Wir betrachten ein Wort z, welches von A akzeptiert wird und für das $|z| \geq k$ gilt. Beim Einlesen von z durchläuft der DEA offensichtlich $|z| + 1 \geq k + 1$ Zustände (einschließlich des Startzustands). Da der Automat aber nur k Zustände hat, muß der DEA A beim Einlesen von z zweimal denselben Zustand durchlaufen, also durchläuft der Automat eine Schleife. Wir bezeichnen mit u den Teil des Wortes, den A vor Erreichen der Schleife eingelesen hat, mit v den Teil, der während der Schleife eingelesen wurde, mit w den Rest des Wortes. Offensichtlich gilt $|uv| \leq k$ und $v \geq 1$.

Da wir eine Schleife durchlaufen haben, kann der Automat natürlich auch die Schleife meiden ($i = 0$) oder zweimal durchlaufen ($i = 2$) oder beliebig oft durchlaufen. Mit anderen Worten, für alle $i \geq 0$ gilt $uv^iw \in L$, was zu beweisen war. □

Bemerkung 4.52 Schaut man sich die logische Struktur des Pumping Lemmas an, erkennt man, dass es sich um eine Implikation handelt, dass heißt, die Umkehrung *muss nicht* wahr sein. Hier ist es tatsächlich so, es gibt Sprachen, die nicht regulär sind, für die aber die Behauptungen des Pumping Lemmas erfüllt sind (siehe Abbildung 4.19). Damit eignet sich das Pumping Lemma gut, um zu zeigen, dass gewisse Sprachen *nicht regulär* sind, man kann aber auf *keinen Fall* zeigen, dass gewisse Sprachen *regulär* sind!!!.

Beispiel 4.53 Wir betrachten die Sprache

$$L = \{a^n b^n \mid n \geq 1\},$$

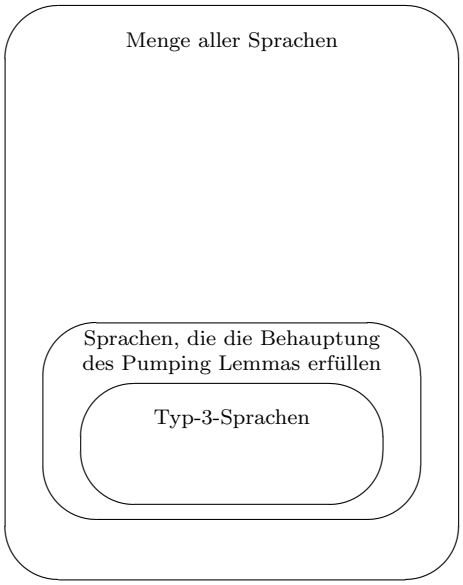


Abbildung 4.19: Hierarchie mit Menge der Sprachen, die die Behauptung des Pumping Lemmas erfüllen

von der wir schon länger vermuten, dass sie nicht regulär ist. Das wollen wir jetzt mit dem Pumping Lemma durch die indirekte Beweismethode beweisen.

Dazu nehmen wir an, L sei regulär. Dann gibt es laut Pumping Lemma eine Konstante k, so dass für alle Wörter $z \in L$ mit $|z| \geq k$ Wörter u, v und w existieren, so dass gilt:

(i) $z = uvw$,

(ii) $|uv| \leq k$ and $|v| \geq 1$,

(iii) für alle $i \in \mathbb{N}$ gilt $uv^i w \in L$.

Betrachten wir speziell das Wort $z = a^k b^k$, dann gilt offensichtlich $|z| = 2\,k \geq k$, also gibt es eine Zerlegung $z = a^k b^k = uvw$ mit $|uv| \leq k$ and $|v| \geq 1$. Daraus ergibt sich:

$$\begin{aligned} uv &= a^r \quad \text{für } r \leq k, \\ w &= a^{k-r} b^k, \\ v &= a^s \quad \text{für } 1 \leq s \leq r \leq k. \end{aligned}$$

Nach (iii) ist jedes Wort $uv^i w \in L$, also auch

$$uv^2 w \in L. \tag{4.4}$$

Andererseits gilt

$$uv^2 w = uvvw = a^r a^s a^{k-r} b^k = a^{k+s} b^k$$

mit $s \geq 1$, also $k + s \neq k$, das heißt

$$uv^2 w \notin L, \tag{4.5}$$

was einen Widerspruch zu 4.4 darstellt. Folglich war unsere Annahme (L regulär) falsch, also kann $L = \{a^n b^n \mid n \geq 1\}$ *nicht regulär* sein.

Beispiel 4.54 Mit Hilfe des Pumping Lemmas kann man auch die Nichtregularität der Sprachen

$L = \{a^{n^2} \mid n \geq 1\}$,
$L = \{a^{2^n} \mid n \geq 1\}$,
$L = \{a^p \mid p \text{ Primzahl}\}$,
$L = \{a^n b^n c^n \mid n \geq 1\}$,
$L = \{ww^R \mid w \in \{a, b\}^*\}$,
$L = \{ww \mid w \in \{a, b\}^*\}$,
$L = \text{EXPR}$

und vielen anderen zeigen.

4.3.5 Abschlusseigenschaften

Wir wollen in diesem Kapitel untersuchen, unter welchen Operationen die Menge der regulären Sprachen abgeschlossen ist. Eine Menge M ist *unter der k-stelligen Operation* op *abgeschlossen*, falls für alle $a_1, a_2, \ldots, a_k \in M$ auch $\text{op}(a_1, a_2, \ldots, a_k) \in M$ gilt.

Satz 4.55 *Die Menge der regulären Sprachen ist unter den Operationen*

(i) *Vereinigung,*
(ii) *Durchschnitt,*
(iii) *Komplement,*
(iv) *Produkt (Konkatenation) und*
(v) *Kleene-Stern*

abgeschlossen.

Beweis. (i), (iv) und (v): Der Abschluss unter den Operationen Vereinigung, Produkt und Kleene-Stern ist eine direkte Folgerung aus der Definition der regulären Ausdrücke, da mit zwei gegebenen regulären Ausdrücken α und β auch $\alpha\beta$, $(\alpha|\beta)$ sowie $(\alpha)^*$ reguläre Ausdrücke sind und somit die beschriebenen Sprachen regulär sind.

(iii): Der Abschluss unter der Operation Komplementbildung wird folgendermaßen nachgewiesen. Sei L eine reguläre Sprache. Dann existiert ein DEA $A = (Z, \Sigma, \delta, z_0, E)$ mit $T(A) = L$. Wir konstruieren den DEA $A' = (Z, \Sigma, \delta, z_0, \Sigma \setminus E)$. Offensichtlich gilt dann $T(A') = \overline{L}$, d. h., durch das Vertauschen von akzeptierenden und nichtakzeptierenden Zuständen akzeptiert A' genau die Wörter, die A nicht akzeptiert hat, also das Komplemt von L.

(ii): Offensichtlich gilt

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

(folgt direkt aus Gleichung 1.42 auf Seite 11), damit ist durch den Abschluss unter Komplement und Vereinigung auch der Abschluss unter Durchschnitt gegeben.

Der Abschluss unter Durchschnittsbildung kann allerdings auch direkt durch Konstruktion des sogenannten *Produktautomaten* nachgewiesen werden (siehe z. B. [13]). □

4.3.6 Wortproblem und andere Entscheidbarkeitsprobleme

Nach Satz 4.21 ist das Wortproblem für Typ-3-Grammatiken *entscheidbar*, dann natürlich *auch für deterministische endliche Automaten*, denn wir können zu einem deterministischen endlichen Automaten eine äquivalente Typ-3-Grammatik konstruieren. Allerdings weist der Algorithmus, der im Beweis des Satzes 4.21 benutzt wurde, eine exponentielle Laufzeit aus, für praktische Anwendungen nicht brauchbar.

Wir betrachten jetzt das Wortproblem für DEA.

Gegeben: DEA $A = (Z, \Sigma, \delta, z_0, E)$, und Wort $w \in \Sigma^*$,

Frage: Gilt $w \in T(A)$?

Dann gilt.

Satz 4.56 *Das Wortproblem für deterministische endliche Automaten ist in linearer Zeit entscheidbar.*

Beweis. Der Beweis ist trivial: Nach dem „Einlesen“ des Wortes w wissen wir, ob $w \in T(A)$ gilt oder nicht (je nachdem ob ein akzeptierender Zustand erreicht wurde oder nicht). Und das Einlesen benötigt genau n Schritte, falls das Wort w die Länge n hat. □

Man betrachtet noch andere Entscheidbarkeitsprobleme für deterministische endliche Automaten.

Leerheitsproblem:

Gegeben: DEA A.

Frage: Gilt $T(A) = \emptyset$?

Endlichkeitsproblem:

Gegeben: DEA A.

Frage: Ist $T(A)$ endlich?

Schnittproblem:

Gegeben: Zwei DEA's A_1 und A_2.

Frage: Gilt $T(A_1) \cap T(A_2) = \emptyset$?

Äquivalenzproblem:

Gegeben: Zwei DEA's A_1 und A_2.

Frage: Gilt $T(A_1) = (A_2)$?

Zusammenfassend gilt folgender Satz, den wir aber hier nicht beweisen wollen.

Satz 4.57 *Das Leerheitsproblem, das Endlichkeitsproblem, das Schnittproblem sowie das Äquivalenzproblem für DEA sind entscheidbar.* □

4.4 Kontextfreie Sprachen

In diesem Kapitel wollen wir uns näher mit den kontextfreien Sprachen beschäftigen. Wir wissen aus dem vorhergehenden Kapitel, dass die Menge der regulären Sprachen zwar einfach handhabbar ist, aber leider gehören schon etwas komplexere Sprachen nicht zu den regulären Sprachen, wie folgende Beispiele zeigen.

Beispiel 4.58 Die nichtreguläre Sprache $L = \{a^n b^n \mid n \geq 1\}$ wird offensichtlich durch die kontextfreie Grammatik $G = (\{S\}, \{a, b\}, \{S \to aSb,\ S \to ab\}, S)$ erzeugt.

Beispiel 4.59 Wir betrachten die kontextfreie Grammatik

$$G = (\{S, X, C\}, \{x, c, 0, +, -, ;, :=, \neq, \text{LOOP}, \text{WHILE}, \text{DO}, \text{END}\}, P, S)$$

mit folgenden Regeln in der Menge P.

$$\begin{aligned}
&S \to S; S\\
&S \to \text{LOOP } X \text{ DO } S \text{ END}\\
&S \to \text{WHILE } X \neq 0 \text{ DO } S \text{ END}\\
&S \to X := X + C\\
&S \to X := X - C\\
&X \to x\\
&C \to c
\end{aligned}$$

Die erzeugte Sprache ist die Menge aller korrekten LOOP/WHILE-Programme. Ein korrektes Programm muss auch korrekt *geklammert* sein, nämlich die Klammerung durch die Schlüsselworte DO und END. Man kann zeigen, dass solche Sprachen nicht regulär sind.

Ein weiteres Beispiel für eine kontextfreie Sprache, die nicht regulär ist, ist die Menge der korrekt geklammerten arithmetischen Ausdrücke EXPR (siehe Beispiele 4.7 und 4.1).

4.4.1 Normalformen

Wir wollen jetzt Normalformen kontextfreier Grammatiken betrachten, das heißt Grammatiken, in denen die Regeln von sehr einfacher Natur sind.

Definition 4.60 (Chomsky Normalform) *Eine kontextfreie Grammatik $G = (V, \Sigma, P, S)$ heißt in Chomsky Normalform, falls jede Regel in P von der Form*

(i) *$A \to BC$ mit $A, B, C \in V$ oder*

(ii) *$A \to a$ mit $A \in V$ und $a \in \Sigma$ oder*

(iii) *$S \to \varepsilon$, falls S auf keiner rechten Seite einer Regel vorkommt,*

ist.

Wir sprechen von einer *Normalform*, da folgender Satz gilt.

Satz 4.61 *Zu jeder kontextfreien Grammatik G kann man eine äquivalente kontextfreie Grammatik G' in Chomsky Normalform konstruieren, das heißt, es gilt dann $L(G) = L(G')$.* □

Wir wollen den Beweis hier nicht erbringen, sondern auf die Literatur verweisen. Man findet die Konstruktionen in fast jedem Lehrbuch. Wir möchten an dieser Stelle nur kurz die einzelnen Schritte skizzieren.

Wenn man eine Grammatik G hat, so macht man sie zunächst „ε-frei“, wir haben schon darauf hingewiesen. Dann beseitigt man alle „Kettenregeln“, das heißt Regeln vom Typ $A \to B$. Schließlich führt man für jedes Terminalzeichen a eine neue Regel $X_a \to a$ ein und ersetzt in allen rechten

Seiten, die länger als eins sind, alle Terminalzeichen a durch das jeweilige Nichtterminalzeichen X_a und letztlich „verkürzt“ man noch alle rechten Seiten, die mehr als zwei Nichtterminale enthalten. Wir möchten an dieser Stelle für den interessierten Leser ein Beispiel für solch eine Konstruktion angeben.

Beispiel 4.62 Wir betrachten die Grammatik $G = (\{S, A, B\}, \{a, b, c\}, P, S)$ mit folgenden Regeln in P.

$$S \to cSc \mid AB,$$
$$A \to aAb \mid ab,$$
$$B \to cBb \mid \varepsilon.$$

Das Konstruieren einer äquivalenten kontextfreien Grammatik G' in Chomsky-Normalform zu G geschieht folgendermaßen.

Das Beseitigen der ε-Regeln (also von Regeln $A \to \varepsilon$ für $A \neq S$) geschieht in zwei Teilschritten, zuerst wird eine Grammatik G_1 konstruiert, in der in keiner rechten Seite einer Regel das Startsymbol S auftaucht: $G_1 = (\{S, S', A, B\}, \{a, b, c\}, P_1, S)$ mit folgenden Regeln in P_1:

$$S \to cS'c \mid AB, \qquad A \to aAb \mid ab,$$
$$S' \to cS'c \mid AB, \qquad B \to cBb \mid \varepsilon.$$

Dann beseitigen wir die ε-Regeln, d. h. wir konstruieren $G_2 = (\{S, S', A, B\}, \{a, b, c\}, P_2, S)$ „ohne“ ε-Regeln. Falls $Y \overset{*}{\Longrightarrow} \varepsilon$ gilt, nehmen wir für jede Regel $X \to \alpha Y \beta$ mit $\alpha\beta \neq \varepsilon$ auch die Regel $X \to \alpha\beta$ hinzu. Konkret erhalten wir:

$$S \to cS'c \mid AB \mid A, \qquad A \to aAb \mid ab,$$
$$S' \to cS'c \mid AB \mid A, \qquad B \to cBb \mid cb.$$

Jetzt beseitigt man die Kettenregeln. Falls eine Regel $X \to Y$ wegfällt, und $Y \to \alpha$ existiert, gibt es Ableitungen $X \Longrightarrow Y \Longrightarrow \alpha$, die natürlich nicht ersatzlos wegfallen dürfen. Deshalb führen wir die direkte Regel $X \to \alpha$ ein. Konkret für unser Beispiel erhalten wir $G_3 = (\{S, S', A, B\}, \{a, b, c\}, P_3, S)$, wobei P_3 aus den Regeln

$$S \to cS'c \mid AB \mid aAb \mid ab, \qquad A \to aAb \mid ab,$$
$$S' \to cS'c \mid AB \mid aAb \mid ab, \qquad B \to cBb \mid cb$$

besteht.

Dann wird sichergestellt, dass die rechten Seiten der Regeln entweder genau aus einem Terminal oder aber aus einem nonterminalen Wort bestehen. Dabei ergibt sich die Grammatik $G_4 = (\{S, S', A, B, X_a, X_b, X_c\}, \{a, b, c\}, P_4, S)$, wobei in P_4 genau die Regeln

$$S \to X_cS'X_c \mid AB \mid X_aAX_b \mid X_aX_b, \qquad X_a \to a,$$
$$S' \to X_cS'X_c \mid AB \mid X_aAX_b \mid X_aX_b, \qquad X_b \to b,$$
$$A \to X_aAX_b \mid X_aX_b, \qquad X_c \to c,$$
$$B \to X_cBX_b \mid X_cX_b$$

enthalten sind.

Im letzten Schritt werden jetzt die nonterminalen Wörter auf den rechten Seiten der Regeln „verkürzt“. Damit ist $G' = (\{S, S', A, B, X_a, X_b, X_c, D_1, D_2, D_3, D_4, D_5, D_6\}, \{a, b, c\}, P', S)$, wo-

bei in P' genau die Regeln

$$\begin{aligned}
S &\to X_cD_1 \mid AB \mid X_aD_2 \mid X_aX_b, & D_1 &\to S'X_c,\\
S' &\to X_cD_3 \mid AB \mid X_aD_4 \mid X_aX_b, & D_2 &\to AX_b,\\
A &\to X_aD_5 \mid X_aX_b, & D_3 &\to S'X_c,\\
B &\to X_cD_6 \mid X_cX_b, & D_4 &\to AX_b,\\
X_a &\to a, & D_5 &\to AX_b,\\
X_b &\to b, & D_6 &\to BX_b,\\
X_c &\to c
\end{aligned}$$

enthalten sind.

Nachdem wir alle Schritte durchgeführt haben, ist obige Grammatik G' eine zu G äquivalente kontextfreie Grammatik in Chomsky-Normalform.

Für eine kontextfreie Grammatik in Chomsky Normalform kann man folgende Beobachtungen machen:

1. Der Syntaxbaum einer Ableitung ist ein binärer Baum (nur die Blätter hängen immer einzeln am darüberliegenden Knoten).
2. Damit kann man für die Tiefe t eines Syntaxbaumes für ein Wort w abschätzen: $\log_2 |w| < t$.
3. Jede Ableitung für ein Wort w der Sprache besteht aus genau $2 \cdot |w| - 1$ Schritten.

Wir möchten noch eine weitere Normalform für kontextfreie Sprachen, nämlich die GREIBACH[10] *Normalform*, nennen.

Definition 4.63 (Greibach Normalform) *Eine kontextfreie Grammatik $G = (V, \Sigma, P, S)$ heißt in Greibach Normalform, falls jede Regel in P von der Form*

(i) *$A \to aB_1B_2 \dots B_n$ mit $n \geq 0$, $A, B_1, \dots, B_n \in V$ und $a \in \Sigma$ oder*

(ii) *$S \to \varepsilon$, falls S auf keiner rechten Seite einer Regel vorkommt,*

ist.

Auch hier gilt folgender Satz, den wir ohne Beweis angeben wollen.

Satz 4.64 *Zu jeder kontextfreien Grammatik G kann man eine äquivalente kontextfreie Grammatik G' in Greibach Normalform konstruieren, das heißt, es gilt dann $L(G) = L(G')$.* □

Wir bemerken, dass die Greibach Normalform in gewisser Weise eine Verallgemeinerung der regulären Grammatik darstellt und dass man noch weiter $0 \leq n \leq 2$ fordern kann.

4.4.2 Das Pumping Lemma

In Analogie zum Pumping Lemma für reguläre Sprachen kann man ein Pumping Lemma für kontextfreie Sprachen aufstellen.

Satz 4.65 (Pumping Lemma) *Sei L eine kontextfreie Sprache. Dann gibt es eine Konstante $k \in \mathbb{N}$, so dass für alle Wörter $z \in L$ mit $|z| \geq k$ Wörter u, v, w, x und y existieren, so dass gilt:*

(i) *$z = uvwxy$,*

(ii) *$|vwx| \leq k$ und $|vx| \geq 1$,*

(iii) *für alle $i \in \mathbb{N}$ gilt $uv^iwx^iy \in L$.* □

[10] S. A. GREIBACH, amerikanische Mathematikerin

Wir wollen keinen Beweis angeben. Mit dem Pumping Lemma für kontextfreie Sprachen kann man beweisen, dass Sprachen *nicht* kontextfrei sind, zum Beispiel die Sprachen

$L = \{a^{n^2} \mid n \geq 1\}$,
$L = \{a^{2^n} \mid n \geq 1\}$,
$L = \{a^p \mid p \text{ Primzahl}\}$,
$L = \{a^n b^n c^n \mid n \geq 1\}$,
$L = \{ww \mid w \in \{a, b\}^*\}$.

Wir möchten auch hier darauf hinweisen, dass das Pumping Lemma eine notwendige Eigenschaft für kontextfreie Sprachen formuliert. Also auch hier gilt in Analogie zum regulären Fall:

Mit dem Pumping Lemma für kontextfreie Sprachen kann man nicht beweisen, dass eine Sprache kontextfrei ist; sondern nur, dass eine Sprache nicht kontextfrei ist!

4.4.3 Abschlusseigenschaften

Auch die Menge der kontextfreien Sprachen wollen wir wieder auf Abschlusseigenschaften hin untersuchen.

Satz 4.66 *Die Menge der kontextfreien Sprachen ist unter den Operationen*

(i) *Vereinigung,*

(ii) *Produkt (Konkatenation) und*

(iii) *Kleene-Stern*

abgeschlossen. Sie ist unter den Operationen

(iv) *Durchschnitt und*

(v) *Komplement*

nicht abgeschlossen.

Beweis. Wir wollen für die positiven Aussagen ohne weitere Bemerkungen nur die Konstruktionen angeben. Es seien zwei kontextfreie Grammatiken $G_1 = (V_1, \Sigma, P_1, S_1)$ und $G_2 = (V_2, \Sigma, P_2, S_2)$ für zwei kontextfreie Sprachen L_1 und L_2 gegeben.

(i) Wir konstruieren $G = (V, \Sigma, P, S)$ durch

$V = V_1 \cup V_2 \cup \{S\}$,
$P = P_1 \cup P_2 \cup \{S \to S_1,\, S \to S_2\}$.

Offensichtlich ist dann auch G kontextfrei und es gilt $L(G) = L_1 \cup L_2$.

(ii) Wir konstruieren $G' = (V', \Sigma, P', S')$ durch

$V' = V_1 \cup V_2 \cup \{S'\}$,
$P' = P_1 \cup P_2 \cup \{S' \to S_1 S_2\}$.

Offensichtlich ist dann auch G kontextfrei und es gilt $L(G) = L_1 \cdot L_2$.

(iii) Wir konstruieren $G'' = (V'', \Sigma, P'', S'')$ durch

$V'' = V_1 \cup \{S''\}$,
$P'' = P_1 \cup \{S'' \to \varepsilon,\, S'' \to S_1,\, S_1 \to S_1 S_1\}$.

Offensichtlich ist dann auch G kontextfrei und es gilt $L(G) = L_1^*$.

(iv) Kommen wir jetzt zum Beweis, dass die Menge der kontextfreien Sprachen nicht unter Durchschnittbildung abgeschlossen ist. Dazu reicht die Angabe eines Gegenbeispiels. Sei

$L_1 = \{a^n b^n c^m \mid m, n \geq 1\}$ und $L_2 = \{a^n b^m c^m \mid m, n \geq 1\}$. Offensichtlich sind L_1 und L_2 kontextfreie Sprachen (siehe Übungsaufgaben oder Beispiel 4.68). Der Durchschnitt beider Mengen $L_1 \cap L_2 = \{a^n b^n c^n \mid n \geq 1\}$ ist aber nicht kontextfrei.

(v) Zum Schluss der Nachweis der Nichtabgeschlossenheit der Menge der kontextfreien Sprachen unter Komplementbildung. Wir werden den Nachweis indirekt führen. Also Annahme: die Menge der kontextfreien Sprachen ist unter Komplement abgeschlossen. Dann gilt gemäß der Gleichung (1.42) für den Durchschnitt zweier Mengen

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

also wäre mit den kontextfreien Sprachen L_1 und L_2 wegen (i) und unserer Annahme auch $L_1 \cap L_2$ kontextfrei, was einen Widerspruch zur obigen Aussage (iv) darstellt. Damit ist unsere Annahme falsch, die Aussage (v) bewiesen und der Beweis des Satzes somit komplett. □

4.4.4 Entscheidbarkeit und der Algorithmus von Cocke, Younger und Kasami

Das Wortproblem für Typ-1-Sprachen ist entscheidbar, ein entsprechender Algorithmus benötigt aber exponentiellen Zeitaufwand. Für kontextfreie Sprachen können wir einen Algorithmus angeben, der das Wortproblem effizienter löst. Allerdings muss dazu die gegebene Grammatik in Chomsky Normalform vorliegen.

Eingabe: Grammatik $G = (V, \Sigma, P, S)$ in Chomsky Normalform und Wort $x = a_1 a_2 \dots a_n$
Ausgabe: **JA**, falls $x \in L(G)$; **Nein**, sonst
`FOR` $i := 1$ `TO` n `DO`
  $T[i,1] := \{A \mid A \to a_i \in P\}$
`END;`
`FOR` $j := 2$ `TO` n `DO`
  `FOR` $i := 1$ `TO` $n + 1 - j$ `DO`
    $T[i,j] := \emptyset$;
    `FOR` $k := 1$ `TO` $j - 1$ `DO`
      $T[i,j] := T[i,j] \cup \{A \mid A \to BC \in P,\ B \in T[i,k],\ C \in T[i+k, j-k]\}$
    `END`
  `END`
`END;`
`IF` $S \in T[1,n]$ `THEN`
  `return JA`
`ELSE`
  `return NEIN`
`END`

Abbildung 4.20: Der Algorithmus von Cocke, Younger und Kasami

Satz 4.67 *Der Algorithmus von Cocke, Younger und Kasami löst das Wortproblem für kontextfreie Grammatiken in Chomsky Normalform mit einem Zeitaufwand von $O(n^3)$.* □

Beweisen wollen wir den Satz nicht, eine genaue Analyse des Algorithmus liefert aber relativ schnell das gewünschte Ergebnis. Die wesentlich Beobachtung ist, dass $T[i,j]$ die Menge aller Nichtterminale ist, die das an der Stelle i beginnende Teilwort der Länge j, also $a_i a_{i+1} \cdots a_{i+j-1}$, erzeugen. Zum Zeitverhalten muss man sagen, dass natürlich auch das Herstellen der Chomsky Normalform Zeit kostet und somit beachtet werden muss, allerdings fällt der Aufwand zum Herstellen der Chomsky Normalform ja nur einmal an und ist nur abhängig von der Größe der Grammatik.

Wir wollen den Algorithmus von Cocke, Younger und Kasami an einem Beispiel anwenden.

Beispiel 4.68 Die Sprache $L = \{a^n b^n c^m \mid m, n \geq 1\}$ ist kontextfrei und wird von der Grammatik $G = (V, \Sigma, P, S)$ mit den Regeln

$$S \to AB,$$
$$A \to aAb \mid ab,$$
$$B \to cB \mid c$$

erzeugt. Eine kontextfreie Grammatik in Chomsky Normalform für L hätte folgende Menge von Regeln:

$$S \to AB, \qquad C \to a,$$
$$A \to CD \mid CF, \qquad D \to b,$$
$$B \to EB \mid c, \qquad E \to c,$$
$$F \to AD.$$

Sei $x = aaabbbcc$. Dann erzeugt der Algorithmus die Tabelle in Abbildung 4.21.

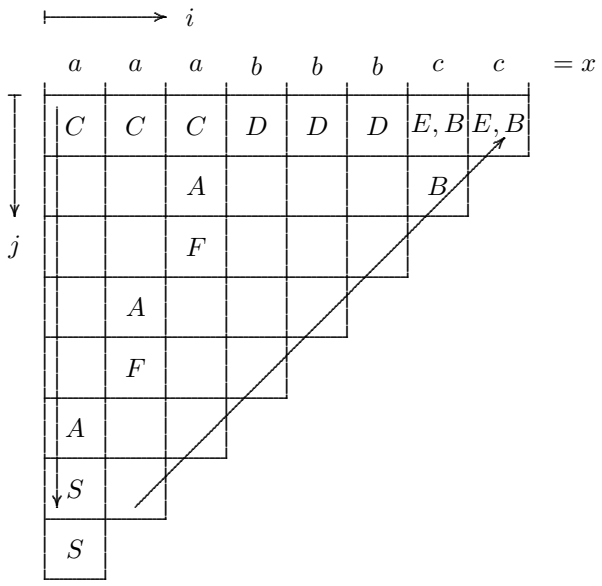


Abbildung 4.21: Tabelle zum Algorithmus von CYK

In der Tabelle stellt jede Zelle eine Menge $T[i, j]$ von Nichtterminalen dar. Die Koordinatenachsen für i und j sind eingezeichnet.

Der Algorithmus besteht aus drei Teilen. Im ersten Teil wird die oberste Zeile ($T[i, 1]$) berechnet.

Im zweiten Teil werden zeilenweise die weiteren Mengen $T[i, j]$ berechnet, indem immer die darüberliegende Spalte von oben und die nach rechts oben führende Diagonale betrachtet werden. Die Menge $T[1, 8]$ würde dann zum Beispiel folgendermaßen ermittelt werden (zur besseren Lesbarkeit wurden die Variablen A, B, C hier mit X, Y, Z bezeichnet, um sie von den Nichterminalzeichen

abzugrenzen). Leere Zellen stehen dabei für die leere Menge.

$$\begin{aligned}
T[1,8] &= \{X \mid X \to YZ \in P,\ Y \in T[1,1],\ Z \in T[2,7]\} && \text{(für } k=1\text{)}\\
&\quad \cup \{X \mid X \to YZ \in P,\ Y \in T[1,2],\ Z \in T[3,6]\} && \text{(für } k=2\text{)}\\
&\quad \cup \{X \mid X \to YZ \in P,\ Y \in T[1,3],\ Z \in T[4,5]\} && \text{(für } k=3\text{)}\\
&\quad \cup \{X \mid X \to YZ \in P,\ Y \in T[1,4],\ Z \in T[5,4]\} && \text{(für } k=4\text{)}\\
&\quad \cup \{X \mid X \to YZ \in P,\ Y \in T[1,5],\ Z \in T[6,3]\} && \text{(für } k=5\text{)}\\
&\quad \cup \{X \mid X \to YZ \in P,\ Y \in T[1,6],\ Z \in T[7,2]\} && \text{(für } k=6\text{)}\\
&\quad \cup \{X \mid X \to YZ \in P,\ Y \in T[1,7],\ Z \in T[8,1]\} && \text{(für } k=7\text{)}\\
&= \{X \mid X \to YZ \in P,\ Y \in \{C\},\ Z \in \emptyset\}\\
&\quad \cup \{X \mid X \to YZ \in P,\ Y \in \emptyset,\ Z \in \emptyset\}\\
&\quad \cup \{X \mid X \to YZ \in P,\ Y \in \emptyset,\ Z \in \emptyset\}\\
&\quad \cup \{X \mid X \to YZ \in P,\ Y \in \emptyset,\ Z \in \emptyset\}\\
&\quad \cup \{X \mid X \to YZ \in P,\ Y \in \emptyset,\ Z \in \emptyset\}\\
&\quad \cup \{X \mid X \to YZ \in P,\ Y \in \{A\},\ Z \in \{B\}\}\\
&\quad \cup \{X \mid X \to YZ \in P,\ Y \in \{S\},\ Z \in \{E,B\}\}\\
&= \emptyset \cup \emptyset \cup \emptyset \cup \emptyset \cup \emptyset \cup \{S\} \cup \emptyset\\
&= \{S\}.
\end{aligned}$$

Im dritten Teil des Algorithmus schließlich wird sich die Menge $T[1,n]$ angeschaut, hier $T[1,8]$. Gilt $S \in T[1,n]$, dann und nur dann gibt der Algorithmus „ja“ aus, das heißt, es gibt eine Ableitung $S \overset{*}{\Longrightarrow} x$, hier $S \overset{*}{\Longrightarrow} aaabbbcc$, somit erzeugt die gegebene Grammatik $x = aaabbbcc$.

Noch eine Bemerkung: aus der aufgestellten Tabelle kann man auch die Ableitung für das Wort x ablesen, indem wir rückwärts die Regeln anwenden, die auf S in $T[1,n]$ geführt haben. Hier sieht die Ableitung für *aaabbbc* dann folgendermaßen aus (die Nonterminale, die ersetzt werden, sind jeweils unterstrichen).

$$\begin{aligned}
&\underline{S} \Longrightarrow \underline{A}B \Longrightarrow C\underline{F}B \Longrightarrow C\underline{A}DB \Longrightarrow CC\underline{F}DB \Longrightarrow CC\underline{A}DDB \Longrightarrow \underline{C}CCDDDB\\
&\quad \Longrightarrow a\underline{C}CDDDB \Longrightarrow aa\underline{C}DDDB \Longrightarrow aaa\underline{D}DDB \Longrightarrow aaab\underline{D}DB \Longrightarrow aaabb\underline{D}B\\
&\quad \Longrightarrow aaabbb\underline{B} \Longrightarrow aaabbbc
\end{aligned}$$

Für die anderen Entscheidbarkeitsprobleme, die wir noch im Kapitel 4.3.6 für die Menge der regulären Sprachen formuliert haben kann man folgenden Satz angeben.

Satz 4.69 *Das Leerheitsproblem und das Endlichkeitsproblem für kontextfreie Grammatiken sind entscheidbar. Das Schnittproblem und das Äquivalenzproblem für kontextfreie Grammatiken sind unentscheidbar.* □

Wir wollen den Satz nicht beweisen, möchten nur bemerken, dass die Unentscheidbarkeit des Äquivalenzproblems für die hohe Komplexität schon der kontextfreien Grammatiken spricht.

4.4.5 Kellerautomaten

Wir wollen jetzt versuchen, ein Automatenmodell für die kontextfreien Grammatiken zu finden. Der Mangel der endlichen Automaten ist, dass sie im Prinzip nicht „zählen“ können, sie können sich nur etwas im Zustand „merken“, aber es gibt nur endlich viele Zustände. Deshalb können sie zum Beispiel die Sprache

$$\begin{aligned}
L &= \{wcw^R \mid w \in \{a,b\}^*\}\\
&= \{a_1a_2\ldots a_nca_n\ldots a_2a_1 \mid n \geq 0,\ a_i \in \{a,b\},\ 1 \leq i \leq n\}
\end{aligned}$$

nicht akzeptieren.

Wir führen jetzt das Modell des *Kellerautomaten* ein, indem wir den endlichen Automaten mit einem zusätzlichen Speicher ausstatten, dem sogenannten *Keller*. Der *Kellerautomat* (englisch *pushdown automaton*, deshalb mit PDA abgekürzt) kann im Speicher schreiben und natürlich lesen, wobei die Überführungen nun auch vom gelesenen Kellersymbol abhängen. Der Keller funktioniert nach dem Prinzip „Last In First Out“ (LIFO), das heißt, er kann nur immer das oberste Symbol lesen. Wir statten den Keller noch mit einem Anfangssymbol aus (#). Zusätzlich erlauben wir auch noch sogenannte „ε-Übergänge“, das heißt der Automat „liest“ auf dem Eingabeband ein ε, also nichts und kann so im Keller „rechnen“, ohne die Eingabe zu verarbeiten. Der Kellerautomat wird als Standardversion immer als *nichtdeterministischer Automat* definiert. Formal sieht die Definition wie folgt aus.

Definition 4.70 (Kellerautomat) *Ein (nichtdeterministischer) Kellerautomat (mit PDA abgekürzt von pushdown automaton) M ist ein 6-Tupel $M = (Z, \Sigma, \Gamma, \delta, z_0, \#)$. Dabei ist*

- *Z das Zustandsalphabet,*
- *Σ das Eingabealphabet,*
- *Γ das Kelleralphabet,*
- *$z_0 \in Z$ der Anfangszustand,*
- *$\delta\colon Z \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \to 2_E^{Z \times (\Gamma \setminus \{\#\})^*}$ die Zustandsüberführungsfunktion (2_E^X bedeutet dabei die Menge der endlichen Teilmengen von X),*
- *$\# \in \Gamma$ das Kelleranfangssymbol.*

Die Arbeitsweise wird folgendermaßen interpretiert:

- Bei $(z', B_1 B_2 \dots B_k) \in \delta(z, a, A)$ mit $a \in \Sigma$ befindet sich der Automat im Zustand z, liest auf dem Eingabeband ein a, liest im Keller ein A (das oberste Symbol) und entfernt dieses A aus dem Keller, geht in den Zustand z' über, bewegt den Lesekopf auf dem Eingabeband einen Schritt nach rechts und schreibt auf den Keller das Wort $B_1 B_2 \dots B_k$ derart, dass B_1 das neue oberste Symbol des Kellers ist.
- Bei $(z', B_1 B_2 \dots B_k) \in \delta(z, \varepsilon, A)$ liest er im Gegensatz zur obigen Aktion auf dem Eingabeband nichts ein und bewegt also seinen Lesekopf auf dem Eingabeband nicht.
- Der Kellerautomat beginnt seine Arbeit immer im Zustand z_0 und mit dem Kellerinhalt #.
- Er akzeptiert die zu Beginn der Arbeit auf dem Eingabeband stehende Zeichenkette, wenn er diese Eingabe vollständig eingelesen hat und der Keller leer ist (auch das Kelleranfangssymbol # steht nicht mehr im Keller).

Formal führen wir wie bei den Turingmaschinen *Konfigurationen* zur augenblicklichen Situationsbeschreibung ein.

Definition 4.71 (Konfiguration eines PDA) *Sei $M = (Z, \Sigma, \Gamma, \delta, z_0, \#)$ ein Kellerautomat, unter einer Konfiguration k verstehen wir ein Tripel $k \in Z \times \Sigma^* \times \Gamma^*$.*

Anschaulich beschreibt eine Konfiguration $k = (z, v, \gamma)$ folgende augenblickliche Situation des PDA: er befindet sich im Zustand z, v ist die noch nicht verarbeitete Eingabe auf dem Eingabeband und im Keller steht das Wort γ, wobei das erste Symbol von γ das oberste Kellersymbol sein soll.

In der Menge der Konfigurationen eines Kellerautomaten können wir jetzt die Überführungsrelation $\vdash$ einführen, wobei

$$k_1 \vdash k_2$$

bedeutet, der Kellerautomat überführt die Konfiguration k_1 in genau einem Schritt in die Konfiguration k_2; auf die formale Definition verzichten wir hier, zur Interpretation siehe oben und siehe Abbildung 4.22.

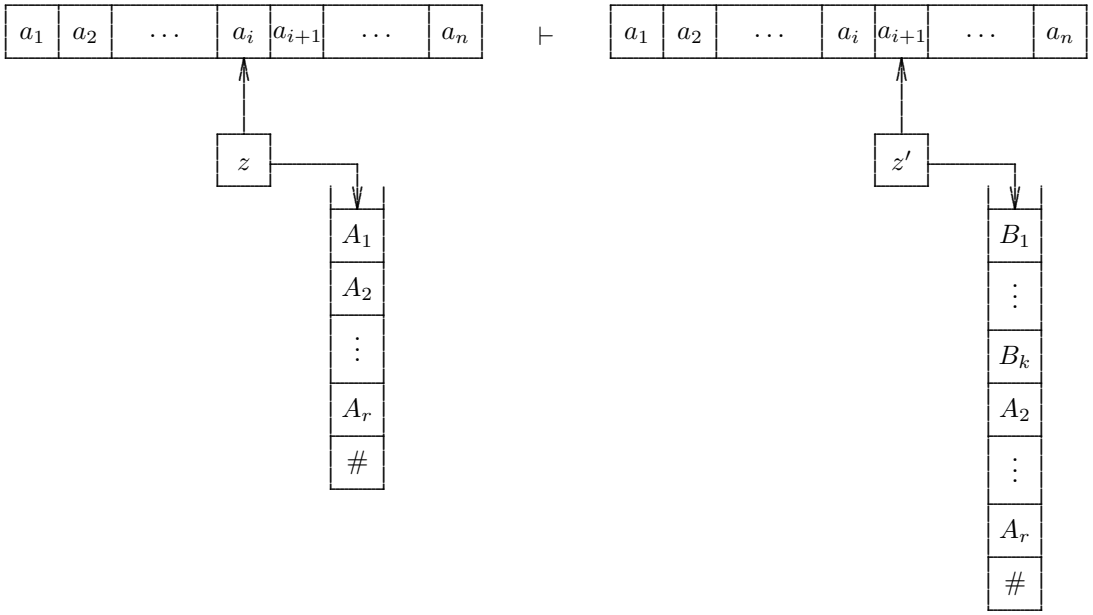


Abbildung 4.22: Interpretation der Überführung $(z', B_1 \dots B_k) \in \delta(z, a_i, A_1)$ eines Kellerautomaten

Unter $k_1 \vdash^* k_2$ wollen wir wieder verstehen, dass der PDA k_1 in endlich vielen Schritten (auch null) in k_2 überführt.

Dann können wir die von einem Kellerautomaten akzeptierte Sprache definieren.

Definition 4.72 (Akzeptierte Sprache eines PDA) *Für einen PDA* $M = (Z, \Sigma, \Gamma, \delta, z_0, \#)$ *sei die von ihm akzeptierte Sprache* $T(M)$ *durch*

$$T(M) = \{w \in \Sigma^* \mid (z_0, w, \#) \vdash^* (z, \varepsilon, \varepsilon) \text{ für ein } z \in Z\}$$

definiert

Es ist Zeit für ein Beispiel.

Beispiel 4.73 Wir geben einen Kellerautomaten für die Sprache

$$L = \{wcw^R \mid w \in \{a, b\}^*\} = \{a_1 a_2 \dots a_n c a_n \dots a_2 a_1 \mid n \geq 0,\ a_i \in \{a, b\},\ 1 \leq i \leq n\}$$

an. Sei

$$M = (\{z_0, z_1\}, \{a, b, c\}, \{A, B, \#\}, \delta, z_0, \#)$$

ein Kellerautomat, wobei δ durch

$$\begin{aligned}
&\delta(z_0, a, X) = \{(z_0, AX)\} \qquad \text{für } X \in \{A, B, \#\},\\
&\delta(z_0, b, X) = \{(z_0, BX)\} \qquad \text{für } X \in \{A, B, \#\},\\
&\delta(z_0, c, X) = \{(z_1, X)\} \qquad \text{für } X \in \{A, B, \#\},\\
&\delta(z_1, a, A) = \{(z_1, \varepsilon)\},\\
&\delta(z_1, b, B) = \{(z_1, \varepsilon)\},\\
&\delta(z_1, \varepsilon, \#) = \{(z_1, \varepsilon)\}
\end{aligned}$$

definiert ist. Dann gilt bei der Eingabe *bacab*

$$\begin{aligned}
(z_0, bacab, \#) &\vdash (z_0, acab, B\#) \vdash (z_0, cab, AB\#) \vdash (z_1, ab, AB\#) \vdash (z_1, b, B\#) \vdash (z_1, \varepsilon, \#)\\
&\vdash (z_1, \varepsilon, \varepsilon),
\end{aligned}$$

also $bacab \in T(M)$.

Folgenden Satz wollen wir ohne Beweis angeben.

Satz 4.74 *Eine Sprache L ist kontextfrei genau dann, wenn ein (nichtdeterministischer) Kellerautomat M mit $T(M) = L$ existiert.* □

Wenn man sich den Kellerautomat im Beispiel 4.73 genauer anschaut, hat jede Konfiguration dieses Automaten höchstens eine mögliche Folgekonfiguration, also ist dieser PDA sogar ein deterministischer. Betrachten wir ein Beispiel für einen echt nichtdeterministischen Kellerautomaten.

Beispiel 4.75 Sei

$$M = (\{z_0, z_1\}, \{a, b\}, \{A, B, \#\}, \delta, z_0, \#)$$

ein Kellerautomat, mit

$$\begin{aligned}
&\delta(z_0, a, X) = \{(z_0, AX)\} \qquad \text{für } X \in \{B, \#\},\\
&\delta(z_0, a, A) = \{(z_1, AA), (z_1, \varepsilon)\},\\
&\delta(z_0, b, X) = \{(z_0, BX)\} \qquad \text{für } X \in \{A, \#\},\\
&\delta(z_0, b, B) = \{(z_0, BB), (z_1, \varepsilon)\},\\
&\delta(z_0, \varepsilon, \#) = \{(z_1, \varepsilon)\},\\
&\delta(z_1, a, A) = \{(z_1, \varepsilon)\},\\
&\delta(z_1, b, B) = \{(z_1, \varepsilon)\},\\
&\delta(z_1, \varepsilon, \#) = \{(z_1, \varepsilon)\},
\end{aligned}$$

dann gilt offensichtlich

$$T(M) = \{ww^R \mid w \in \{a, b\}^*\} = \{a_1 a_2 \dots a_n a_n \dots a_2 a_1 \mid n \geq 0,\ a_i \in \{a, b\},\ 1 \leq i \leq n\}.$$

Man kann zeigen, dass für die Sprache $\{ww^R \mid w \in \{a, b\}^*\}$ *kein* deterministischer Kellerautomat existiert. Die von deterministischen Kellerautomaten akzeptierten Sprachen heißen deterministisch kontextfrei. Es gilt dann folgender Satz.

Satz 4.76 *Die Menge der deterministisch kontextfreien Sprachen ist eine echte Teilmenge der Menge der kontextfreien Sprachen.*

Die deterministisch kontextfreien Sprachen spielen eine wichtige Rolle im Compilerbau. Sie werden durch sogenannte $LR(k)$-Grammatiken beschrieben, die ausreichen, um die Syntax von Programmiersprachen zu beschreiben. Da das Wortproblem für deterministisch kontextfreie Sprachen in linearer Zeit entscheidbar ist, ist somit die Syntaxüberprüfung von Programmen auch in linearer Zeit möglich, ein wesentlicher Vorteil gegenüber der $O(n^3)$-Laufzeit des CYK-Algorithmus, der das Wortproblem für kontextfreie Sprachen entscheidet.

4.5 Rekursiv aufzählbare und kontextabhängige Sprachen

Schließlich sollen noch Automatencharakterisierungen für die beiden umfangreichsten Familien von Sprachen in der Chomsky-Hierarchie angegeben werden. Dabei kommen wir auf ein bekanntes und bereits ausführlich diskutiertes Modell zurück: die Turingmaschine.

Satz 4.77 *Eine Sprache L ist genau dann rekursiv aufzählbar (d. h. vom Typ 0), wenn es eine (nichtdeterministische) Turingmaschine M gibt, die sie akzeptiert, also für die $T(M) = L$ gilt.*

Beweis. Die Beweisidee soll hier nur kurz angedeutet werden. Zu einer Grammatik $G = (V, \Sigma, P, S)$ konstruiert man eine nichtdeterministische Turingmaschine M mit folgender Arbeitsweise. Als Eingabe erhält M ein Wort $w \in \Sigma^*$. Solange auf dem Band von M nicht das Wort S steht, werden nichtdeterministisch eine Regel $\alpha to \beta$ aus P sowie ein Vorkommen von β im aktuellen Bandwort (sofern ein solches existiert) gewählt, das dann durch α ersetzt wird. Es ist offensichtlich, dass ein akzeptierender Lauf von M einer Ableitung bezüglich G in umgekehrter Reihenfolge entspricht.

Umgekehrt sei $M = (Z, \Sigma, \Gamma, \delta, z_0, \Box, E)$ eine NTM. Wir konstruieren die Grammatik $G = (V, \Sigma \cup \{\$, \&\}, P, S)$ mit dem Nichtterminal-Aphabet $V = Z \cup (\Gamma \setminus \Sigma) \cup \{S, A, B\}$. Die Regelmenge P besteht aus folgenden Regeln:

- $S \to \$A, A \to xA, A \to qB, B \to xB, B \to \$$ für alle $x \in \Gamma, q \in E$.
- $bz' \to za$, falls $(z', b, R) \in \delta(z, a)$,
  $z'b \to za$, falls $(z', b, N) \in \delta(z, a)$,
  $z'xb \to xza$, falls $(z', b, R) \in \delta(z, a), x \in \Gamma$.
- $\$\Box \to \$, \Box\$ \to \$$.
- $\$z_0 \to \$\&$.

Mit Hilfe der Regeln der ersten Art erzeugt man ein Wort der Form $\$k\$$, wobei k eine beliebige Endkonfiguration von M ist. Die Regeln der zweiten Art sind so konstruiert, dass $\$k'\$ \Leftarrow_G \$k\$$ genau dann gilt, wenn k' bezüglich M eine Nachfolgekonfiguration von k ist. Durch die Regeln der dritten Art lassen sich führende und abschließende Blanksymbole der Konfiguration entfernen. Schließlich kann man durch Anwendung der letzten Regel ein Wort $\$z_0w\$$ mit $w \in \Sigma^*$ in das Terminalwort $\$\&w\$$ überführen. Nach unserer Konstruktion kann man das Wort $\$\&w\$$ genau dann in G erzeugen, wenn w von M akzeptiert wird. Mit einigen weiteren technischen Änderungen kann man aus G eine Grammatik G' erhalten, so dass $L(G') = T(M)$ gilt. □

Der Unterschied zwischen kontextabhängigen und allgemeinen Grammatiken besteht nur darin, dass bei kontextabhängigen Grammatiken keine verkürzenden Regeln zugelassen sind. Damit sind die in einer Ableitung eines Terminalwortes w auftretenden Satzformen in ihrer Länge durch $|w|$ beschränkt.

Vollzieht man den ersten Teil des Beweises von Satz 4.77 mit einer kontextabhängigen Grammatik nach, so entsteht eine nichtdeterministische Turingmaschine, die nur auf den Zellen der Eingabe sowie zwei zusätzlichen Randzellen links und rechts der Eingabe arbeitet. Eine derart beschränkte NTM heißt *linear beschränkter Automat* (kurz LBA). Auf eine formale Definition des linear beschränkten Automaten wollen hier verzichten.

Führt man umgekehrt die Konstruktion des zweiten Teils des Beweises mit einem LBA durch, so kann man auf die verkürzenden Regeln $\$\Box \to \$$ und $\Box\$ \to \$$ verzichten. Es entsteht also eine Grammatik vom Typ 1 (wobei einige technische Details wiederum noch zu klären wären). Wir können deshalb formulieren:

Satz 4.78 *Eine Sprache L ist genau dann kontextabhängig (d. h. vom Typ 1), wenn es einen (nichtdeterministischen) linear beschränkten Automaten M gibt, die sie akzeptiert, also für den $T(M) = L$ gilt.*

Die Frage nach der Äquivalenz von deterministischen und nichtdeterministischen LBA muss leider offen bleiben; sie ist bis heute nicht beantwortet und ist in der Literatur unter dem Namen *LBA-Problem* bekannt.

4.6 Tabellarischer Überblick

Hier wollen wir die wichtigsten Ergebnisse des Kapitels 4 in tabellarischer Form zusammenfassen. Dabei tauchen in den Tabellen auch Ergebnisse auf, die wir vorher noch nicht explizit genannt, geschweige denn bewiesen haben.

Zunächst betrachten wir die Sprachfamilien der Chomsky Hierarchie (zusätzlich wurden die deterministisch kontextfreien Sprachen aufgenommen, die nicht zur eigentlichen Chomsky Hierarchie zählen) und deren *Charakterisierungsmöglichkeiten*.

Sprache	Grammatik	Automat	Andere
Regulär	Typ 3	Endlicher Automat (EA)	Regulärer Ausdruck
Deterministisch kontextfrei	$LR(k)$	Deterministischer Kellerautomat (DPDA)	
Kontextfrei	Typ 2	(Nichtdeterministischer) Kellerautomat (PDA)	
Kontextabhängig	Typ 1	(Nichtdeterministischer) Linear beschränkter Automat (LBA)	
Rekursiv aufzählbar	Typ 0	Turingmaschine (TM)	

Tabelle 4.2: Charakterisierungen der Sprachfamilien der Chomsky Hierarchie

In der Tabelle 4.2 werden teilweise die Automatentypen hinsichtlich Determinismus spezifiziert, teilweise nicht. Einen Überblick über die Problematik des Verhältnisses von *deterministischen und nichtdeterministischen Varianten* von Automaten gibt die Tabelle 4.3. Die offene Frage der Äquivalenz von deterministischen und nichtdeterministischen linear beschränkten Automaten wird als sogenanntes LBA-Problem bezeichnet.

Nichtdeterminismus	Determinismus	Äquivalenz
NEA	DEA	ja
PDA	DPDA	nein
LBA	DLBA	?
NTM	DTM	ja

Tabelle 4.3: Beziehungen zwischen deterministischen und nichtdeterministischen Automaten

Einen Überblick über die *Abschlusseigenschaften* der Sprachen der Chomsky Hierarchie (erweitert um die deterministisch kontextfreien Sprachen) bezüglich ausgewählter Operationen liefert Tabelle 4.4.

	Typ 3	Det. kf.	Typ 2	Typ 1	Typ 0
Vereinigung	ja	nein	ja	ja	ja
Durchschnitt	ja	nein	nein	ja	ja
Komplement	ja	ja	nein	ja	nein
Konkatenation	ja	nein	ja	ja	ja
Kleene-Stern	ja	nein	ja	ja	ja

Tabelle 4.4: Abschlusseigenschaften

Ausgewählte *Entscheidbarkeitsprobleme* der Sprachen der Chomsky Hierarchie werden in der Tabelle 4.5 betrachtet, wobei „ja“ für entscheidbar und „nein“ für unentscheidbar steht.

	Typ 3	Det. kf.	Typ 2	Typ 1	Typ 0
Wortproblem	ja	ja	ja	ja	nein
Leerheitsproblem	ja	ja	ja	nein	nein
Schnittproblem	ja	nein	nein	nein	nein
Äquivalenzproblem	ja	ja	nein	nein	nein

Tabelle 4.5: Entscheidbarkeitsprobleme

In der Tabelle 4.6 sind die Ergebnisse über die *Zeitkomplexität des Wortproblems* für die Sprachfamilien der Chomsky Hierarchie zusammengefasst, wobei Erwähnung finden sollte, dass man das Wortproblem für kontextfreie Sprachen, gegeben durch kontextfreie Grammatiken in Chomsky Normalform (CNF), durch verbesserte Algorithmen heute bereits in einer Zeitkomplexität von $O(n^{2,38})$ lösen kann. Für das Wortproblem der Typ-1-Sprachen ist bis heute kein Algorithmus mit polynomialem Zeitverhalten bekannt.

Sprache	Komplexität
Typ 3 (DEA gegeben)	linear
Deterministisch kontextfrei	linear
Typ 2 (CNF gegeben)	$O(n^3)$
Typ 1	exponentiell
Typ 0	unentscheidbar

Tabelle 4.6: Komplexität des Wortproblems